Contents

What is Azure Machine Learning?

1/22/2020 • 6 minutes to read • Edit Online

In this article, you learn about Azure Machine Learning, a cloud-based environment you can use to train, deploy, automate, manage, and track ML models.

Azure Machine Learning can be used for any kind of machine learning, from classical ml to deep learning, supervised, and unsupervised learning. Whether you prefer to write Python or R code or zero-code/low-code options such as the designer, you can build, train, and track highly accurate machine learning and deep-learning models in an Azure Machine Learning Workspace.

Start training on your local machine and then scale out to the cloud.

The service also interoperates with popular open-source tools, such as PyTorch, TensorFlow, and scikit-learn.

> **TIP**
>
> **Free trial!** If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today. You get credits to spend on Azure services. After they're used up, you can keep the account and use free Azure services. Your credit card is never charged unless you explicitly change your settings and ask to be charged.

What is machine learning?

Machine learning is a data science technique that allows computers to use existing data to forecast future behaviors, outcomes, and trends. By using machine learning, computers learn without being explicitly programmed.

Forecasts or predictions from machine learning can make apps and devices smarter. For example, when you shop online, machine learning helps recommend other products you might want based on what you've bought. Or when your credit card is swiped, machine learning compares the transaction to a database of transactions and helps detect fraud. And when your robot vacuum cleaner vacuums a room, machine learning helps it decide whether the job is done.

Machine learning tools to fit each task

Azure Machine Learning provides all the tools developers and data scientists need for their machine learning workflows, including:

- The Azure Machine Learning designer (preview): drag-n-drop modules to build your experiments and then deploy pipelines.
- Jupyter notebooks: use our example notebooks or create your own notebooks to leverage our SDK for Python samples for your machine learning.
- R scripts or notebooks in which you use the SDK for R to write your own code, or use the R modules in the designer.
- Visual Studio Code extension
- Machine learning CLI

- Open-source frameworks such as PyTorch, TensorFlow, and scikit-learn and many more

You can even use MLflow to track metrics and deploy models or Kubeflow to build end-to-end workflow pipelines.

Build ML models in Python or R

Start training on your local machine using the Azure Machine Learning Python SDK or R SDK. Then, you can scale out to the cloud.

With many available compute targets, like Azure Machine Learning Compute and Azure Databricks, and with advanced hyperparameter tuning services, you can build better models faster by using the power of the cloud.

You can also automate model training and tuning using the SDK.

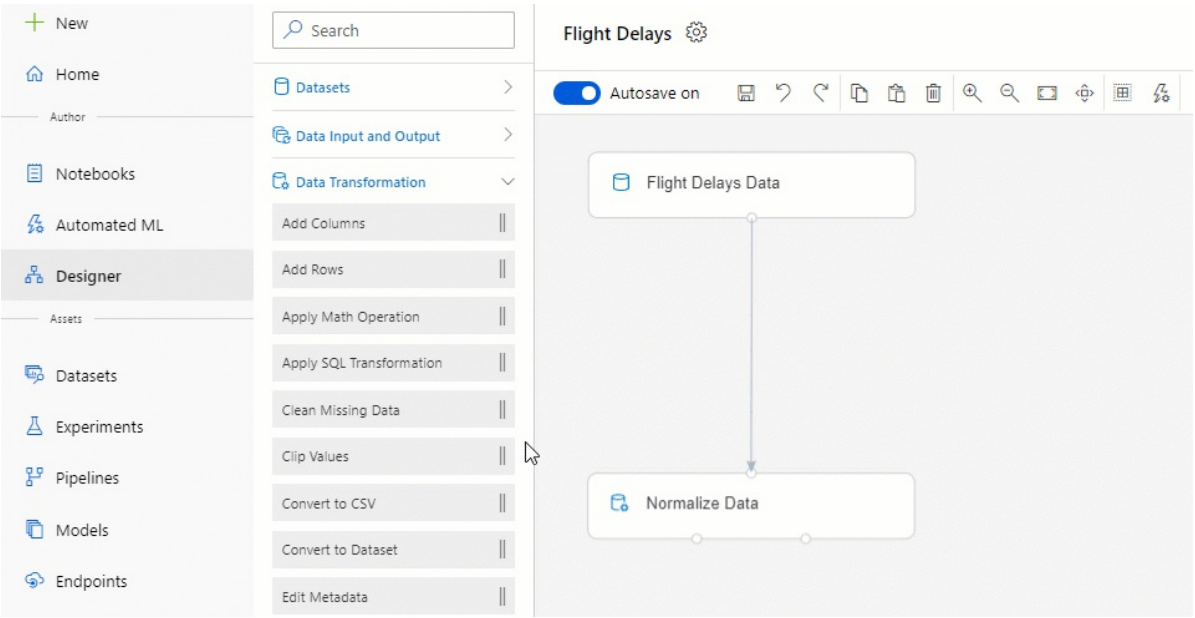
Build ML models with no-code tools

For code-free or low-code training and deployment, try:

- **Azure Machine Learning designer (preview)**

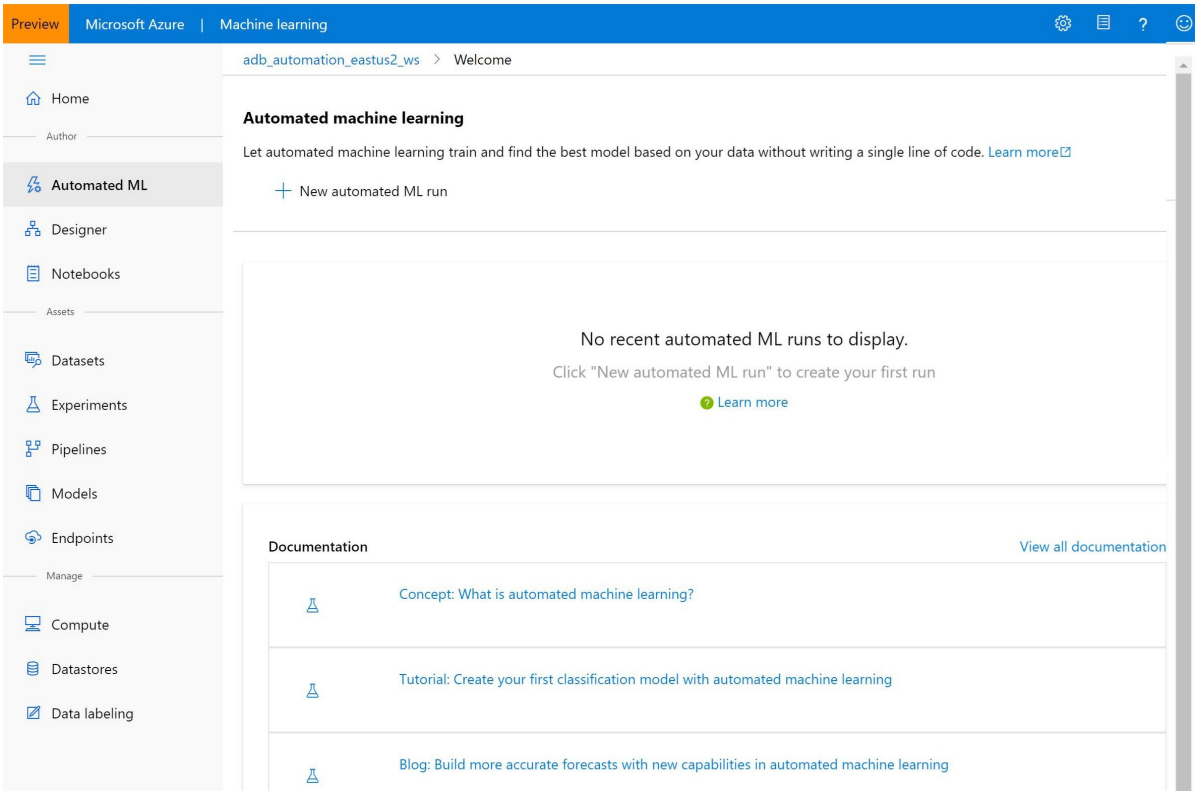
  Use the designer to prep data, train, test, deploy, manage, and track machine learning models without writing any code. There is no programming required, you visually connect datasets and modules to construct your model. Try out the designer tutorial.

  Learn more in the Azure Machine Learning designer overview article.



- **Automated machine learning UI**

  Learn how to create automated ML experiments in the easy-to-use interface.



MLOps: Deploy & lifecycle management

When you have the right model, you can easily use it in a web service, on an IoT device, or from Power BI. For more information, see the article on how to deploy and where.

Then you can manage your deployed models by using the Azure Machine Learning SDK for Python, Azure Machine Learning studio, or the machine learning CLI.

These models can be consumed and return predictions in real time or asynchronously on large quantities of data.

And with advanced machine learning pipelines, you can collaborate on each step from data preparation, model training and evaluation, through deployment. Pipelines allow you to:

- Automate the end-to-end machine learning process in the cloud
- Reuse components and only rerun steps when needed
- Use different compute resources in each step
- Run batch scoring tasks

If you want to use scripts to automate your machine learning workflow, the machine learning CLI provides command-line tools that perform common tasks, such as submitting a training run or deploying a model.

To get started using Azure Machine Learning, see Next steps.

Integration with other services

Azure Machine Learning works with other services on the Azure platform, and also integrates with open source tools such as Git and MLFlow.

- Compute targets such as **Azure Kubernetes Service**, **Azure Container Instances**, **Azure Databricks**, **Azure Data Lake Analytics**, and **Azure HDInsight**. For more information on compute targets, see What are compute targets?.
- **Azure Event Grid**. For more information, see Consume Azure Machine Learning events.
- **Azure Monitor**. For more information, see Monitoring Azure Machine Learning.
- Data stores such as **Azure Storage accounts**, **Azure Data Lake Storage**, **Azure SQL Database**, **Azure**

Database for PostgreSQL, and **Azure Open Datasets**. For more information, see Access data in Azure storage services and Create datasets with Azure Open Datasets.

- **Azure Virtual Networks**. For more information, see Secure experimentation and inference in a virtual network.
- **Azure Pipelines**. For more information, see Train and deploy machine learning models.
- **Git repository logs**. For more information, see Git integration.
- **MLFlow**. For more information, see MLflow to track metrics and deploy models
- **Kubeflow**. For more information, see build end-to-end workflow pipelines.

Secure communications

Your Azure Storage account, compute targets, and other resources can be used securely inside a virtual network to train models and perform inference. For more information, see Secure experimentation and inference in a virtual network.

Basic & Enterprise editions

Azure Machine Learning offers two editions tailored for your machine learning needs:

- Basic (generally available)
- Enterprise (preview)

These editions determine which machine learning tools are available to developers and data scientists from their workspace.

Basic workspaces allow you to continue using Azure Machine Learning and pay for only the Azure resources consumed during the machine learning process. Enterprise edition workspaces will be charged only for their Azure consumption while the edition is in preview. Learn more about what's available in the Azure Machine Learning edition overview & pricing page.

You assign the edition whenever you create a workspace. And, pre-existing workspaces have been converted to the Basic edition for you. Basic edition includes all features that were already generally available as of October 2019. Any experiments in those workspaces that were built using Enterprise edition features will continue to be available to you in read-only until you upgrade to Enterprise. Learn how to upgrade a Basic workspace to Enterprise edition.

Customers are responsible for costs incurred on compute and other Azure resources during this time.

Next steps

- Create your first experiment with your preferred method:
  - Use Python notebooks to train & deploy ML models
  - Use R Markdown to train & deploy ML models
  - Use automated machine learning to train & deploy ML models
  - Use the designer's drag & drop capabilities to train & deploy
  - Use the machine learning CLI to train and deploy a model
- Learn about machine learning pipelines to build, optimize, and manage your machine learning scenarios.
- Read the in-depth Azure Machine Learning architecture and concepts article.

Azure Machine Learning vs Machine Learning Studio (classic)

3/27/2020 • 2 minutes to read • Edit Online

In this article, you learn the difference between Azure Machine Learning and Machine Learning Studio (classic).

Azure Machine Learning provides Python and R SDKs **and** the "drag-and-drop" designer to build and deploy machine learning models. Studio (classic) only offers a standalone drag-and-drop experience.

We recommend that new users choose Azure Machine Learning for the widest range of cutting-edge machine learning tools.

Quick comparison

The following table summarizes some of the key differences between Azure Machine Learning and Studio (classic):

	MACHINE LEARNING STUDIO (CLASSIC)	AZURE MACHINE LEARNING
Drag and drop interface	Supported	Supported - Azure Machine Learning designer (preview)
Experiment	Scalable (10-GB training data limit)	Scale with compute target
Training compute targets	Proprietary compute target, CPU support only	Wide range of customizable training compute targets. Includes GPU and CPU support
Deployment compute targets	Proprietary web service format, not customizable	Wide range of customizable deployment compute targets. Includes GPU and CPU support
ML Pipeline	Not supported	Build flexible, modular pipelines to automate workflows
MLOps	Basic model management and deployment	Entity versioning (model, data, workflows), workflow automation, integration with CICD tooling, and more
Model format	Proprietary format, Studio (classic) only	Multiple supported formats depending on training job type
Automated model training and hyperparameter tuning	Not supported	Supported in the SDK and visual workspace
Data drift detection	Not supported	Supported in SDK and visual workspace

Migrate from Machine Learning Studio (classic)

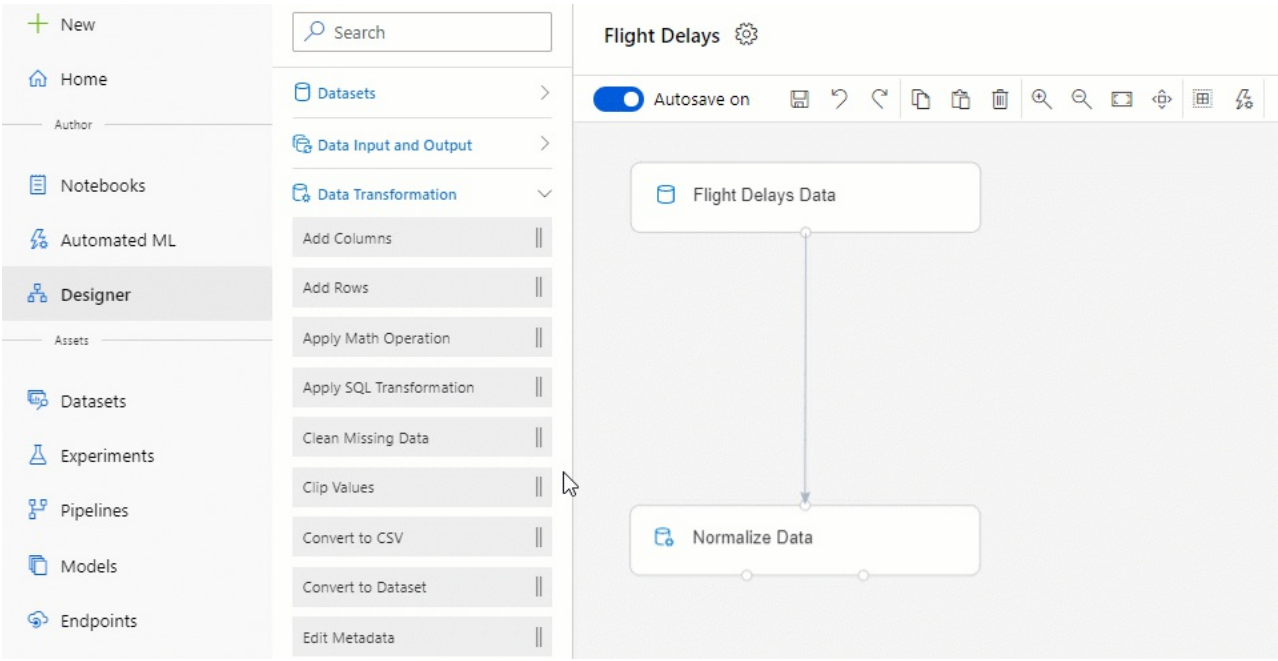
Currently, there's no way to migrate Studio (classic) assets to Azure Machine Learning designer (preview). Current Studio (classic) users can continue to use their machine learning assets. However, we encourage all users to considering using the designer, which provides a familiar drag-and-drop experience with improved workflow **plus**

scalability, version control, and enterprise security.

Get started with Azure Machine Learning

The following resources can help you get started with Azure Machine Learning.

- Read the Azure Machine Learning overview.
- Create your first experiment with the Python SDK.
- Create your first designer pipeline to predict auto prices.



Next steps

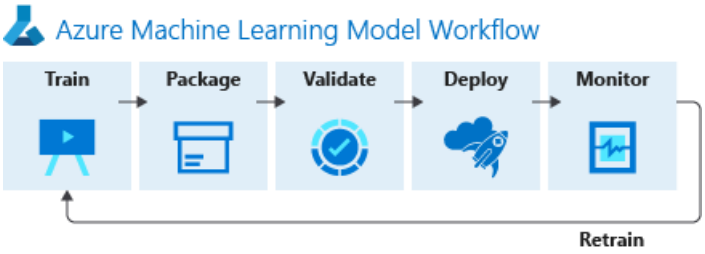
In addition to the drag-and-drop capabilities in the designer, Azure Machine Learning has other tools available:

- Use Python notebooks to train & deploy ML models
- Use R Markdown to train & deploy ML models
- Use automated machine learning to train & deploy ML models
- Use the machine learning CLI to train and deploy a model

How Azure Machine Learning works: Architecture and concepts

4/10/2020 • 11 minutes to read • Edit Online

Learn about the architecture, concepts, and workflow for Azure Machine Learning. The major components of the service and the general workflow for using the service are shown in the following diagram:



Workflow

The machine learning model workflow generally follows this sequence:

1. **Train**
   - Develop machine learning training scripts in **Python**, **R**, or with the visual designer.
   - Create and configure a **compute target**.
   - **Submit the scripts** to a configured compute target to run in that environment. During training, the scripts can read from or write to **datastores**. The logs and output produced during training are saved as **runs** in the **workspace** and grouped under **experiments**.
2. **Package** - After a satisfactory run is found, register the persisted model in the **model registry**.
3. **Validate** - **Query the experiment** for logged metrics from the current and past runs. If the metrics don't indicate a desired outcome, loop back to step 1 and iterate on your scripts.
4. **Deploy** - Develop a scoring script that uses the model and **Deploy the model** as a **web service** in Azure, or to an **IoT Edge device**.
5. **Monitor** - Monitor for **data drift** between the training dataset and inference data of a deployed model. When necessary, loop back to step 1 to retrain the model with new training data.

Tools for Azure Machine Learning

Use these tools for Azure Machine Learning:

- Interact with the service in any Python environment with the Azure Machine Learning SDK for Python.
- Interact with the service in any R environment with the Azure Machine Learning SDK for R.
- Automate your machine learning activities with the Azure Machine Learning CLI.
- Use Azure Machine Learning designer (preview) to perform the workflow steps without writing code.

> **NOTE**
>
> Although this article defines terms and concepts used by Azure Machine Learning, it does not define terms and concepts for the Azure platform. For more information about Azure platform terminology, see the Microsoft Azure glossary.

Glossary

- Activity
- Workspace
  - Experiments
    - Run
      - Run configuration
      - Snapshot
      - Git tracking
      - Logging
  - ML pipelines
  - Models
    - Environments
    - Training script
    - Estimators
  - Endpoints
    - Web service
    - IoT modules
  - Dataset & datastores
  - Compute targets

Activities

An activity represents a long running operation. The following operations are examples of activities:

- Creating or deleting a compute target
- Running a script on a compute target

Activities can provide notifications through the SDK or the web UI so that you can easily monitor the progress of these operations.

Workspaces

The workspace is the top-level resource for Azure Machine Learning. It provides a centralized place to work with all the artifacts you create when you use Azure Machine Learning. You can share a workspace with others. For a detailed description of workspaces, see What is an Azure Machine Learning workspace?.

Experiments

An experiment is a grouping of many runs from a specified script. It always belongs to a workspace. When you submit a run, you provide an experiment name. Information for the run is stored under that experiment. If you submit a run and specify an experiment name that doesn't exist, a new experiment with that newly specified name is automatically created.

For an example of using an experiment, see Tutorial: Train your first model.

Runs

A run is a single execution of a training script. An experiment will typically contain multiple runs.

Azure Machine Learning records all runs and stores the following information in the experiment:

- Metadata about the run (timestamp, duration, and so on)
- Metrics that are logged by your script
- Output files that are autocollected by the experiment or explicitly uploaded by you
- A snapshot of the directory that contains your scripts, prior to the run

You produce a run when you submit a script to train a model. A run can have zero or more child runs. For example, the top-level run might have two child runs, each of which might have its own child run.

Run configurations

A run configuration is a set of instructions that defines how a script should be run in a specified compute target. The configuration includes a wide set of behavior definitions, such as whether to use an existing Python environment or to use a Conda environment that's built from a specification.

A run configuration can be persisted into a file inside the directory that contains your training script, or it can be constructed as an in-memory object and used to submit a run.

For example run configurations, see Select and use a compute target to train your model.

Snapshots

When you submit a run, Azure Machine Learning compresses the directory that contains the script as a zip file and sends it to the compute target. The zip file is then extracted, and the script is run there. Azure Machine Learning also stores the zip file as a snapshot as part of the run record. Anyone with access to the workspace can browse a run record and download the snapshot.

> **NOTE**
>
> To prevent unnecessary files from being included in the snapshot, make an ignore file (.gitignore or .amlignore). Place this file in the Snapshot directory and add the filenames to ignore in it. The .amlignore file uses the same syntax and patterns as the .gitignore file. If both files exist, the .amlignore file takes precedence.

GitHub tracking and integration

When you start a training run where the source directory is a local Git repository, information about the repository is stored in the run history. This works with runs submitted using an estimator, ML pipeline, or script run. It also works for runs submitted from the SDK or Machine Learning CLI.

For more information, see Git integration for Azure Machine Learning.

Logging

When you develop your solution, use the Azure Machine Learning Python SDK in your Python script to log arbitrary metrics. After the run, query the metrics to determine whether the run has produced the model you want to deploy.

ML Pipelines

You use machine learning pipelines to create and manage workflows that stitch together machine learning phases. For example, a pipeline might include data preparation, model training, model deployment, and inference/scoring phases. Each phase can encompass multiple steps, each of which can run unattended in various compute targets.

Pipeline steps are reusable, and can be run without rerunning the previous steps if the output of those steps hasn't changed. For example, you can retrain a model without rerunning costly data preparation steps if the data hasn't changed. Pipelines also allow data scientists to collaborate while working on separate areas of a machine learning workflow.

For more information about machine learning pipelines with this service, see Pipelines and Azure Machine Learning.

Models

At its simplest, a model is a piece of code that takes an input and produces output. Creating a machine learning model involves selecting an algorithm, providing it with data, and tuning hyperparameters. Training is an iterative process that produces a trained model, which encapsulates what the model learned during the training process.

A model is produced by a run in Azure Machine Learning. You can also use a model that's trained outside of Azure

Machine Learning. You can register a model in an Azure Machine Learning workspace.

Azure Machine Learning is framework agnostic. When you create a model, you can use any popular machine learning framework, such as Scikit-learn, XGBoost, PyTorch, TensorFlow, and Chainer.

For an example of training a model using Scikit-learn and an estimator, see Tutorial: Train an image classification model with Azure Machine Learning.

The **model registry** keeps track of all the models in your Azure Machine Learning workspace.

Models are identified by name and version. Each time you register a model with the same name as an existing one, the registry assumes that it's a new version. The version is incremented, and the new model is registered under the same name.

When you register the model, you can provide additional metadata tags and then use the tags when you search for models.

> **TIP**
> A registered model is a logical container for one or more files that make up your model. For example, if you have a model that is stored in multiple files, you can register them as a single model in your Azure Machine Learning workspace. After registration, you can then download or deploy the registered model and receive all the files that were registered.

You can't delete a registered model that is being used by an active deployment.

For an example of registering a model, see Train an image classification model with Azure Machine Learning.

Environments

Azure ML Environments are used to specify the configuration (Docker / Python / Spark / etc.) used to create a reproducible environment for data preparation, model training and model serving. They are managed and versioned entities within your Azure Machine Learning workspace that enable reproducible, auditable, and portable machine learning workflows across different compute targets.

You can use an environment object on your local compute to develop your training script, reuse that same environment on Azure Machine Learning Compute for model training at scale, and even deploy your model with that same environment.

Learn how to create and manage a reusable ML environment for training and inference.

Training scripts

To train a model, you specify the directory that contains the training script and associated files. You also specify an experiment name, which is used to store information that's gathered during training. During training, the entire directory is copied to the training environment (compute target), and the script that's specified by the run configuration is started. A snapshot of the directory is also stored under the experiment in the workspace.

For an example, see Tutorial: Train an image classification model with Azure Machine Learning.

Estimators

To facilitate model training with popular frameworks, the estimator class allows you to easily construct run configurations. You can create and use a generic Estimator to submit training scripts that use any learning framework you choose (such as scikit-learn).

For PyTorch, TensorFlow, and Chainer tasks, Azure Machine Learning also provides respective PyTorch, TensorFlow, and Chainer estimators to simplify using these frameworks.

For more information, see the following articles:

- Train ML models with estimators.

- Train Pytorch deep learning models at scale with Azure Machine Learning.
- Train and register TensorFlow models at scale with Azure Machine Learning.
- Train and register Chainer models at scale with Azure Machine Learning.

Endpoints

An endpoint is an instantiation of your model into either a web service that can be hosted in the cloud or an IoT module for integrated device deployments.

Web service endpoint

When deploying a model as a web service the endpoint can be deployed on Azure Container Instances, Azure Kubernetes Service, or FPGAs. You create the service from your model, script, and associated files. These are placed into a base container image which contains the execution environment for the model. The image has a load-balanced, HTTP endpoint that receives scoring requests that are sent to the web service.

Azure helps you monitor your web service by collecting Application Insights telemetry or model telemetry, if you've chosen to enable this feature. The telemetry data is accessible only to you, and it's stored in your Application Insights and storage account instances.

If you've enabled automatic scaling, Azure automatically scales your deployment.

For an example of deploying a model as a web service , see Deploy an image classification model in Azure Container Instances.

IoT module endpoints

A deployed IoT module endpoint is a Docker container that includes your model and associated script or application and any additional dependencies. You deploy these modules by using Azure IoT Edge on edge devices.

If you've enabled monitoring, Azure collects telemetry data from the model inside the Azure IoT Edge module. The telemetry data is accessible only to you, and it's stored in your storage account instance.

Azure IoT Edge ensures that your module is running, and it monitors the device that's hosting it.

Compute instance (preview)

An **Azure Machine Learning compute instance** (formerly Notebook VM) is a fully managed cloud-based workstation that includes multiple tools and environments installed for machine learning. Compute instances can be used as a compute target for training and inferencing jobs. For large tasks, Azure Machine Learning compute clusters with multi-node scaling capabilities is a better compute target choice.

Learn more about compute instances.

Datasets and datastores

Azure Machine Learning Datasets (preview) make it easier to access and work with your data. Datasets manage data in various scenarios such as model training and pipeline creation. Using the Azure Machine Learning SDK, you can access underlying storage, explore data, and manage the life cycle of different Dataset definitions.

Datasets provide methods for working with data in popular formats, such as using `from_delimited_files()` or `to_pandas_dataframe()`.

For more information, see Create and register Azure Machine Learning Datasets. For more examples using Datasets, see the sample notebooks.

A **datastore** is a storage abstraction over an Azure storage account. The datastore can use either an Azure blob container or an Azure file share as the back-end storage. Each workspace has a default datastore, and you can register additional datastores. Use the Python SDK API or the Azure Machine Learning CLI to store and retrieve files from the datastore.

Compute targets

A compute target lets you specify the compute resource where you run your training script or host your service

deployment. This location may be your local machine or a cloud-based compute resource.

Learn more about the available compute targets for training and deployment.

Next steps

To get started with Azure Machine Learning, see:

- What is Azure Machine Learning?
- Create an Azure Machine Learning workspace
- Tutorial (part 1): Train a model

Tutorial: Predict automobile price with the designer (preview)

3/30/2020 • 13 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this two-part tutorial, you learn how to use the Azure Machine Learning designer to train and deploy a machine learning model that predicts the price of any car. The designer is a drag-and-drop tool that lets you create machine learning models without a single line of code.

In part one of the tutorial, you'll learn how to:

- Create a new pipeline.
- Import data.
- Prepare data.
- Train a machine learning model.
- Evaluate a machine learning model.

In part two of the tutorial, you'll deploy your model as a real-time inferencing endpoint to predict the price of any car based on technical specifications you send it.

> **NOTE**
>
> A completed version of this tutorial is available as a sample pipeline.
>
> To find it, go to the designer in your workspace. In the **New pipeline** section, select **Sample 1 - Regression: Automobile Price Prediction(Basic)**.

Create a new pipeline

Azure Machine Learning pipelines organize multiple machine learning and data processing steps into a single resource. Pipelines let you organize, manage, and reuse complex machine learning workflows across projects and users.

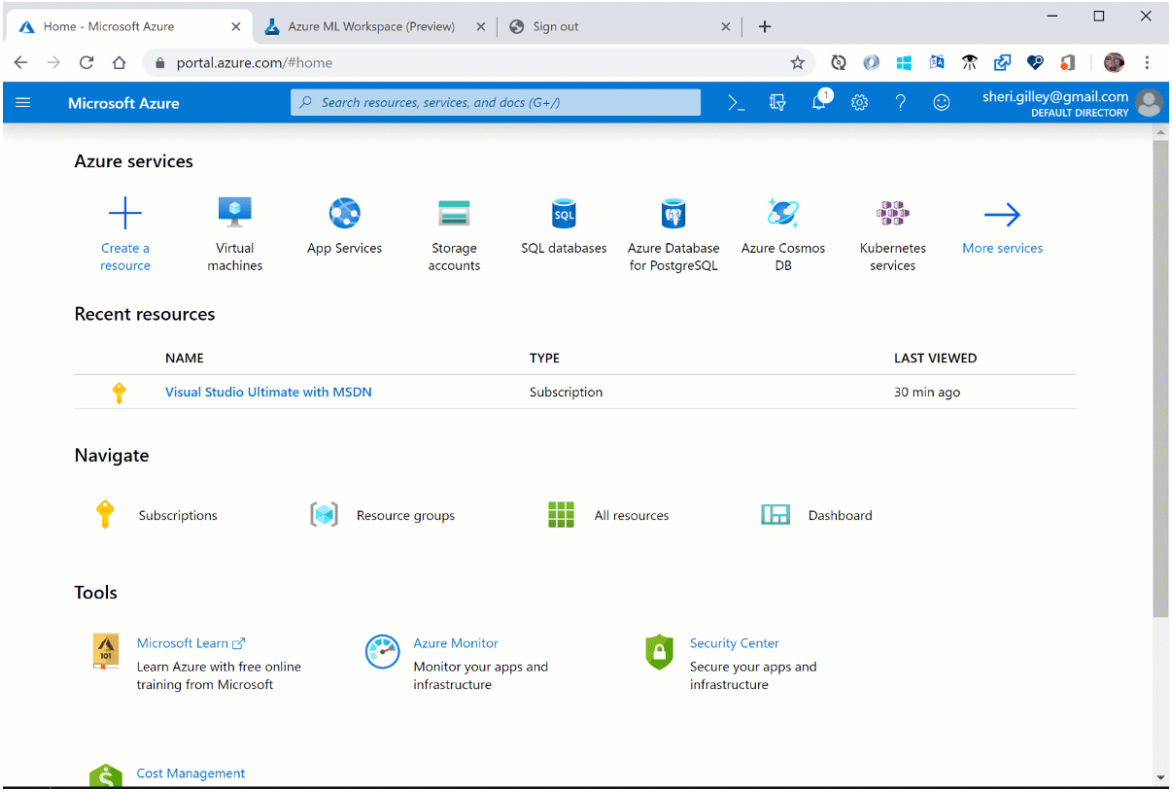
To create an Azure Machine Learning pipeline, you need an Azure Machine Learning workspace. In this section, you learn how to create both these resources.

Create a new workspace

In order to use the designer, you first need an Azure Machine Learning workspace. The workspace is the top-level resource for Azure Machine Learning, it provides a centralized place to work with all the artifacts you create in Azure Machine Learning.

If you have an Azure Machine Learning workspace with an Enterprise edition, skip to the next section.

1. Sign in to the Azure portal by using the credentials for your Azure subscription.
2. In the upper-left corner of the Azure portal, select **+ Create a resource**.



3. Use the search bar to find **Machine Learning**.
4. Select **Machine Learning**.
5. In the **Machine Learning** pane, select **Create** to begin.
6. Provide the following information to configure your new workspace:

FIELD	DESCRIPTION
Workspace name	Enter a unique name that identifies your workspace. In this example, we use **docs-ws**. Names must be unique across the resource group. Use a name that's easy to recall and to differentiate from workspaces created by others.
Subscription	Select the Azure subscription that you want to use.
Resource group	Use an existing resource group in your subscription, or enter a name to create a new resource group. A resource group holds related resources for an Azure solution. In this example, we use **docs-aml**.
Location	Select the location closest to your users and the data resources to create your workspace.
Workspace edition	Select **Enterprise**. This tutorial requires the use of the Enterprise edition. The Enterprise edition is in preview and doesn't currently add any extra costs.

7. After you're finished configuring the workspace, select **Create**.

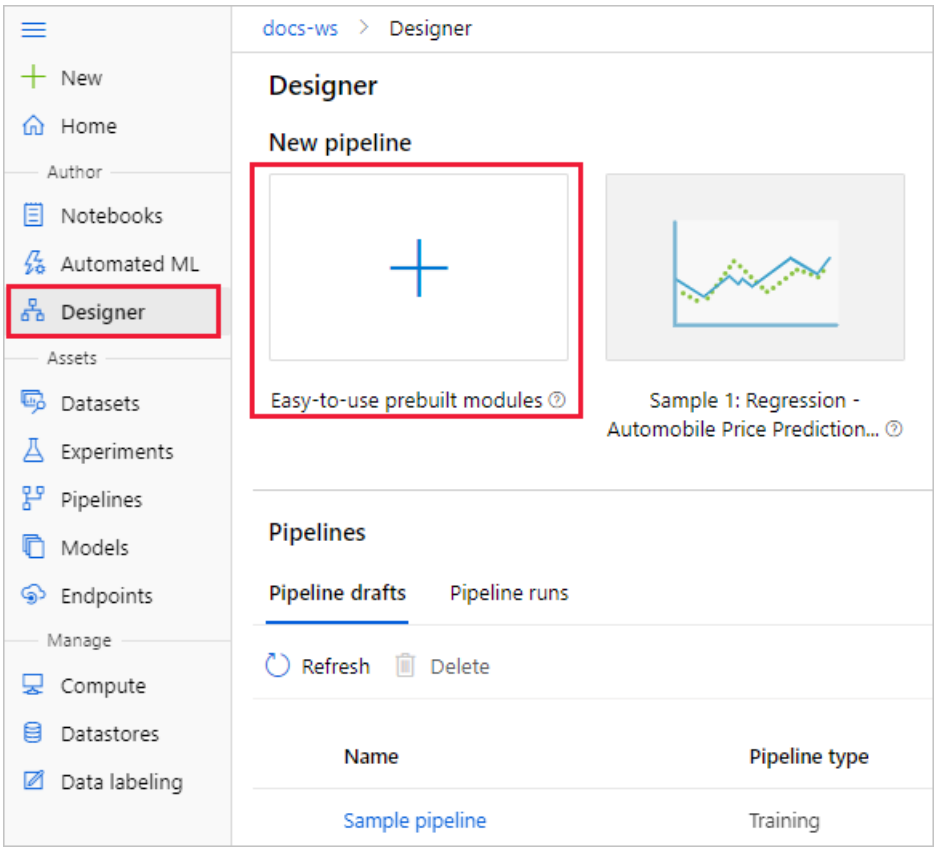
> **WARNING**
> It can take several minutes to create your workspace in the cloud.

When the process is finished, a deployment success message appears.

8. To view the new workspace, select **Go to resource**.

Create the pipeline

1. Sign in to ml.azure.com, and select the workspace you want to work with.
2. Select **Designer**.



3. Select **Easy-to-use prebuilt modules**.
4. At the top of the canvas, select the default pipeline name **Pipeline-Created-on**. Rename it to *Automobile price prediction*. The name doesn't need to be unique.

Set the default compute target

A pipeline runs on a compute target, which is a compute resource that's attached to your workspace. After you create a compute target, you can reuse it for future runs.

You can set a **Default compute target** for the entire pipeline, which will tell every module to use the same compute target by default. However, you can specify compute targets on a per-module basis.

1. Next to the pipeline name, select the **Gear icon**  at the top of the canvas to open the **Settings** pane.
2. In the **Settings** pane to the right of the canvas, select **Select compute target**.

   If you already have an available compute target, you can select it to run this pipeline.

   > **NOTE**
   > The designer can run experiments only on Azure Machine Learning Compute targets. Other compute targets won't be shown.

3. Enter a name for the compute resource.

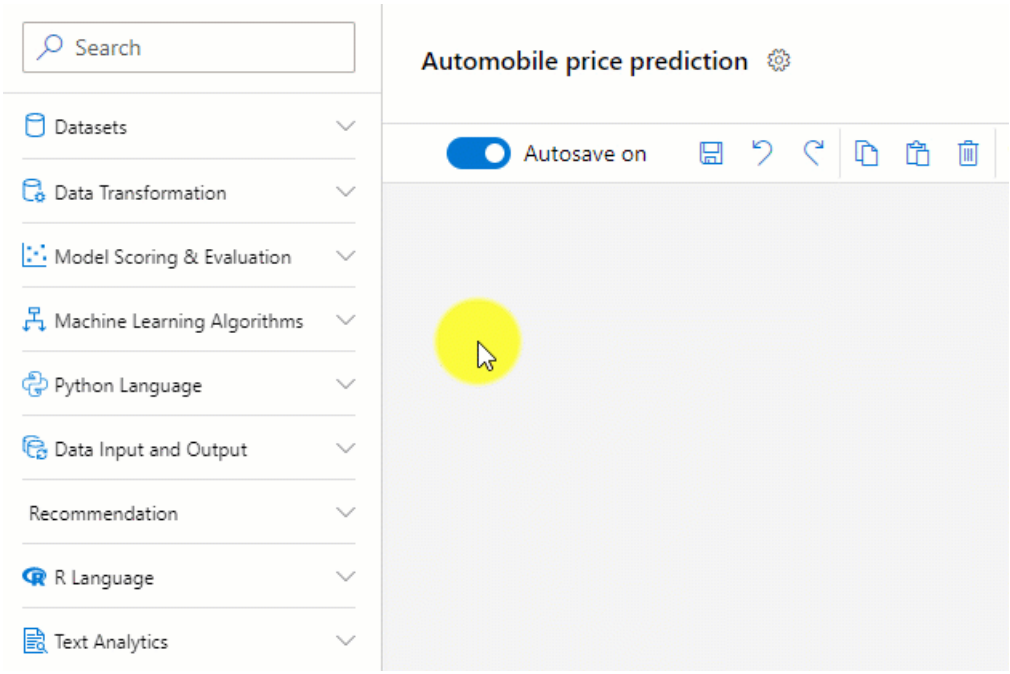
4. Select **Save**.

> **NOTE**
>
> It takes approximately five minutes to create a compute resource. After the resource is created, you can reuse it and skip this wait time for future runs.
>
> The compute resource autoscales to zero nodes when it's idle to save cost. When you use it again after a delay, you might experience approximately five minutes of wait time while it scales back up.

Import data

There are several sample datasets included in the designer for you to experiment with. For this tutorial, use **Automobile price data (Raw)**.

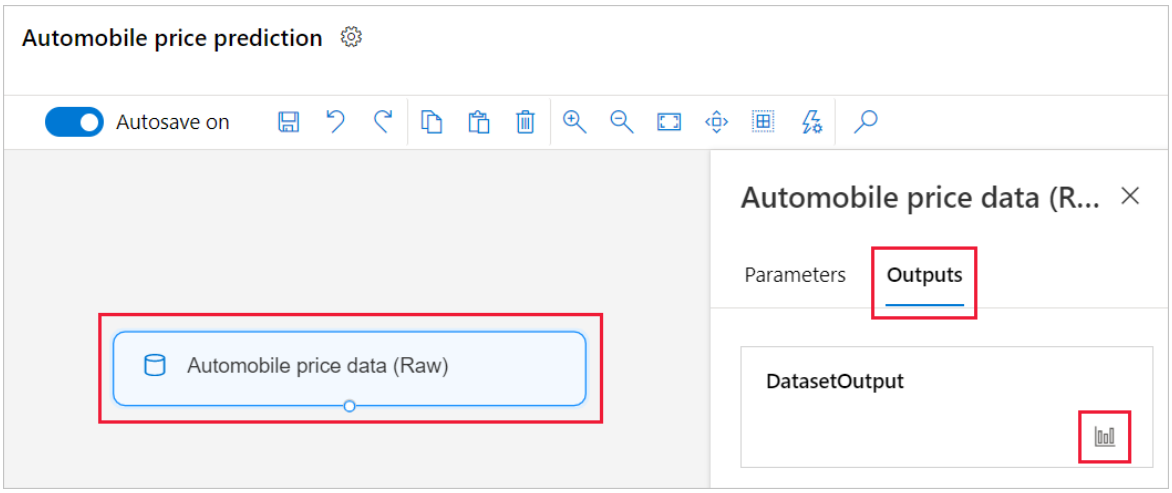
1. To the left of the pipeline canvas is a palette of datasets and modules. Select **Datasets**, and then view the **Samples** section to view the available sample datasets.
2. Select the dataset **Automobile price data (Raw)**, and drag it onto the canvas.



Visualize the data

You can visualize the data to understand the dataset that you'll use.

1. Select the **Automobile price data (Raw)** module.
2. In the module details pane to the right of the canvas, select **Outputs + log**.
3. Select the graph icon to visualize the data.



4. Select the different columns in the data window to view information about each one.

   Each row represents an automobile, and the variables associated with each automobile appear as columns. There are 205 rows and 26 columns in this dataset.

Prepare data

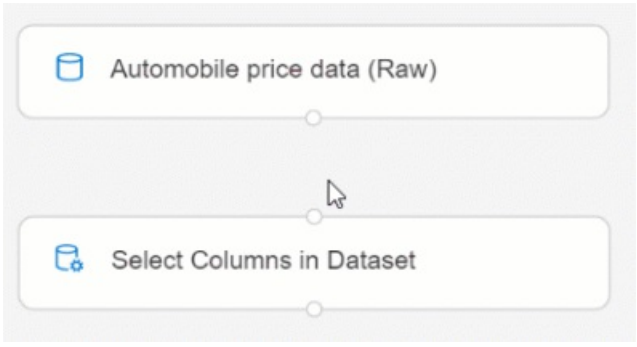
Datasets typically require some preprocessing before analysis. You might have noticed some missing values when you inspected the dataset. These missing values must be cleaned so that the model can analyze the data correctly.

Remove a column

When you train a model, you have to do something about the data that's missing. In this dataset, the **normalized-losses** column is missing many values, so you will exclude that column from the model altogether.

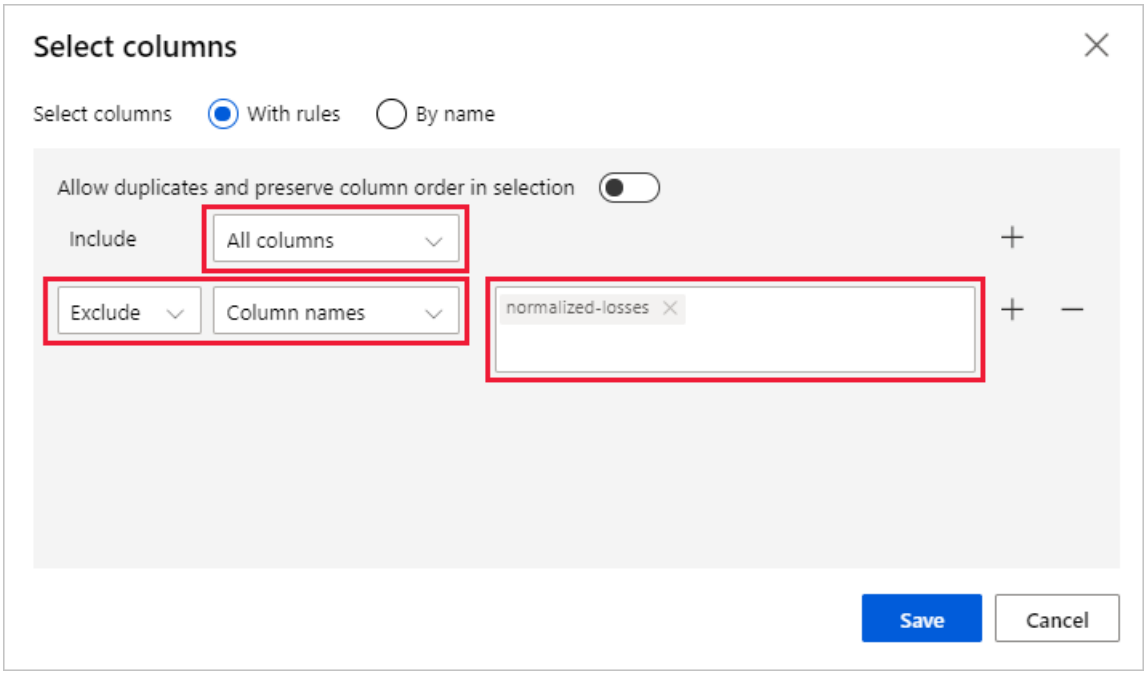
1. In the module palette to the left of the canvas, expand the **Data Transformation** section and find the **Select Columns in Dataset** module.
2. Drag the **Select Columns in Dataset** module onto the canvas. Drop the module below the dataset module.
3. Connect the **Automobile price data (Raw)** dataset to the **Select Columns in Dataset** module. Drag from the dataset's output port, which is the small circle at the bottom of the dataset on the canvas, to the input port of **Select Columns in Dataset**, which is the small circle at the top of the module.

> **TIP**
> You create a flow of data through your pipeline when you connect the output port of one module to an input port of another.



4. Select the **Select Columns in Dataset** module.

5. In the module details pane to the right of the canvas, select **Edit column**.
6. Expand the **Column names** drop down next to **Include**, and select **All columns**.
7. Select the **+** to add a new rule.
8. From the drop-down menus, select **Exclude** and **Column names**.
9. Enter *normalized-losses* in the text box.
10. In the lower right, select **Save** to close the column selector.



11. Select the **Select Columns in Dataset** module.
12. In the module details pane to the right of the canvas, select the **Comment** text box and enter *Exclude normalized losses*.

    Comments will appear on the graph to help you organize your pipeline.

Clean missing data

Your dataset still has missing values after you remove the **normalized-losses** column. You can remove the remaining missing data by using the **Clean Missing Data** module.

> **TIP**
> Cleaning the missing values from input data is a prerequisite for using most of the modules in the designer.

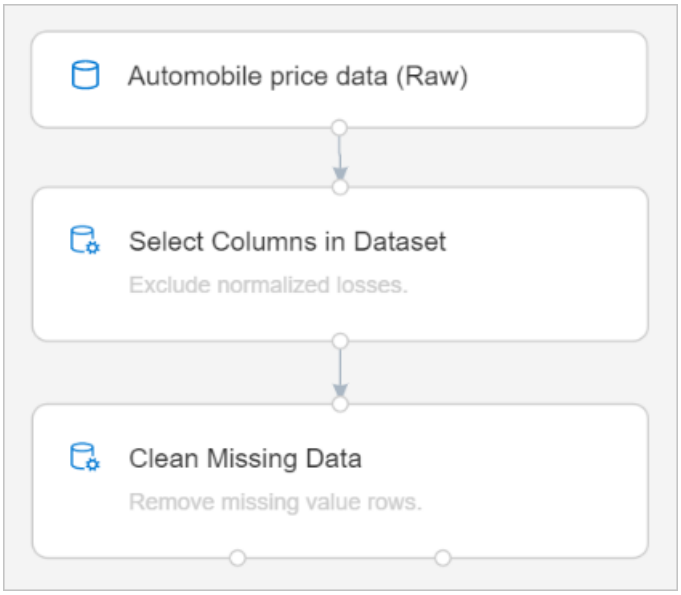
1. In the module palette to the left of the canvas, expand the section **Data Transformation**, and find the **Clean Missing Data** module.
2. Drag the **Clean Missing Data** module to the pipeline canvas. Connect it to the **Select Columns in Dataset** module.
3. Select the **Clean Missing Data** module.
4. In the module details pane to the right of the canvas, select **Edit Column**.
5. In the **Columns to be cleaned** window that appears, expand the drop-down menu next to **Include**. Select, **All columns**

6. Select **Save**

7. In the module details pane to the right of the canvas, select **Remove entire row** under **Cleaning mode**.

8. In the module details pane to the right of the canvas, select the **Comment** box, and enter *Remove missing value rows*.

   Your pipeline should now look something like this:



Train a machine learning model

Now that you have the modules in place to process the data, you can set up the training modules.

Because you want to predict price, which is a number, you can use a regression algorithm. For this example, you use a linear regression model.

Split the data

Splitting data is a common task in machine learning. You will split your data into two separate datasets. One dataset will train the model and the other will test how well the model performed.

1. In the module palette, expand the section **Data Transformation** and find the **Split Data** module.

2. Drag the **Split Data** module to the pipeline canvas.

3. Connect the left port of the **Clean Missing Data** module to the **Split Data** module.

   > **IMPORTANT**
   > Be sure that the left output ports of **Clean Missing Data** connects to **Split Data**. The left port contains the the cleaned data. The right port contains the discarted data.

4. Select the **Split Data** module.

5. In the module details pane to the right of the canvas, set the **Fraction of rows in the first output dataset** to 0.7.

   This option splits 70 percent of the data to train the model and 30 percent for testing it. The 70 percent dataset will be accessible through the left output port. The remaining data will be available through the right output port.

6. In the module details pane to the right of the canvas, select the **Comment** box, and enter *Split the dataset*

into training set (0.7) and test set (0.3).

Train the model

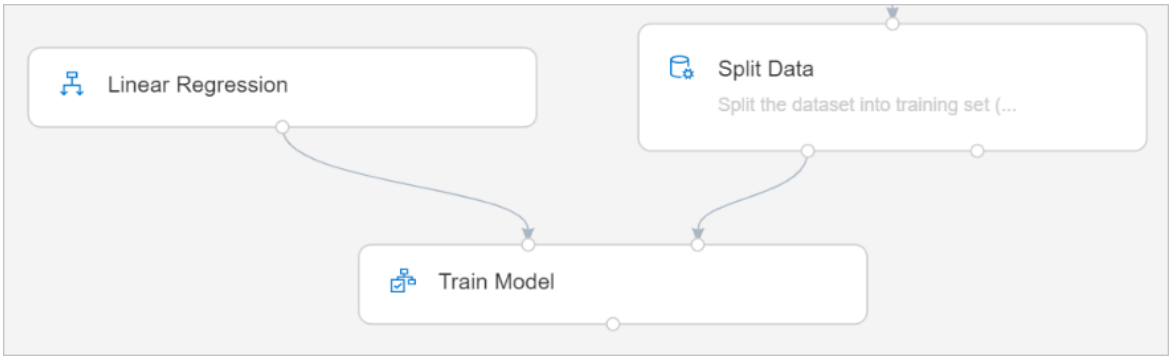
Train the model by giving it a dataset that includes the price. The algorithm constructs a model that explains the relationship between the features and the price as presented by the training data.

1. In the module palette, expand **Machine Learning Algorithms**.

   This option displays several categories of modules that you can use to initialize learning algorithms.

2. Select **Regression** > **Linear Regression**, and drag it to the pipeline canvas.
3. Connect the output of the **Linear Regression** module to the left input of the **Train Model** module.
4. In the module palette, expand the section **Module training**, and drag the **Train Model** module to the canvas.
5. Select the **Train Model** module, and drag it to the pipeline canvas.
6. Connect the training data output (left port) of the **Split Data** module to the right input of the **Train Model** module.

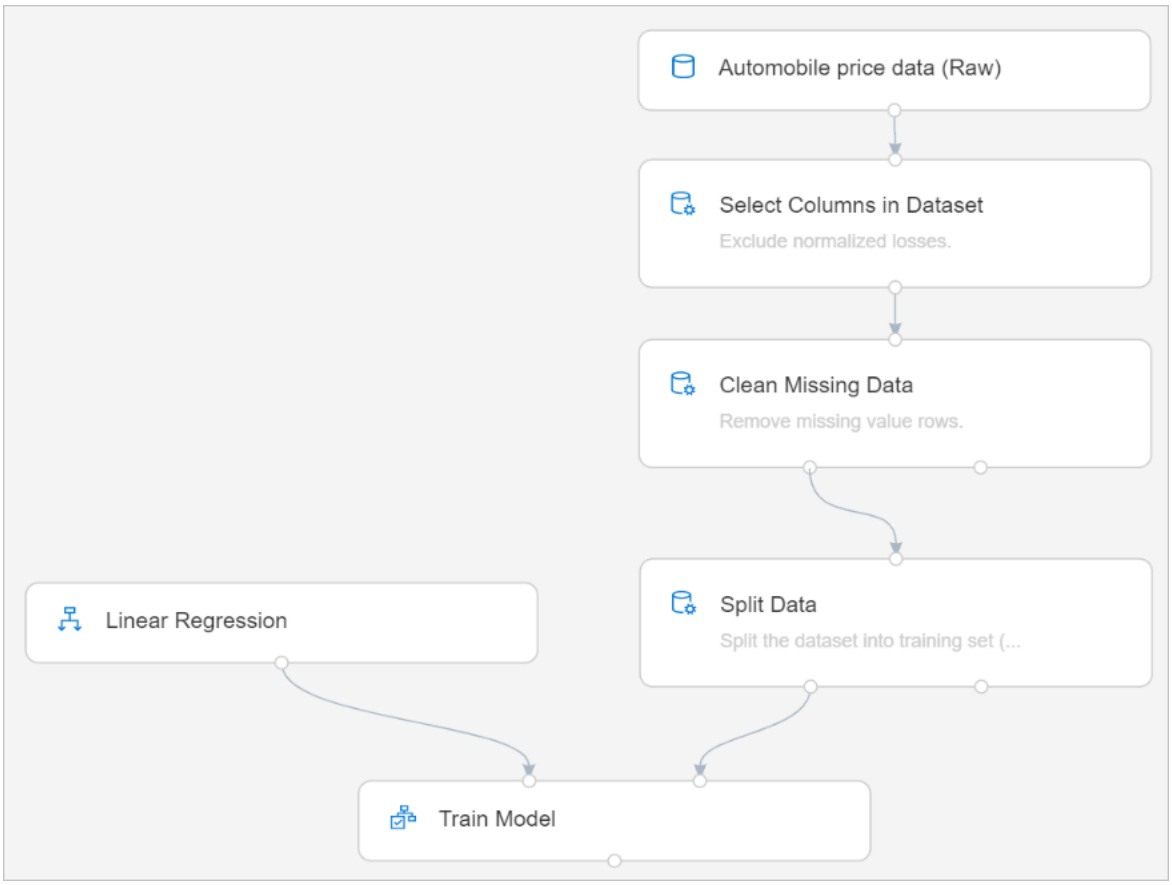
   > **IMPORTANT**
   > Be sure that the left output ports of **Split Data** connects to **Train Model**. The left port contains the the training set. The right port contains the test set.

7. Select the **Train Model** module.
8. In the module details pane to the right of the canvas, select **Edit column** selector.
9. In the **Label column** dialog box, expand the drop-down menu and select **Column names**.
10. In the text box, enter *price* to specify the value that your model is going to predict.

    > **IMPORTANT**
    > Make sure you enter the column name exactly. Do not capitalize **price**.

    Your pipeline should look like this:



Add the Score Model module

After you train your model by using 70 percent of the data, you can use it to score the other 30 percent to see how well your model functions.

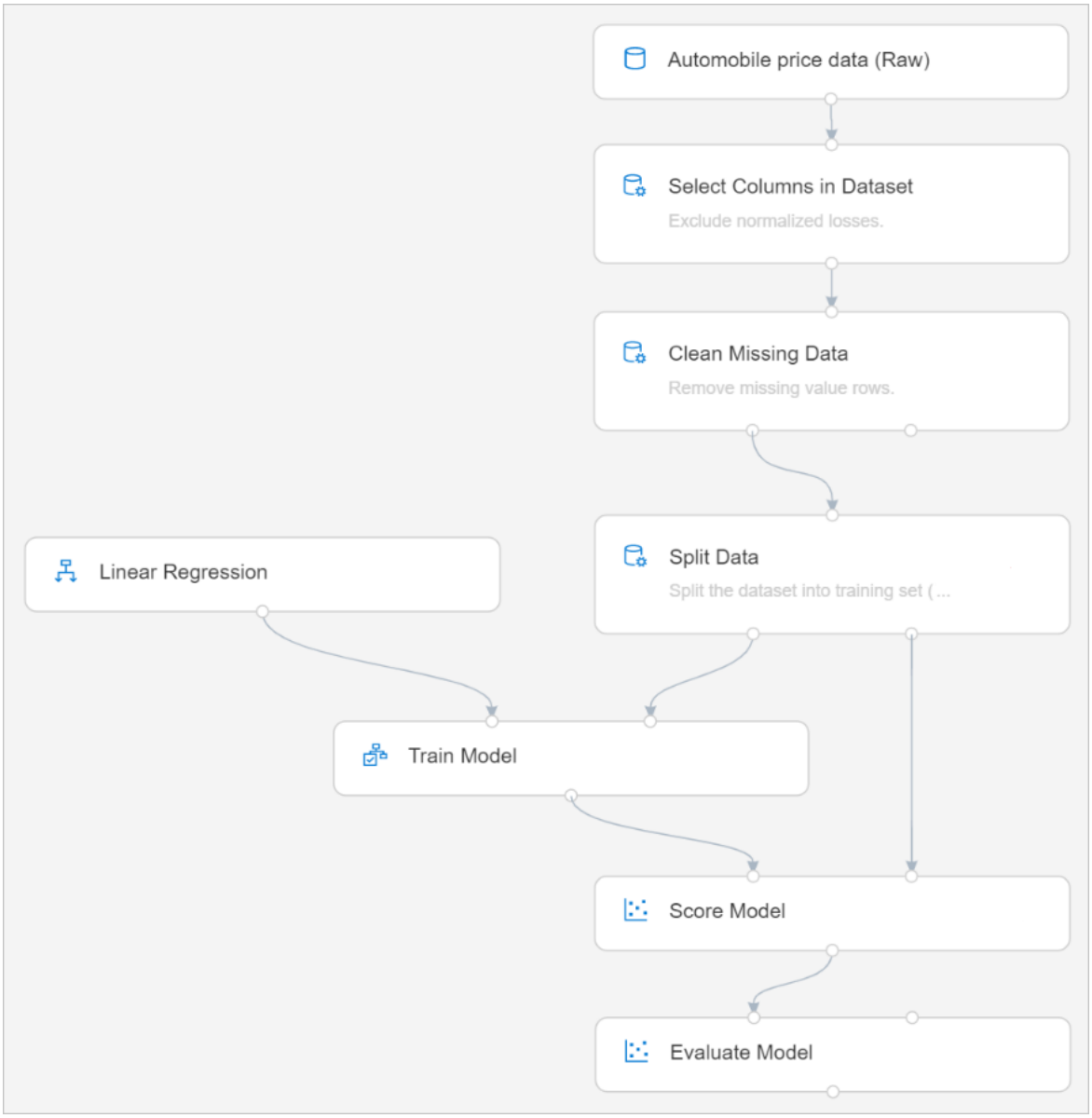
1. Enter *score model* in the search box to find the **Score Model** module. Drag the module to the pipeline canvas.
2. Connect the output of the **Train Model** module to the left input port of **Score Model**. Connect the test data output (right port) of the **Split Data** module to the right input port of **Score Model**.

Add the Evaluate Model module

Use the **Evaluate Model** module to evaluate how well your model scored the test dataset.

1. Enter *evaluate* in the search box to find the **Evaluate Model** module. Drag the module to the pipeline canvas.
2. Connect the output of the **Score Model** module to the left input of **Evaluate Model**.

   The final pipeline should look something like this:



Submit the pipeline

Now that your pipeline is all setup, you can submit a pipeline run to train your machine learning model. You can submit a valid pipeline run at any point, which can be used to review changes to your pipeline during development.

1. At the top of the canvas, select **Submit**.
2. In the **Set up pipeline run** dialog box, select **Create new**.

> **NOTE**
> Experiments group similar pipeline runs together. If you run a pipeline multiple times, you can select the same experiment for successive runs.

   a. Enter a descriptive name for **New experiment Name**.

   b. Select **Submit**.

You can view run status and details at the top right of the canvas.

If is the first run, it may take up to 20 minutes for your pipeline to finish running. The default compute settings have a minimum node size of 0, which means that the designer must allocate resources after

being idle. Repeated pipeline runs will take less time since the compute resources are already allocated. Additionally, the designer uses cached results for each module to further improve efficiency.

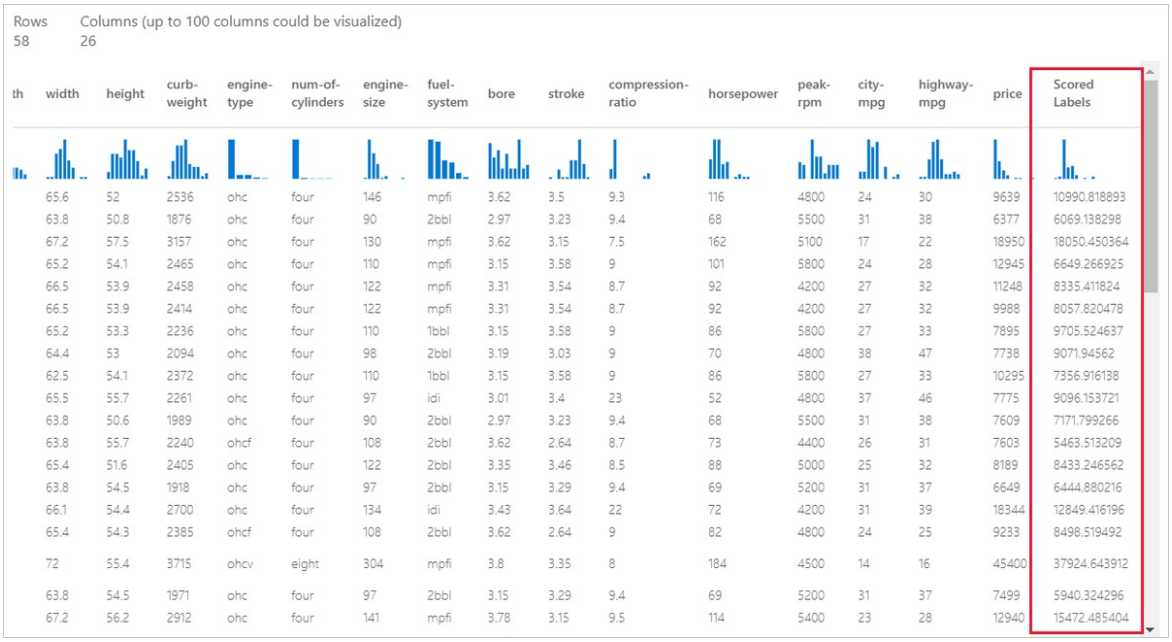
View scored labels

After the run completes, you can view the results of the pipeline run. First, look at the predictions generated by the regression model.

1. Select the **Score Model** module to view its output.

2. In the module details pane to the right of the canvas, select **Outputs + logs** > graph icon  to view results.

   Here you can see the predicted prices and the actual prices from the testing data.



Rows 58 | Columns (up to 100 columns could be visualized) 26

th	width	height	curb-weight	engine-type	num-of-cylinders	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	Scored Labels
	65.6	52	2536	ohc	four	146	mpfi	3.62	3.5	9.3	116	4800	24	30	9639	10990.818893
	63.8	50.8	1876	ohc	four	90	2bbl	2.97	3.23	9.4	68	5500	31	38	6377	6069.138298
	67.2	57.5	3157	ohc	four	130	mpfi	3.62	3.15	7.5	162	5100	17	22	18950	18050.450364
	65.2	54.1	2465	ohc	four	110	mpfi	3.15	3.58	9	101	5800	24	28	12945	6649.266925
	66.5	53.9	2458	ohc	four	122	mpfi	3.31	3.54	8.7	92	4200	27	32	11248	8335.411824
	66.5	53.9	2414	ohc	four	122	mpfi	3.31	3.54	8.7	92	4200	27	32	9988	8057.820478
	65.2	53.3	2236	ohc	four	110	1bbl	3.15	3.58	9	86	5800	27	33	7895	9705.524637
	64.4	53	2094	ohc	four	98	2bbl	3.19	3.03	9	70	4800	38	47	7738	9071.94562
	62.5	54.1	2372	ohc	four	110	1bbl	3.15	3.58	9	86	5800	27	33	10295	7356.916138
	65.5	55.7	2261	ohc	four	97	idi	3.01	3.4	23	52	4800	37	46	7775	9096.153721
	63.8	50.6	1989	ohc	four	90	2bbl	2.97	3.23	9.4	68	5500	31	38	7609	7171.799266
	63.8	55.7	2240	ohcf	four	108	2bbl	3.62	2.64	8.7	73	4400	26	31	7603	5463.513209
	65.4	51.6	2405	ohc	four	122	2bbl	3.35	3.46	8.5	88	5000	25	32	8189	8433.246562
	63.8	54.5	1918	ohc	four	97	2bbl	3.15	3.29	9.4	69	5200	31	37	6649	6444.880216
	66.1	54.4	2700	ohc	four	134	idi	3.43	3.64	22	72	4200	31	39	18344	12849.416196
	65.4	54.3	2385	ohcf	four	108	2bbl	3.62	2.64	9	82	4800	24	25	9233	8498.519492
	72	55.4	3715	ohcv	eight	304	mpfi	3.8	3.35	8	184	4500	14	16	45400	37924.643912
	63.8	54.5	1971	ohc	four	97	2bbl	3.15	3.29	9.4	69	5200	31	37	7499	5940.324296
	67.2	56.2	2912	ohc	four	141	mpfi	3.78	3.15	9.5	114	5400	23	28	12940	15472.485404

Evaluate models

Use the **Evaluate Model** to see how well the trained model performed on the test dataset.

1. Select the **Evaluate Model** module to view its output.

2. In the module details pane to the right of the canvas, select **Outputs + logs** > graph icon  to view results.

The following statistics are shown for your model:

- **Mean Absolute Error (MAE)**: The average of absolute errors. An error is the difference between the predicted value and the actual value.
- **Root Mean Squared Error (RMSE)**: The square root of the average of squared errors of predictions made on the test dataset.
- **Relative Absolute Error**: The average of absolute errors relative to the absolute difference between actual values and the average of all actual values.
- **Relative Squared Error**: The average of squared errors relative to the squared difference between the actual values and the average of all actual values.
- **Coefficient of Determination**: Also known as the R squared value, this statistical metric indicates how well a model fits the data.

For each of the error statistics, smaller is better. A smaller value indicates that the predictions are closer to the actual values. For the coefficient of determination, the closer its value is to one (1.0), the better the predictions.

Clean up resources

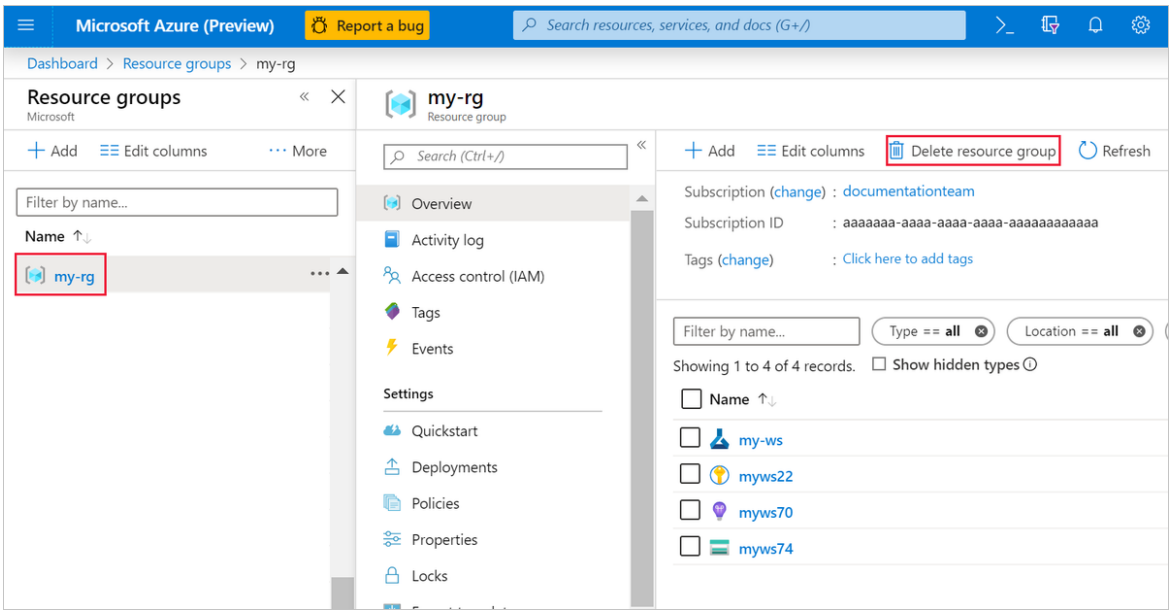
Skip this section if you want to continue on with part 2 of the tutorial, deploying models.

> **IMPORTANT**
>
> You can use the resources that you created as prerequisites for other Azure Machine Learning tutorials and how-to articles.

Delete everything

If you don't plan to use anything that you created, delete the entire resource group so you don't incur any charges.

1. In the Azure portal, select **Resource groups** on the left side of the window.



2. In the list, select the resource group that you created.

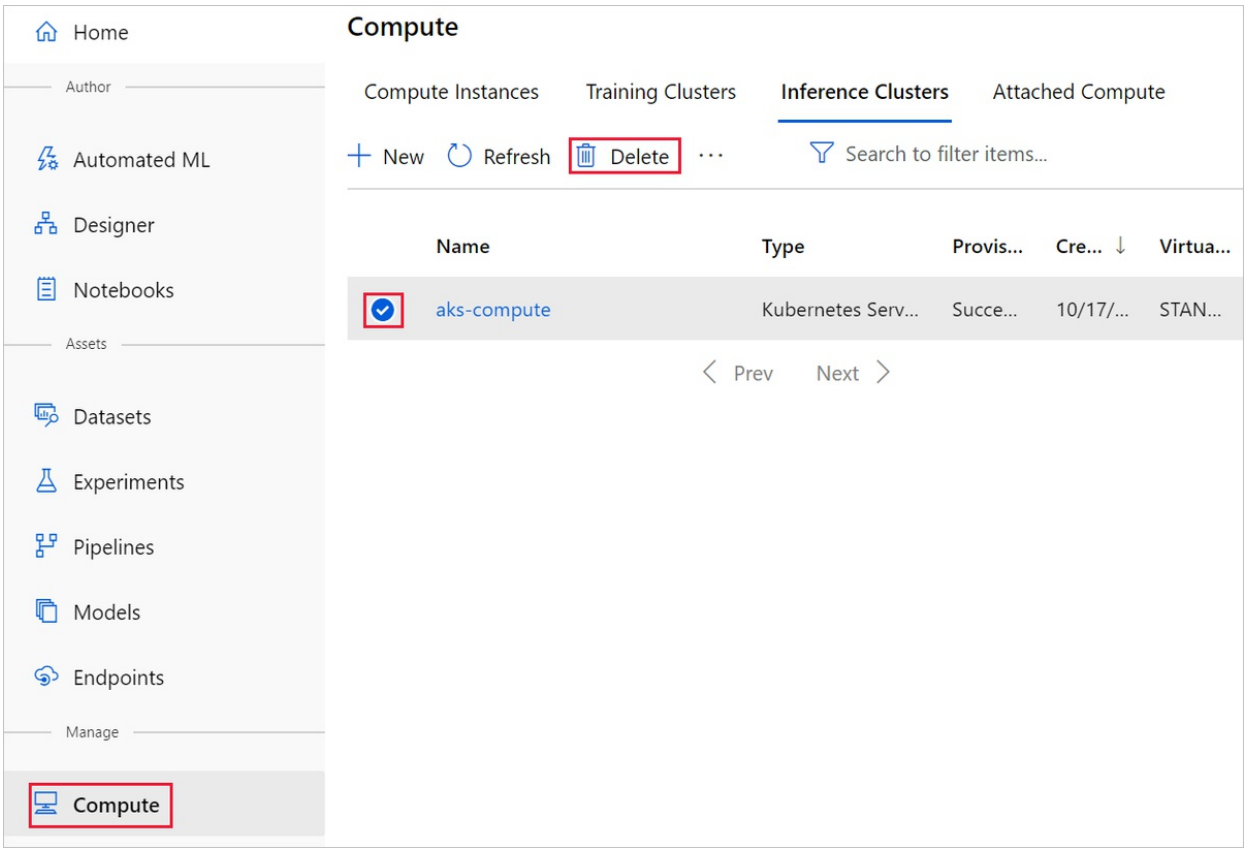
3. Select **Delete resource group**.

Deleting the resource group also deletes all resources that you created in the designer.

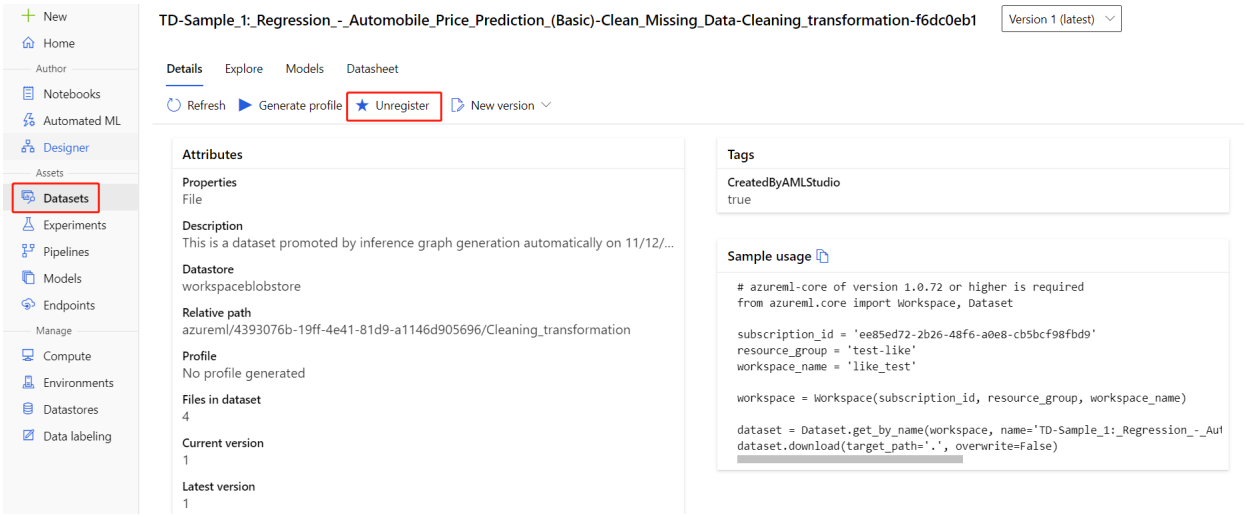
Delete individual assets

In the designer where you created your experiment, delete individual assets by selecting them and then selecting the **Delete** button.

The compute target that you created here *automatically autoscales* to zero nodes when it's not being used. This action is taken to minimize charges. If you want to delete the compute target, take these steps:



You can unregister datasets from your workspace by selecting each dataset and selecting **Unregister**.



To delete a dataset, go to the storage account by using the Azure portal or Azure Storage Explorer and manually delete those assets.

Next steps

In part two, you'll learn how to deploy your model as a real-time endpoint.

Continue to deploying models

Tutorial: Deploy a machine learning model with the designer (preview)

3/30/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

You can deploy the predictive model developed in part one of the tutorial to give others a chance to use it. In part one, you trained your model. Now, it's time to generate new predictions based on user input. In this part of the tutorial, you will:

- Create a real-time inference pipeline.
- Create an inferencing cluster.
- Deploy the real-time endpoint.
- Test the real-time endpoint.

Prerequisites

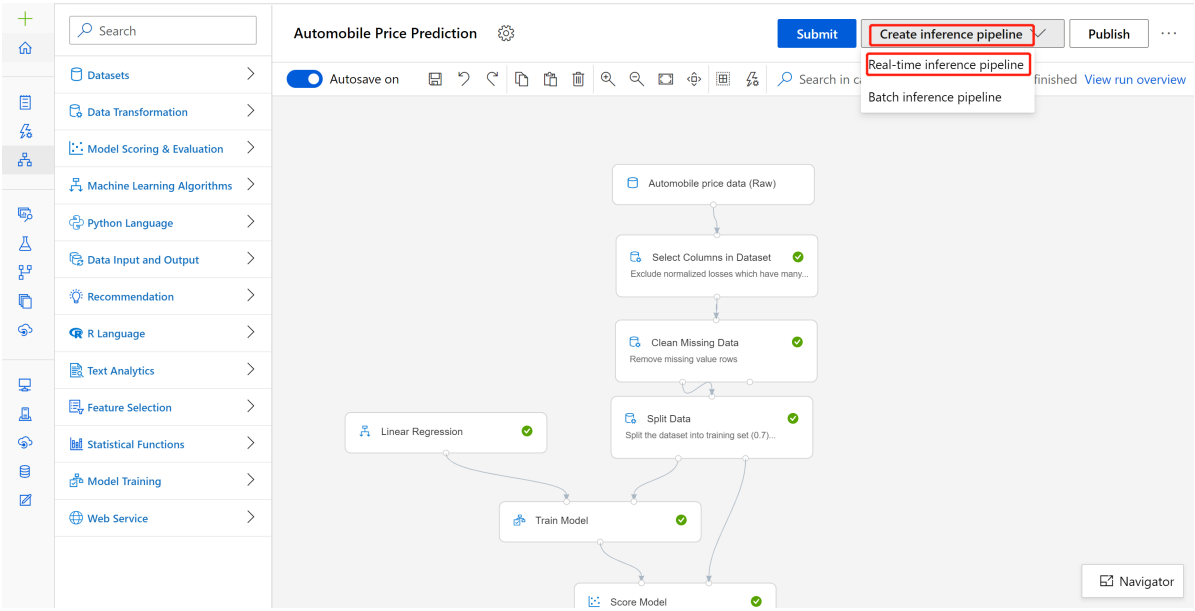
Complete part one of the tutorial to learn how to train and score a machine learning model in the designer.

Create a real-time inference pipeline

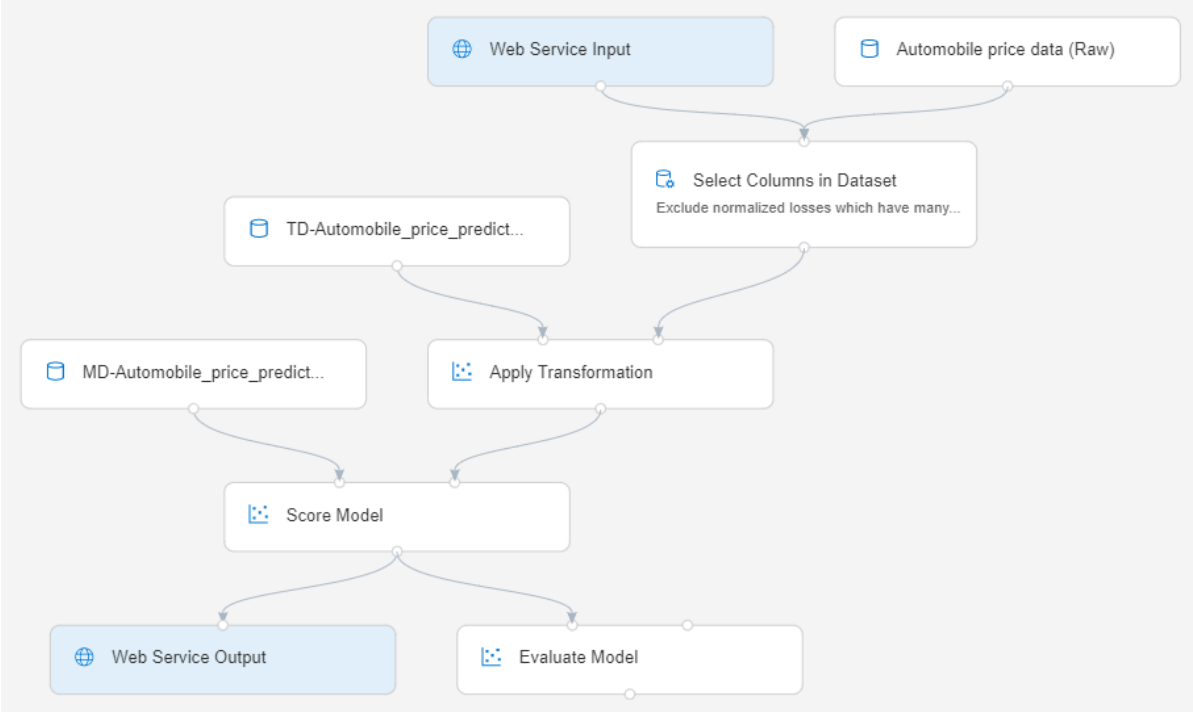
To deploy your pipeline, you must first convert the training pipeline into a real-time inference pipeline. This process removes training modules and adds web service inputs and outputs to handle requests.

Create a real-time inference pipeline

1. Above the pipeline canvas, select **Create inference pipeline** > **Real-time inference pipeline**.



Your pipeline should now look like this:



When you select **Create inference pipeline**, several things happen:

- The trained model is stored as a **Dataset** module in the module palette. You can find it under **My Datasets**.
- Training modules like **Train Model** and **Split Data** are removed.
- The saved trained model is added back into the pipeline.
- **Web Service Input** and **Web Service Output** modules are added. These modules show where user data enters the pipeline and where data is returned.

> **NOTE**
> By default, the **Web Service Input** will expect the same data schema as the training data used to create the predictive pipeline. In this scenario, price is included in the schema. However, price isn't used as a factor during prediction.

2. Select **Submit**, and use the same compute target and experiment that you used in part one.

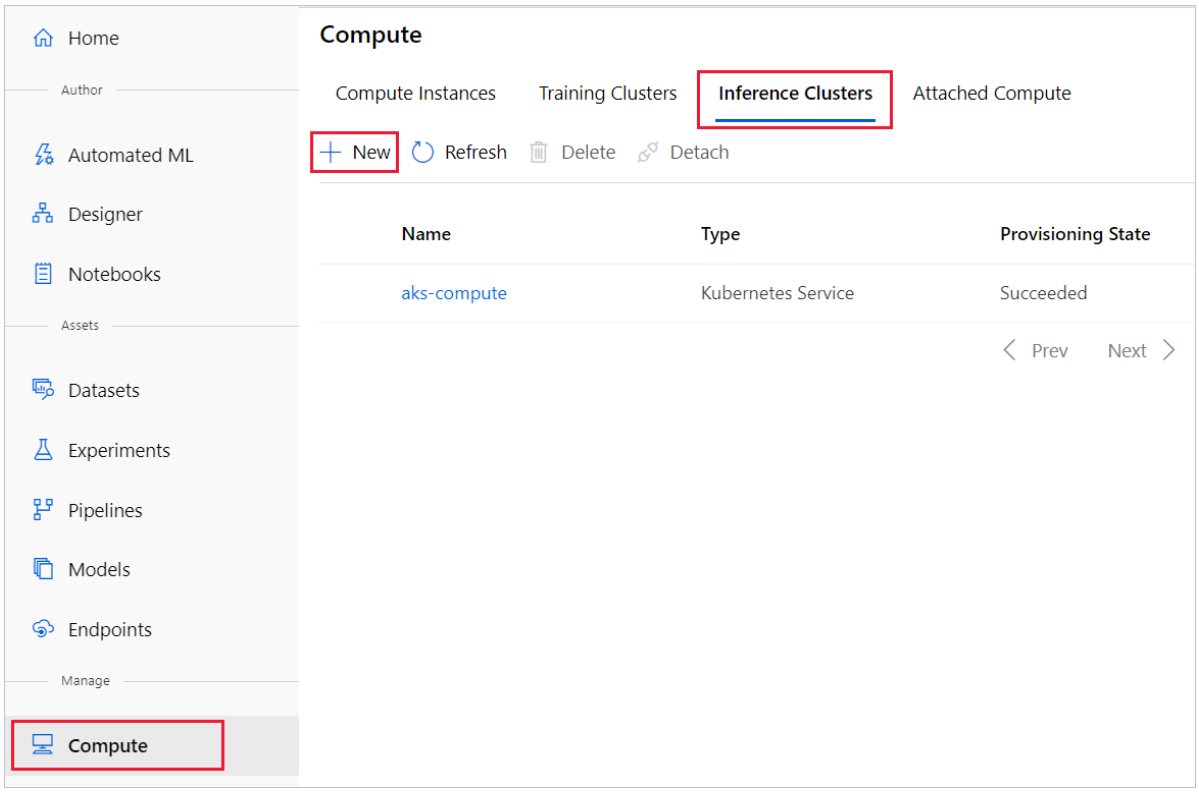
   If is the first run, it may take up to 20 minutes for your pipeline to finish running. The default compute settings have a minimum node size of 0, which means that the designer must allocate resources after being idle. Repeated pipeline runs will take less time since the compute resources are already allocated. Additionally, the designer uses cached results for each module to further improve efficiency.

3. Select **Deploy**.

Create an inferencing cluster

In the dialog box that appears, you can select from any existing Azure Kubernetes Service (AKS) clusters to deploy your model to. If you don't have an AKS cluster, use the following steps to create one.

1. Select **Compute** in the dialog box that appears to go to the **Compute** page.
2. On the navigation ribbon, select **Inference Clusters** > **+ New**.



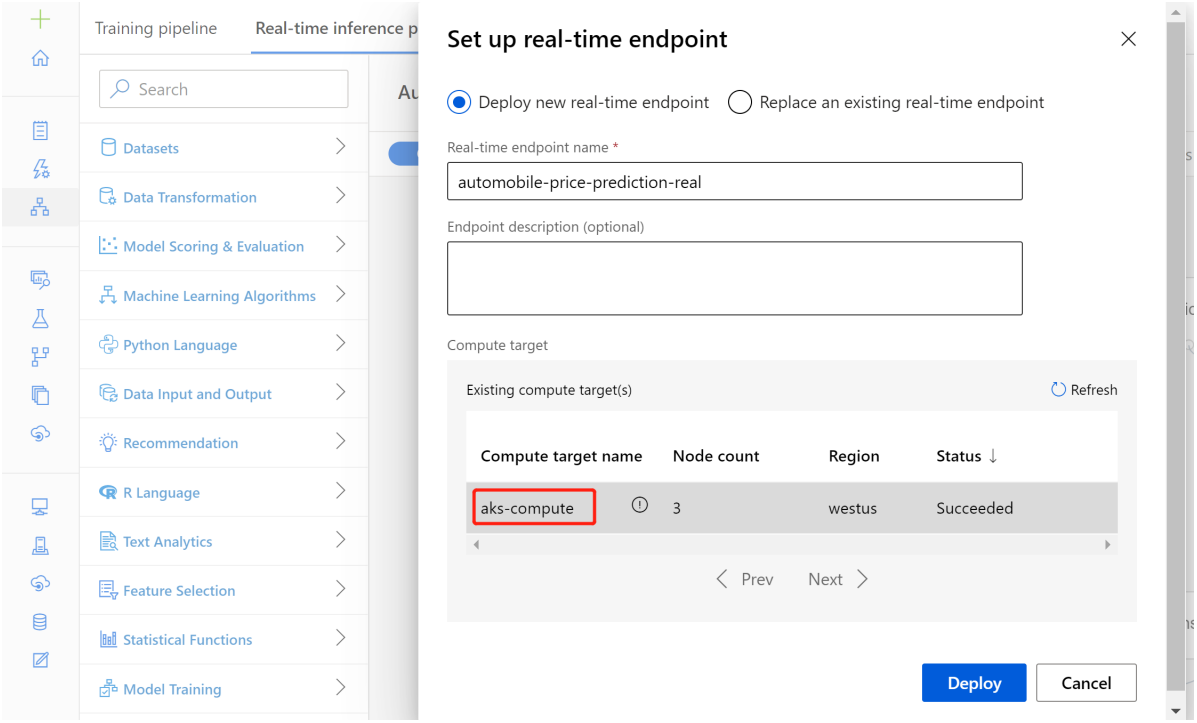
3. In the inference cluster pane, configure a new Kubernetes Service.
4. Enter *aks-compute* for the **Compute name**.
5. Select a nearby region that's available for the **Region**.
6. Select **Create**.

> **NOTE**
> It takes approximately 15 minutes to create a new AKS service. You can check the provisioning state on the **Inference Clusters** page.

Deploy the real-time endpoint

After your AKS service has finished provisioning, return to the real-time inferencing pipeline to complete deployment.

1. Select **Deploy** above the canvas.
2. Select **Deploy new real-time endpoint**.
3. Select the AKS cluster you created.
4. Select **Deploy**.

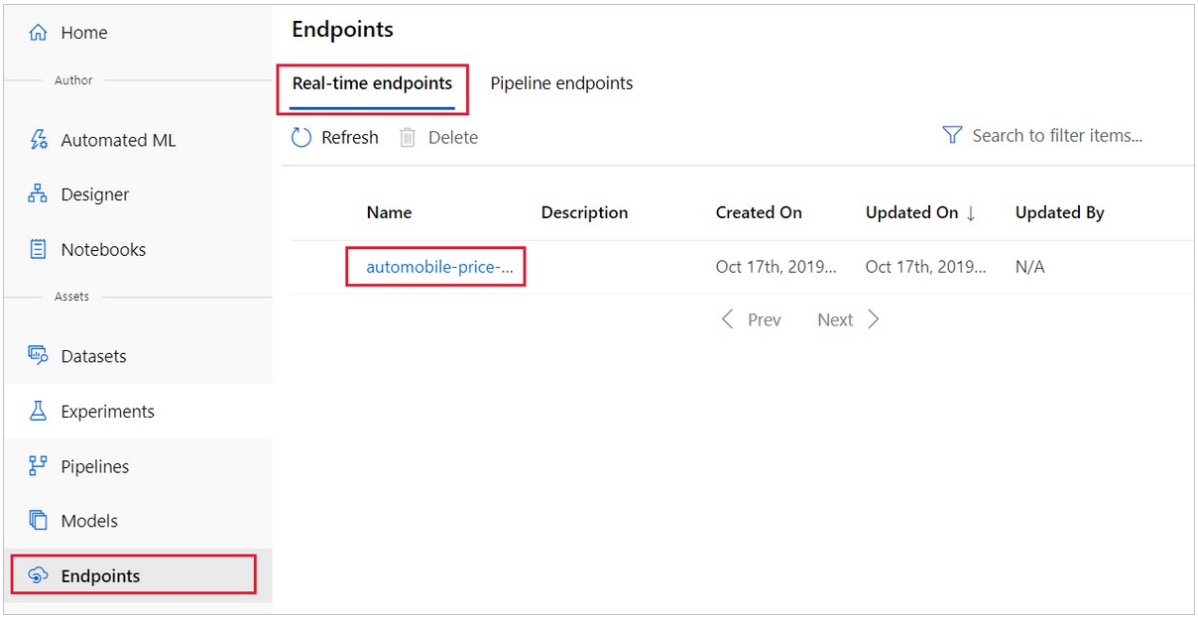


A success notification above the canvas appears after deployment finishes. It might take a few minutes.

Test the real-time endpoint

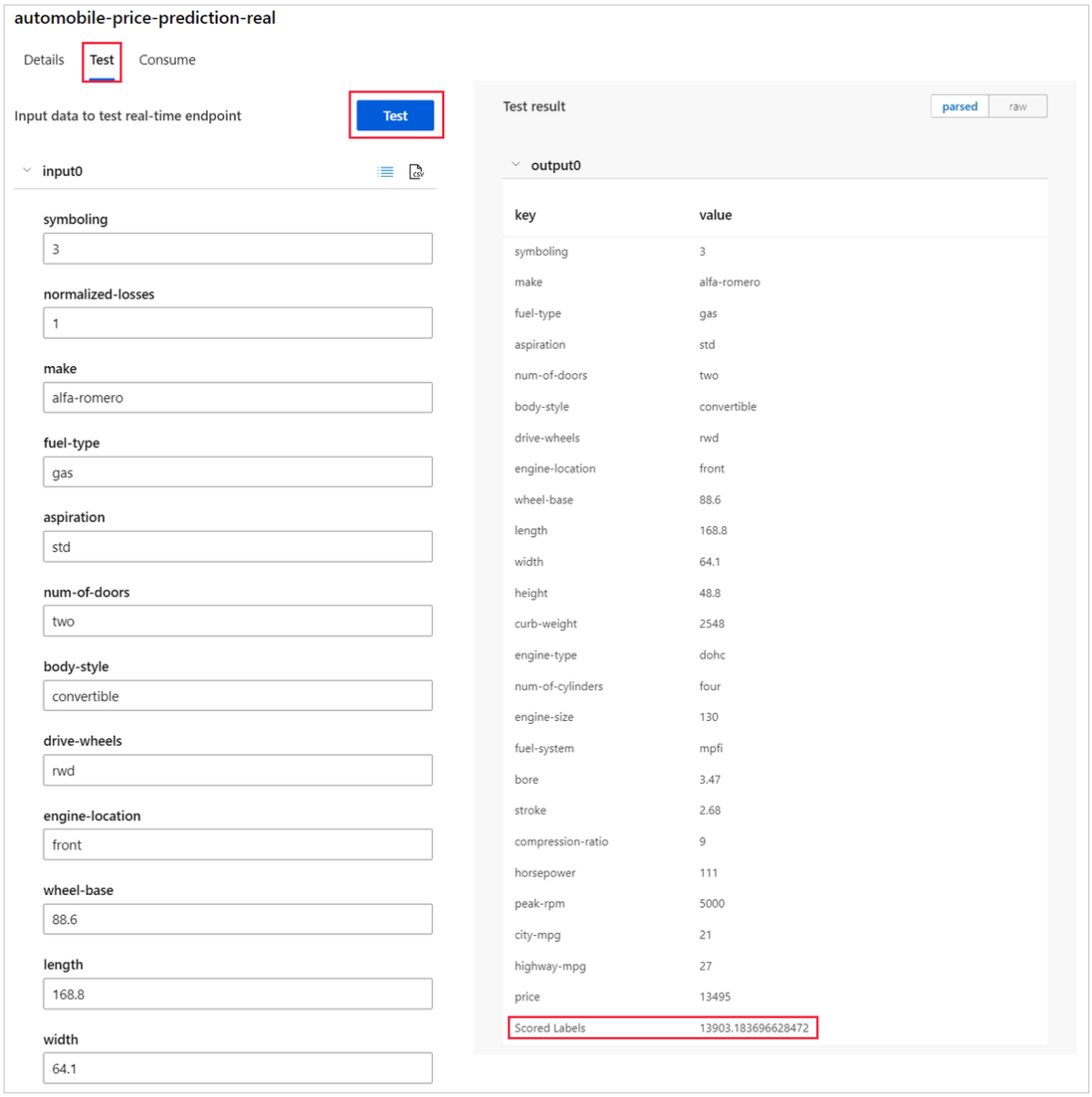
After deployment finishes, you can test your real-time endpoint by going to the **Endpoints** page.

1. On the **Endpoints** page, select the endpoint you deployed.



2. Select **Test**.
3. You can manually input testing data or use the autofilled sample data, and select **Test**.

   The portal submits a test request to the endpoint and shows the results. Although a price value is generated for the input data, it isn't used to generate the prediction value.



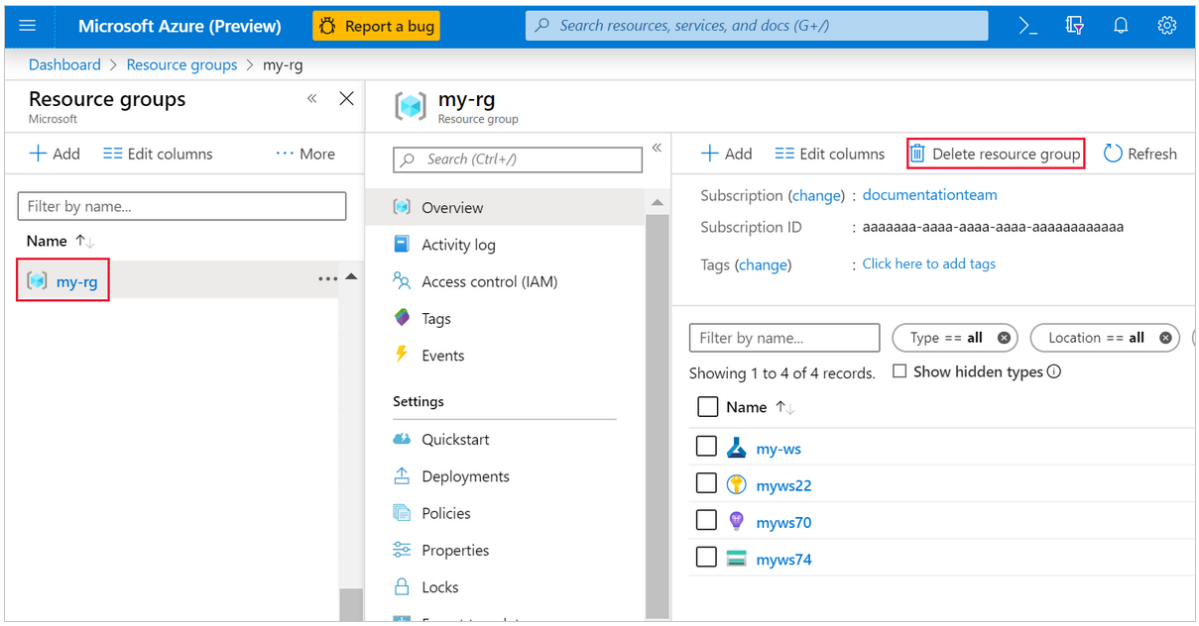
Clean up resources

> **IMPORTANT**
>
> You can use the resources that you created as prerequisites for other Azure Machine Learning tutorials and how-to articles.

Delete everything

If you don't plan to use anything that you created, delete the entire resource group so you don't incur any charges.

1. In the Azure portal, select **Resource groups** on the left side of the window.



2. In the list, select the resource group that you created.

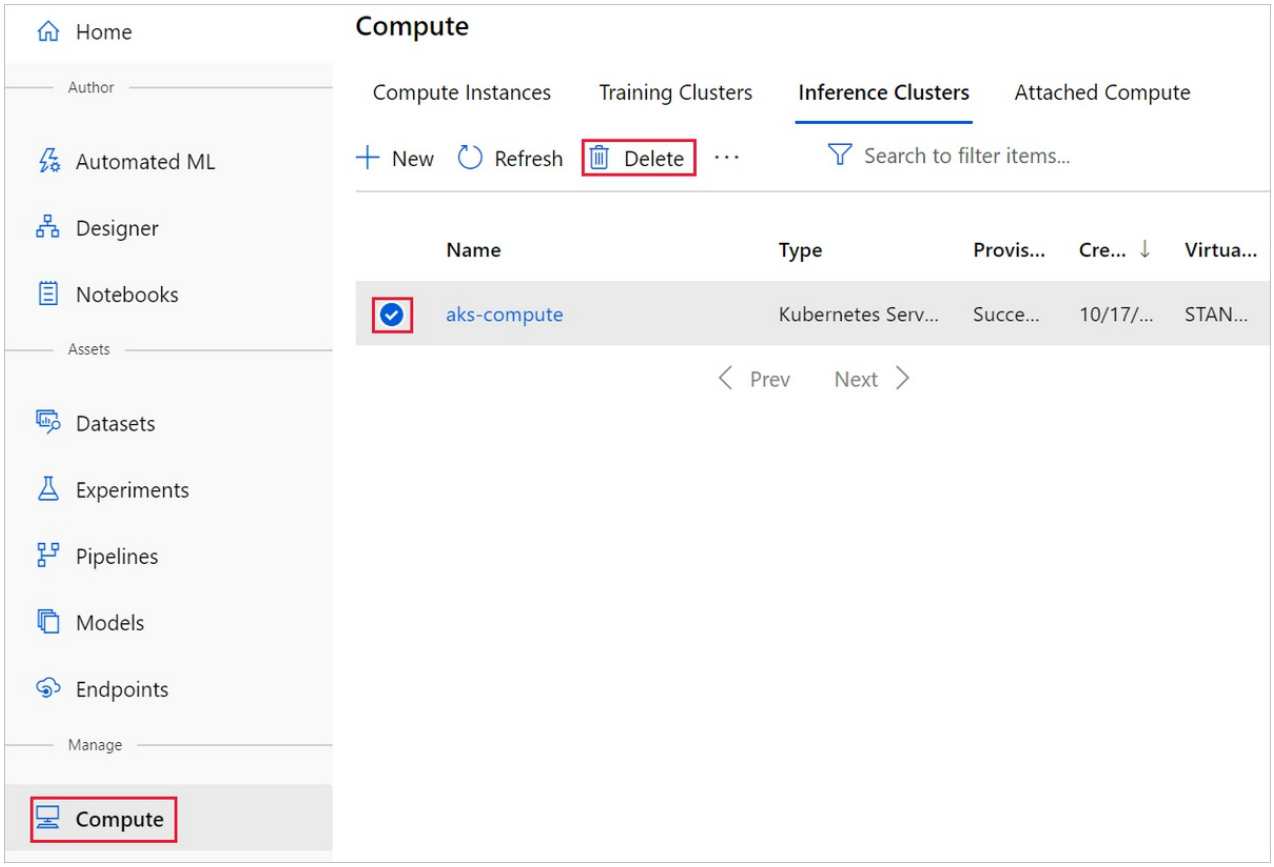
3. Select **Delete resource group**.

Deleting the resource group also deletes all resources that you created in the designer.

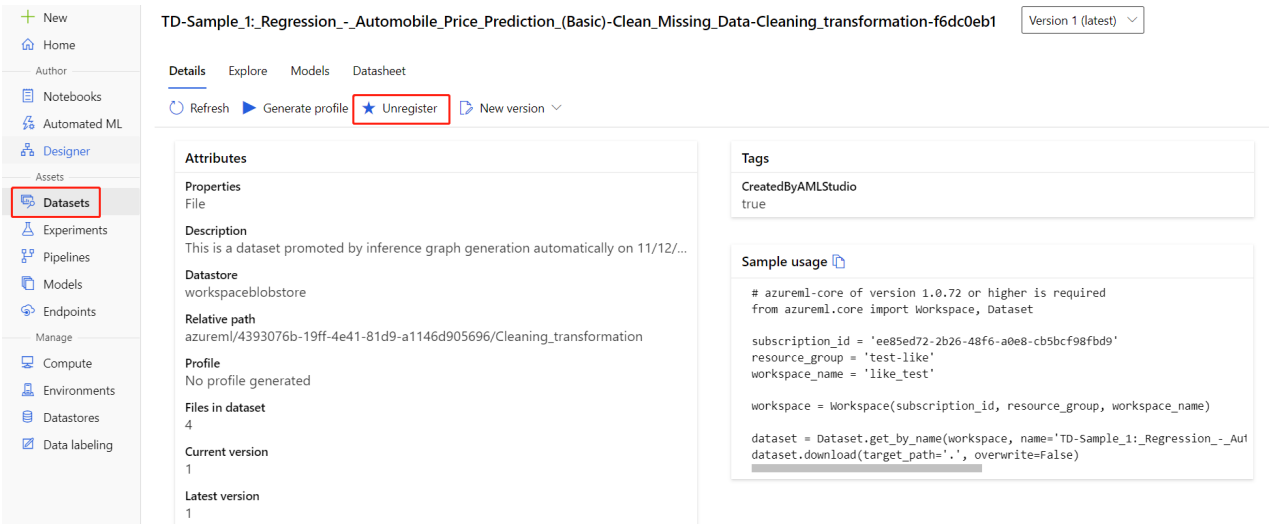
Delete individual assets

In the designer where you created your experiment, delete individual assets by selecting them and then selecting the **Delete** button.

The compute target that you created here *automatically autoscales* to zero nodes when it's not being used. This action is taken to minimize charges. If you want to delete the compute target, take these steps:



You can unregister datasets from your workspace by selecting each dataset and selecting **Unregister**.



To delete a dataset, go to the storage account by using the Azure portal or Azure Storage Explorer and manually delete those assets.

Next steps

In this tutorial, you learned the key steps in how to create, deploy, and consume a machine learning model in the designer. To learn more about how you can use the designer to solve other types of problems, see our other sample pipelines.

Designer samples

Tutorial: Create a classification model with automated ML in Azure Machine Learning

3/27/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this tutorial, you learn how to create a basic classification model without writing a single line of code using Azure Machine Learning's automated machine learning interface. This classification model predicts if a client will subscribe to a fixed term deposit with a financial institution.

With automated machine learning, you can automate away time intensive tasks. Automated machine learning rapidly iterates over many combinations of algorithms and hyperparameters to help you find the best model based on a success metric of your choosing.

In this tutorial, you learn how to do the following tasks:

- Create an Azure Machine Learning workspace.
- Run an automated machine learning experiment.
- View experiment details.
- Deploy the model.

Prerequisites

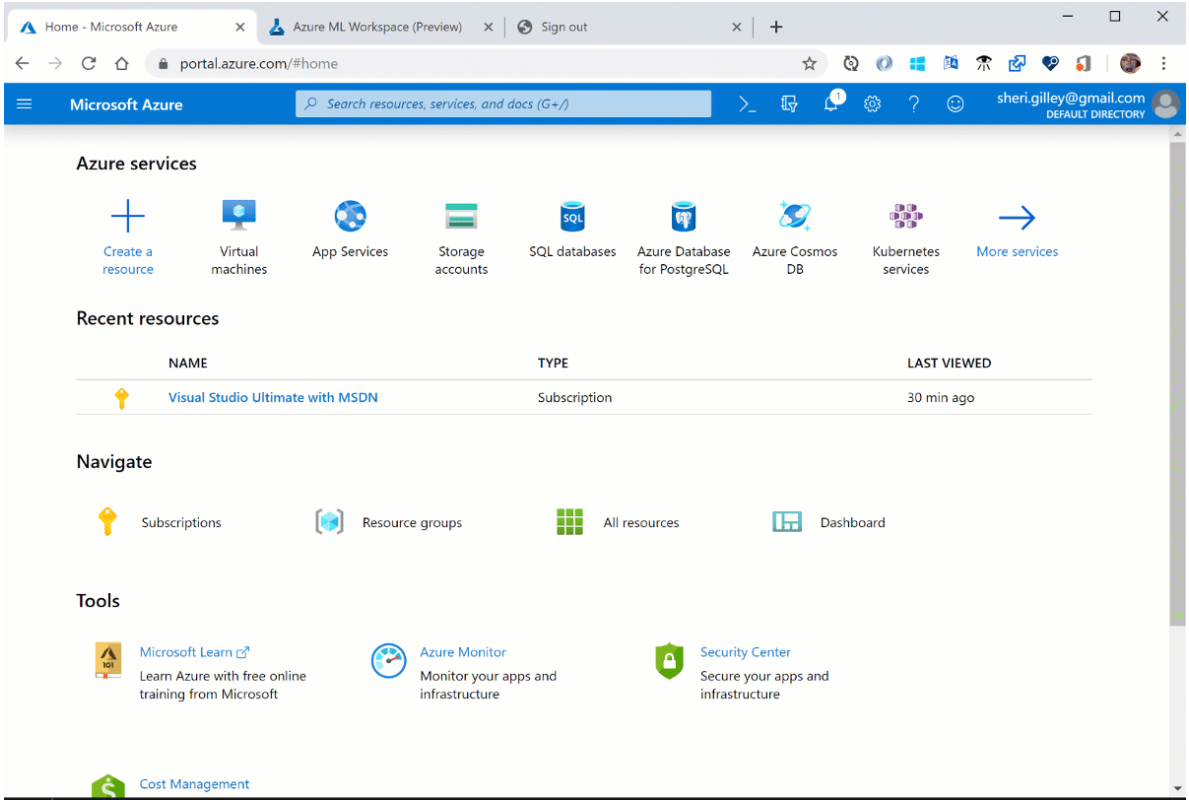
- An Azure subscription. If you don't have an Azure subscription, create a free account.
- Download the **bankmarketing_train.csv** data file. The **y** column indicates if a customer subscribed to a fixed term deposit, which is later identified as the target column for predictions in this tutorial.

Create a workspace

An Azure Machine Learning workspace is a foundational resource in the cloud that you use to experiment, train, and deploy machine learning models. It ties your Azure subscription and resource group to an easily consumed object in the service.

You create a workspace via the Azure portal, a web-based console for managing your Azure resources.

1. Sign in to the Azure portal by using the credentials for your Azure subscription.
2. In the upper-left corner of the Azure portal, select **+ Create a resource**.



3. Use the search bar to find **Machine Learning**.
4. Select **Machine Learning**.
5. In the **Machine Learning** pane, select **Create** to begin.
6. Provide the following information to configure your new workspace:

FIELD	DESCRIPTION
Workspace name	Enter a unique name that identifies your workspace. In this example, we use **docs-ws**. Names must be unique across the resource group. Use a name that's easy to recall and to differentiate from workspaces created by others.
Subscription	Select the Azure subscription that you want to use.
Resource group	Use an existing resource group in your subscription, or enter a name to create a new resource group. A resource group holds related resources for an Azure solution. In this example, we use **docs-aml**.
Location	Select the location closest to your users and the data resources to create your workspace.
Workspace edition	Select **Enterprise**. This tutorial requires the use of the Enterprise edition. The Enterprise edition is in preview and doesn't currently add any extra costs.

7. After you're finished configuring the workspace, select **Create**.

> **WARNING**
> It can take several minutes to create your workspace in the cloud.

When the process is finished, a deployment success message appears.

8. To view the new workspace, select **Go to resource**.

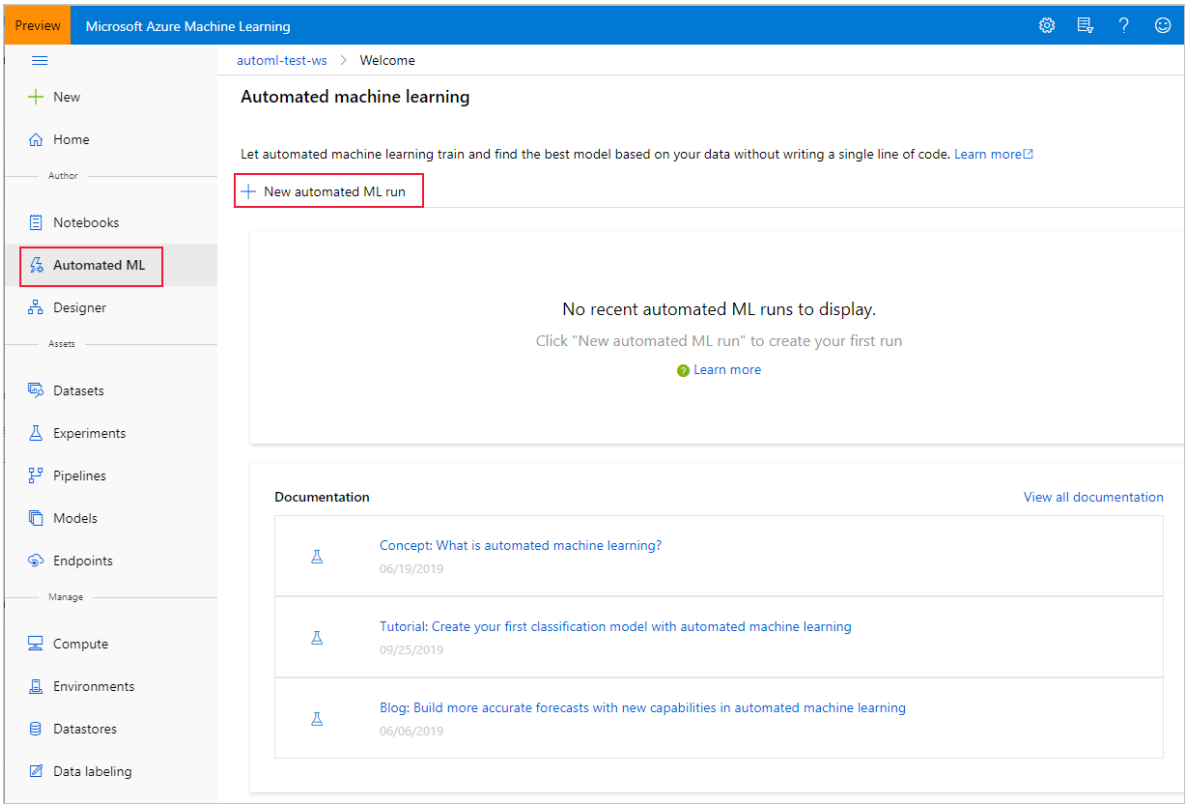
> **IMPORTANT**
> Take note of your **workspace** and **subscription**. You'll need these to ensure you create your experiment in the right place.

Create and run the experiment

You complete the following experiment set-up and run steps via Azure Machine learning at https://ml.azure.com, a consolidated web interface that includes machine learning tools to perform data science scenarios for data science practitioners of all skill levels. This interface is not supported on Internet Explorer browsers.

1. Sign in to Azure Machine Learning.
2. Select your subscription and the workspace you created.
3. Select **Get started**.
4. In the left pane, select **Automated ML** under the **Author** section.

   Since this is your first automated ML experiment, you'll see an empty list and links to documentation.



5. Select **New automated ML run**.
6. Create a new dataset by selecting **From local files** from the **+Create dataset** drop-down.

   a. On the **Basic info** form, give your dataset a name and provide an optional description. The automated ML interface currently only supports TabularDatasets, so the dataset type should default

to *Tabular*.

b. Select **Next** on the bottom left

c. On the **Datastore and file selection** form, select the default datastore that was automatically set up during your workspace creation, **workspaceblobstore (Azure Blob Storage)**. This is where you'll upload your data file to make it available to your workspace.

d. Select **Browse**.

e. Choose the **bankmarketing_train.csv** file on your local computer. This is the file you downloaded as a prerequisite.

f. Give your dataset a unique name and provide an optional description.

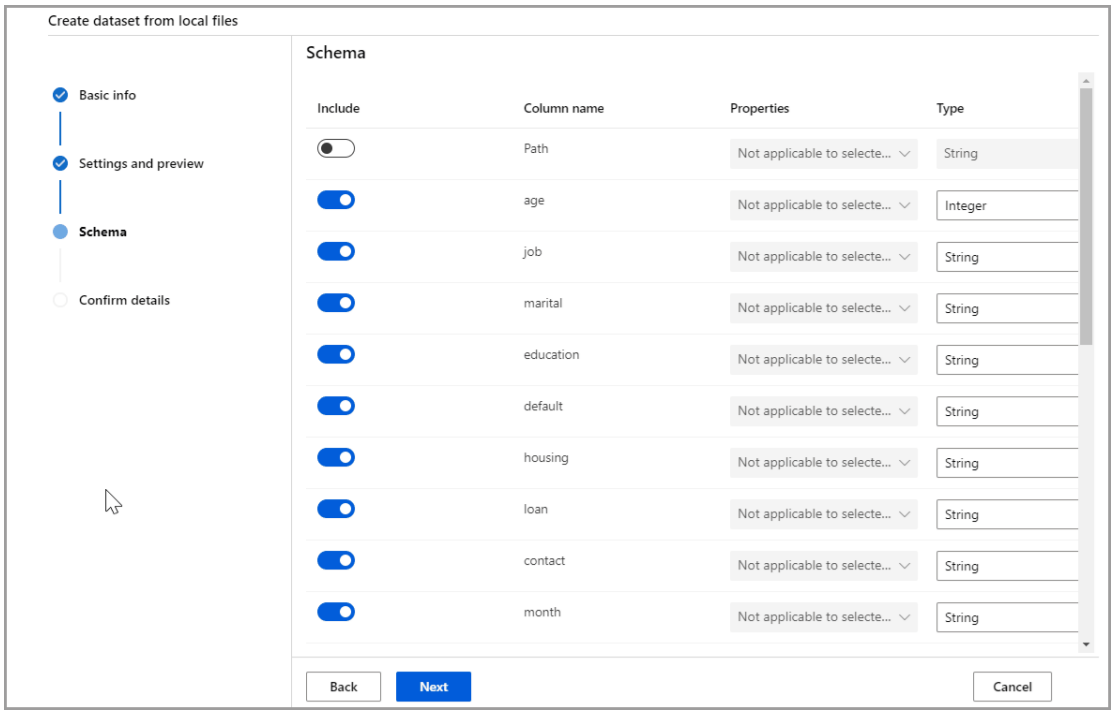
g. Select **Next** on the bottom left, to upload it to the default container that was automatically set up during your workspace creation.

   When the upload is complete, the Settings and preview form is pre-populated based on the file type.

h. Verify that the **Settings and preview** form is populated as follows and select **Next**.

FIELD	DESCRIPTION	VALUE FOR TUTORIAL
File format	Defines the layout and type of data stored in a file.	Delimited
Delimiter	One or more characters for specifying the boundary between separate, independent regions in plain text or other data streams.	Comma
Encoding	Identifies what bit to character schema table to use to read your dataset.	UTF-8
Column headers	Indicates how the headers of the dataset, if any, will be treated.	All files have same headers
Skip rows	Indicates how many, if any, rows are skipped in the dataset.	None

i. The **Schema** form allows for further configuration of your data for this experiment. For this example, select the toggle switch for the **day_of_week** feature, so as to not include it for this experiment. Select **Next**.



j. On the **Confirm details** form, verify the information matches what was previously populated on the **Basic info** and **Settings and preview** forms.

k. Select **Create** to complete the creation of your dataset.

l. Select your dataset once it appears in the list.

m. Review the **Data preview** to ensure you didn't include **day_of_week** then, select **OK**.

n. Select **Next**.

7. Populate the **Configure Run** form as follows:

a. Enter this experiment name: `my-1st-automl-experiment`

b. Select **y** as the target column, what you want to predict. This column indicates whether the client subscribed to a term deposit or not.

c. Select **Create a new compute** and configure your compute target. A compute target is a local or cloud-based resource environment used to run your training script or host your service deployment. For this experiment, we use a cloud-based compute.

FIELD	DESCRIPTION	VALUE FOR TUTORIAL
Compute name	A unique name that identifies your compute context.	automl-compute
Virtual machine size	Select the virtual machine size for your compute.	Standard_DS12_V2
Min / Max nodes (in Advanced Settings)	To profile data, you must specify 1 or more nodes.	Min nodes: 1 Max nodes: 6

a. Select **Create** to get the compute target.

This takes a couple minutes to complete.

b. After creation, select your new compute target from the drop-down list.

d. Select **Next**.

8. On the **Task type and settings** form, select **Classification** as the machine learning task type.

   a. Select **View additional configuration settings** and populate the fields as follows. These settings are to better control the training job. Otherwise, defaults are applied based on experiment selection and data.

   > **NOTE**
   > In this tutorial, you won't set a metric score or max cores per iterations threshold. Nor will you block algorithms from being tested.

ADDITIONAL CONFIGURATIONS	DESCRIPTION	VALUE FOR TUTORIAL
Primary metric	Evaluation metric that the machine learning algorithm will be measured by.	AUC_weighted
Automatic featurization	Enables preprocessing. This includes automatic data cleansing, preparing, and transformation to generate synthetic features.	Enable
Blocked algorithms	Algorithms you want to exclude from the training job	None
Exit criterion	If a criteria is met, the training job is stopped.	Training job time (hours): 1 Metric score threshold: None
Validation	Choose a cross-validation type and number of tests.	Validation type: k-fold cross-validation Number of validations: 2
Concurrency	The maximum number of parallel iterations executed per iteration	Max concurrent iterations: 5

   Select **Save**.

9. Select **Finish** to run the experiment. The **Run Detail** screen opens with the **Run status** at the top as the experiment preparation begins.

> **IMPORTANT**
> Preparation takes **10-15 minutes** to prepare the experiment run. Once running, it takes **2-3 minutes more for each iteration**.
> Select **Refresh** periodically to see the status of the run as the experiment progresses.
>
> In production, you'd likely walk away for a bit. But for this tutorial, we suggest you start exploring the tested algorithms on the **Models** tab as they complete while the others are still running.

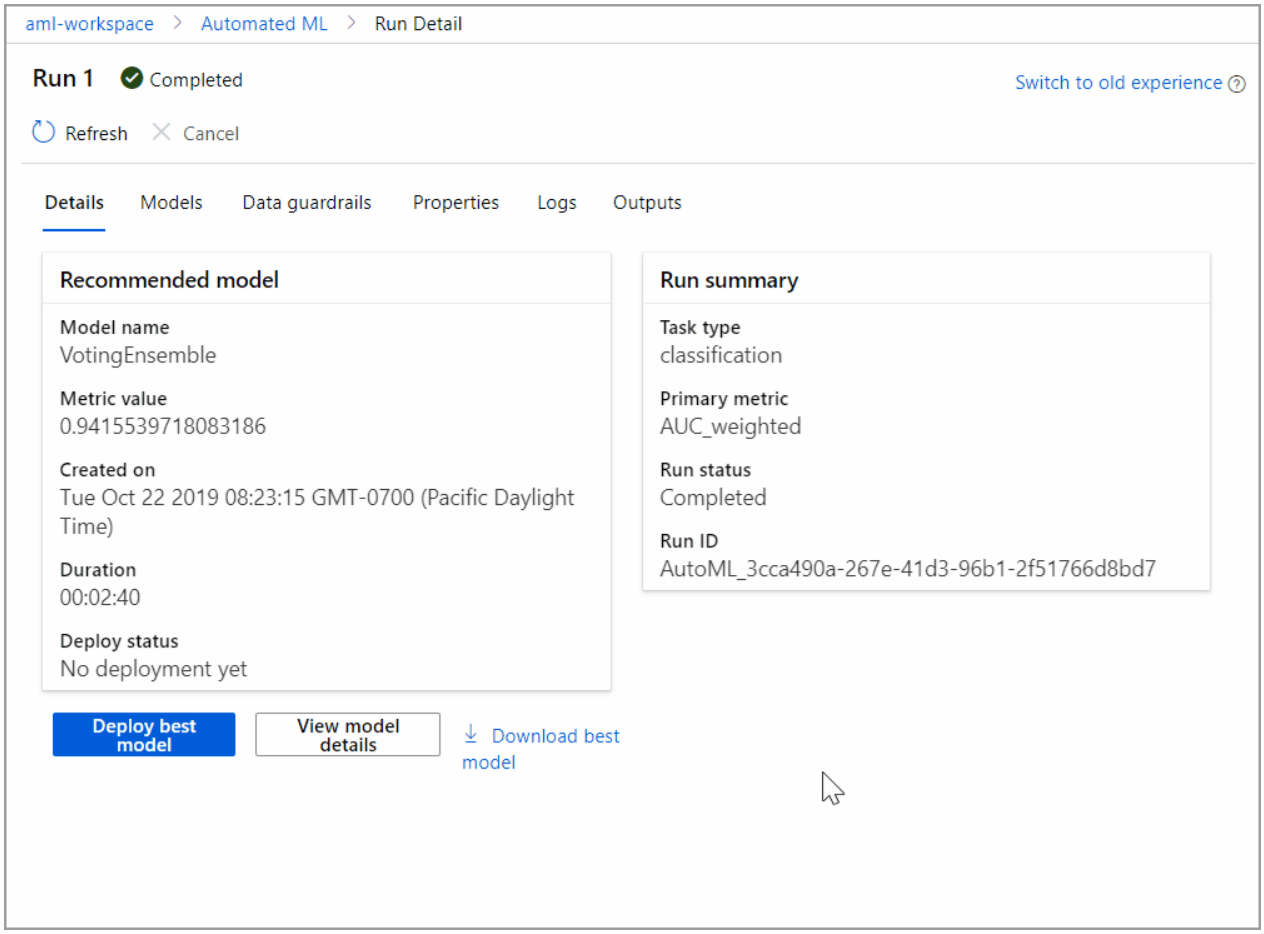
Explore models

Navigate to the **Models** tab to see the algorithms (models) tested. By default, the models are ordered by metric score as they complete. For this tutorial, the model that scores the highest based on the chosen **AUC_weighted**

metric is at the top of the list.

While you wait for all of the experiment models to finish, select the **Algorithm name** of a completed model to explore its performance details.

The following navigates through the **Model details** and the **Visualizations** tabs to view the selected model's properties, metrics, and performance charts.



Deploy the best model

The automated machine learning interface allows you to deploy the best model as a web service in a few steps. Deployment is the integration of the model so it can predict on new data and identify potential areas of opportunity.

For this experiment, deployment to a web service means that the financial institution now has an iterative and scalable web solution for identifying potential fixed term deposit customers.

Once the run is complete, navigate back to the **Run Detail** page and select the **Models** tab.

In this experiment context, **VotingEnsemble** is considered the best model, based on the **AUC_weighted** metric. We deploy this model, but be advised, deployment takes about 20 minutes to complete. The deployment process entails several steps including registering the model, generating resources, and configuring them for the web service.

1. Select the **Deploy best model** button in the bottom-left corner.
2. Populate the **Deploy a model** pane as follows:

FIELD	VALUE
Deployment name	my-automl-deploy

FIELD	VALUE
Deployment description	My first automated machine learning experiment deployment
Compute type	Select Azure Compute Instance (ACI)
Enable authentication	Disable.
Use custom deployments	Disable. Allows for the default driver file (scoring script) and environment file to be autogenerated.

For this example, we use the defaults provided in the *Advanced* menu.

3. Select **Deploy**.

   A green success message appears at the top of the **Run** screen, and in the **Recommended model** pane, a status message appears under **Deploy status**. Select **Refresh** periodically to check the deployment status.

Now you have an operational web service to generate predictions.

Proceed to the **Next Steps** to learn more about how to consume your new web service, and test your predictions using Power BI's built in Azure Machine Learning support.

Clean up resources

Deployment files are larger than data and experiment files, so they cost more to store. Delete only the deployment files to minimize costs to your account, or if you want to keep your workspace and experiment files. Otherwise, delete the entire resource group, if you don't plan to use any of the files.

Delete the deployment instance

Delete just the deployment instance from Azure Machine Learning at https://ml.azure.com/, if you want to keep the resource group and workspace for other tutorials and exploration.

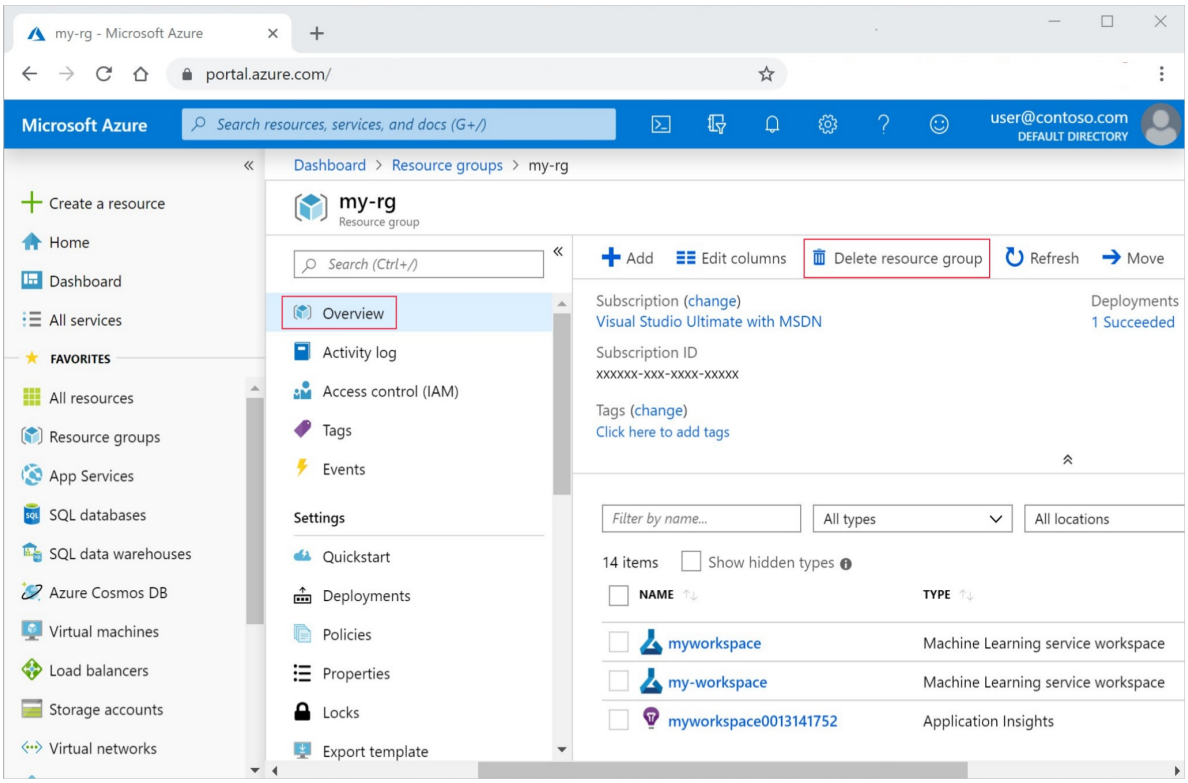
1. Go to Azure Machine Learning. Navigate to your workspace and on the left under the **Assets** pane, select **Endpoints**.
2. Select the deployment you want to delete and select **Delete**.
3. Select **Proceed**.

Delete the resource group

> **IMPORTANT**
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

Next steps

In this automated machine learning tutorial, you used Azure Machine Learning's automated ML interface to create and deploy a classification model. See these articles for more information and next steps:

Consume a web service

- Learn more about automated machine learning.
- For more information on classification metrics and charts, see the Understand automated machine learning results article.+ Learn more about featurization.
- Learn more about data profiling.

> **NOTE**
>
> This Bank Marketing dataset is made available under the Creative Commons (CCO: Public Domain) License. Any rights in individual contents of the database are licensed under the Database Contents License and available on Kaggle. This dataset was originally available within the UCI Machine Learning Database.
>
> [Moro et al., 2014] S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014.

Tutorial: Forecast bike sharing demand with automated machine learning

2/7/2020 • 8 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this tutorial, you use automated machine learning, or automated ML, in the Azure Machine Learning studio to create a time series forecasting model to predict rental demand for a bike sharing service.

In this tutorial, you learn how to do the following tasks:

- Create and load a dataset.
- Configure and run an automated ML experiment.
- Explore the experiment results.
- Deploy the best model.

Prerequisites

- An Enterprise edition Azure Machine Learning workspace. If you don't have a workspace, create an Enterprise edition workspace.
  - Automated machine learning in the Azure Machine Learning studio is only available for Enterprise edition workspaces.
- Download the bike-no.csv data file

Get started in Azure Machine Learning studio

For this tutorial, you create your automated ML experiment run in Azure Machine Learning studio, a consolidated interface that includes machine learning tools to perform data science scenarios for data science practitioners of all skill levels. The studio is not supported on Internet Explorer browsers.

1. Sign in to Azure Machine Learning studio.
2. Select your subscription and the workspace you created.
3. Select **Get started**.
4. In the left pane, select **Automated ML** under the **Author** section.
5. Select **+New automated ML run**.

Create and load dataset

Before you configure your experiment, upload your data file to your workspace in the form of an Azure Machine Learning dataset. Doing so, allows you to ensure that your data is formatted appropriately for your experiment.

1. On the **Select dataset** form, select **From local files** from the **+Create dataset** drop-down.
   a. On the **Basic info** form, give your dataset a name and provide an optional description. The dataset type should default to **Tabular**, since automated ML in Azure Machine Learning studio currently only supports tabular datasets.
   b. Select **Next** on the bottom left

c. On the **Datastore and file selection** form, select the default datastore that was automatically set up during your workspace creation, **workspaceblobstore (Azure Blob Storage)**. This is the storage location where you'll upload your data file.

d. Select **Browse**.

e. Choose the **bike-no.csv** file on your local computer. This is the file you downloaded as a prerequisite.

f. Select **Next**

When the upload is complete, the Settings and preview form is pre-populated based on the file type.

g. Verify that the **Settings and preview** form is populated as follows and select **Next**.

FIELD	DESCRIPTION	VALUE FOR TUTORIAL
File format	Defines the layout and type of data stored in a file.	Delimited
Delimiter	One or more characters for specifying the boundary between separate, independent regions in plain text or other data streams.	Comma
Encoding	Identifies what bit to character schema table to use to read your dataset.	UTF-8
Column headers	Indicates how the headers of the dataset, if any, will be treated.	Use headers from the first file
Skip rows	Indicates how many, if any, rows are skipped in the dataset.	None

h. The **Schema** form allows for further configuration of your data for this experiment.

a. For this example, choose to ignore the **casual** and **registered** columns. These columns are a breakdown of the **cnt** column so, therefore we don't include them.

b. Also for this example, leave the defaults for the **Properties** and **Type**.

c. Select **Next**.

i. On the **Confirm details** form, verify the information matches what was previously populated on the **Basic info** and **Settings and preview** forms.

j. Select **Create** to complete the creation of your dataset.

k. Select your dataset once it appears in the list.

l. Select **Next**.

Configure experiment run

After you load and configure your data, set up your remote compute target and select which column in your data you want to predict.

1. Populate the **Configure run** form as follows:
   a. Enter an experiment name: `automl-bikeshare`

b. Select **cnt** as the target column, what you want to predict. This column indicates the number of total bike share rentals.

c. Select **Create a new compute** and configure your compute target. Automated ML only supports Azure Machine Learning compute.

FIELD	DESCRIPTION	VALUE FOR TUTORIAL
Compute name	A unique name that identifies your compute context.	bike-compute
Virtual machine size	Select the virtual machine size for your compute.	Standard_DS12_V2
Min / Max nodes (in Advanced Settings)	To profile data, you must specify 1 or more nodes.	Min nodes: 1 Max nodes: 6

a. Select **Create** to get the compute target.

This takes a couple minutes to complete.

b. After creation, select your new compute target from the drop-down list.

d. Select **Next**.

Select task type and settings

Complete the setup for your automated ML experiment by specifying the machine learning task type and configuration settings.

1. On the **Task type and settings** form, select **Time series forecasting** as the machine learning task type.
2. Select **date** as your **Time column** and leave **Group by column(s)** blank.

a. Select **View additional configuration settings** and populate the fields as follows. These settings are to better control the training job. Otherwise, defaults are applied based on experiment selection and data.

ADDITIONAL CONFIGURATIONS	DESCRIPTION	VALUE FOR TUTORIAL
Primary metric	Evaluation metric that the machine learning algorithm will be measured by.	Normalized root mean squared error
Automatic featurization	Enables preprocessing. This includes automatic data cleansing, preparing, and transformation to generate synthetic features.	Enable
Explain best model (preview)	Automatically shows explainability on the best model created by automated ML.	Enable
Blocked algorithms	Algorithms you want to exclude from the training job	Extreme Random Trees

ADDITIONAL CONFIGURATIONS	DESCRIPTION	VALUE FOR TUTORIAL
Additional forecasting settings	These settings help improve the accuracy of your model ***Forecast horizon***: length of time into the future you want to predict ***Forecast target lags:*** how far back you want to construct the lags of a the target variable ***Target rolling window***: specifies the size of the rolling window over which features, such as the *max*, *min* and *sum*, will be generated.	Forecast horizon: 14 Forecast target lags: None Target rolling window size: None
Exit criterion	If a criteria is met, the training job is stopped.	Training job time (hours): 3 Metric score threshold: None
Validation	Choose a cross-validation type and number of tests.	Validation type: k-fold cross-validation Number of validations: 5
Concurrency	The maximum number of parallel iterations executed per iteration	Max concurrent iterations: 6

Select **Save**.

Run experiment

To run your experiment, select **Finish**. The **Run details** screen opens with the **Run status** at the top next to the run number. This status updates as the experiment progresses.

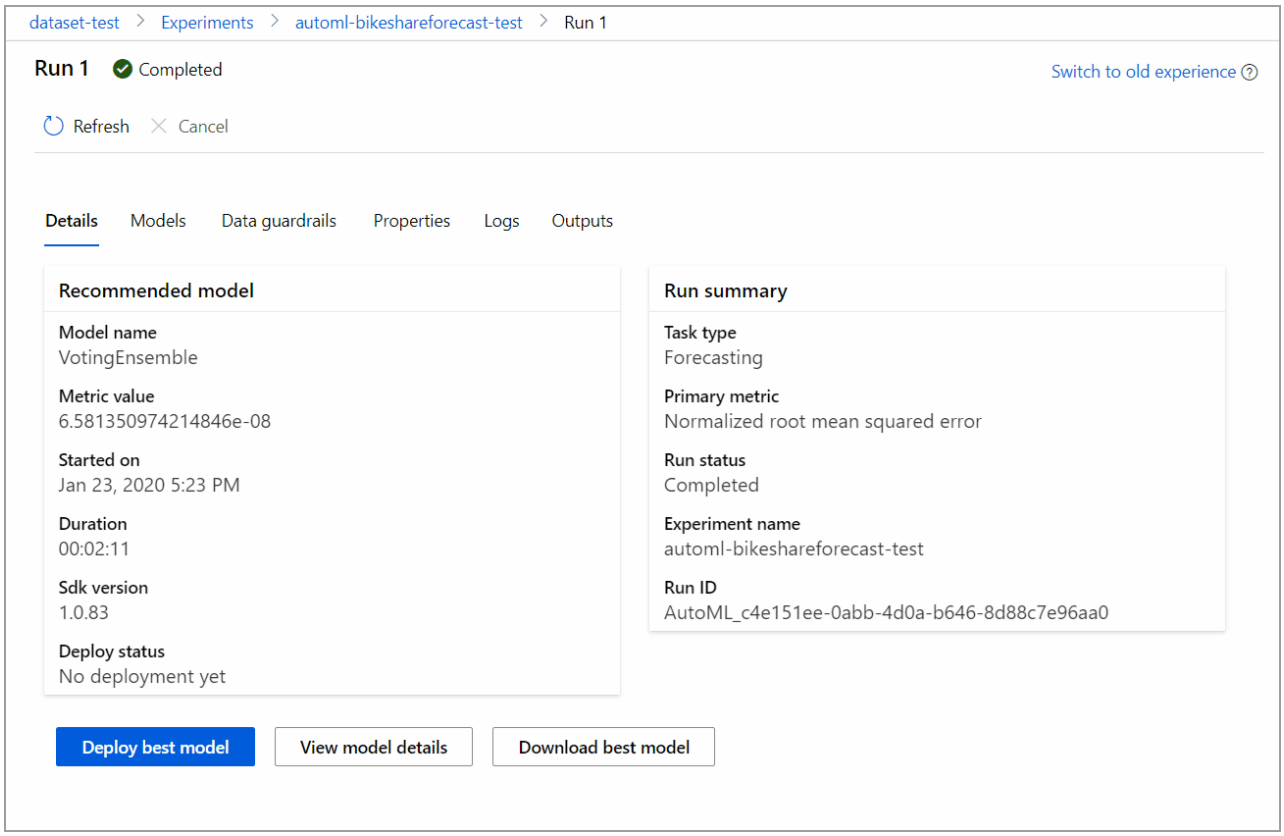
> **IMPORTANT**
> Preparation takes **10-15 minutes** to prepare the experiment run. Once running, it takes **2-3 minutes more for each iteration**.
>
> In production, you'd likely walk away for a bit as this process takes time. While you wait, we suggest you start exploring the tested algorithms on the **Models** tab as they complete.

Explore models

Navigate to the **Models** tab to see the algorithms (models) tested. By default, the models are ordered by metric score as they complete. For this tutorial, the model that scores the highest based on the chosen **Normalized root mean squared error** metric is at the top of the list.

While you wait for all of the experiment models to finish, select the **Algorithm name** of a completed model to explore its performance details.

The following example navigates through the **Model details** and the **Visualizations** tabs to view the selected model's properties, metrics and performance charts.



Deploy the model

Automated machine learning in Azure Machine Learning studio allows you to deploy the best model as a web service in a few steps. Deployment is the integration of the model so it can predict on new data and identify potential areas of opportunity.

For this experiment, deployment to a web service means that the bike share company now has an iterative and scalable web solution for forecasting bike share rental demand.

Once the run is complete, navigate back to the **Run detail** page and select the **Models** tab.

In this experiment context, **StackEnsemble** is considered the best model, based on the **Normalized root mean squared error** metric. We deploy this model, but be advised, deployment takes about 20 minutes to complete. The deployment process entails several steps including registering the model, generating resources, and configuring them for the web service.

1. Select the **Deploy best model** button in the bottom-left corner.
2. Populate the **Deploy a model** pane as follows:

FIELD	VALUE
Deployment name	bikeshare-deploy
Deployment description	bike share demand deployment
Compute type	Select Azure Compute Instance (ACI)
Enable authentication	Disable.
Use custom deployment assets	Disable. Disabling allows for the default driver file (scoring script) and environment file to be autogenerated.

For this example, we use the defaults provided in the *Advanced* menu.

3. Select **Deploy**.

   A green success message appears at the top of the **Run** screen stated that the deployment was started successfully. The progress of the deployment can be found in the **Recommended model** pane under **Deploy status**.

Once deployment succeeds, you have an operational web service to generate predictions.

Proceed to the **Next steps** to learn more about how to consume your new web service, and test your predictions using Power BI's built in Azure Machine Learning support.

Clean up resources

Deployment files are larger than data and experiment files, so they cost more to store. Delete only the deployment files to minimize costs to your account, or if you want to keep your workspace and experiment files. Otherwise, delete the entire resource group, if you don't plan to use any of the files.

Delete the deployment instance

Delete just the deployment instance from the Azure Machine Learning studio, if you want to keep the resource group and workspace for other tutorials and exploration.

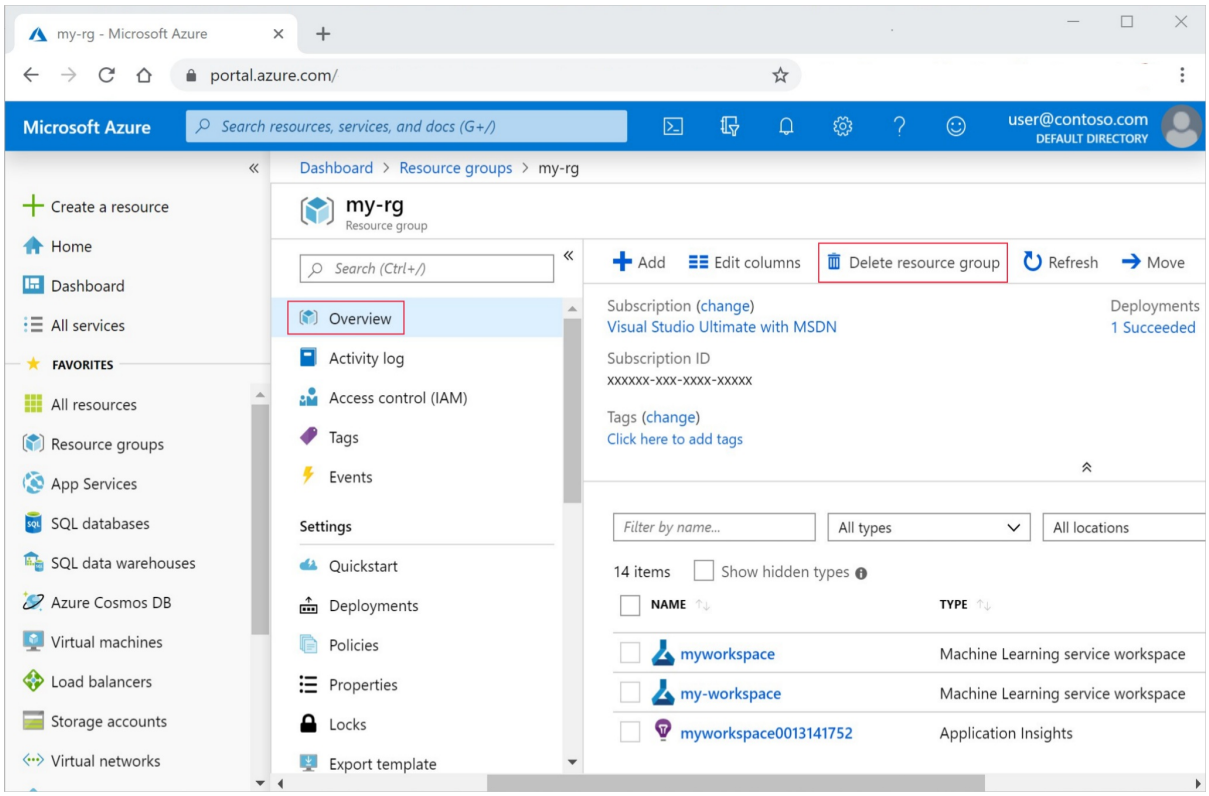
1. Go to the Azure Machine Learning studio. Navigate to your workspace and on the left under the **Assets** pane, select **Endpoints**.
2. Select the deployment you want to delete and select **Delete**.
3. Select **Proceed**.

Delete the resource group

> **IMPORTANT**
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

Next steps

In this tutorial, you used automated ML in the Azure Machine Learning studio to create and deploy a time series forecasting model that predicts bike share rental demand.

See this article for steps on how to create a Power BI supported schema to facilitate consumption of your newly deployed web service:

Consume a web service

> **NOTE**
>
> This bike share dataset has been modified for this tutorial. This dataset was made available as part of a Kaggle competition and was originally available via Capital Bikeshare. It can also be found within the UCI Machine Learning Database.
>
> Source: Fanaee-T, Hadi, and Gama, Joao, Event labeling combining ensemble detectors and background knowledge, Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg.

Tutorial: Create a labeling project for multi-class image classification

4/7/2020 • 9 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This tutorial shows you how to manage the process of labeling (also referred to as tagging) images to be used as data for building machine learning models. Data labeling in Azure Machine Learning is in public preview.

If you want to train a machine learning model to classify images, you need hundreds or even thousands of images that are correctly labeled. Azure Machine Learning helps you manage the progress of your private team of domain experts as they label your data.

In this tutorial, you'll use images of cats and dogs. Since each image is either a cat or a dog, this is a *multi-class* labeling project. You'll learn how to:

- Create an Azure storage account and upload images to the account.
- Create an Azure Machine Learning workspace.
- Create a multi-class image labeling project.
- Label your data. Either you or your labelers can perform this task.
- Complete the project by reviewing and exporting the data.

Prerequisites

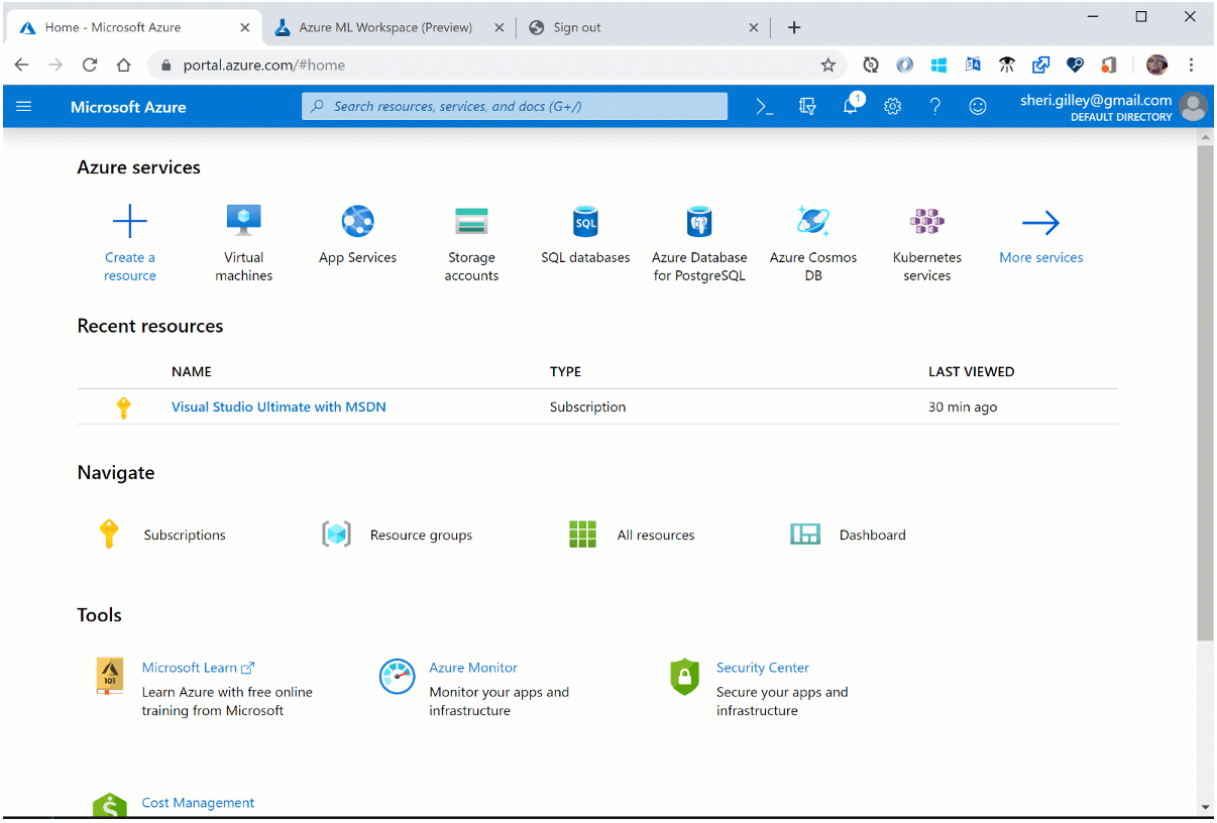
- An Azure subscription. If you don't have an Azure subscription, create a free account.

Create a workspace

An Azure Machine Learning workspace is a foundational resource in the cloud that you use to experiment, train, and deploy machine learning models. It ties your Azure subscription and resource group to an easily consumed object in the service.

You create a workspace via the Azure portal, a web-based console for managing your Azure resources.

1. Sign in to Azure portal by using the credentials for your Azure subscription.
2. In the upper-left corner of Azure portal, select **+ Create a resource**.



3. Use the search bar to find **Machine Learning**.
4. Select **Machine Learning**.
5. In the **Machine Learning** pane, select **Create** to begin.
6. Provide the following information to configure your new workspace:

FIELD	DESCRIPTION
Workspace name	Enter a unique name that identifies your workspace. In this example, we use **docs-ws**. Names must be unique across the resource group. Use a name that's easy to recall and to differentiate from workspaces created by others.
Subscription	Select the Azure subscription that you want to use.
Resource group	Use an existing resource group in your subscription or enter a name to create a new resource group. A resource group holds related resources for an Azure solution. In this example, we use **docs-aml**.
Location	Select the location closest to your users and the data resources to create your workspace.
Workspace edition	Select **Basic** as the workspace type for this tutorial. The workspace type (Basic & Enterprise) determines the features to which you'll have access and pricing. Everything in this tutorial can be performed with either a Basic or Enterprise workspace.

7. After you are finished configuring the workspace, select **Review + Create**.

> **WARNING**
> It can take several minutes to create your workspace in the cloud.

When the process is finished, a deployment success message appears.

8. To view the new workspace, select **Go to resource**.

Start a labeling project

Next you will manage the data labeling project in Azure Machine Learning studio, a consolidated interface that includes machine learning tools to perform data science scenarios for data science practitioners of all skill levels. The studio is not supported on Internet Explorer browsers.

1. Sign in to Azure Machine Learning studio.
2. Select your subscription and the workspace you created.

Create a datastore

Azure Machine Learning datastores are used to store connection information, like your subscription ID and token authorization. Here you use a datastore to connect to the storage account that contains the images for this tutorial.

1. On the left side of your workspace, select **Datastores**.
2. Select **+ New datastore**.
3. Fill out the form with these settings:

FIELD	DESCRIPTION
Datastore name	Give the datastore a name. Here we use **labeling_tutorial**.
Datastore type	Select the type of storage. Here we use **Azure Blob Storage**, the preferred storage for images.
Account selection method	Select **Enter manually**.
URL	`https://azureopendatastorage.blob.core.windows.net/openimagescontaine`
Authentication type	Select **SAS token**.
Account key	`?sv=2019-02-02&ss=bfqt&srt=sco&sp=rl&se=2025-03-25T04:51:17Z&st=2020-24T20:51:17Z&spr=https&sig=7D7SdkQidGT6pURQ9R4SUzWGxZ%2BHlNPCstoSRRVg`

4. Select **Create** to create the datastore.

Add labelers to workspace

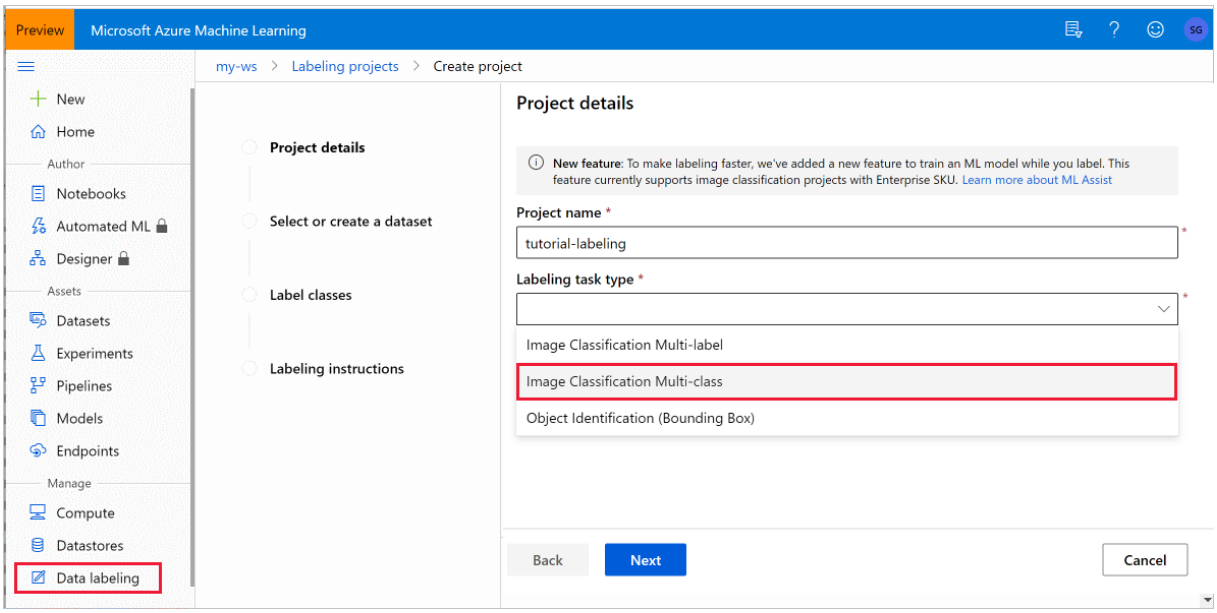
Set up your workspace to include all the people who will label data for any of your projects. Later you'll add these labelers to your specific labeling project.

1. On the left side, select **Data labeling**.
2. At the top of the page, select **Labelers**.
3. Select **Add labeler** to add the email address of a labeler.
4. Continue to add more labelers until you're done.

Create a labeling project

Now that you have your list of labelers and access to the data you want to have labeled, create your labeling project.

1. At the top of the page, select **Projects**.
2. Select **+ Add project**.



Project details

1. Use the following input for the **Project details** form:

FIELD	DESCRIPTION
Project name	Give your project a name. Here we'll use **tutorial-cats-n-dogs**.
Labeling task type	Select **Image Classification Multi-class**.

   Select **Next** to continue creating the project.

Select or create a dataset

1. On the **Select or create a dataset** form, select the second choice, **Create a dataset**, then select the link **From datastore**.
2. Use the following input for the **Create dataset from datastore** form:
   a. On the **Basic info** form, add a name, here we'll use **images-for-tutorial**. Add a description if you wish. Then select **Next**.
   b. On the **Datastore selection** form, use the dropdown to select your **Previously created datastore**, for example **tutorial_images (Azure Blob Storage)**
   c. Next, still on the **Datastore selection** form, select **Browse** and then select **MultiClass - DogsCats**. Select **Save** to use **/MultiClass - DogsCats** as the path.
   d. Select **Next** to confirm details and then **Create** to create the dataset.
   e. Select the circle next to the dataset name in the list, for example **images-for-tutorial**.
3. Select **Next** to continue creating the project.

Label classes

1. On the **Label classes** form, type a label name, then select **+Add label** to type the next label. For this project, the labels are **Cat**, **Dog**, and **Uncertain**.
2. Select **Next** when have added all the labels.

Labeling instructions

1. On the **Labeling instructions** form, you can provide a link to a website that provides detailed instructions for your labelers. We'll leave it blank for this tutorial.
2. You can also add a short description of the task directly on the form. Type **Labeling tutorial - Cats & Dogs.**
3. Select **Next**.
4. On the **ML assisted labeling** form, leave the checkbox unchecked. ML assisted labeling requires more data than you'll be using in this tutorial.

5. Select **Create project**.

This page doesn't automatically refresh. After a pause, manually refresh the page until the project's status changes to **Created**.

Add labelers to your project

Add some or all of your labelers to this project.

1. Select the project name to open the project.
2. At the top of the page, select **Teams**.
3. Select the **labeling_tutorial Default Team** link.
4. Now use **Assign labelers** to add the labelers you want to participate in this project.
5. Select from the list of labelers you created earlier. Once you've selected all the labelers you wish to use, select **Assign labelers** to add them to your default project team.

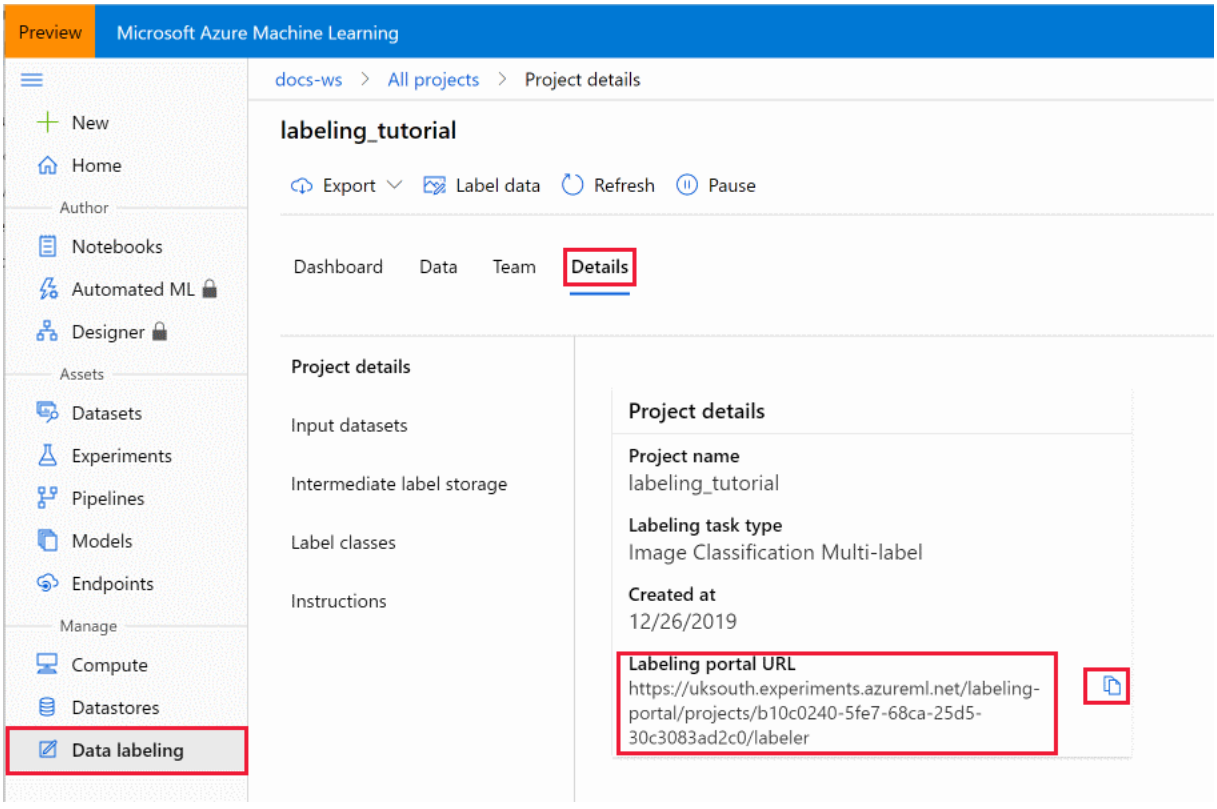
Start labeling

You have now set up your Azure resources, and configured a data labeling project. It's time to add labels to your data.

Notify labelers

If you have lots of images to label, hopefully you also have lots of labelers to complete the task. You'll now want to send them instructions so they can access the data and start labeling.

1. In Machine Learning studio, select **Data labeling** on the left-hand side to find your project.
2. Select the project name link.
3. At the top of the page, select **Details**. You see a summary of your project.



4. Copy the **Labeling portal URL** link to send to your labelers.
5. Now select **Team** at the top to find your labeling team.
6. Select the team name link.
7. At the top of the page, select **Email team** to start your email. Paste in the labeling portal URL you just copied.

Each time a labeler goes to the portal URL, they'll be presented with more images to label, until the queue is empty.

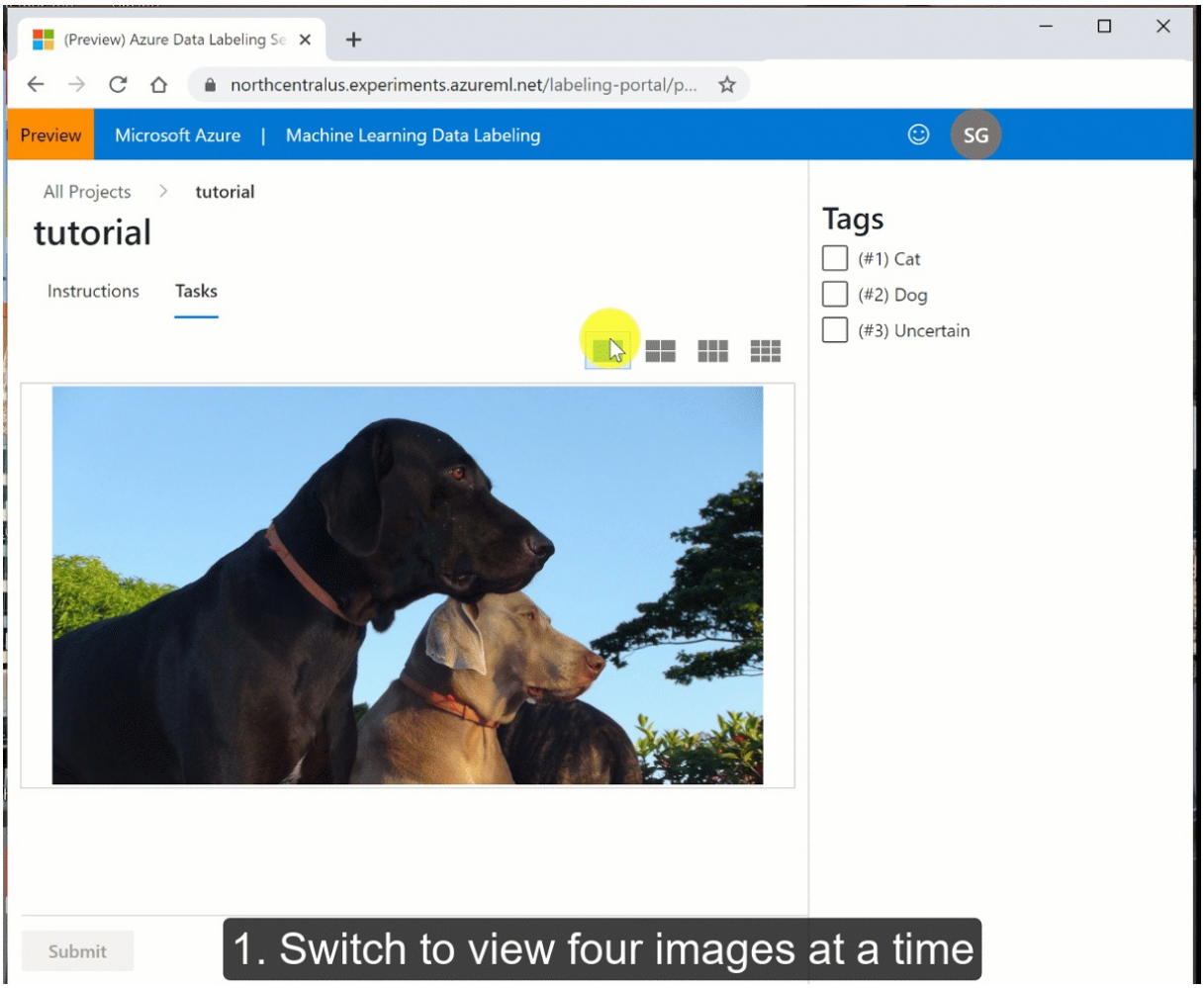
Tag the images

In this part of the tutorial, you'll switch roles from the *project administrator* to that of a *labeler*. Use the URL you sent to the team. This URL brings you to the labeling portal for your project. If you had added instructions, you'd see them here when you arrive on the page.

1. At the top of the page, select **Tasks** to start labeling.
2. Select a thumbnail image on the right to display the number of images you wish to label in one go. You must label all these images before you can move on. Only switch layouts when you have a fresh page of unlabeled data. Switching layouts clears the page's in-progress tagging work.
3. Select one or more images, then select a tag to apply to the selection. The tag appears below the image. Continue to select and tag all images on the page. To select all the displayed images simultaneously, select **Select all**. Select at least one image to apply a tag.

   > **TIP**
   > You can select the first nine tags by using the number keys on your keyboard.

4. Once all the images on the page are tagged, select **Submit** to submit these labels.



5. After you submit tags for the data at hand, Azure refreshes the page with a new set of images from the work queue.

Complete the project

Now you'll switch roles back to the *project administrator* for the labeling project.

As a manager, you may want to review the work of your labeler.

Review labeled data

1. In Machine Learning studio, select **Data labeling** on the left-hand side to find your project.

2. Select the project name link.
3. The Dashboard shows you the progress of your project.
4. At the top of the page, select **Data**.
5. On the left side, select **Labeled data** to see your tagged images.
6. When you disagree with a label, select the image and then select **Reject** at the bottom of the page. The tags will be removed and the image is put back in the queue of unlabeled images.

Export labeled data

You can export the label data for Machine Learning experimentation at any time. Users often export multiple times and train different models, rather than wait for all the images to be labeled.

Image labels can be exported in COCO format or as an Azure Machine Learning dataset. The dataset format makes it easy to use for training in Azure Machine Learning.

1. In Machine Learning studio, select **Data labeling** on the left-hand side to find your project.
2. Select the project name link.
3. Select **Export** and choose **Export as Azure ML Dataset**.

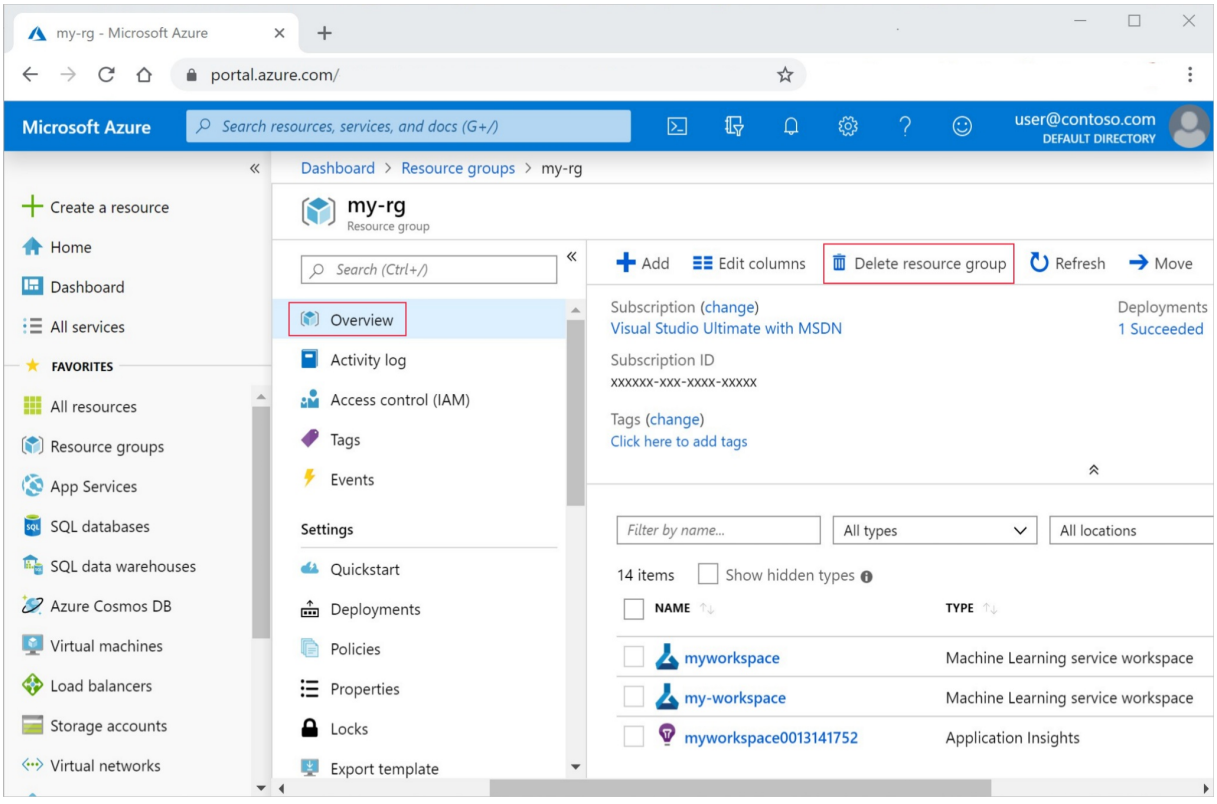
   The status of the export appears just below the **Export** button.
4. Once the labels are successfully exported, select **Datasets** on the left side to view the results.

Clean up resources

> **IMPORTANT**
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.

4. Enter the resource group name. Then select **Delete**.

Next steps

In this tutorial, you labeled images. Now use your labeled data:

Train a machine learning image recognition model.

Tutorial: Get started creating your first ML experiment with the Python SDK

4/13/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this tutorial, you complete the end-to-end steps to get started with the Azure Machine Learning Python SDK running in Jupyter notebooks. This tutorial is **part one of a two-part tutorial series**, and covers Python environment setup and configuration, as well as creating a workspace to manage your experiments and machine learning models. **Part two** builds on this to train multiple machine learning models and introduce the model management process using both Azure Machine Learning studio and the SDK.

In this tutorial, you:

- Create an Azure Machine Learning Workspace to use in the next tutorial.
- Clone the tutorials notebook to your folder in the workspace.
- Create a cloud-based compute instance with Azure Machine Learning Python SDK installed and pre-configured.

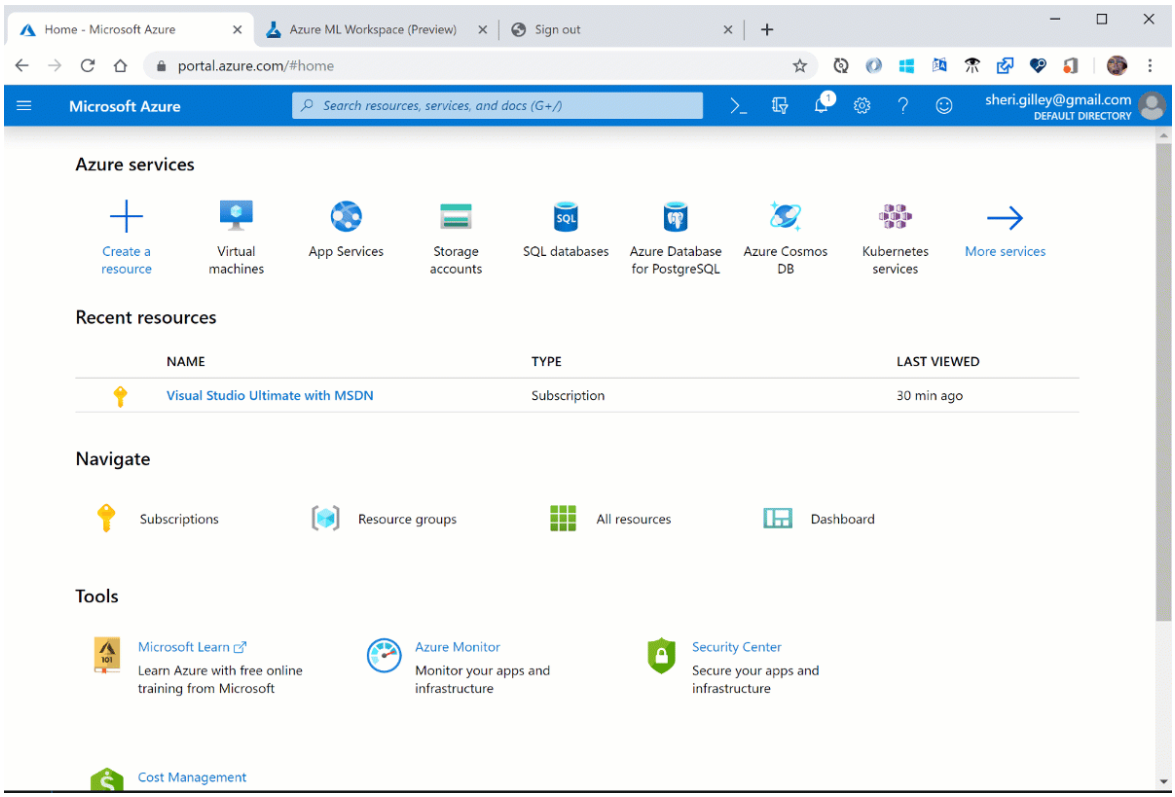
If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.

Create a workspace

An Azure Machine Learning workspace is a foundational resource in the cloud that you use to experiment, train, and deploy machine learning models. It ties your Azure subscription and resource group to an easily consumed object in the service.

You create a workspace via the Azure portal, a web-based console for managing your Azure resources.

1. Sign in to Azure portal by using the credentials for your Azure subscription.
2. In the upper-left corner of Azure portal, select **+ Create a resource**.



3. Use the search bar to find **Machine Learning**.
4. Select **Machine Learning**.
5. In the **Machine Learning** pane, select **Create** to begin.
6. Provide the following information to configure your new workspace:

FIELD	DESCRIPTION
Workspace name	Enter a unique name that identifies your workspace. In this example, we use **docs-ws**. Names must be unique across the resource group. Use a name that's easy to recall and to differentiate from workspaces created by others.
Subscription	Select the Azure subscription that you want to use.
Resource group	Use an existing resource group in your subscription or enter a name to create a new resource group. A resource group holds related resources for an Azure solution. In this example, we use **docs-aml**.
Location	Select the location closest to your users and the data resources to create your workspace.
Workspace edition	Select **Basic** as the workspace type for this tutorial. The workspace type (Basic & Enterprise) determines the features to which you'll have access and pricing. Everything in this tutorial can be performed with either a Basic or Enterprise workspace.

7. After you are finished configuring the workspace, select **Review + Create**.

> **WARNING**
> It can take several minutes to create your workspace in the cloud.

When the process is finished, a deployment success message appears.

8. To view the new workspace, select **Go to resource**.

> **IMPORTANT**
> Take note of your **workspace** and **subscription**. You'll need these to ensure you create your experiment in the right place.

Run notebook in your workspace

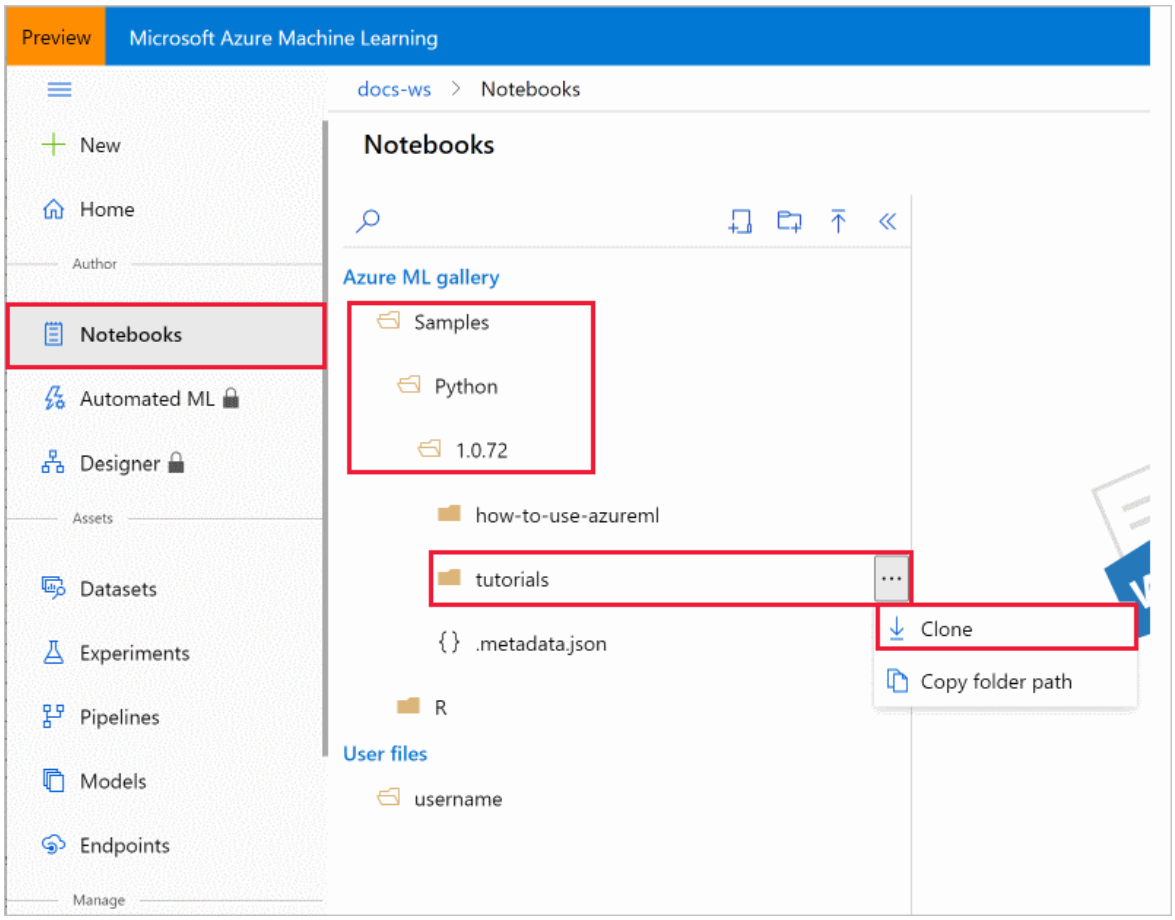
This tutorial uses the cloud notebook server in your workspace for an install-free and pre-configured experience. Use your own environment if you prefer to have control over your environment, packages and dependencies.

Follow along with this video or use the detailed steps below to clone and run the tutorial from your workspace.

Clone a notebook folder

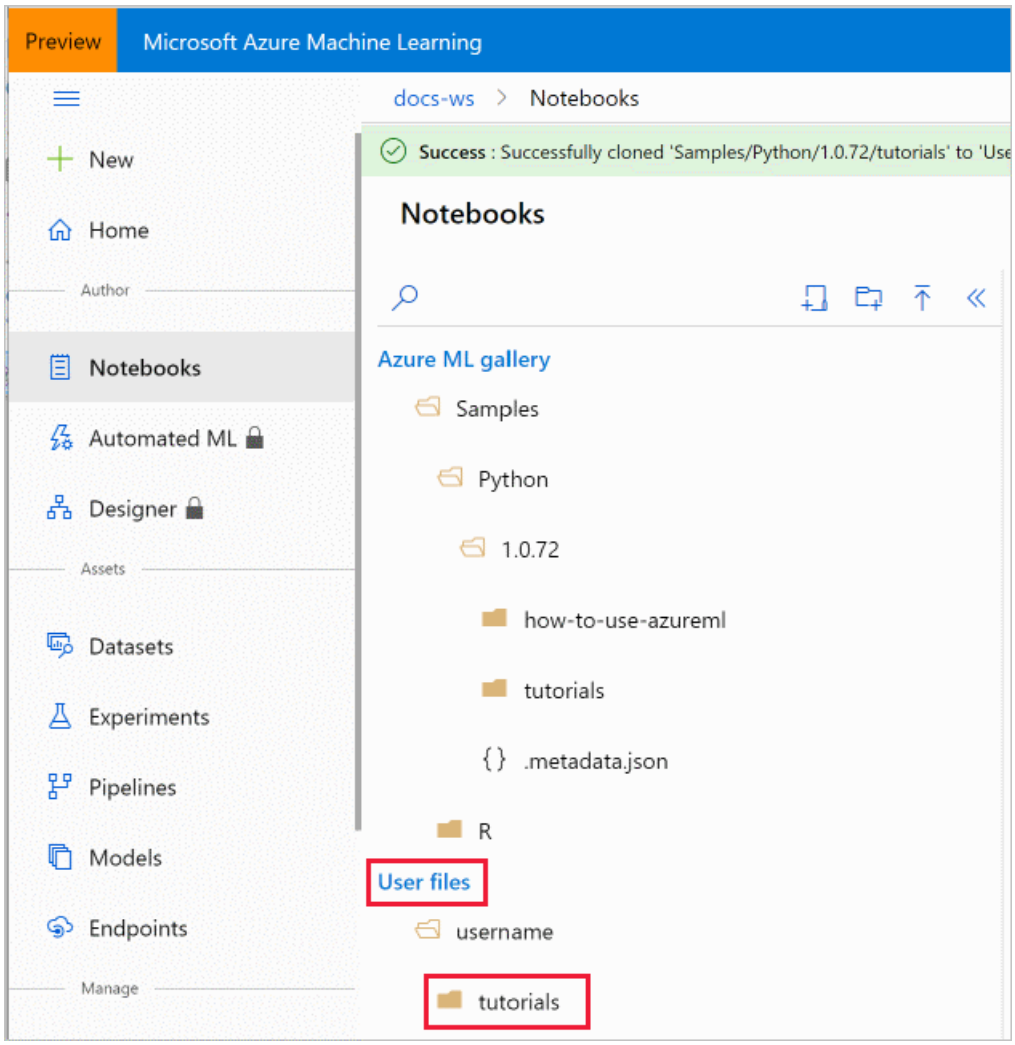
You complete the following experiment set-up and run steps in Azure Machine Learning studio, a consolidated interface that includes machine learning tools to perform data science scenarios for data science practitioners of all skill levels.

1. Sign in to Azure Machine Learning studio.
2. Select your subscription and the workspace you created.
3. Select **Notebooks** on the left.
4. Open the **Samples** folder.
5. Open the **Python** folder.
6. Open the folder with a version number on it. This number represents the current release for the Python SDK.
7. Select the **"..."** at the right of the **tutorials** folder and then select **Clone**.



8. A list of folders displays showing each user who accesses the workspace. Select your folder to clone the **tutorials** folder there.

1. Under **User Files** open your folder and then open the cloned **tutorials** folder.



IMPORTANT

You can view notebooks in the **samples** folder but you cannot run a notebook from there. In order to run a notebook, make sure you open the cloned version of the notebook in the **User Files** section.

2. Select the **tutorial-1st-experiment-sdk-train.ipynb** file in your **tutorials/create-first-ml-experiment** folder.
3. On the top bar, select a compute instance to use to run the notebook. These VMs are pre-configured with everything you need to run Azure Machine Learning.
4. If no VMs are found, select **+ Add** to create the compute instance VM.
    a. When you create a VM, provide a name. The name must be between 2 to 16 characters. Valid characters are letters, digits, and the - character, and must also be unique across your Azure subscription.
    b. Select the Virtual Machine size from the available choices.
    c. Then select **Create**. It can take approximately 5 minutes to set up your VM.
5. Once the VM is available it will be displayed in the top toolbar. You can now run the notebook either by using **Run all** in the toolbar, or by using **Shift+Enter** in the code cells of the notebook.

If you have custom widgets or prefer using Jupyter/JupyterLab select the **Jupyter** drop down on the far right, then select **Jupyter** or **JupyterLab**. The new browser window will be opened.

Next steps

In this tutorial, you completed these tasks:

- Created an Azure Machine Learning workspace.
- Created and configured a cloud notebook server in your workspace.

In **part two** of the tutorial you run the code in `tutorial-1st-experiment-sdk-train.ipynb` to train a machine learning model.

Tutorial: Train your first model

> **IMPORTANT**
>
> If you do not plan on following part 2 of this tutorial or any other tutorials, you should stop the cloud notebook server VM when you are not using it to reduce cost.

Tutorial: Train your first ML model

4/1/2020 • 7 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This tutorial is **part two of a two-part tutorial series**. In the previous tutorial, you created a workspace and chose a development environment. In this tutorial, you learn the foundational design patterns in Azure Machine Learning, and train a simple scikit-learn model based on the diabetes data set. After completing this tutorial, you will have the practical knowledge of the SDK to scale up to developing more-complex experiments and workflows.

In this tutorial, you learn the following tasks:

- Connect your workspace and create an experiment
- Load data and train scikit-learn models
- View training results in the studio
- Retrieve the best model

Prerequisites

The only prerequisite is to run part one of this tutorial, Setup environment and workspace.

In this part of the tutorial, you run the code in the sample Jupyter notebook *tutorials/create-first-ml-experiment/tutorial-1st-experiment-sdk-train.ipynb* opened at the end of part one. This article walks through the same code that is in the notebook.

Open the notebook

1. Sign in to Azure Machine Learning studio.
2. Open the **tutorial-1st-experiment-sdk-train.ipynb** in your folder as shown in part one.

> **WARNING**
> Do **not** create a *new* notebook in the Jupyter interface! The notebook *tutorials/create-first-ml-experiment/tutorial-1st-experiment-sdk-train.ipynb* is inclusive of **all code and data needed** for this tutorial.

Connect workspace and create experiment

> **IMPORTANT**
> The rest of this article contains the same content as you see in the notebook.
>
> Switch to the Jupyter notebook now if you want to read along as you run the code. To run a single code cell in a notebook, click the code cell and hit **Shift+Enter**. Or, run the entire notebook by choosing **Run all** from the top toolbar.

Import the `Workspace` class, and load your subscription information from the file `config.json` using the function `from_config()`. This looks for the JSON file in the current directory by default, but you can also specify a path parameter to point to the file using `from_config(path="your/file/path")`. In a cloud notebook server, the file is automatically in the root directory.

If the following code asks for additional authentication, simply paste the link in a browser and enter the

authentication token.

```
from azureml.core import Workspace
ws = Workspace.from_config()
```

Now create an experiment in your workspace. An experiment is another foundational cloud resource that represents a collection of trials (individual model runs). In this tutorial you use the experiment to create runs and track your model training in the Azure Machine Learning studio. Parameters include your workspace reference, and a string name for the experiment.

```
from azureml.core import Experiment
experiment = Experiment(workspace=ws, name="diabetes-experiment")
```

Load data and prepare for training

For this tutorial, you use the diabetes data set, which uses features like age, gender, and BMI to predict diabetes disease progression. Load the data from the Azure Open Datasets class, and split it into training and test sets using `train_test_split()`. This function segregates the data so the model has unseen data to use for testing following training.

```
from azureml.opendatasets import Diabetes
from sklearn.model_selection import train_test_split

x_df = Diabetes.get_tabular_dataset().to_pandas_dataframe().dropna()
y_df = x_df.pop("Y")

X_train, X_test, y_train, y_test = train_test_split(x_df, y_df, test_size=0.2, random_state=66)
```

Train a model

Training a simple scikit-learn model can easily be done locally for small-scale training, but when training many iterations with dozens of different feature permutations and hyperparameter settings, it is easy to lose track of what models you've trained and how you trained them. The following design pattern shows how to leverage the SDK to easily keep track of your training in the cloud.

Build a script that trains ridge models in a loop through different hyperparameter alpha values.

```
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.externals import joblib
import math

alphas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

for alpha in alphas:
    run = experiment.start_logging()
    run.log("alpha_value", alpha)

    model = Ridge(alpha=alpha)
    model.fit(X=X_train, y=y_train)
    y_pred = model.predict(X=X_test)
    rmse = math.sqrt(mean_squared_error(y_true=y_test, y_pred=y_pred))
    run.log("rmse", rmse)

    model_name = "model_alpha_" + str(alpha) + ".pkl"
    filename = "outputs/" + model_name

    joblib.dump(value=model, filename=filename)
    run.upload_file(name=model_name, path_or_stream=filename)
    run.complete()
```

The above code accomplishes the following:

1. For each alpha hyperparameter value in the `alphas` array, a new run is created within the experiment. The alpha value is logged to differentiate between each run.
2. In each run, a Ridge model is instantiated, trained, and used to run predictions. The root-mean-squared-error is calculated for the actual versus predicted values, and then logged to the run. At this point the run has metadata attached for both the alpha value and the rmse accuracy.
3. Next, the model for each run is serialized and uploaded to the run. This allows you to download the model file from the run in the studio.
4. At the end of each iteration the run is completed by calling `run.complete()`.

After the training has completed, call the `experiment` variable to fetch a link to the experiment in the studio.

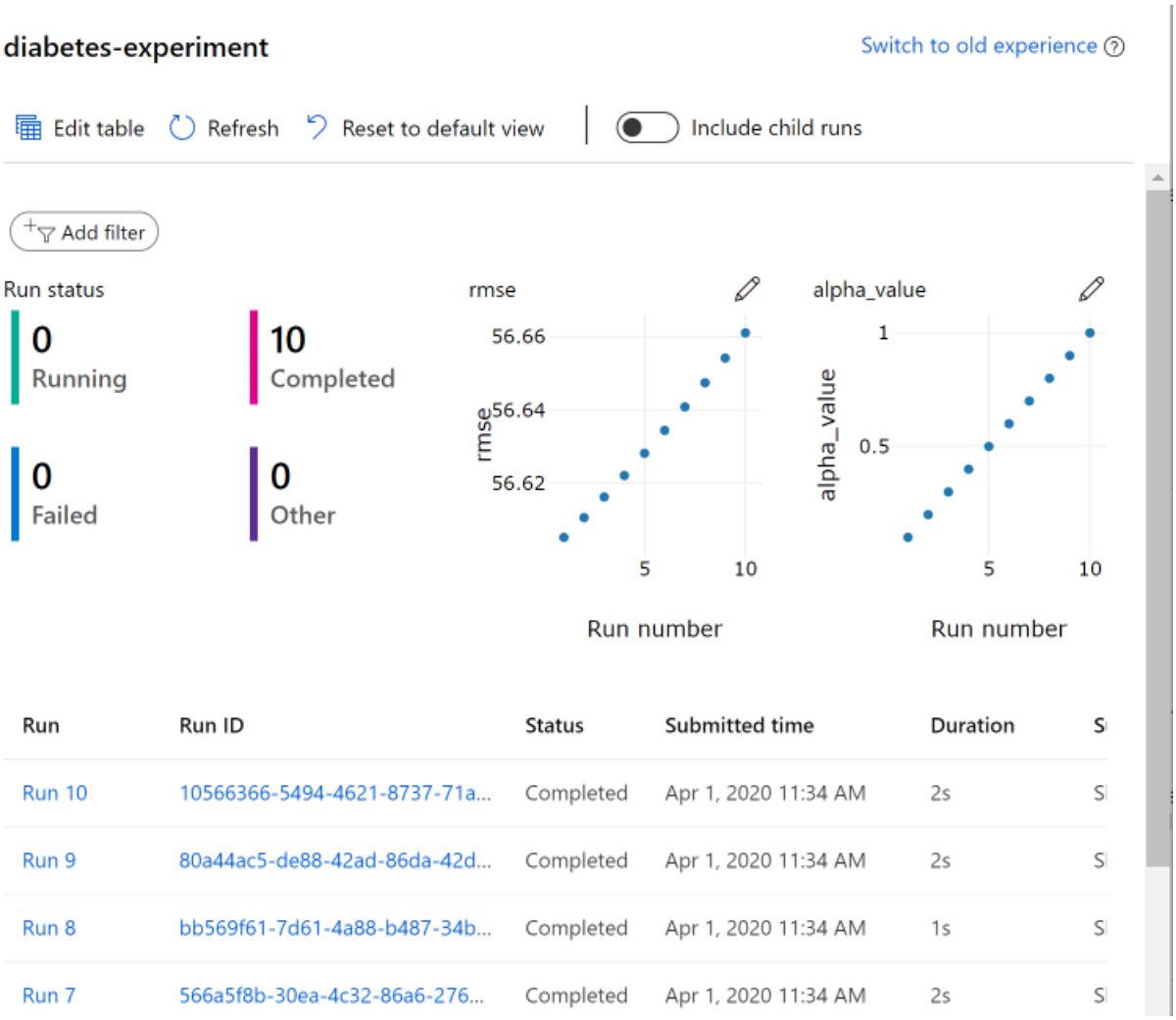
```
experiment
```

NAME	WORKSPACE	REPORT PAGE	DOCS PAGE
diabetes-experiment	your-workspace-name	Link to Azure Machine Learning studio	Link to Documentation

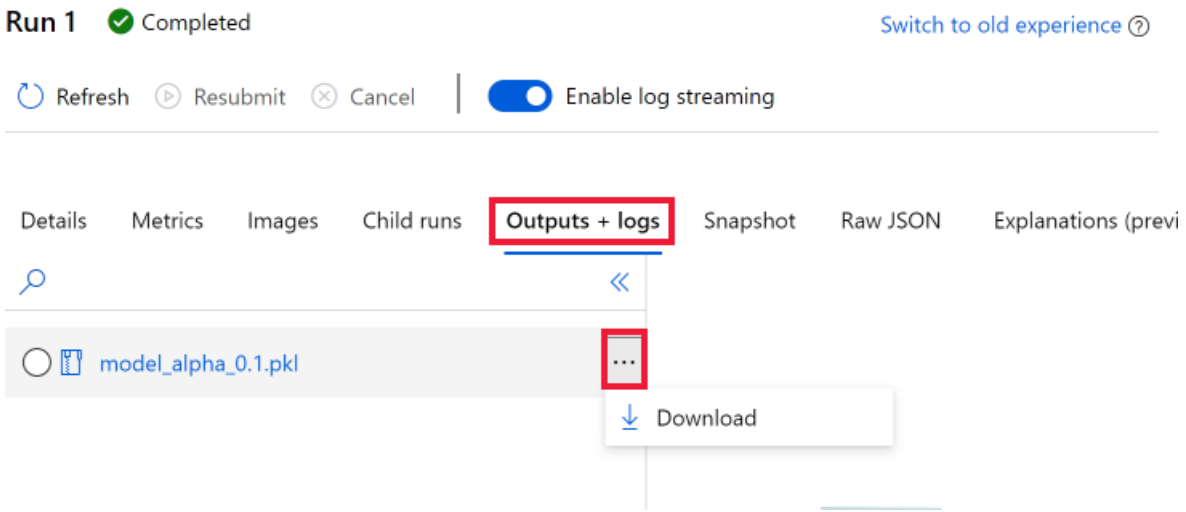
View training results in studio

Following the **Link to Azure Machine Learning studio** takes you to the main experiment page. Here you see all the individual runs in the experiment. Any custom-logged values (`alpha_value` and `rmse`, in this case) become fields for each run, and also become available for the charts and tiles at the top of the experiment page. To add a logged metric to a chart or tile, hover over it, click the edit button, and find your custom-logged metric.

When training models at scale over hundreds and thousands of separate runs, this page makes it easy to see every model you trained, specifically how they were trained, and how your unique metrics have changed over time.



Select a run number link in the `RUN NUMBER` column to see the page for an individual run. The default tab **Details** shows you more-detailed information on each run. Navigate to the **Outputs + logs** tab, and you see the `.pkl` file for the model that was uploaded to the run during each training iteration. Here you can download the model file, rather than having to retrain it manually.



Get the best model

In addition to being able to download model files from the experiment in the studio, you can also download them programmatically. The following code iterates through each run in the experiment, and accesses both the logged run metrics and the run details (which contains the run_id). This keeps track of the best run, in this case the run with the lowest root-mean-squared-error.

```
minimum_rmse_runid = None
minimum_rmse = None

for run in experiment.get_runs():
    run_metrics = run.get_metrics()
    run_details = run.get_details()
    # each logged metric becomes a key in this returned dict
    run_rmse = run_metrics["rmse"]
    run_id = run_details["runId"]

    if minimum_rmse is None:
        minimum_rmse = run_rmse
        minimum_rmse_runid = run_id
    else:
        if run_rmse < minimum_rmse:
            minimum_rmse = run_rmse
            minimum_rmse_runid = run_id

print("Best run_id: " + minimum_rmse_runid)
print("Best run_id rmse: " + str(minimum_rmse))
```

```
Best run_id: 864f5ce7-6729-405d-b457-83250da99c80
Best run_id rmse: 57.234760283951765
```

Use the best run ID to fetch the individual run using the `Run` constructor along with the experiment object. Then call `get_file_names()` to see all the files available for download from this run. In this case, you only uploaded one file for each run during training.

```
from azureml.core import Run
best_run = Run(experiment=experiment, run_id=minimum_rmse_runid)
print(best_run.get_file_names())
```

```
['model_alpha_0.1.pkl']
```

Call `download()` on the run object, specifying the model file name to download. By default this function downloads to the current directory.

```
best_run.download_file(name="model_alpha_0.1.pkl")
```

Clean up resources

Do not complete this section if you plan on running other Azure Machine Learning tutorials.

Stop the compute instance

If you used a compute instance or Notebook VM, stop the VM when you are not using it to reduce cost.

1. In your workspace, select **Compute**.
2. From the list, select the VM.
3. Select **Stop**.
4. When you're ready to use the server again, select **Start**.

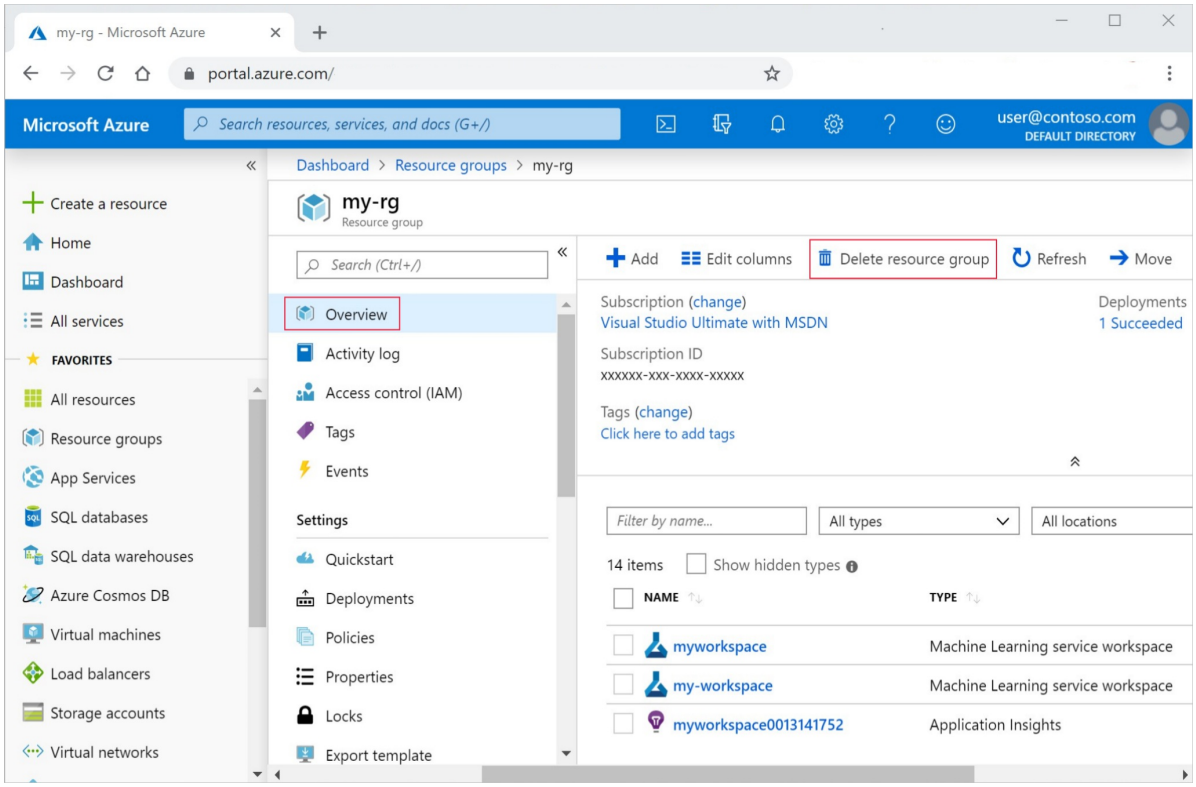
Delete everything

IMPORTANT

The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties and select **Delete**.

Next steps

In this tutorial, you did the following tasks:

- Connected your workspace and created an experiment
- Loaded data and trained scikit-learn models
- Viewed training results in the studio and retrieved models

Deploy your model with Azure Machine Learning. Learn how to develop automated machine learning experiments.

Tutorial: Train image classification models with MNIST data and scikit-learn

4/24/2020 • 13 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this tutorial, you train a machine learning model on remote compute resources. You'll use the training and deployment workflow for Azure Machine Learning in a Python Jupyter notebook. You can then use the notebook as a template to train your own machine learning model with your own data. This tutorial is **part one of a two-part tutorial series**.

This tutorial trains a simple logistic regression by using the MNIST dataset and scikit-learn with Azure Machine Learning. MNIST is a popular dataset consisting of 70,000 grayscale images. Each image is a handwritten digit of 28 x 28 pixels, representing a number from zero to nine. The goal is to create a multi-class classifier to identify the digit a given image represents.

Learn how to take the following actions:

- Set up your development environment.
- Access and examine the data.
- Train a simple logistic regression model on a remote cluster.
- Review training results and register the best model.

You learn how to select a model and deploy it in part two of this tutorial.

If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.

> **NOTE**
> Code in this article was tested with Azure Machine Learning SDK version 1.0.83.

Prerequisites

- Complete the Tutorial: Get started creating your first Azure ML experiment to:
  - Create a workspace
  - Clone the tutorials notebook to your folder in the workspace.
  - Create a cloud-based compute instance.
- In your cloned *tutorials/image-classification-mnist-data* folder, open the *img-classification-part1-training.ipynb* notebook.

The tutorial and accompanying **utils.py** file is also available on GitHub if you wish to use it on your own local environment. Run `pip install azureml-sdk[notebooks] azureml-opendatasets matplotlib` to install dependencies for this tutorial.

> **IMPORTANT**
>
> The rest of this article contains the same content as you see in the notebook.
>
> Switch to the Jupyter notebook now if you want to read along as you run the code. To run a single code cell in a notebook, click the code cell and hit **Shift+Enter**. Or, run the entire notebook by choosing **Run all** from the top toolbar.

Set up your development environment

All the setup for your development work can be accomplished in a Python notebook. Setup includes the following actions:

- Import Python packages.
- Connect to a workspace, so that your local computer can communicate with remote resources.
- Create an experiment to track all your runs.
- Create a remote compute target to use for training.

Import packages

Import Python packages you need in this session. Also display the Azure Machine Learning SDK version:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

import azureml.core
from azureml.core import Workspace

# check core SDK version number
print("Azure ML SDK Version: ", azureml.core.VERSION)
```

Connect to a workspace

Create a workspace object from the existing workspace. `Workspace.from_config()` reads the file **config.json** and loads the details into an object named `ws`:

```
# load workspace configuration from the config.json file in the current folder.
ws = Workspace.from_config()
print(ws.name, ws.location, ws.resource_group, sep='\t')
```

Create an experiment

Create an experiment to track the runs in your workspace. A workspace can have multiple experiments:

```
from azureml.core import Experiment
experiment_name = 'sklearn-mnist'

exp = Experiment(workspace=ws, name=experiment_name)
```

Create or attach an existing compute target

By using Azure Machine Learning Compute, a managed service, data scientists can train machine learning models on clusters of Azure virtual machines. Examples include VMs with GPU support. In this tutorial, you create Azure Machine Learning Compute as your training environment. You will submit Python code to run on this VM later in the tutorial.

The code below creates the compute clusters for you if they don't already exist in your workspace.

Creation of the compute target takes about five minutes. If the compute resource is already in the workspace, the code uses it and skips the creation process.

```
from azureml.core.compute import AmlCompute
from azureml.core.compute import ComputeTarget
import os

# choose a name for your cluster
compute_name = os.environ.get("AML_COMPUTE_CLUSTER_NAME", "cpucluster")
compute_min_nodes = os.environ.get("AML_COMPUTE_CLUSTER_MIN_NODES", 0)
compute_max_nodes = os.environ.get("AML_COMPUTE_CLUSTER_MAX_NODES", 4)

# This example uses CPU VM. For using GPU VM, set SKU to STANDARD_NC6
vm_size = os.environ.get("AML_COMPUTE_CLUSTER_SKU", "STANDARD_D2_V2")


if compute_name in ws.compute_targets:
    compute_target = ws.compute_targets[compute_name]
    if compute_target and type(compute_target) is AmlCompute:
        print('found compute target. just use it. ' + compute_name)
else:
    print('creating a new compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size=vm_size,
                                                                 min_nodes=compute_min_nodes,
                                                                 max_nodes=compute_max_nodes)

    # create the cluster
    compute_target = ComputeTarget.create(
        ws, compute_name, provisioning_config)

    # can poll for a minimum number of nodes and for a specific timeout.
    # if no min node count is provided it will use the scale settings for the cluster
    compute_target.wait_for_completion(
        show_output=True, min_node_count=None, timeout_in_minutes=20)

    # For a more detailed view of current AmlCompute status, use get_status()
    print(compute_target.get_status().serialize())
```

You now have the necessary packages and compute resources to train a model in the cloud.

Explore data

Before you train a model, you need to understand the data that you use to train it. In this section you learn how to:

- Download the MNIST dataset.
- Display some sample images.

Download the MNIST dataset

Use Azure Open Datasets to get the raw MNIST data files. Azure Open Datasets are curated public datasets that you can use to add scenario-specific features to machine learning solutions for more accurate models. Each dataset has a corresponding class, `MNIST` in this case, to retrieve the data in different ways.

This code retrieves the data as a `FileDataset` object, which is a subclass of `Dataset`. A `FileDataset` references single or multiple files of any format in your datastores or public urls. The class provides you with the ability to download or mount the files to your compute by creating a reference to the data source location. Additionally, you register the Dataset to your workspace for easy retrieval during training.

Follow the how-to to learn more about Datasets and their usage in the SDK.

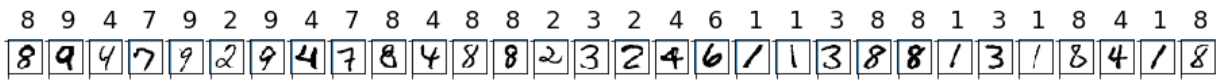
```
from azureml.core import Dataset
from azureml.opendatasets import MNIST

data_folder = os.path.join(os.getcwd(), 'data')
os.makedirs(data_folder, exist_ok=True)

mnist_file_dataset = MNIST.get_file_dataset()
mnist_file_dataset.download(data_folder, overwrite=True)

mnist_file_dataset = mnist_file_dataset.register(workspace=ws,
                                                 name='mnist_opendataset',
                                                 description='training and test dataset',
                                                 create_new_version=True)
```

Display some sample images

Load the compressed files into `numpy` arrays. Then use `matplotlib` to plot 30 random images from the dataset with their labels above them. This step requires a `load_data` function that's included in an `util.py` file. This file is included in the sample folder. Make sure it's placed in the same folder as this notebook. The `load_data` function simply parses the compressed files into numpy arrays.

```
# make sure utils.py is in the same directory as this code
from utils import load_data
import glob


# note we also shrink the intensity values (X) from 0-255 to 0-1. This helps the model converge faster.
X_train = load_data(glob.glob(os.path.join(data_folder,"**/train-images-idx3-ubyte.gz"), recursive=True)
[0], False) / 255.0
X_test = load_data(glob.glob(os.path.join(data_folder,"**/t10k-images-idx3-ubyte.gz"), recursive=True)[0],
False) / 255.0
y_train = load_data(glob.glob(os.path.join(data_folder,"**/train-labels-idx1-ubyte.gz"), recursive=True)
[0], True).reshape(-1)
y_test = load_data(glob.glob(os.path.join(data_folder,"**/t10k-labels-idx1-ubyte.gz"), recursive=True)[0],
True).reshape(-1)


# now let's show some randomly chosen images from the traininng set.
count = 0
sample_size = 30
plt.figure(figsize=(16, 6))
for i in np.random.permutation(X_train.shape[0])[:sample_size]:
    count = count + 1
    plt.subplot(1, sample_size, count)
    plt.axhline('')
    plt.axvline('')
    plt.text(x=10, y=-10, s=y_train[i], fontsize=18)
    plt.imshow(X_train[i].reshape(28, 28), cmap=plt.cm.Greys)
plt.show()
```

A random sample of images displays:



Now you have an idea of what these images look like and the expected prediction outcome.

Train on a remote cluster

For this task, you submit the job to run on the remote training cluster you set up earlier. To submit a job you:

- Create a directory

- Create a training script
- Create an estimator object
- Submit the job

Create a directory

Create a directory to deliver the necessary code from your computer to the remote resource.

```
import os
script_folder = os.path.join(os.getcwd(), "sklearn-mnist")
os.makedirs(script_folder, exist_ok=True)
```

Create a training script

To submit the job to the cluster, first create a training script. Run the following code to create the training script called `train.py` in the directory you just created.

```
%%writefile $script_folder/train.py

import argparse
import os
import numpy as np
import glob

from sklearn.linear_model import LogisticRegression
import joblib

from azureml.core import Run
from utils import load_data

# let user feed in 2 parameters, the dataset to mount or download, and the regularization rate of the
logistic regression model
parser = argparse.ArgumentParser()
parser.add_argument('--data-folder', type=str, dest='data_folder', help='data folder mounting point')
parser.add_argument('--regularization', type=float, dest='reg', default=0.01, help='regularization rate')
args = parser.parse_args()

data_folder = args.data_folder
print('Data folder:', data_folder)

# load train and test set into numpy arrays
# note we scale the pixel intensity values to 0-1 (by dividing it with 255.0) so the model can converge
faster.
X_train = load_data(glob.glob(os.path.join(data_folder, '**/train-images-idx3-ubyte.gz'), recursive=True)
[0], False) / 255.0
X_test = load_data(glob.glob(os.path.join(data_folder, '**/t10k-images-idx3-ubyte.gz'), recursive=True)[0],
False) / 255.0
y_train = load_data(glob.glob(os.path.join(data_folder, '**/train-labels-idx1-ubyte.gz'), recursive=True)
[0], True).reshape(-1)
y_test = load_data(glob.glob(os.path.join(data_folder, '**/t10k-labels-idx1-ubyte.gz'), recursive=True)[0],
True).reshape(-1)

print(X_train.shape, y_train.shape, X_test.shape, y_test.shape, sep = '\n')

# get hold of the current run
run = Run.get_context()

print('Train a logistic regression model with regularization rate of', args.reg)
clf = LogisticRegression(C=1.0/args.reg, solver="liblinear", multi_class="auto", random_state=42)
clf.fit(X_train, y_train)

print('Predict the test set')
y_hat = clf.predict(X_test)

# calculate accuracy on the prediction
acc = np.average(y_hat == y_test)
print('Accuracy is', acc)

run.log('regularization rate', np.float(args.reg))
run.log('accuracy', np.float(acc))

os.makedirs('outputs', exist_ok=True)
# note file saved in the outputs folder is automatically uploaded into experiment record
joblib.dump(value=clf, filename='outputs/sklearn_mnist_model.pkl')
```

Notice how the script gets data and saves models:

- The training script reads an argument to find the directory that contains the data. When you submit the job later, you point to the datastore for this argument:
  ```
  parser.add_argument('--data-folder', type=str, dest='data_folder', help='data directory mounting
  point')
  ```
- The training script saves your model into a directory named **outputs**. Anything written in this directory

is automatically uploaded into your workspace. You access your model from this directory later in the tutorial. `joblib.dump(value=clf, filename='outputs/sklearn_mnist_model.pkl')`

- The training script requires the file `utils.py` to load the dataset correctly. The following code copies `utils.py` into `script_folder` so that the file can be accessed along with the training script on the remote resource.

```
import shutil
shutil.copy('utils.py', script_folder)
```

Create an estimator

An estimator object is used to submit the run. Azure Machine Learning has pre-configured estimators for common machine learning frameworks, as well as generic Estimator. Create an estimator by specifying

- The name of the estimator object, `est`.
- The directory that contains your scripts. All the files in this directory are uploaded into the cluster nodes for execution.
- The compute target. In this case, you use the Azure Machine Learning compute cluster you created.
- The training script name, **train.py**.
- An environment that contains the libraries needed to run the script.
- Parameters required from the training script.

In this tutorial, this target is AmlCompute. All files in the script folder are uploaded into the cluster nodes for run. The **data_folder** is set to use the dataset. "First, create the environment that contains: the scikit-learn library, azureml-dataprep required for accessing the dataset, and azureml-defaults which contains the dependencies for logging metrics. The azureml-defaults also contains the dependencies required for deploying the model as a web service later in the part 2 of the tutorial.

Once the environment is defined, register it with the Workspace to re-use it in part 2 of the tutorial.

```
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies

# to install required packages
env = Environment('tutorial-env')
cd = CondaDependencies.create(pip_packages=['azureml-dataprep[pandas,fuse]>=1.1.14', 'azureml-defaults'],
conda_packages = ['scikit-learn==0.22.1'])

env.python.conda_dependencies = cd

# Register environment to re-use later
env.register(workspace = ws)
```

Then create the estimator with the following code.

```
from azureml.train.estimator import Estimator

script_params = {
    # to mount files referenced by mnist dataset
    '--data-folder': mnist_file_dataset.as_named_input('mnist_opendataset').as_mount(),
    '--regularization': 0.5
}

est = Estimator(source_directory=script_folder,
              script_params=script_params,
              compute_target=compute_target,
              environment_definition=env,
              entry_script='train.py')
```

Submit the job to the cluster

Run the experiment by submitting the estimator object:

```
run = exp.submit(config=est)
run
```

Because the call is asynchronous, it returns a **Preparing** or **Running** state as soon as the job is started.

Monitor a remote run

In total, the first run takes **about 10 minutes**. But for subsequent runs, as long as the script dependencies don't change, the same image is reused. So the container startup time is much faster.

What happens while you wait:

- **Image creation**: A Docker image is created that matches the Python environment specified by the estimator. The image is uploaded to the workspace. Image creation and uploading takes **about five minutes**.

  This stage happens once for each Python environment because the container is cached for subsequent runs. During image creation, logs are streamed to the run history. You can monitor the image creation progress by using these logs.

- **Scaling**: If the remote cluster requires more nodes to do the run than currently available, additional nodes are added automatically. Scaling typically takes **about five minutes.**

- **Running**: In this stage, the necessary scripts and files are sent to the compute target. Then datastores are mounted or copied. And then the **entry_script** is run. While the job is running, **stdout** and the **./logs** directory are streamed to the run history. You can monitor the run's progress by using these logs.

- **Post-processing**: The **./outputs** directory of the run is copied over to the run history in your workspace, so you can access these results.

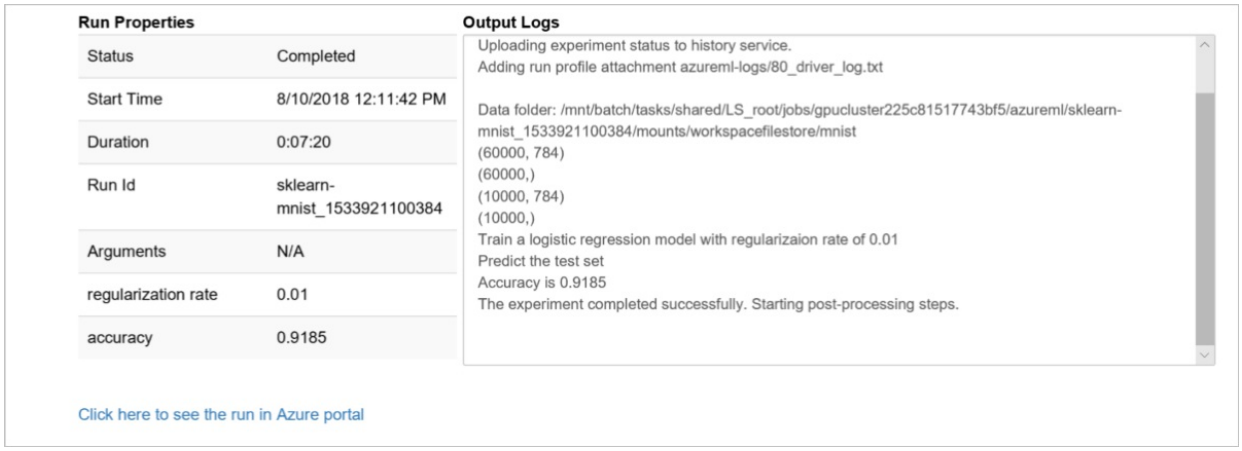
You can check the progress of a running job in several ways. This tutorial uses a Jupyter widget and a `wait_for_completion` method.

Jupyter widget

Watch the progress of the run with a Jupyter widget. Like the run submission, the widget is asynchronous and provides live updates every 10 to 15 seconds until the job finishes:

```
from azureml.widgets import RunDetails
RunDetails(run).show()
```

The widget will look like the following at the end of training:



If you need to cancel a run, you can follow these instructions.

Get log results upon completion

Model training and monitoring happen in the background. Wait until the model has finished training before you run more code. Use `wait_for_completion` to show when the model training is finished:

```
run.wait_for_completion(show_output=False)  # specify True for a verbose log
```

Display run results

You now have a model trained on a remote cluster. Retrieve the accuracy of the model:

```
print(run.get_metrics())
```

The output shows the remote model has accuracy of 0.9204:

```
{'regularization rate': 0.8, 'accuracy': 0.9204}
```

In the next tutorial, you explore this model in more detail.

Register model

The last step in the training script wrote the file `outputs/sklearn_mnist_model.pkl` in a directory named `outputs` in the VM of the cluster where the job is run. `outputs` is a special directory in that all content in this directory is automatically uploaded to your workspace. This content appears in the run record in the experiment under your workspace. So the model file is now also available in your workspace.

You can see files associated with that run:

```
print(run.get_file_names())
```

Register the model in the workspace, so that you or other collaborators can later query, examine, and deploy this model:

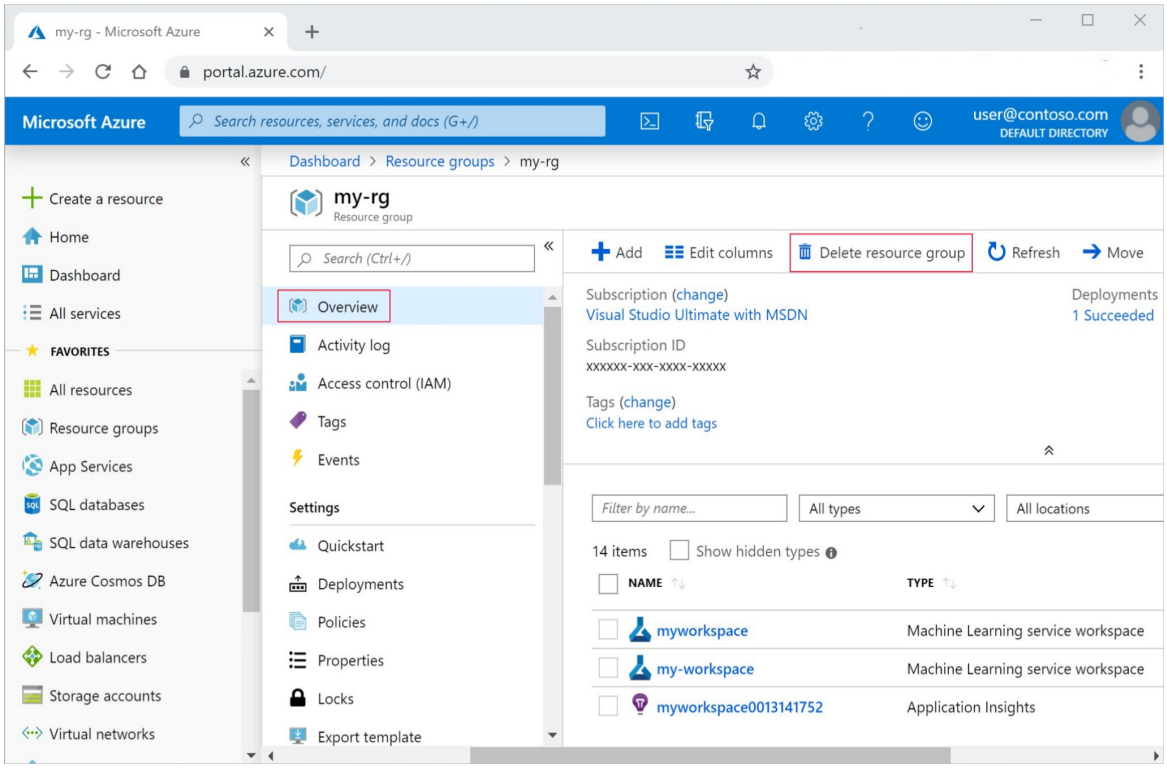
```
# register model
model = run.register_model(model_name='sklearn_mnist',
                           model_path='outputs/sklearn_mnist_model.pkl')
print(model.name, model.id, model.version, sep='\t')
```

Clean up resources

> **IMPORTANT**
>
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

You can also delete just the Azure Machine Learning Compute cluster. However, autoscale is turned on, and the cluster minimum is zero. So this particular resource won't incur additional compute charges when not in use:

```
# Optionally, delete the Azure Machine Learning Compute cluster
compute_target.delete()
```

Next steps

In this Azure Machine Learning tutorial, you used Python for the following tasks:

- Set up your development environment.
- Access and examine the data.
- Train multiple models on a remote cluster using the popular scikit-learn machine learning library
- Review training details and register the best model.

You're ready to deploy this registered model by using the instructions in the next part of the tutorial series:

Tutorial 2 - Deploy models

Tutorial: Deploy an image classification model in Azure Container Instances

4/24/2020 • 7 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This tutorial is **part two of a two-part tutorial series**. In the previous tutorial, you trained machine learning models and then registered a model in your workspace on the cloud. Now you're ready to deploy the model as a web service. A web service is an image, in this case a Docker image. It encapsulates the scoring logic and the model itself.

In this part of the tutorial, you use Azure Machine Learning for the following tasks:

- Set up your testing environment.
- Retrieve the model from your workspace.
- Deploy the model to Container Instances.
- Test the deployed model.

Container Instances is a great solution for testing and understanding the workflow. For scalable production deployments, consider using Azure Kubernetes Service. For more information, see how to deploy and where.

> **NOTE**
> Code in this article was tested with Azure Machine Learning SDK version 1.0.83.

Prerequisites

To run the notebook, first complete the model training in Tutorial (part 1): Train an image classification model. Then open the *img-classification-part2-deploy.ipynb* notebook in your cloned *tutorials/image-classification-mnist-data* folder.

This tutorial is also available on GitHub if you wish to use it on your own local environment. Make sure you have installed `matplotlib` and `scikit-learn` in your environment.

> **IMPORTANT**
> The rest of this article contains the same content as you see in the notebook.
>
> Switch to the Jupyter notebook now if you want to read along as you run the code. To run a single code cell in a notebook, click the code cell and hit **Shift+Enter**. Or, run the entire notebook by choosing **Run all** from the top toolbar.

Set up the environment

Start by setting up a testing environment.

Import packages

Import the Python packages needed for this tutorial.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

import azureml.core

# Display the core SDK version number
print("Azure ML SDK Version: ", azureml.core.VERSION)
```

Deploy as web service

Deploy the model as a web service hosted in ACI.

To build the correct environment for ACI, provide the following:

- A scoring script to show how to use the model
- A configuration file to build the ACI
- The model you trained before

Create scoring script

Create the scoring script, called score.py, used by the web service call to show how to use the model.

You must include two required functions into the scoring script:

- The `init()` function, which typically loads the model into a global object. This function is run only once when the Docker container is started.
- The `run(input_data)` function uses the model to predict a value based on the input data. Inputs and outputs to the run typically use JSON for serialization and de-serialization, but other formats are supported.

```
%%writefile score.py
import json
import numpy as np
import os
import pickle
import joblib

def init():
    global model
    # AZUREML_MODEL_DIR is an environment variable created during deployment.
    # It is the path to the model folder (./azureml-models/$MODEL_NAME/$VERSION)
    # For multiple models, it points to the folder containing all deployed models (./azureml-models)
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_mnist_model.pkl')
    model = joblib.load(model_path)

def run(raw_data):
    data = np.array(json.loads(raw_data)['data'])
    # make prediction
    y_hat = model.predict(data)
    # you can return any data type as long as it is JSON-serializable
    return y_hat.tolist()
```

Create configuration file

Create a deployment configuration file and specify the number of CPUs and gigabyte of RAM needed for your ACI container. While it depends on your model, the default of 1 core and 1 gigabyte of RAM is usually sufficient for many models. If you feel you need more later, you would have to recreate the image and redeploy the service.

```
from azureml.core.webservice import AciWebservice

aciconfig = AciWebservice.deploy_configuration(cpu_cores=1,
                                               memory_gb=1,
                                               tags={"data": "MNIST",  "method" : "sklearn"},
                                               description='Predict MNIST with sklearn')
```

Deploy in ACI

Estimated time to complete: **about 2-5 minutes**

Configure the image and deploy. The following code goes through these steps:

1. Create environment object containing dependencies needed by the model using the environment (`tutorial-env`) saved during training.
2. Create inference configuration necessary to deploy the model as a web service using:
   - The scoring file (`score.py`)
   - environment object created in previous step
3. Deploy the model to the ACI container.
4. Get the web service HTTP endpoint.

```
%%time
from azureml.core.webservice import Webservice
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment
from azureml.core import Workspace
from azureml.core.model import Model

ws = Workspace.from_config()
model = Model(ws, 'sklearn_mnist')


myenv = Environment.get(workspace=ws, name="tutorial-env", version="1")
inference_config = InferenceConfig(entry_script="score.py", environment=tutorial-env)

service = Model.deploy(workspace=ws,
                       name='sklearn-mnist-svc3',
                       models=[model],
                       inference_config=inference_config,
                       deployment_config=aciconfig)

service.wait_for_deployment(show_output=True)
```

Get the scoring web service's HTTP endpoint, which accepts REST client calls. This endpoint can be shared with anyone who wants to test the web service or integrate it into an application.

```
print(service.scoring_uri)
```

Test the model

Download test data

Download the test data to the **./data/** directory

```
import os
from azureml.core import Dataset
from azureml.opendatasets import MNIST

data_folder = os.path.join(os.getcwd(), 'data')
os.makedirs(data_folder, exist_ok=True)

mnist_file_dataset = MNIST.get_file_dataset()
mnist_file_dataset.download(data_folder, overwrite=True)
```

Load test data

Load the test data from the **./data/** directory created during the training tutorial.

```
from utils import load_data
import os
import glob

data_folder = os.path.join(os.getcwd(), 'data')
# note we also shrink the intensity values (X) from 0-255 to 0-1. This helps the neural network converge
faster
X_test = load_data(glob.glob(os.path.join(data_folder,"**/t10k-images-idx3-ubyte.gz"), recursive=True)[0],
False) / 255.0
y_test = load_data(glob.glob(os.path.join(data_folder,"**/t10k-labels-idx1-ubyte.gz"), recursive=True)[0],
True).reshape(-1)
```

Predict test data

Feed the test dataset to the model to get predictions.

The following code goes through these steps:

1. Send the data as a JSON array to the web service hosted in ACI.
2. Use the SDK's `run` API to invoke the service. You can also make raw calls using any HTTP tool such as curl.

```
import json
test = json.dumps({"data": X_test.tolist()})
test = bytes(test, encoding='utf8')
y_hat = service.run(input_data=test)
```

Examine the confusion matrix

Generate a confusion matrix to see how many samples from the test set are classified correctly. Notice the mis-classified value for the incorrect predictions.

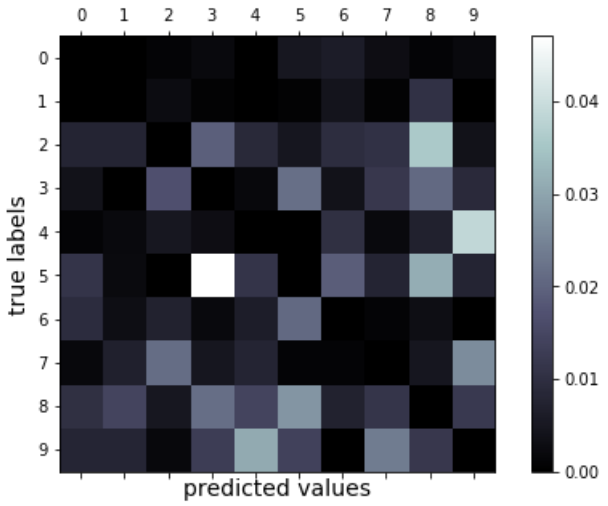
```
from sklearn.metrics import confusion_matrix

conf_mx = confusion_matrix(y_test, y_hat)
print(conf_mx)
print('Overall accuracy:', np.average(y_hat == y_test))
```

The output shows the confusion matrix:

```
[[ 960    0    1    2    1    5    6    3    1    1]
 [   0 1112    3    1    0    1    5    1   12    0]
 [   9    8  920   20   10    4   10   11   37    3]
 [   4    0   17  921    2   21    4   12   20    9]
 [   1    2    5    3  915    0   10    2    6   38]
 [  10    2    0   41   10  770   17    7   28    7]
 [   9    3    7    2    6   20  907    1    3    0]
 [   2    7   22    5    8    1    1  950    5   27]
 [  10   15    5   21   15   27    7   11  851   12]
 [   7    8    2   13   32   13    0   24   12  898]]
Overall accuracy: 0.9204
```

Use `matplotlib` to display the confusion matrix as a graph. In this graph, the X axis represents the actual values, and the Y axis represents the predicted values. The color in each grid represents the error rate. The lighter the color, the higher the error rate is. For example, many 5's are mis-classified as 3's. So you see a bright grid at (5,3).

```
# normalize the diagonal cells so that they don't overpower the rest of the cells when visualized
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
np.fill_diagonal(norm_conf_mx, 0)

fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(111)
cax = ax.matshow(norm_conf_mx, cmap=plt.cm.bone)
ticks = np.arange(0, 10, 1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(ticks)
ax.set_yticklabels(ticks)
fig.colorbar(cax)
plt.ylabel('true labels', fontsize=14)
plt.xlabel('predicted values', fontsize=14)
plt.savefig('conf.png')
plt.show()
```



Show predictions

Test the deployed model with a random sample of 30 images from the test data.

1. Print the returned predictions and plot them along with the input images. Red font and inverse image (white on black) is used to highlight the misclassified samples.

Since the model accuracy is high, you might have to run the following code a few times before you can see a misclassified sample.

```
import json

# find 30 random samples from test set
n = 30
sample_indices = np.random.permutation(X_test.shape[0])[0:n]

test_samples = json.dumps({"data": X_test[sample_indices].tolist()})
test_samples = bytes(test_samples, encoding='utf8')

# predict using the deployed model
result = service.run(input_data=test_samples)

# compare actual value vs. the predicted values:
i = 0
plt.figure(figsize = (20, 1))

for s in sample_indices:
    plt.subplot(1, n, i + 1)
    plt.axhline('')
    plt.axvline('')

    # use different color for misclassified sample
    font_color = 'red' if y_test[s] != result[i] else 'black'
    clr_map = plt.cm.gray if y_test[s] != result[i] else plt.cm.Greys

    plt.text(x=10, y=-10, s=result[i], fontsize=18, color=font_color)
    plt.imshow(X_test[s].reshape(28, 28), cmap=clr_map)

    i = i + 1
plt.show()
```

You can also send raw HTTP request to test the web service.

```
import requests

# send a random row from the test set to score
random_index = np.random.randint(0, len(X_test)-1)
input_data = "{\"data\": [" + str(list(X_test[random_index])) + "]}"

headers = {'Content-Type': 'application/json'}

# for AKS deployment you'd need to the service key in the header as well
# api_key = service.get_key()
# headers = {'Content-Type':'application/json',  'Authorization':('Bearer '+ api_key)}

resp = requests.post(service.scoring_uri, input_data, headers=headers)

print("POST to url", service.scoring_uri)
#print("input data:", input_data)
print("label:", y_test[random_index])
print("prediction:", resp.text)
```

Clean up resources

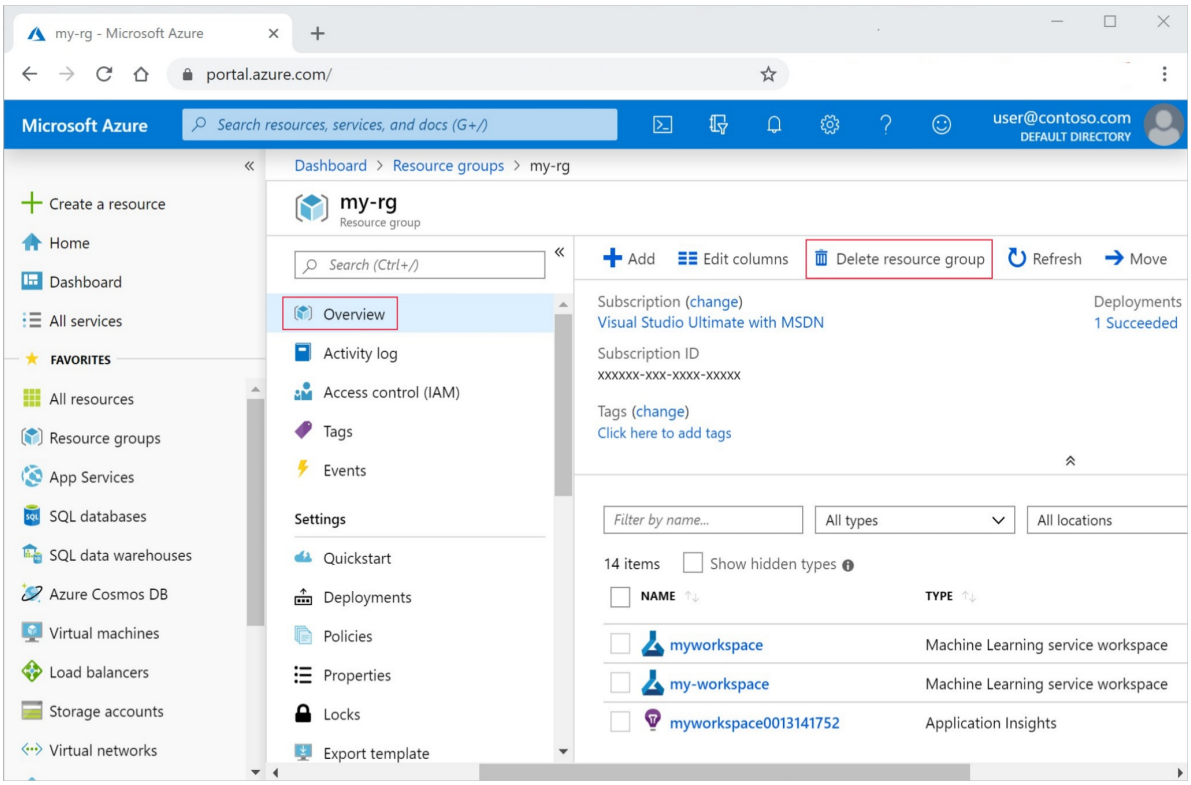
To keep the resource group and workspace for other tutorials and exploration, you can delete only the Container Instances deployment by using this API call:

```
service.delete()
```

> **IMPORTANT**
>
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

Next steps

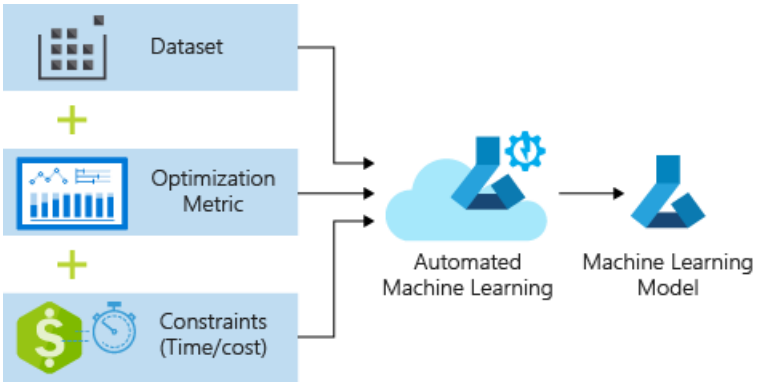
- Learn about all of the deployment options for Azure Machine Learning.
- Learn how to create clients for the web service.
- Make predictions on large quantities of data asynchronously.
- Monitor your Azure Machine Learning models with Application Insights.
- Try out the automatic algorithm selection tutorial.

Tutorial: Use automated machine learning to predict taxi fares

2/10/2020 • 14 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this tutorial, you use automated machine learning in Azure Machine Learning to create a regression model to predict NYC taxi fare prices. This process accepts training data and configuration settings, and automatically iterates through combinations of different feature normalization/standardization methods, models, and hyperparameter settings to arrive at the best model.



In this tutorial you learn the following tasks:

- Download, transform, and clean data using Azure Open Datasets
- Train an automated machine learning regression model
- Calculate model accuracy

If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.

Prerequisites

- Complete the setup tutorial if you don't already have an Azure Machine Learning workspace or notebook virtual machine.
- After you complete the setup tutorial, open the *tutorials/regression-automl-nyc-taxi-data/regression-automated-ml.ipynb* notebook using the same notebook server.

This tutorial is also available on GitHub if you wish to run it in your own local environment. Run `pip install azureml-sdk[automl] azureml-opendatasets azureml-widgets` to get the required packages.

Download and prepare data

Import the necessary packages. The Open Datasets package contains a class representing each data source (`NycTlcGreen` for example) to easily filter date parameters before downloading.

```
from azureml.opendatasets import NycTlcGreen
import pandas as pd
from datetime import datetime
from dateutil.relativedelta import relativedelta
```

Begin by creating a dataframe to hold the taxi data. When working in a non-Spark environment, Open Datasets only allows downloading one month of data at a time with certain classes to avoid `MemoryError` with large datasets.

To download taxi data, iteratively fetch one month at a time, and before appending it to `green_taxi_df` randomly sample 2,000 records from each month to avoid bloating the dataframe. Then preview the data.

```
green_taxi_df = pd.DataFrame([])
start = datetime.strptime("1/1/2015","%m/%d/%Y")
end = datetime.strptime("1/31/2015","%m/%d/%Y")

for sample_month in range(12):
    temp_df_green = NycTlcGreen(start + relativedelta(months=sample_month), end +
relativedelta(months=sample_month)) \
        .to_pandas_dataframe()
    green_taxi_df = green_taxi_df.append(temp_df_green.sample(2000))

green_taxi_df.head(10)
```

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	PAYMENTTYPE	FAREAMOUNT	EXTRA	MTATAX	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
131969	2	2015-01-10 5:34:44	2015-01-10 5:45:03	3	4.84	None	None	-73.88	40.84	-73.94	...	2	15.00	0.50	0.50	0.3	0.00	0.00	nan	16.30	1.00

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	PAYMENTTYPE	FAREAMOUNT	EXTRA	MTATAX	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
11298117	2	2015-01-20 16:26:29	2015-01-20 16:30:26	1	0.69	None	None	-73.96	40.81	-73.96	...	2	4.50	1.00	0.50	0.3	0.00	0.00	nan	6.30	1.00
12786220	2	2015-01-10 05:58:10	2015-01-10 06:00:55	1	0.45	None	None	-73.92	40.76	-73.91	...	2	4.00	0.00	0.50	0.3	0.00	0.00	nan	4.80	1.00

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	PAYMENTTYPE	FAREAMOUNT	EXTRA	MTATAX	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
348430	2	2015-01-17 02:20:50	2015-01-17 02:41:38	1	0.00	None	None	-73.81	40.70	-73.82	...	2	12.50	0.50	0.50	0.3	0.00	0.00	nan	13.80	1.00
12696627	1	2015-01-01 04:10	2015-01-01 06:23	1	0.50	None	None	-73.92	40.76	-73.92	...	2	4.00	0.50	0.50	0	0.00	0.00	nan	5.00	1.00

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	PAYMENTTYPE	FAREAMOUNT	EXTRA	MTATAX	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
811755	1	2015-01-04 19:57:51	2015-01-04 20:05:45	2	1.10	None	None	-73.96	40.72	-73.95	...	2	6.50	0.50	0.50	0.3	0.00	0.00	nan	7.80	1.00
737281	1	2015-01-03 12:27:31	2015-01-03 12:33:52	1	0.90	None	None	-73.88	40.76	-73.87	...	2	6.00	0.00	0.50	0.3	0.00	0.00	nan	6.80	1.00

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	PAYMENTTYPE	FAREAMOUNT	EXTRA	MTATAX	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
113951	1	2015-01-09 23:25:51	2015-01-09 23:39:52	1	3.30	None	None	-73.96	40.72	-73.91	...	2	12.50	0.50	0.50	0.3	0.00	0.00	nan	13.80	1.00
150436	2	2015-01-11 17:15:14	2015-01-11 17:22:57	1	1.19	None	None	-73.94	40.71	-73.95	...	1	7.00	0.00	0.50	0.3	1.75	0.00	nan	9.55	1.00

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	PAYMENTTYPE	FAREAMOUNT	EXTRA	MTATAX	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
4321336	2	2015-01-22 23:16:33	2015-01-23:20:13	1	0.65	None	None	-73.94	40.71	-73.94	...	2	5.00	0.50	0.50	0.3	0.00	0.00	nan	6.30	1.00

10 rows × 23 columns

Now that the initial data is loaded, define a function to create various time-based features from the pickup datetime field. This will create new fields for the month number, day of month, day of week, and hour of day, and will allow the model to factor in time-based seasonality. Use the `apply()` function on the dataframe to iteratively apply the `build_time_features()` function to each row in the taxi data.

```
def build_time_features(vector):
    pickup_datetime = vector[0]
    month_num = pickup_datetime.month
    day_of_month = pickup_datetime.day
    day_of_week = pickup_datetime.weekday()
    hour_of_day = pickup_datetime.hour

    return pd.Series((month_num, day_of_month, day_of_week, hour_of_day))

green_taxi_df[["month_num", "day_of_month","day_of_week", "hour_of_day"]] =
green_taxi_df[["lpepPickupDatetime"]].apply(build_time_features, axis=1)
green_taxi_df.head(10)
```

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE	MONTH_NUM	DAY_OF_MONTH	DAY_OF_WEEK	HOUR_OF_DAY
1319699	2	2015-01-10 5:34:44	2015-01-10 5:45:03	3	4.84	None	None	-73.88	40.84	-73.94	...	0.3	0.00	0.00	nan	16.30	1.00	1	11	6	5
11298117	2	2015-01-20 16:26:29	2015-01-20 16:30:26	1	0.69	None	None	-73.96	40.81	-73.96	...	0.3	0.00	0.00	nan	6.30	1.00	1	20	1	16

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE	MONTH_NUM	DAY_OF_MONTH	DAY_OF_WEEK	HOUR_OF_DAY
1278620	2	2015-01-01 01:05:58:10	2015-01-01 01:06:00:00:55	1	0.45	None	None	-73.92	40.76	-73.91	...	0.3	0.00	0.00	nan	4.80	1.00	1	1	3	5
348430	2	2015-01-17 02:20:50	2015-01-17 02:41:38	1	0.00	None	None	-73.81	40.70	-73.82	...	0.3	0.00	0.00	nan	13.80	1.00	1	17	5	2

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE	MONTH_NUM	DAY_OF_MONTH	DAY_OF_WEEK	HOUR_OF_DAY
12696227	1	2015-01-01:04:10	2015-01-01:06:23	1	0.50	None	None	-73.92	40.76	-73.92	...	0	0.00	0.00	nan	5.00	1.00	1	1	3	5
8117555	1	2015-01-04:57:51	2015-01-04:20:05:45	2	1.10	None	None	-73.96	40.72	-73.95	...	0.3	0.00	0.00	nan	7.80	1.00	1	4	6	19

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE	MONTH_NUM	DAY_OF_MONTH	DAY_OF_WEEK	HOUR_OF_DAY
737281	1	2015-01-03 12:27:31	2015-01-03 12:33:52	1	0.90	None	None	-73.88	40.76	-73.87	...	0.3	0.00	0.00	nan	6.80	1.00	1	3	5	12
113951	1	2015-01-09 23:25:51	2015-01-09 23:39:52	1	3.30	None	None	-73.96	40.72	-73.91	...	0.3	0.00	0.00	nan	13.80	1.00	1	9	4	23

	VENDORID	LPEPPICKUPDATETIME	LPEPDROPOFFDATETIME	PASSENGERCOUNT	TRIPDISTANCE	PULOCATIONID	DOLOCATIONID	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	...	IMPROVEMENTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE	MONTH_NUM	DAY_OF_MONTH	DAY_OF_WEEK	HOUR_OF_DAY
150436	2	2015-01-11 17:15:14	2015-01-11 17:22:57	1	1.19	None	None	-73.94	40.71	-73.95	...	0.3	1.75	0.00	nan	9.55	1.00	1	11	6	17
432136	2	2015-01-22 23:16:33	2015-01-22 23:20:13	1	0.65	None	None	-73.94	40.71	-73.94	...	0.3	0.00	0.00	nan	6.30	1.00	1	22	3	23

10 rows × 27 columns

Remove some of the columns that you won't need for training or additional feature building.

```
columns_to_remove = ["lpepPickupDatetime", "lpepDropoffDatetime", "puLocationId", "doLocationId", "extra",
"mtaTax",
                     "improvementSurcharge", "tollsAmount", "ehailFee", "tripType", "rateCodeID",
                     "storeAndFwdFlag", "paymentType", "fareAmount", "tipAmount"
                    ]
for col in columns_to_remove:
    green_taxi_df.pop(col)

green_taxi_df.head(5)
```

Cleanse data

Run the `describe()` function on the new dataframe to see summary statistics for each field.

```
green_taxi_df.describe()
```

	VENDORID	PASSENGERCOUNT	TRIPDISTANCE	PICKUPLONGITUDE	PICKUPLATITUDE	DROPOFFLONGITUDE	DROPOFFLATITUDE	TOTALAMOUNT	MONTH_NUM	DAY_OF_MONTH	DAY_OF_WEEK	HOUR_OF_DAY
COUNT	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00	480000.00
MEAN	1.78	1.37	2.87	-73.83	40.69	-73.84	40.70	14.75	6.50	15.13	3.27	13.52
STD	0.41	1.04	2.93	2.76	1.52	2.61	1.44	12.08	3.45	8.45	1.95	6.83
MIN	1.00	0.00	0.00	-74.66	0.00	-74.66	0.00	-300.00	1.00	1.00	0.00	0.00
25%	2.00	1.00	1.06	-73.96	40.70	-73.97	40.70	7.80	3.75	8.00	2.00	9.00
50%	2.00	1.00	1.90	-73.94	40.75	-73.94	40.75	11.30	6.50	15.00	3.00	15.00
75%	2.00	1.00	3.60	-73.92	40.80	-73.91	40.79	17.80	9.25	22.00	5.00	19.00
MAX	2.00	9.00	97.57	0.00	41.93	0.00	41.94	450.00	12.00	30.00	6.00	23.00

From the summary statistics, you see that there are several fields that have outliers or values that will reduce model accuracy. First filter the lat/long fields to be within the bounds of the Manhattan area. This will filter out longer taxi trips or trips that are outliers in respect to their relationship with other features.

Additionally filter the `tripDistance` field to be greater than zero but less than 31 miles (the haversine distance between the two lat/long pairs). This eliminates long outlier trips that have inconsistent trip cost.

Lastly, the `totalAmount` field has negative values for the taxi fares, which don't make sense in the context of our model, and the `passengerCount` field has bad data with the minimum values being zero.

Filter out these anomalies using query functions, and then remove the last few columns unnecessary for training.

```
final_df = green_taxi_df.query("pickupLatitude>=40.53 and pickupLatitude<=40.88")
final_df = final_df.query("pickupLongitude>=-74.09 and pickupLongitude<=-73.72")
final_df = final_df.query("tripDistance>=0.25 and tripDistance<31")
final_df = final_df.query("passengerCount>0 and totalAmount>0")

columns_to_remove_for_training = ["pickupLongitude", "pickupLatitude", "dropoffLongitude",
"dropoffLatitude"]
for col in columns_to_remove_for_training:
    final_df.pop(col)
```

Call `describe()` again on the data to ensure cleansing worked as expected. You now have a prepared and cleansed set of taxi, holiday, and weather data to use for machine learning model training.

```
final_df.describe()
```

Configure workspace

Create a workspace object from the existing workspace. A Workspace is a class that accepts your Azure subscription and resource information. It also creates a cloud resource to monitor and track your model runs. `Workspace.from_config()` reads the file **config.json** and loads the authentication details into an object named `ws`. `ws` is used throughout the rest of the code in this tutorial.

```
from azureml.core.workspace import Workspace
ws = Workspace.from_config()
```

Split the data into train and test sets

Split the data into training and test sets by using the `train_test_split` function in the `scikit-learn` library. This function segregates the data into the x (**features**) data set for model training and the y (**values to predict**) data set for testing.

The `test_size` parameter determines the percentage of data to allocate to testing. The `random_state` parameter sets a seed to the random generator, so that your train-test splits are deterministic.

```
from sklearn.model_selection import train_test_split

y_df = final_df.pop("totalAmount")
x_df = final_df

x_train, x_test, y_train, y_test = train_test_split(x_df, y_df, test_size=0.2, random_state=223)
```

The purpose of this step is to have data points to test the finished model that haven't been used to train the model, in order to measure true accuracy.

In other words, a well-trained model should be able to accurately make predictions from data it hasn't already seen. You now have data prepared for auto-training a machine learning model.

Automatically train a model

To automatically train a model, take the following steps:

1. Define settings for the experiment run. Attach your training data to the configuration, and modify settings that control the training process.
2. Submit the experiment for model tuning. After submitting the experiment, the process iterates through different machine learning algorithms and hyperparameter settings, adhering to your defined constraints. It chooses the best-fit model by optimizing an accuracy metric.

Define training settings

Define the experiment parameter and model settings for training. View the full list of settings. Submitting the experiment with these default settings will take approximately 5-20 min, but if you want a shorter run time, reduce the `experiment_timeout_minutes` parameter.

PROPERTY	VALUE IN THIS TUTORIAL	DESCRIPTION
iteration_timeout_minutes	2	Time limit in minutes for each iteration. Reduce this value to decrease total runtime.
experiment_timeout_minutes	20	Maximum amount of time in minutes that all iterations combined can take before the experiment terminates.
enable_early_stopping	True	Flag to enable early termination if the score is not improving in the short term.
primary_metric	spearman_correlation	Metric that you want to optimize. The best-fit model will be chosen based on this metric.
featurization	auto	By using **auto**, the experiment can preprocess the input data (handling missing data, converting text to numeric, etc.)
verbosity	logging.INFO	Controls the level of logging.
n_cross_validations	5	Number of cross-validation splits to perform when validation data is not specified.

```
import logging

automl_settings = {
    "iteration_timeout_minutes": 2,
    "experiment_timeout_minutes": 20,
    "enable_early_stopping": True,
    "primary_metric": 'spearman_correlation',
    "featurization": 'auto',
    "verbosity": logging.INFO,
    "n_cross_validations": 5
}
```

Use your defined training settings as a `**kwargs` parameter to an `AutoMLConfig` object. Additionally, specify your training data and the type of model, which is `regression` in this case.

```
from azureml.train.automl import AutoMLConfig

automl_config = AutoMLConfig(task='regression',
                             debug_log='automated_ml_errors.log',
                             X=x_train.values,
                             y=y_train.values.flatten(),
                             **automl_settings)
```

NOTE

Automated machine learning pre-processing steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same pre-processing steps applied during training are applied to your input data automatically.

Train the automatic regression model

Create an experiment object in your workspace. An experiment acts as a container for your individual runs. Pass the defined `automl_config` object to the experiment, and set the output to `True` to view progress during the run.

After starting the experiment, the output shown updates live as the experiment runs. For each iteration, you see the model type, the run duration, and the training accuracy. The field `BEST` tracks the best running training score based on your metric type.

```
from azureml.core.experiment import Experiment
experiment = Experiment(ws, "taxi-experiment")
local_run = experiment.submit(automl_config, show_output=True)
```

```
Running on local machine
Parent Run ID: AutoML_1766cdf7-56cf-4b28-a340-c4aeee15b12b
Current status: DatasetFeaturization. Beginning to featurize the dataset.
Current status: DatasetEvaluation. Gathering dataset statistics.
Current status: FeaturesGeneration. Generating features for the dataset.
Current status: DatasetFeaturizationCompleted. Completed featurizing the dataset.
Current status: DatasetCrossValidationSplit. Generating individually featurized CV splits.
Current status: ModelSelection. Beginning model selection.

********************************************************************************************
ITERATION: The iteration being evaluated.
PIPELINE: A summary description of the pipeline being evaluated.
DURATION: Time taken for the current iteration.
METRIC: The result of computing score on the fitted pipeline.
BEST: The best observed score thus far.
********************************************************************************************

 ITERATION   PIPELINE                                       DURATION      METRIC      BEST
         0   StandardScalerWrapper RandomForest             0:00:16       0.8746    0.8746
         1   MinMaxScaler RandomForest                      0:00:15       0.9468    0.9468
         2   StandardScalerWrapper ExtremeRandomTrees       0:00:09       0.9303    0.9468
         3   StandardScalerWrapper LightGBM                 0:00:10       0.9424    0.9468
         4   RobustScaler DecisionTree                      0:00:09       0.9449    0.9468
         5   StandardScalerWrapper LassoLars                0:00:09       0.9440    0.9468
         6   StandardScalerWrapper LightGBM                 0:00:10       0.9282    0.9468
         7   StandardScalerWrapper RandomForest             0:00:12       0.8946    0.9468
         8   StandardScalerWrapper LassoLars                0:00:16       0.9439    0.9468
         9   MinMaxScaler ExtremeRandomTrees                0:00:35       0.9199    0.9468
        10   RobustScaler ExtremeRandomTrees                0:00:19       0.9411    0.9468
        11   StandardScalerWrapper ExtremeRandomTrees       0:00:13       0.9077    0.9468
        12   StandardScalerWrapper LassoLars                0:00:15       0.9433    0.9468
        13   MinMaxScaler ExtremeRandomTrees                0:00:14       0.9186    0.9468
        14   RobustScaler RandomForest                      0:00:10       0.8810    0.9468
        15   StandardScalerWrapper LassoLars                0:00:55       0.9433    0.9468
        16   StandardScalerWrapper ExtremeRandomTrees       0:00:13       0.9026    0.9468
        17   StandardScalerWrapper RandomForest             0:00:13       0.9140    0.9468
        18   VotingEnsemble                                 0:00:23       0.9471    0.9471
        19   StackEnsemble                                  0:00:27       0.9463    0.9471
```

Explore the results

Explore the results of automatic training with a Jupyter widget. The widget allows you to see a graph and table of all individual run iterations, along with training accuracy metrics and metadata. Additionally, you can filter on different accuracy metrics than your primary metric with the dropdown selector.

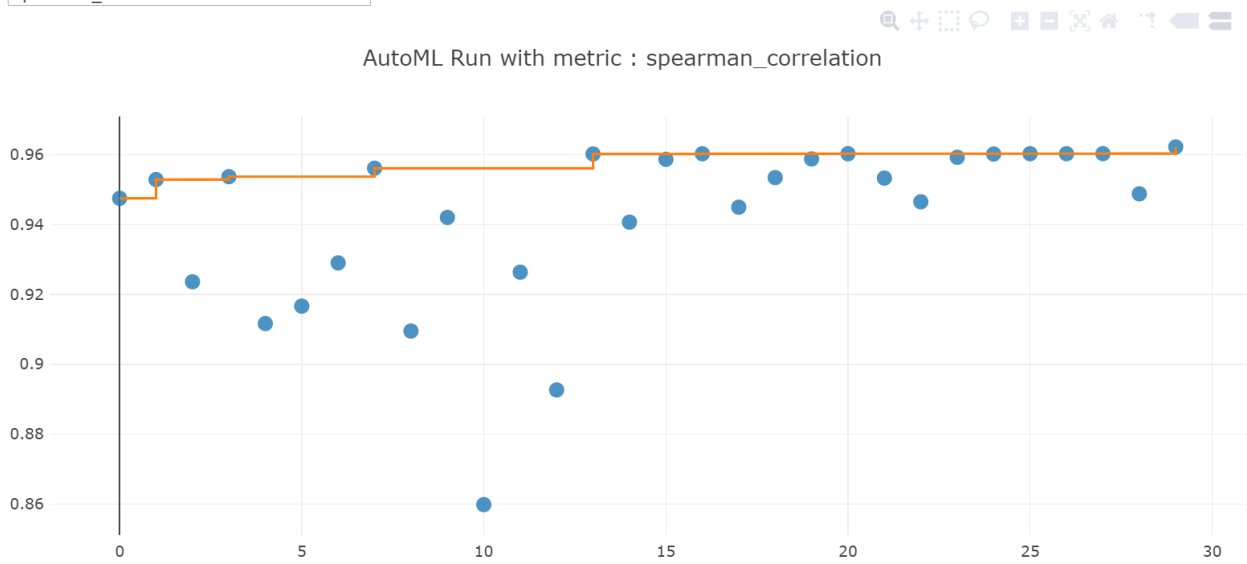
```
from azureml.widgets import RunDetails
RunDetails(local_run).show()
```

AutoML_797ff8a7-a369-4986-8180-e9bbed938259:
Status: Completed

Iteration	Pipeline	Iteration metric	Best metric	Status	Duration	Started	Ru
29	Ensemble	0.96225654	0.96225654	Completed	0:01:10	Dec 6, 2018 6:12 PM	
27	MaxAbsScaler, SGD	0.96033526	0.96033526	Completed	0:00:15	Dec 6, 2018 6:11 PM	
25	StandardScalerWrapper, ElasticNet	0.96031892	0.96031892	Completed	0:00:11	Dec 6, 2018 6:11 PM	
20	MaxAbsScaler, SGD	0.96031661	0.96031661	Completed	0:00:20	Dec 6, 2018 6:09 PM	
26	StandardScalerWrapper, ElasticNet	0.96031391	0.96031892	Completed	0:00:12	Dec 6, 2018 6:11 PM	

Pages: 1 2 3 4 5 6 Next Last 5 per page

spearman_correlation



Click here to see the run in Azure portal

Retrieve the best model

Select the best model from your iterations. The `get_output` function returns the best run and the fitted model for the last fit invocation. By using the overloads on `get_output`, you can retrieve the best run and fitted model for any logged metric or a particular iteration.

```
best_run, fitted_model = local_run.get_output()
print(best_run)
print(fitted_model)
```

Test the best model accuracy

Use the best model to run predictions on the test data set to predict taxi fares. The function `predict` uses the best model and predicts the values of y, **trip cost**, from the `x_test` data set. Print the first 10 predicted cost values from `y_predict`.

```
y_predict = fitted_model.predict(x_test.values)
print(y_predict[:10])
```

Calculate the `root mean squared error` of the results. Convert the `y_test` dataframe to a list to compare to the predicted values. The function `mean_squared_error` takes two arrays of values and calculates the average squared error between them. Taking the square root of the result gives an error in the same units as the y variable, **cost**. It indicates roughly how far the taxi fare predictions are from the actual fares.

```
from sklearn.metrics import mean_squared_error
from math import sqrt

y_actual = y_test.values.flatten().tolist()
rmse = sqrt(mean_squared_error(y_actual, y_predict))
rmse
```

Run the following code to calculate mean absolute percent error (MAPE) by using the full `y_actual` and `y_predict` data sets. This metric calculates an absolute difference between each predicted and actual value and sums all the differences. Then it expresses that sum as a percent of the total of the actual values.

```
sum_actuals = sum_errors = 0

for actual_val, predict_val in zip(y_actual, y_predict):
    abs_error = actual_val - predict_val
    if abs_error < 0:
        abs_error = abs_error * -1

    sum_errors = sum_errors + abs_error
    sum_actuals = sum_actuals + actual_val

mean_abs_percent_error = sum_errors / sum_actuals
print("Model MAPE:")
print(mean_abs_percent_error)
print()
print("Model Accuracy:")
print(1 - mean_abs_percent_error)
```

```
Model MAPE:
0.14353867606052823

Model Accuracy:
0.8564613239394718
```

From the two prediction accuracy metrics, you see that the model is fairly good at predicting taxi fares from the data set's features, typically within +- $4.00, and approximately 15% error.

The traditional machine learning model development process is highly resource-intensive, and requires significant domain knowledge and time investment to run and compare the results of dozens of models. Using automated machine learning is a great way to rapidly test many different models for your scenario.

Clean up resources

Do not complete this section if you plan on running other Azure Machine Learning tutorials.

Stop the compute instance

If you used a compute instance or Notebook VM, stop the VM when you are not using it to reduce cost.

1. In your workspace, select **Compute**.
2. From the list, select the VM.
3. Select **Stop**.
4. When you're ready to use the server again, select **Start**.

Delete everything

If you don't plan to use the resources you created, delete them, so you don't incur any charges.

1. In the Azure portal, select **Resource groups** on the far left.
2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties and select **Delete**.

Next steps

In this automated machine learning tutorial, you did the following tasks:

- Configured a workspace and prepared data for an experiment.
- Trained by using an automated regression model locally with custom parameters.
- Explored and reviewed training results.

Deploy your model with Azure Machine Learning.

Tutorial: Build an Azure Machine Learning pipeline for batch scoring

3/17/2020 • 11 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to build a pipeline in Azure Machine Learning to run a batch scoring job. Machine learning pipelines optimize your workflow with speed, portability, and reuse, so you can focus on machine learning instead of infrastructure and automation. After you build and publish a pipeline, you configure a REST endpoint that you can use to trigger the pipeline from any HTTP library on any platform.

The example uses a pretrained Inception-V3 convolutional neural network model implemented in Tensorflow to classify unlabeled images. Learn more about machine learning pipelines.

In this tutorial, you complete the following tasks:

- Configure workspace
- Download and store sample data
- Create dataset objects to fetch and output data
- Download, prepare, and register the model in your workspace
- Provision compute targets and create a scoring script
- Use the `ParallelRunStep` class for async batch scoring
- Build, run, and publish a pipeline
- Enable a REST endpoint for the pipeline

If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.

Prerequisites

- If you don't already have an Azure Machine Learning workspace or notebook virtual machine, complete Part 1 of the setup tutorial.
- When you finish the setup tutorial, use the same notebook server to open the *tutorials/machine-learning-pipelines-advanced/tutorial-pipeline-batch-scoring-classification.ipynb* notebook.

If you want to run the setup tutorial in your own local environment, you can access the tutorial on GitHub. Run `pip install azureml-sdk[notebooks] azureml-pipeline-core azureml-contrib-pipeline-steps pandas requests` to get the required packages.

Configure workspace and create a datastore

Create a workspace object from the existing Azure Machine Learning workspace.

- A workspace is a class that accepts your Azure subscription and resource information. The workspace also creates a cloud resource you can use to monitor and track your model runs.
- `Workspace.from_config()` reads the `config.json` file and then loads the authentication details into an object named `ws`. The `ws` object is used in the code throughout this tutorial.

```
from azureml.core import Workspace
ws = Workspace.from_config()
```

Create a datastore for sample images

On the `pipelinedata` account, get the ImageNet evaluation public data sample from the `sampledata` public blob container. Call `register_azure_blob_container()` to make the data available to the workspace under the name `images_datastore`. Then, set the workspace default datastore as the output datastore. Use the output datastore to score output in the pipeline.

```
from azureml.core.datastore import Datastore

batchscore_blob = Datastore.register_azure_blob_container(ws,
                      datastore_name="images_datastore",
                      container_name="sampledata",
                      account_name="pipelinedata",
                      overwrite=True)

def_data_store = ws.get_default_datastore()
```

Create dataset objects

When building pipelines, `Dataset` objects are used for reading data from workspace datastores, and `PipelineData` objects are used for transferring intermediate data between pipeline steps.

> **IMPORTANT**
> The batch scoring example in this tutorial uses only one pipeline step. In use cases that have multiple steps, the typical flow will include these steps:
>
> 1. Use `Dataset` objects as *inputs* to fetch raw data, perform some transformation, and then *output* a `PipelineData` object.
> 2. Use the `PipelineData` *output object* in the preceding step as an *input object*. Repeat it for subsequent steps.

In this scenario, you create `Dataset` objects that correspond to the datastore directories for both the input images and the classification labels (y-test values). You also create a `PipelineData` object for the batch scoring output data.

```
from azureml.core.dataset import Dataset
from azureml.pipeline.core import PipelineData

input_images = Dataset.File.from_files((batchscore_blob, "batchscoring/images/"))
label_ds = Dataset.File.from_files((batchscore_blob, "batchscoring/labels/*.txt"))
output_dir = PipelineData(name="scores",
                          datastore=def_data_store,
                          output_path_on_compute="batchscoring/results")
```

Next, register the datasets to the workspace.

```
input_images = input_images.register(workspace = ws, name = "input_images")
label_ds = label_ds.register(workspace = ws, name = "label_ds")
```

Download and register the model

Download the pretrained Tensorflow model to use it for batch scoring in a pipeline. First, create a local directory where you store the model. Then, download and extract the model.

```
import os
import tarfile
import urllib.request

if not os.path.isdir("models"):
    os.mkdir("models")

response = urllib.request.urlretrieve("http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz",
"model.tar.gz")
tar = tarfile.open("model.tar.gz", "r:gz")
tar.extractall("models")
```

Next, register the model to your workspace, so you can easily retrieve the model in the pipeline process. In the `register()` static function, the `model_name` parameter is the key you use to locate your model throughout the SDK.

```
from azureml.core.model import Model

model = Model.register(model_path="models/inception_v3.ckpt",
                       model_name="inception",
                       tags={"pretrained": "inception"},
                       description="Imagenet trained tensorflow inception",
                       workspace=ws)
```

Create and attach the remote compute target

Machine learning pipelines can't be run locally, so you run them on cloud resources or *remote compute targets*. A remote compute target is a reusable virtual compute environment where you run experiments and machine learning workflows.

Run the following code to create a GPU-enabled `AmlCompute` target, and then attach it to your workspace. For more information about compute targets, see the conceptual article.

```
from azureml.core.compute import AmlCompute, ComputeTarget
from azureml.exceptions import ComputeTargetException
compute_name = "gpu-cluster"

# checks to see if compute target already exists in workspace, else create it
try:
    compute_target = ComputeTarget(workspace=ws, name=compute_name)
except ComputeTargetException:
    config = AmlCompute.provisioning_configuration(vm_size="STANDARD_NC6",
                                                   vm_priority="lowpriority",
                                                   min_nodes=0,
                                                   max_nodes=1)

    compute_target = ComputeTarget.create(workspace=ws, name=compute_name, provisioning_configuration=config)
    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

Write a scoring script

To do the scoring, create a batch scoring script called `batch_scoring.py`, and then write it to the current directory. The script takes input images, applies the classification model, and then outputs the predictions to a results file.

The `batch_scoring.py` script takes the following parameters, which get passed from the `ParallelRunStep` you create later:

- `--model_name`: The name of the model being used.
- `--labels_name`: The name of the `Dataset` that holds the `labels.txt` file.

The pipeline infrastructure uses the `ArgumentParser` class to pass parameters into pipeline steps. For example, in the following code, the first argument `--model_name` is given the property identifier `model_name`. In the `init()` function, `Model.get_model_path(args.model_name)` is used to access this property.

```
%%writefile batch_scoring.py

import os
import argparse
import datetime
import time
import tensorflow as tf
from math import ceil
import numpy as np
import shutil
from tensorflow.contrib.slim.python.slim.nets import inception_v3

from azureml.core import Run
from azureml.core.model import Model
from azureml.core.dataset import Dataset

slim = tf.contrib.slim

image_size = 299
num_channel = 3


def get_class_label_dict():
    label = []
    proto_as_ascii_lines = tf.gfile.GFile("labels.txt").readlines()
    for l in proto_as_ascii_lines:
        label.append(l.rstrip())
    return label


def init():
    global g_tf_sess, probabilities, label_dict, input_images

    parser = argparse.ArgumentParser(description="Start a tensorflow model serving")
    parser.add_argument('--model_name', dest="model_name", required=True)
    parser.add_argument('--labels_name', dest="labels_name", required=True)
    args, _ = parser.parse_known_args()

    workspace = Run.get_context(allow_offline=False).experiment.workspace
    label_ds = Dataset.get_by_name(workspace=workspace, name=args.labels_name)
    label_ds.download(target_path='.', overwrite=True)

    label_dict = get_class_label_dict()
    classes_num = len(label_dict)

    with slim.arg_scope(inception_v3.inception_v3_arg_scope()):
        input_images = tf.placeholder(tf.float32, [1, image_size, image_size, num_channel])
        logits, _ = inception_v3.inception_v3(input_images,
                                              num_classes=classes_num,
                                              is_training=False)
        probabilities = tf.argmax(logits, 1)

    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    g_tf_sess = tf.Session(config=config)
    g_tf_sess.run(tf.global_variables_initializer())
```

```
    g_tf_sess.run(tf.local_variables_initializer())

    model_path = Model.get_model_path(args.model_name)
    saver = tf.train.Saver()
    saver.restore(g_tf_sess, model_path)


def file_to_tensor(file_path):
    image_string = tf.read_file(file_path)
    image = tf.image.decode_image(image_string, channels=3)

    image.set_shape([None, None, None])
    image = tf.image.resize_images(image, [image_size, image_size])
    image = tf.divide(tf.subtract(image, [0]), [255])
    image.set_shape([image_size, image_size, num_channel])
    return image


def run(mini_batch):
    result_list = []
    for file_path in mini_batch:
        test_image = file_to_tensor(file_path)
        out = g_tf_sess.run(test_image)
        result = g_tf_sess.run(probabilities, feed_dict={input_images: [out]})
        result_list.append(os.path.basename(file_path) + ": " + label_dict[result[0]])
    return result_list
```

TIP

The pipeline in this tutorial has only one step, and it writes the output to a file. For multi-step pipelines, you also use `ArgumentParser` to define a directory to write output data for input to subsequent steps. For an example of passing data between multiple pipeline steps by using the `ArgumentParser` design pattern, see the notebook.

Build the pipeline

Before you run the pipeline, create an object that defines the Python environment and creates the dependencies that your `batch_scoring.py` script requires. The main dependency required is Tensorflow, but you also install `azureml-defaults` for background processes. Create a `RunConfiguration` object by using the dependencies. Also, specify Docker and Docker-GPU support.

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import DEFAULT_GPU_IMAGE

cd = CondaDependencies.create(pip_packages=["tensorflow-gpu==1.13.1", "azureml-defaults"])
env = Environment(name="parallelenv")
env.python.conda_dependencies = cd
env.docker.base_image = DEFAULT_GPU_IMAGE
```

Create the configuration to wrap the script

Create the pipeline step using the script, environment configuration, and parameters. Specify the compute target you already attached to your workspace.

```
from azureml.contrib.pipeline.steps import ParallelRunConfig

parallel_run_config = ParallelRunConfig(
    environment=env,
    entry_script="batch_scoring.py",
    source_directory=".",
    output_action="append_row",
    mini_batch_size="20",
    error_threshold=1,
    compute_target=compute_target,
    process_count_per_node=2,
    node_count=1
)
```

Create the pipeline step

A pipeline step is an object that encapsulates everything you need to run a pipeline, including:

- Environment and dependency settings
- The compute resource to run the pipeline on
- Input and output data, and any custom parameters
- Reference to a script or SDK logic to run during the step

Multiple classes inherit from the parent class `PipelineStep`. You can choose classes to use specific frameworks or stacks to build a step. In this example, you use the `ParallelRunStep` class to define your step logic by using a custom Python script. If an argument to your script is either an input to the step or an output of the step, the argument must be defined *both* in the `arguments` array *and* in either the `input` or the `output` parameter, respectively.

In scenarios where there is more than one step, an object reference in the `outputs` array becomes available as an *input* for a subsequent pipeline step.

```
from azureml.contrib.pipeline.steps import ParallelRunStep

batch_score_step = ParallelRunStep(
    name="parallel-step-test",
    inputs=[input_images.as_named_input("input_images")],
    output=output_dir,
    models=[model],
    arguments=["--model_name", "inception",
               "--labels_name", "label_ds"],
    parallel_run_config=parallel_run_config,
    allow_reuse=False
)
```

For a list of all the classes you can use for different step types, see the steps package.

Submit the pipeline

Now, run the pipeline. First, create a `Pipeline` object by using your workspace reference and the pipeline step you created. The `steps` parameter is an array of steps. In this case, there's only one step for batch scoring. To build pipelines that have multiple steps, place the steps in order in this array.

Next, use the `Experiment.submit()` function to submit the pipeline for execution. You also specify the custom parameter `param_batch_size`. The `wait_for_completion` function outputs logs during the pipeline build process. You can use the logs to see current progress.

IMPORTANT

The first pipeline run takes roughly *15 minutes*. All dependencies must be downloaded, a Docker image is created, and the Python environment is provisioned and created. Running the pipeline again takes significantly less time because those resources are reused instead of created. However, total run time for the pipeline depends on the workload of your scripts and the processes that are running in each pipeline step.

```
from azureml.core import Experiment
from azureml.pipeline.core import Pipeline

pipeline = Pipeline(workspace=ws, steps=[batch_score_step])
pipeline_run = Experiment(ws, 'batch_scoring').submit(pipeline)
pipeline_run.wait_for_completion(show_output=True)
```

Download and review output

Run the following code to download the output file that's created from the `batch_scoring.py` script. Then, explore the scoring results.

```
import pandas as pd

batch_run = next(pipeline_run.get_children())
batch_output = batch_run.get_output_data("scores")
batch_output.download(local_path="inception_results")

for root, dirs, files in os.walk("inception_results"):
    for file in files:
        if file.endswith("parallel_run_step.txt"):
            result_file = os.path.join(root, file)

df = pd.read_csv(result_file, delimiter=":", header=None)
df.columns = ["Filename", "Prediction"]
print("Prediction has ", df.shape[0], " rows")
df.head(10)
```

Publish and run from a REST endpoint

Run the following code to publish the pipeline to your workspace. In your workspace in Azure Machine Learning studio, you can see metadata for the pipeline, including run history and durations. You can also run the pipeline manually from the studio.

Publishing the pipeline enables a REST endpoint that you can use to run the pipeline from any HTTP library on any platform.

```
published_pipeline = pipeline_run.publish_pipeline(
    name="Inception_v3_scoring", description="Batch scoring using Inception v3 model", version="1.0")

published_pipeline
```

To run the pipeline from the REST endpoint, you need an OAuth2 Bearer-type authentication header. The following example uses interactive authentication (for illustration purposes), but for most production scenarios that require automated or headless authentication, use service principal authentication as described in this article.

Service principal authentication involves creating an *App Registration* in *Azure Active Directory*. First, you generate a client secret, and then you grant your service principal *role access* to your machine learning workspace. Use the `ServicePrincipalAuthentication` class to manage your authentication flow.

Both `InteractiveLoginAuthentication` and `ServicePrincipalAuthentication` inherit from `AbstractAuthentication`. In both cases, use the `get_authentication_header()` function in the same way to fetch the header:

```
from azureml.core.authentication import InteractiveLoginAuthentication

interactive_auth = InteractiveLoginAuthentication()
auth_header = interactive_auth.get_authentication_header()
```

Get the REST URL from the `endpoint` property of the published pipeline object. You can also find the REST URL in your workspace in Azure Machine Learning studio.

Build an HTTP POST request to the endpoint. Specify your authentication header in the request. Add a JSON payload object that has the experiment name and the batch size parameter. As noted earlier in the tutorial, `param_batch_size` is passed through to your `batch_scoring.py` script because you defined it as a `PipelineParameter` object in the step configuration.

Make the request to trigger the run. Include code to access the `Id` key from the response dictionary to get the value of the run ID.

```
import requests

rest_endpoint = published_pipeline.endpoint
response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "batch_scoring",
                               "ParameterAssignments": {"param_batch_size": 50}})
run_id = response.json()["Id"]
```

Use the run ID to monitor the status of the new run. The new run takes another 10-15 min to finish.

The new run will look similar to the pipeline you ran earlier in the tutorial. You can choose not to view the full output.

```
from azureml.pipeline.core.run import PipelineRun
from azureml.widgets import RunDetails

published_pipeline_run = PipelineRun(ws.experiments["batch_scoring"], run_id)
RunDetails(published_pipeline_run).show()
```

Clean up resources

Don't complete this section if you plan to run other Azure Machine Learning tutorials.

Stop the compute instance

If you used a compute instance or Notebook VM, stop the VM when you are not using it to reduce cost.

1. In your workspace, select **Compute**.
2. From the list, select the VM.
3. Select **Stop**.
4. When you're ready to use the server again, select **Start**.

Delete everything

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, in the left menu, select **Resource groups**.

2. In the list of resource groups, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then, select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties, and then select **Delete**.

Next steps

In this machine learning pipelines tutorial, you did the following tasks:

- Built a pipeline with environment dependencies to run on a remote GPU compute resource.
- Created a scoring script to run batch predictions by using a pretrained Tensorflow model.
- Published a pipeline and enabled it to be run from a REST endpoint.

For more examples of how to build pipelines by using the machine learning SDK, see the notebook repository.

Tutorial: Convert ML experimental code to production code

4/1/2020 • 10 minutes to read • Edit Online

A machine learning project requires experimentation where hypotheses are tested with agile tools like Jupyter Notebook using real datasets. Once the model is ready for production, the model code should be placed in a production code repository. In some cases, the model code must be converted to Python scripts to be placed in the production code repository. This tutorial covers a recommended approach on how to export experimentation code to Python scripts.

In this tutorial, you learn how to:

- Clean nonessential code
- Refactor Jupyter Notebook code into functions
- Create Python scripts for related tasks
- Create unit tests

Prerequisites

- Generate the MLOpsPython template and use the `experimentation/Diabetes Ridge Regression Training.ipynb` and `experimentation/Diabetes Ridge Regression Scoring.ipynb` notebooks. These notebooks are used as an example of converting from experimentation to production. You can find these notebooks at https://github.com/microsoft/MLOpsPython/tree/master/experimentation.
- Install `nbconvert`. Follow only the installation instructions under section **Installing nbconvert** on the Installation page.

Remove all nonessential code

Some code written during experimentation is only intended for exploratory purposes. Therefore, the first step to convert experimental code into production code is to remove this nonessential code. Removing nonessential code will also make the code more maintainable. In this section, you'll remove code from the `experimentation/Diabetes Ridge Regression Training.ipynb` notebook. The statements printing the shape of `X` and `y` and the cell calling `features.describe` are just for data exploration and can be removed. After removing nonessential code, `experimentation/Diabetes Ridge Regression Training.ipynb` should look like the following code without markdown:

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import joblib
import pandas as pd

sample_data = load_diabetes()

df = pd.DataFrame(
    data=sample_data.data,
    columns=sample_data.feature_names)
df['Y'] = sample_data.target

X = df.drop('Y', axis=1).values
y = df['Y'].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=0)
data = {"train": {"X": X_train, "y": y_train},
        "test": {"X": X_test, "y": y_test}}

args = {
    "alpha": 0.5
}

reg_model = Ridge(**args)
reg.fit(data["train"]["X"], data["train"]["y"])

preds = reg_model.predict(data["test"]["X"])
mse = mean_squared_error(preds, y_test)
metrics = {"mse": mse}
print(metrics)

model_name = "sklearn_regression_model.pkl"
joblib.dump(value=reg, filename=model_name)
```

Refactor code into functions

Second, the Jupyter code needs to be refactored into functions. Refactoring code into functions makes unit testing easier and makes the code more maintainable. In this section, you'll refactor:

- The Diabetes Ridge Regression Training notebook(`experimentation/Diabetes Ridge Regression Training.ipynb`)
- The Diabetes Ridge Regression Scoring notebook(`experimentation/Diabetes Ridge Regression Scoring.ipynb`)

Refactor Diabetes Ridge Regression Training notebook into functions

In `experimentation/Diabetes Ridge Regression Training.ipynb` , complete the following steps:

1. Create a function called `split_data` to split the data frame into test and train data. The function should take the dataframe `df` as a parameter, and return a dictionary containing the keys `train` and `test` .

   Move the code under the *Split Data into Training and Validation Sets* heading into the `split_data` function and modify it to return the `data` object.

2. Create a function called `train_model` , which takes the parameters `data` and `args` and returns a trained model.

   Move the code under the heading *Training Model on Training Set* into the `train_model` function and modify it to return the `reg_model` object. Remove the `args` dictionary, the values will come from the `args` parameter.

3. Create a function called `get_model_metrics`, which takes parameters `reg_model` and `data`, and evaluates the model then returns a dictionary of metrics for the trained model.

   Move the code under the *Validate Model on Validation Set* heading into the `get_model_metrics` function and modify it to return the `metrics` object.

The three functions should be as follows:

```
# Split the dataframe into test and train data
def split_data(df):
    X = df.drop('Y', axis=1).values
    y = df['Y'].values

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0)
    data = {"train": {"X": X_train, "y": y_train},
            "test": {"X": X_test, "y": y_test}}
    return data


# Train the model, return the model
def train_model(data, args):
    reg_model = Ridge(**args)
    reg_model.fit(data["train"]["X"], data["train"]["y"])
    return reg_model


# Evaluate the metrics for the model
def get_model_metrics(reg_model, data):
    preds = reg_model.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    metrics = {"mse": mse}
    return metrics
```

Still in `experimentation/Diabetes Ridge Regression Training.ipynb`, complete the following steps:

1. Create a new function called `main`, which takes no parameters and returns nothing.
2. Move the code under the "Load Data" heading into the `main` function.
3. Add invocations for the newly written functions into the `main` function:

   ```
   # Split Data into Training and Validation Sets
   data = split_data(df)
   ```

   ```
   # Train Model on Training Set
   args = {
       "alpha": 0.5
   }
   reg = train_model(data, args)
   ```

   ```
   # Validate Model on Validation Set
   metrics = get_model_metrics(reg, data)
   ```

4. Move the code under the "Save Model" heading into the `main` function.

The `main` function should look like the following code:

```
def main():
    # Load Data
    sample_data = load_diabetes()

    df = pd.DataFrame(
        data=sample_data.data,
        columns=sample_data.feature_names)
    df['Y'] = sample_data.target

    # Split Data into Training and Validation Sets
    data = split_data(df)

    # Train Model on Training Set
    args = {
        "alpha": 0.5
    }
    reg = train_model(data, args)

    # Validate Model on Validation Set
    metrics = get_model_metrics(reg, data)

    # Save Model
    model_name = "sklearn_regression_model.pkl"

    joblib.dump(value=reg, filename=model_name)
```

At this stage, there should be no code remaining in the notebook that isn't in a function, other than import statements in the first cell.

Add a statement that calls the `main` function.

```
main()
```

After refactoring, `experimentation/Diabetes Ridge Regression Training.ipynb` should look like the following code without the markdown:

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import pandas as pd
import joblib


# Split the dataframe into test and train data
def split_data(df):
    X = df.drop('Y', axis=1).values
    y = df['Y'].values

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0)
    data = {"train": {"X": X_train, "y": y_train},
            "test": {"X": X_test, "y": y_test}}
    return data


# Train the model, return the model
def train_model(data, args):
    reg_model = Ridge(**args)
    reg_model.fit(data["train"]["X"], data["train"]["y"])
    return reg_model


# Evaluate the metrics for the model
def get_model_metrics(reg_model, data):
    preds = reg_model.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    metrics = {"mse": mse}
    return metrics


def main():
    # Load Data
    sample_data = load_diabetes()

    df = pd.DataFrame(
        data=sample_data.data,
        columns=sample_data.feature_names)
    df['Y'] = sample_data.target

    # Split Data into Training and Validation Sets
    data = split_data(df)

    # Train Model on Training Set
    args = {
        "alpha": 0.5
    }
    reg = train_model(data, args)

    # Validate Model on Validation Set
    metrics = get_model_metrics(reg, data)

    # Save Model
    model_name = "sklearn_regression_model.pkl"

    joblib.dump(value=reg, filename=model_name)

main()
```

Refactor Diabetes Ridge Regression Scoring notebook into functions

In `experimentation/Diabetes Ridge Regression Scoring.ipynb`, complete the following steps:

1. Create a new function called `init`, which takes no parameters and return nothing.
2. Copy the code under the "Load Model" heading into the `init` function.

The `init` function should look like the following code:

```
def init():
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)
```

Once the `init` function has been created, replace all the code under the heading "Load Model" with a single call to `init` as follows:

```
init()
```

In `experimentation/Diabetes Ridge Regression Scoring.ipynb`, complete the following steps:

1. Create a new function called `run`, which takes `raw_data` and `request_headers` as parameters and returns a dictionary of results as follows:

   ```
   {"result": result.tolist()}
   ```

2. Copy the code under the "Prepare Data" and "Score Data" headings into the `run` function.

   The `run` function should look like the following code (Remember to remove the statements that set the variables `raw_data` and `request_headers`, which will be used later when the `run` function is called):

   ```
   def run(raw_data, request_headers):
       data = json.loads(raw_data)["data"]
       data = numpy.array(data)
       result = model.predict(data)

       return {"result": result.tolist()}
   ```

Once the `run` function has been created, replace all the code under the "Prepare Data" and "Score Data" headings with the following code:

```
raw_data = '{"data":[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}'
request_header = {}
prediction = run(raw_data, request_header)
print("Test result: ", prediction)
```

The previous code sets variables `raw_data` and `request_header`, calls the `run` function with `raw_data` and `request_header`, and prints the predictions.

After refactoring, `experimentation/Diabetes Ridge Regression Scoring.ipynb` should look like the following code without the markdown:

```
import json
import numpy
from azureml.core.model import Model
import joblib

def init():
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)

def run(raw_data, request_headers):
    data = json.loads(raw_data)["data"]
    data = numpy.array(data)
    result = model.predict(data)

    return {"result": result.tolist()}

init()
test_row = '{"data":[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}'
request_header = {}
prediction = run(test_row, {})
print("Test result: ", prediction)
```

Combine related functions in Python files

Third, related functions need to be merged into Python files to better help code reuse. In this section, you'll be creating Python files for the following notebooks:

- The Diabetes Ridge Regression Training notebook(`experimentation/Diabetes Ridge Regression Training.ipynb`)
- The Diabetes Ridge Regression Scoring notebook(`experimentation/Diabetes Ridge Regression Scoring.ipynb`)

Create Python file for the Diabetes Ridge Regression Training notebook

Convert your notebook to an executable script by running the following statement in a command prompt, which uses the `nbconvert` package and the path of `experimentation/Diabetes Ridge Regression Training.ipynb`:

```
jupyter nbconvert -- to script "Diabetes Ridge Regression Training.ipynb" –output train
```

Once the notebook has been converted to `train.py`, remove any unwanted comments. Replace the call to `main()` at the end of the file with a conditional invocation like the following code:

```
if __name__ == '__main__':
    main()
```

Your `train.py` file should look like the following code:

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import pandas as pd
import joblib


# Split the dataframe into test and train data
def split_data(df):
    X = df.drop('Y', axis=1).values
    y = df['Y'].values

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0)
    data = {"train": {"X": X_train, "y": y_train},
            "test": {"X": X_test, "y": y_test}}
    return data


# Train the model, return the model
def train_model(data, args):
    reg_model = Ridge(**args)
    reg_model.fit(data["train"]["X"], data["train"]["y"])
    return reg_model


# Evaluate the metrics for the model
def get_model_metrics(reg_model, data):
    preds = reg_model.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    metrics = {"mse": mse}
    return metrics


def main():
    # Load Data
    sample_data = load_diabetes()

    df = pd.DataFrame(
        data=sample_data.data,
        columns=sample_data.feature_names)
    df['Y'] = sample_data.target

    # Split Data into Training and Validation Sets
    data = split_data(df)

    # Train Model on Training Set
    args = {
        "alpha": 0.5
    }
    reg = train_model(data, args)

    # Validate Model on Validation Set
    metrics = get_model_metrics(reg, data)

    # Save Model
    model_name = "sklearn_regression_model.pkl"

    joblib.dump(value=reg, filename=model_name)

if __name__ == '__main__':
    main()
```

`train.py` can now be invoked from a terminal by running `python train.py`. The functions from `train.py` can also be called from other files.

The `train_aml.py` file found in the `diabetes_regression/training` directory in the MLOpsPython repository calls the functions defined in `train.py` in the context of an Azure Machine Learning experiment run. The functions can also be called in unit tests, covered later in this guide.

Create Python file for the Diabetes Ridge Regression Scoring notebook

Covert your notebook to an executable script by running the following statement in a command prompt that which uses the `nbconvert` package and the path of `experimentation/Diabetes Ridge Regression Scoring.ipynb`:

```
jupyter nbconvert -- to script "Diabetes Ridge Regression Scoring.ipynb" –output score
```

Once the notebook has been converted to `score.py`, remove any unwanted comments. Your `score.py` file should look like the following code:

```
import json
import numpy
from azureml.core.model import Model
import joblib

def init():
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)

def run(raw_data, request_headers):
    data = json.loads(raw_data)["data"]
    data = numpy.array(data)
    result = model.predict(data)

    return {"result": result.tolist()}

init()
test_row = '{"data":[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}'
request_header = {}
prediction = run(test_row, request_header)
print("Test result: ", prediction)
```

The `model` variable needs to be global so that it's visible throughout the script. Add the following statement at the beginning of the `init` function:

```
global model
```

After adding the previous statement, the `init` function should look like the following code:

```
def init():
    global model

    # load the model from file into a global object
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)
```

Create unit tests for each Python file

Fourth, create unit tests for your Python functions. Unit tests protect code against functional regressions and make it easier to maintain. In this section, you'll be creating unit tests for the functions in `train.py`.

`train.py` contains multiple functions, but we'll only create a single unit test for the `train_model` function using the

Pytest framework in this tutorial. Pytest isn't the only Python unit testing framework, but it's one of the most commonly used. For more information, visit Pytest.

A unit test usually contains three main actions:

- Arrange object - creating and setting up necessary objects
- Act on an object
- Assert what is expected

The unit test will call `train_model` with some hard-coded data and arguments, and validate that `train_model` acted as expected by using the resulting trained model to make a prediction and comparing that prediction to an expected value.

```
import numpy as np
from code.training.train import train_model


def test_train_model():
    # Arrange
    X_train = np.array([1, 2, 3, 4, 5, 6]).reshape(-1, 1)
    y_train = np.array([10, 9, 8, 8, 6, 5])
    data = {"train": {"X": X_train, "y": y_train}}

    # Act
    reg_model = train_model(data, {"alpha": 1.2})

    # Assert
    preds = reg_model.predict([[1], [2]])
    np.testing.assert_almost_equal(preds, [9.93939393939394, 9.03030303030303])
```

Next steps

Now that you understand how to convert from an experiment to production code, see the following links for more information and next steps:

- MLOpsPython: Build a CI/CD pipeline to train, evaluate and deploy your own model using Azure Pipelines and Azure Machine Learning
- Monitor Azure ML experiment runs and metrics
- Monitor and collect data from ML web service endpoints

Tutorial: Use R to create a machine learning model

4/10/2020 • 14 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this tutorial you'll use the Azure Machine Learning R SDK to create a logistic regression model that predicts the likelihood of a fatality in a car accident. You'll see how the Azure Machine Learning cloud resources work with R to provide a scalable environment for training and deploying a model.

In this tutorial, you perform the following tasks:

- Create an Azure Machine Learning workspace
- Clone a notebook folder with the files necessary to run this tutorial into your workspace
- Open RStudio from your workspace
- Load data and prepare for training
- Upload data to a datastore so it is available for remote training
- Create a compute resource to train the model remotely
- Train a `caret` model to predict probability of fatality
- Deploy a prediction endpoint
- Test the model from R

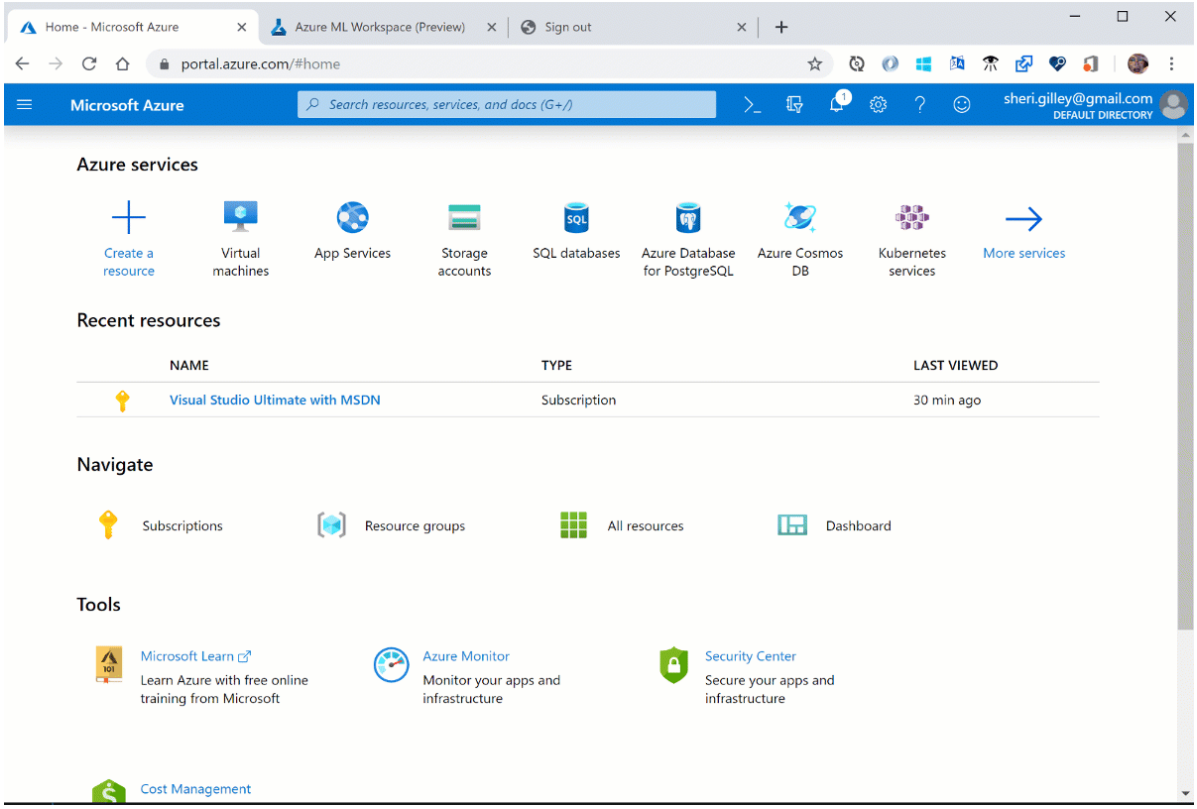
If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.

Create a workspace

An Azure Machine Learning workspace is a foundational resource in the cloud that you use to experiment, train, and deploy machine learning models. It ties your Azure subscription and resource group to an easily consumed object in the service.

You create a workspace via the Azure portal, a web-based console for managing your Azure resources.

1. Sign in to Azure portal by using the credentials for your Azure subscription.
2. In the upper-left corner of Azure portal, select **+ Create a resource**.



3. Use the search bar to find **Machine Learning**.
4. Select **Machine Learning**.
5. In the **Machine Learning** pane, select **Create** to begin.
6. Provide the following information to configure your new workspace:

FIELD	DESCRIPTION
Workspace name	Enter a unique name that identifies your workspace. In this example, we use **docs-ws**. Names must be unique across the resource group. Use a name that's easy to recall and to differentiate from workspaces created by others.
Subscription	Select the Azure subscription that you want to use.
Resource group	Use an existing resource group in your subscription or enter a name to create a new resource group. A resource group holds related resources for an Azure solution. In this example, we use **docs-aml**.
Location	Select the location closest to your users and the data resources to create your workspace.
Workspace edition	Select **Basic** as the workspace type for this tutorial. The workspace type (Basic & Enterprise) determines the features to which you'll have access and pricing. Everything in this tutorial can be performed with either a Basic or Enterprise workspace.

7. After you are finished configuring the workspace, select **Review + Create**.

> **WARNING**
> It can take several minutes to create your workspace in the cloud.

When the process is finished, a deployment success message appears.

8. To view the new workspace, select **Go to resource**.

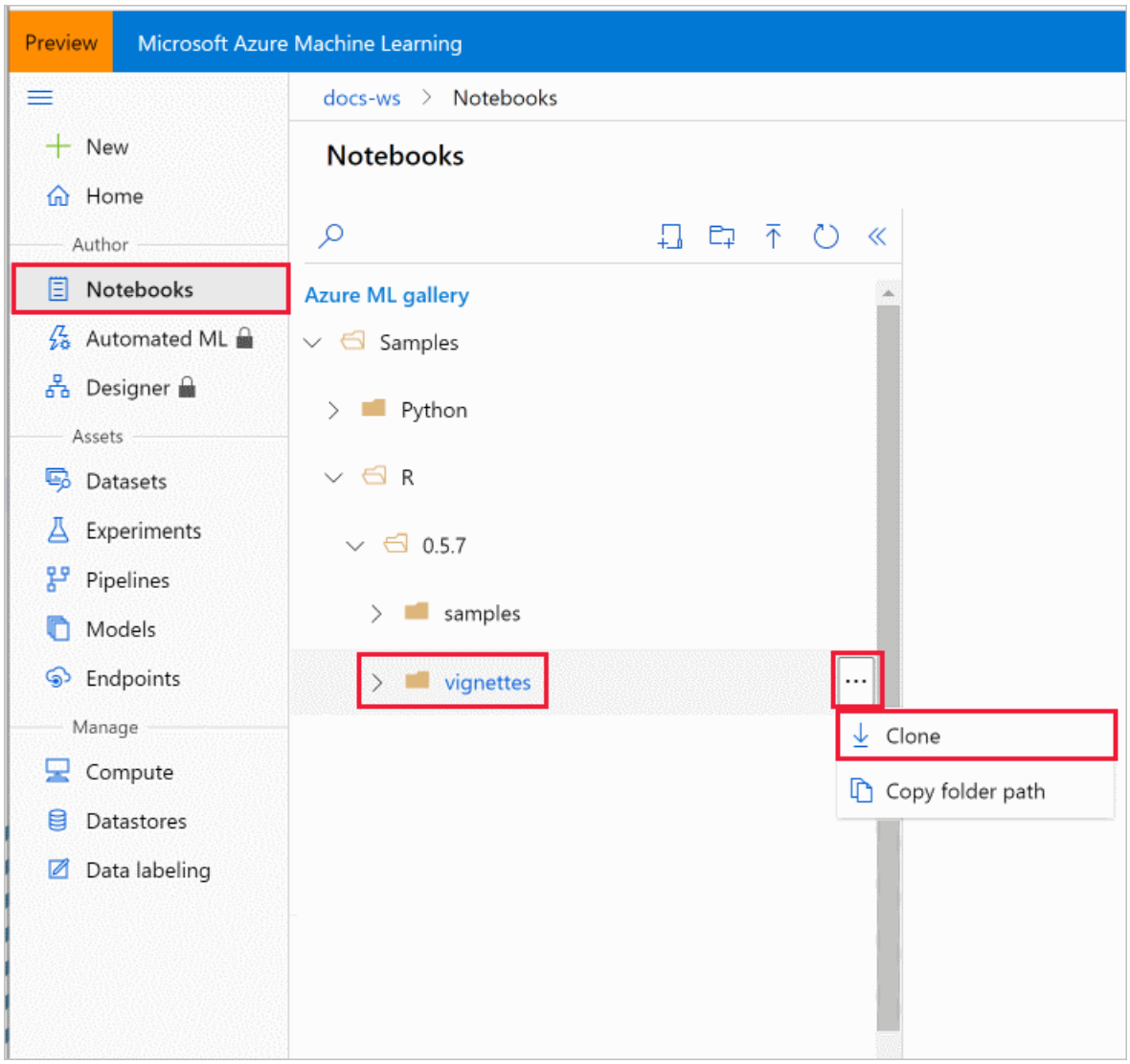
> **IMPORTANT**
> Take note of your **workspace** and **subscription**. You'll need these to ensure you create your experiment in the right place.

Clone a notebook folder

This example uses the cloud notebook server in your workspace for an install-free and pre-configured experience. Use your own environment if you prefer to have control over your environment, packages and dependencies.

You complete the following experiment set-up and run steps in Azure Machine Learning studio, a consolidated interface that includes machine learning tools to perform data science scenarios for data science practitioners of all skill levels.

1. Sign in to Azure Machine Learning studio.
2. Select your subscription and the workspace you created.
3. Select **Notebooks** on the left.
4. Open the **Samples** folder.
5. Open the **R** folder.
6. Open the folder with a version number on it. This number represents the current release for the R SDK.
7. Select the **"..."** at the right of the **vignettes** folder and then select **Clone**.



8. A list of folders displays showing each user who accesses the workspace. Select your folder to clone the **vignettes** folder there.

Use RStudio on a compute instance or Notebook VM to run this tutorial.

1. Select **Compute** on the left.
2. Add a compute resource if one does not already exist.
3. Once the compute is running, use the **RStudio** link to open RStudio.
4. In RStudio, your *vignettes* folder is a few levels down from *Users* in the **Files** section on the lower right. Under *vignettes*, select the *train-and-deploy-to-aci* folder to find the files needed in this tutorial.

> **IMPORTANT**
> The rest of this article contains the same content as you see in the *train-and-deploy-to-aci.Rmd* file. If you are experienced with RMarkdown, feel free to use the code from that file. Or you can copy/paste the code snippets from there, or from this article into an R script or the command line.

Set up your development environment

The setup for your development work in this tutorial includes the following actions:

- Install required packages

- Connect to a workspace, so that your compute instance can communicate with remote resources
- Create an experiment to track your runs
- Create a remote compute target to use for training

Install required packages

- Install the latest version from CRAN.

```
# install the latest version from CRAN
install.packages("azuremlsdk")
azuremlsdk::install_azureml(envname = 'r-reticulate')
```

- Or install the development version from GitHub.

```
# or install the development version from GitHub
remotes::install_github('https://github.com/Azure/azureml-sdk-for-r')
azuremlsdk::install_azureml(envname = 'r-reticulate')
```

Now go ahead and import the **azuremlsdk** package.

```
library(azuremlsdk)
```

The training and scoring scripts (`accidents.R` and `accident_predict.R`) have some additional dependencies. If you plan on running those scripts locally, make sure you have those required packages as well.

Load your workspace

Instantiate a workspace object from your existing workspace. The following code will load the workspace details from the **config.json** file. You can also retrieve a workspace using `get_workspace()` .

```
ws <- load_workspace_from_config()
```

Create an experiment

An Azure ML experiment tracks a grouping of runs, typically from the same training script. Create an experiment to track the runs for training the caret model on the accidents data.

```
experiment_name <- "accident-logreg"
exp <- experiment(ws, experiment_name)
```

Create a compute target

By using Azure Machine Learning Compute (AmlCompute), a managed service, data scientists can train machine learning models on clusters of Azure virtual machines. Examples include VMs with GPU support. In this tutorial, you create a single-node AmlCompute cluster as your training environment. The code below creates the compute cluster for you if it doesn't already exist in your workspace.

You may need to wait a few minutes for your compute cluster to be provisioned if it doesn't already exist.

```
cluster_name <- "rcluster"
compute_target <- get_compute(ws, cluster_name = cluster_name)
if (is.null(compute_target)) {
  vm_size <- "STANDARD_D2_V2"
  compute_target <- create_aml_compute(workspace = ws,
                                       cluster_name = cluster_name,
                                       vm_size = vm_size,
                                       max_nodes = 1)
}

wait_for_provisioning_completion(compute_target)
```

Prepare data for training

This tutorial uses data from the US National Highway Traffic Safety Administration (with thanks to Mary C. Meyer and Tremika Finney). This dataset includes data from over 25,000 car crashes in the US, with variables you can use to predict the likelihood of a fatality. First, import the data into R and transform it into a new dataframe `accidents` for analysis, and export it to an `Rdata` file.

```
nassCDS <- read.csv("nassCDS.csv",
                    colClasses=c("factor","numeric","factor",
                                 "factor","factor","numeric",
                                 "factor","numeric","numeric",
                                 "numeric","character","character",
                                 "numeric","numeric","character"))
accidents <-
na.omit(nassCDS[,c("dead","dvcat","seatbelt","frontal","sex","ageOFocc","yearVeh","airbag","occRole")])
accidents$frontal <- factor(accidents$frontal, labels=c("notfrontal","frontal"))
accidents$occRole <- factor(accidents$occRole)
accidents$dvcat <- ordered(accidents$dvcat,
                           levels=c("1-9km/h","10-24","25-39","40-54","55+"))

saveRDS(accidents, file="accidents.Rd")
```

Upload data to the datastore

Upload data to the cloud so that it can be access by your remote training environment. Each Azure Machine Learning workspace comes with a default datastore that stores the connection information to the Azure blob container that is provisioned in the storage account attached to the workspace. The following code will upload the accidents data you created above to that datastore.

```
ds <- get_default_datastore(ws)

target_path <- "accidentdata"
upload_files_to_datastore(ds,
                          list("./accidents.Rd"),
                          target_path = target_path,
                          overwrite = TRUE)
```

Train a model

For this tutorial, fit a logistic regression model on your uploaded data using your remote compute cluster. To submit a job, you need to:

- Prepare the training script
- Create an estimator
- Submit the job

Prepare the training script

A training script called `accidents.R` has been provided for you in the same directory as this tutorial. Notice the following details **inside the training script** that have been done to leverage Azure Machine Learning for training:

- The training script takes an argument `-d` to find the directory that contains the training data. When you define and submit your job later, you point to the datastore for this argument. Azure ML will mount the storage folder to the remote cluster for the training job.
- The training script logs the final accuracy as a metric to the run record in Azure ML using `log_metric_to_run()`. The Azure ML SDK provides a set of logging APIs for logging various metrics during training runs. These metrics are recorded and persisted in the experiment run record. The metrics can then be accessed at any time or viewed in the run details page in studio. See the reference for the full set of logging methods `log_*()`.
- The training script saves your model into a directory named **outputs**. The `./outputs` folder receives special treatment by Azure ML. During training, files written to `./outputs` are automatically uploaded to your run record by Azure ML and persisted as artifacts. By saving the trained model to `./outputs`, you'll be able to access and retrieve your model file even after the run is over and you no longer have access to your remote training environment.

Create an estimator

An Azure ML estimator encapsulates the run configuration information needed for executing a training script on the compute target. Azure ML runs are run as containerized jobs on the specified compute target. By default, the Docker image built for your training job will include R, the Azure ML SDK, and a set of commonly used R packages. See the full list of default packages included here.

To create the estimator, define:

- The directory that contains your scripts needed for training (`source_directory`). All the files in this directory are uploaded to the cluster node(s) for execution. The directory must contain your training script and any additional scripts required.
- The training script that will be executed (`entry_script`).
- The compute target (`compute_target`), in this case the AmlCompute cluster you created earlier.
- The parameters required from the training script (`script_params`). Azure ML will run your training script as a command-line script with `Rscript`. In this tutorial you specify one argument to the script, the data directory mounting point, which you can access with `ds$path(target_path)`.
- Any environment dependencies required for training. The default Docker image built for training already contains the three packages (`caret`, `e1071`, and `optparse`) needed in the training script. So you don't need to specify additional information. If you are using R packages that are not included by default, use the estimator's `cran_packages` parameter to add additional CRAN packages. See the `estimator()` reference for the full set of configurable options.

```
est <- estimator(source_directory = ".",
                 entry_script = "accidents.R",
                 script_params = list("--data_folder" = ds$path(target_path)),
                 compute_target = compute_target
                 )
```

Submit the job on the remote cluster

Finally submit the job to run on your cluster. `submit_experiment()` returns a Run object that you then use to interface with the run. In total, the first run takes **about 10 minutes**. But for later runs, the same Docker image is reused as long as the script dependencies don't change. In this case, the image is cached and the container startup time is much faster.

```
run <- submit_experiment(exp, est)
```

You can view the run's details in RStudio Viewer. Clicking the "Web View" link provided will bring you to Azure Machine Learning studio, where you can monitor the run in the UI.

```
view_run_details(run)
```

Model training happens in the background. Wait until the model has finished training before you run more code.

```
wait_for_run_completion(run, show_output = TRUE)
```

You -- and colleagues with access to the workspace -- can submit multiple experiments in parallel, and Azure ML will take of scheduling the tasks on the compute cluster. You can even configure the cluster to automatically scale up to multiple nodes, and scale back when there are no more compute tasks in the queue. This configuration is a cost-effective way for teams to share compute resources.

Retrieve training results

Once your model has finished training, you can access the artifacts of your job that were persisted to the run record, including any metrics logged and the final trained model.

Get the logged metrics

In the training script `accidents.R`, you logged a metric from your model: the accuracy of the predictions in the training data. You can see metrics in the studio, or extract them to the local session as an R list as follows:

```
metrics <- get_run_metrics(run)
metrics
```

If you've run multiple experiments (say, using differing variables, algorithms, or hyperparamers), you can use the metrics from each run to compare and choose the model you'll use in production.

Get the trained model

You can retrieve the trained model and look at the results in your local R session. The following code will download the contents of the `./outputs` directory, which includes the model file.

```
download_files_from_run(run, prefix="outputs/")
accident_model <- readRDS("outputs/model.rds")
summary(accident_model)
```

You see some factors that contribute to an increase in the estimated probability of death:

- higher impact speed
- male driver
- older occupant
- passenger

You see lower probabilities of death with:

- presence of airbags
- presence seatbelts
- frontal collision

The vehicle year of manufacture does not have a significant effect.

You can use this model to make new predictions:

```
newdata <- data.frame( # valid values shown below
 dvcat="10-24",        # "1-9km/h" "10-24"   "25-39"   "40-54"   "55+"
 seatbelt="none",      # "none"   "belted"
 frontal="frontal",    # "notfrontal" "frontal"
 sex="f",              # "f" "m"
 ageOFocc=16,          # age in years, 16-97
 yearVeh=2002,         # year of vehicle, 1955-2003
 airbag="none",        # "none"   "airbag"
 occRole="pass"        # "driver" "pass"
 )

## predicted probability of death for these variables, as a percentage
as.numeric(predict(accident_model,newdata, type="response")*100)
```

Deploy as a web service

With your model, you can predict the danger of death from a collision. Use Azure ML to deploy your model as a prediction service. In this tutorial, you will deploy the web service in Azure Container Instances (ACI).

Register the model

First, register the model you downloaded to your workspace with `register_model()`. A registered model can be any collection of files, but in this case the R model object is sufficient. Azure ML will use the registered model for deployment.

```
model <- register_model(ws,
                        model_path = "outputs/model.rds",
                        model_name = "accidents_model",
                        description = "Predict probablity of auto accident")
```

Define the inference dependencies

To create a web service for your model, you first need to create a scoring script (`entry_script`), an R script that will take as input variable values (in JSON format) and output a prediction from your model. For this tutorial, use the provided scoring file `accident_predict.R`. The scoring script must contain an `init()` method that loads your model and returns a function that uses the model to make a prediction based on the input data. See the documentation for more details.

Next, define an Azure ML **environment** for your script's package dependencies. With an environment, you specify R packages (from CRAN or elsewhere) that are needed for your script to run. You can also provide the values of environment variables that your script can reference to modify its behavior. By default, Azure ML will build the same default Docker image used with the estimator for training. Since the tutorial has no special requirements, create an environment with no special attributes.

```
r_env <- r_environment(name = "basic_env")
```

If you want to use your own Docker image for deployment instead, specify the `custom_docker_image` parameter. See the `r_environment()` reference for the full set of configurable options for defining an environment.

Now you have everything you need to create an **inference config** for encapsulating your scoring script and environment dependencies.

```
inference_config <- inference_config(
  entry_script = "accident_predict.R",
  environment = r_env)
```

Deploy to ACI

In this tutorial, you will deploy your service to ACI. This code provisions a single container to respond to inbound requests, which is suitable for testing and light loads. See `aci_webservice_deployment_config()` for additional configurable options. (For production-scale deployments, you can also deploy to Azure Kubernetes Service.)

```
aci_config <- aci_webservice_deployment_config(cpu_cores = 1, memory_gb = 0.5)
```

Now you deploy your model as a web service. Deployment **can take several minutes**.

```
aci_service <- deploy_model(ws,
                            'accident-pred',
                            list(model),
                            inference_config,
                            aci_config)

wait_for_deployment(aci_service, show_output = TRUE)
```

Test the deployed service

Now that your model is deployed as a service, you can test the service from R using `invoke_webservice()`. Provide a new set of data to predict from, convert it to JSON, and send it to the service.

```
library(jsonlite)

newdata <- data.frame( # valid values shown below
 dvcat="10-24",        # "1-9km/h" "10-24"   "25-39"   "40-54"   "55+"
 seatbelt="none",      # "none"   "belted"
 frontal="frontal",    # "notfrontal" "frontal"
 sex="f",              # "f" "m"
 ageOFocc=22,          # age in years, 16-97
 yearVeh=2002,         # year of vehicle, 1955-2003
 airbag="none",        # "none"   "airbag"
 occRole="pass"        # "driver" "pass"
 )

prob <- invoke_webservice(aci_service, toJSON(newdata))
prob
```

You can also get the web service's HTTP endpoint, which accepts REST client calls. You can share this endpoint with anyone who wants to test the web service or integrate it into an application.

```
aci_service$scoring_uri
```

Clean up resources

Delete the resources once you no longer need them. Don't delete any resource you plan to still use.

Delete the web service:

```
delete_webservice(aci_service)
```

Delete the registered model:

```
delete_model(model)
```

Delete the compute cluster:

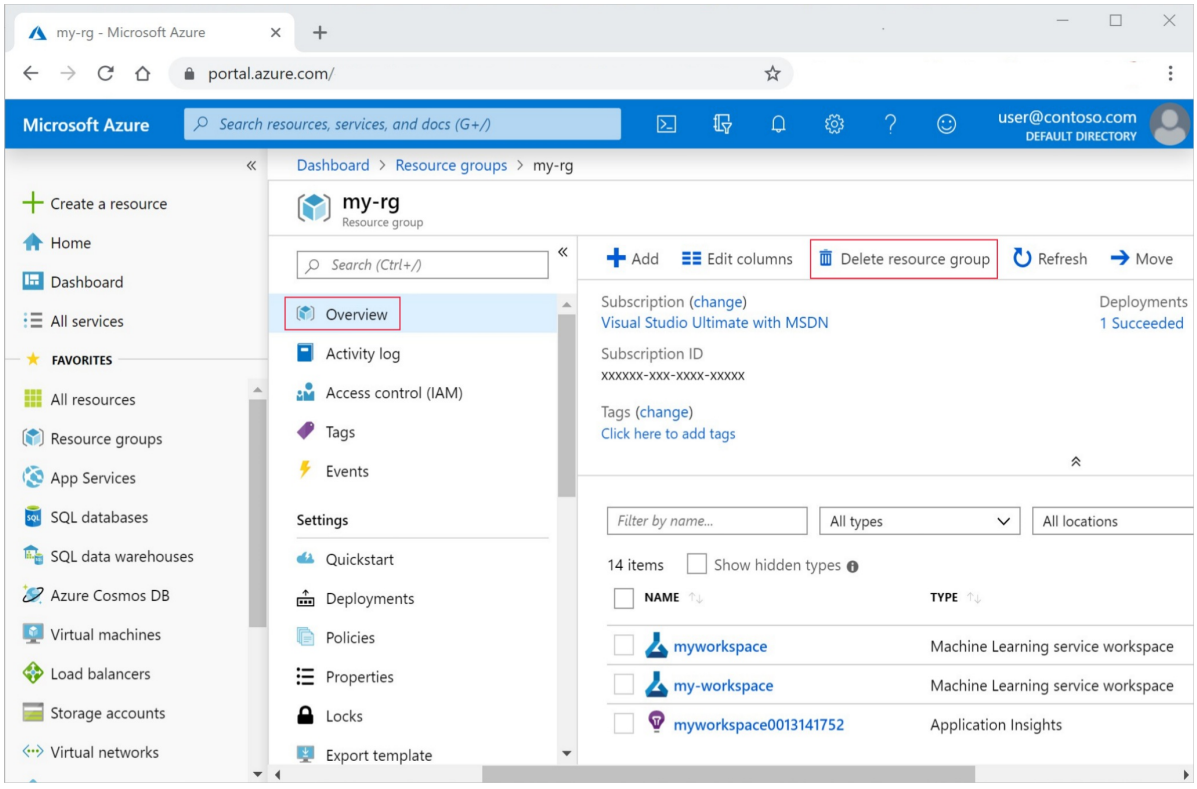
```
delete_compute(compute)
```

Delete everything

> **IMPORTANT**
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties and select **Delete**.

Next steps

- Now that you've completed your first Azure Machine Learning experiment in R, learn more about the Azure

Machine Learning SDK for R.

- Learn more about Azure Machine Learning with R from the examples in the other *vignettes* folders.

Tutorial: Train and deploy a model from the CLI

4/17/2020 • 14 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this tutorial, you use the machine learning extension for the Azure CLI to train, register, and deploy a model.

The Python training scripts in this tutorial use scikit-learn to train a basic model. The focus of this tutorial is not on the scripts or the model, but the process of using the CLI to work with Azure Machine Learning.

Learn how to take the following actions:

- Install the machine learning extension
- Create an Azure Machine Learning workspace
- Create the compute resource used to train the model
- Define and register the dataset used to train the model
- Start a training run
- Register and download a model
- Deploy the model as a web service
- Score data using the web service

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- To use the CLI commands in this document from your **local environment**, you need the Azure CLI.

  If you use the Azure Cloud Shell, the CLI is accessed through the browser and lives in the cloud.

Download the example project

For this tutorial, download the https://github.com/microsoft/MLOps project. The files in the `examples/cli-train-deploy` directory are used by the steps in this tutorial.

To get a local copy of the files, either download a .zip archive, or use the following Git command to clone the repository:

```
git clone https://github.com/microsoft/MLOps.git
```

Training files

The `examples/cli-train-deploy` directory from the project contains the following files, which are used when training a model:

- `.azureml\mnist.runconfig`: A **run configuration** file. This file defines the runtime environment needed to train the model. In this example, it also mounts the data used to train the model into the training environment.
- `scripts\train.py`: The training script. This file trains the model.
- `scripts\utils.py`: A helper file used by the training script.
- `.azureml\conda_dependencies.yml`: Defines the software dependencies needed to run the training script.
- `dataset.json`: The dataset definition. Used to register the MNIST dataset in the Azure Machine Learning

workspace.

Deployment files

The repository contains the following files, which are used to deploy the trained model as a web service:

- `aciDeploymentConfig.yml`: A **deployment configuration** file. This file defines the hosting environment needed for the model.
- `inferenceConfig.json`: An **inference configuration** file. This file defines the software environment used by the service to score data with the model.
- `score.py`: A python script that accepts incoming data, scores it using the model, and then returns a response.
- `scoring-env.yml`: The conda dependencies needed to run the model and `score.py` script.
- `testdata.json`: A data file that can be used to test the deployed web service.

Connect to your Azure subscription

There are several ways that you can authenticate to your Azure subscription from the CLI. The most basic is to interactively authenticate using a browser. To authenticate interactively, open a command line or terminal and use the following command:

```
az login
```

If the CLI can open your default browser, it will do so and load a sign-in page. Otherwise, you need to open a browser and follow the instructions on the command line. The instructions involve browsing to https://aka.ms/devicelogin and entering an authorization code.

> **TIP**
> After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.
>
> To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

Install the machine learning extension

To install the machine learning extension, use the following command:

```
az extension add -n azure-cli-ml
```

If you get a message that the extension is already installed, use the following command to update to the latest version:

```
az extension update -n azure-cli-ml
```

Create a resource group

A resource group is a basic container of resources on the Azure platform. When working with the Azure Machine

Learning, the resource group will contain your Azure Machine Learning workspace. It will also contain other Azure services used by the workspace. For example, if you train your model using a cloud-based compute resource, that resource is created in the resource group.

To **create a new resource group**, use the following command. Replace `<resource-group-name>` with the name to use for this resource group. Replace `<location>` with the Azure region to use for this resource group:

> **TIP**
> You should select a region where the Azure Machine Learning is available. For information, see Products available by region.

```
az group create --name <resource-group-name> --location <location>
```

The response from this command is similar to the following JSON:

```
{
  "id": "/subscriptions/<subscription-GUID>/resourceGroups/<resourcegroupname>",
  "location": "<location>",
  "managedBy": null,
  "name": "<resource-group-name>",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": null
}
```

For more information on working with resource groups, see az group.

Create a workspace

To create a new workspace, use the following command. Replace `<workspace-name>` with the name you want to use for this workspace. Replace `<resource-group-name>` with the name of the resource group:

```
az ml workspace create -w <workspace-name> -g <resource-group-name>
```

The output of this command is similar to the following JSON:

```
{
  "applicationInsights": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.insights/components/<application-insight-name>",
  "containerRegistry": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.containerregistry/registries/<acr-name>",
  "creationTime": "2019-08-30T20:24:19.6984254+00:00",
  "description": "",
  "friendlyName": "<workspace-name>",
  "id": "/subscriptions/<service-GUID>/resourceGroups/<resource-group-
name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>",
  "identityPrincipalId": "<GUID>",
  "identityTenantId": "<GUID>",
  "identityType": "SystemAssigned",
  "keyVault": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.keyvault/vaults/<key-vault-name>",
  "location": "<location>",
  "name": "<workspace-name>",
  "resourceGroup": "<resource-group-name>",
  "storageAccount": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.storage/storageaccounts/<storage-account-name>",
  "type": "Microsoft.MachineLearningServices/workspaces",
  "workspaceid": "<GUID>"
}
```

Connect local project to workspace

From a terminal or command prompt, use the following commands change directories to the `cli-train-deploy` directory, then connect to your workspace:

```
cd ~/MLOps/examples/cli-train-deploy
az ml folder attach -w <workspace-name> -g <resource-group-name>
```

The output of this command is similar to the following JSON:

```
{
  "Experiment name": "model-training",
  "Project path": "/home/user/MLOps/examples/cli-train-deploy",
  "Resource group": "<resource-group-name>",
  "Subscription id": "<subscription-id>",
  "Workspace name": "<workspace-name>"
}
```

This command creates a `.azureml/config.json` file, which contains information needed to connect to your workspace. The rest of the `az ml` commands used in this tutorial will use this file, so you don't have to add the workspace and resource group to all commands.

Create the compute target for training

This example uses an Azure Machine Learning Compute cluster to train the model. To create a new compute cluster, use the following command:

```
az ml computetarget create amlcompute -n cpu-cluster --max-nodes 4 --vm-size Standard_D2_V2
```

The output of this command is similar to the following JSON:

```
{
  "location": "<location>",
  "name": "cpu-cluster",
  "provisioningErrors": null,
  "provisioningState": "Succeeded"
}
```

This command creates a new compute target named `cpu-cluster`, with a maximum of four nodes. The VM size selected provides a VM with a GPU resource. For information on the VM size, see [VM types and sizes].

> **IMPORTANT**
> The name of the compute target (`cpu-cluster` in this case), is important; it is referenced by the `.azureml/mnist.runconfig` file used in the next section.

Define the dataset

To train a model, you can provide the training data using a dataset. To create a dataset from the CLI, you must provide a dataset definition file. The `dataset.json` file provided in the repo creates a new dataset using the MNIST data. The dataset it creates is named `mnist-dataset`.

To register the dataset using the `dataset.json` file, use the following command:

```
az ml dataset register -f dataset.json --skip-validation
```

The output of this command is similar to the following JSON:

```
{
  "definition": [
    "GetFiles"
  ],
  "registration": {
    "description": "mnist dataset",
    "id": "a13a4034-02d1-40bd-8107-b5d591a464b7",
    "name": "mnist-dataset",
    "tags": {
      "sample-tag": "mnist"
    },
    "version": 1,
    "workspace": "Workspace.create(name='myworkspace', subscription_id='mysubscriptionid',
resource_group='myresourcegroup')"
  },
  "source": [
    "http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz",
    "http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz",
    "http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz",
    "http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz"
  ]
}
```

> **IMPORTANT**
> Copy the value of the `id` entry, as it is used in the next section.

To see a more comprehensive template for a dataset, use the following command:

```
az ml dataset register --show-template
```

Reference the dataset

To make the dataset available in the training environment, you must reference it from the runconfig file. The `.azureml/mnist.runconfig` file contains the following YAML entries:

```
# The arguments to the script file.
arguments:
- --data-folder
- DatasetConsumptionConfig:mnist

.....

# The configuration details for data.
data:
  mnist:
# Data Location
    dataLocation:
# the Dataset used for this run.
      dataset:
# Id of the dataset.
        id: a13a4034-02d1-40bd-8107-b5d591a464b7
# the DataPath used for this run.
      datapath:
# Whether to create new folder.
    createOutputDirectories: false
# The mode to handle
    mechanism: mount
# Point where the data is download or mount or upload.
    environmentVariableName: mnist
# relative path where the data is download or mount or upload.
    pathOnCompute:
# Whether to overwrite the data if existing.
    overwrite: false
```

Change the value of the `id` entry to match the value returned when you registered the dataset. This value is used to load the data into the compute target during training.

This YAML results in the following actions during training:

- Mounts the dataset (based on the ID of the dataset) in the training environment, and stores the path to the mount point in the `mnist` environment variable.
- Passes the location of the data (mount point) inside the training environment to the script using the `--data-folder` argument.

The runconfig file also contains information used to configure the environment used by the training run. If you inspect this file, you'll see that it references the `cpu-compute` compute target you created earlier. It also lists the number of nodes to use when training (`"nodeCount": "4"`), and contains a `"condaDependencies"` section that lists the Python packages needed to run the training script.

> **TIP**
> While it is possible to manually create a runconfig file, the one in this example was created using the `generate-runconfig.py` file included in the repository. This file gets a reference to the registered dataset, creates a run config programatically, and then persists it to file.

For more information on run configuration files, see Set up and use compute targets for model training. For a

complete JSON reference, see the runconfigschema.json.

Submit the training run

To start a training run on the `cpu-cluster` compute target, use the following command:

```
az ml run submit-script -c mnist -e myexperiment --source-directory scripts -t runoutput.json
```

This command specifies a name for the experiment (`myexperiment`). The experiment stores information about this run in the workspace.

The `-c mnist` parameter specifies the `.azureml/mnist.runconfig` file.

The `-t` parameter stores a reference to this run in a JSON file, and will be used in the next steps to register and download the model.

As the training run processes, it streams information from the training session on the remote compute resource. Part of the information is similar to the following text:

```
Predict the test set
Accuracy is 0.9185
```

This text is logged from the training script and displays the accuracy of the model. Other models will have different performance metrics.

If you inspect the training script, you'll notice that it also uses the alpha value when it stores the trained model to `outputs/sklearn_mnist_model.pkl`.

The model was saved to the `./outputs` directory on the compute target where it was trained. In this case, the Azure Machine Learning Compute instance in the Azure cloud. The training process automatically uploads the contents of the `./outputs` directory from the compute target where training occurs to your Azure Machine Learning workspace. It's stored as part of the experiment (`myexperiment` in this example).

Register the model

To register the model directly from the stored version in your experiment, use the following command:

```
az ml model register -n mymodel -f runoutput.json --asset-path "outputs/sklearn_mnist_model.pkl" -t
registeredmodel.json
```

This command registers the `outputs/sklearn_mnist_model.pkl` file created by the training run as a new model registration named `mymodel`. The `--assets-path` references a path in an experiment. In this case, the experiment and run information are loaded from the `runoutput.json` file created by the training command. The `-t registeredmodel.json` creates a JSON file that references the new registered model created by this command, and is used by other CLI commands that work with registered models.

The output of this command is similar to the following JSON:

```
{
  "createdTime": "2019-09-19T15:25:32.411572+00:00",
  "description": "",
  "experimentName": "myexperiment",
  "framework": "Custom",
  "frameworkVersion": null,
  "id": "mymodel:1",
  "name": "mymodel",
  "properties": "",
  "runId": "myexperiment_1568906070_5874522d",
  "tags": "",
  "version": 1
}
```

Model versioning

Note the version number returned for the model. The version is incremented each time you register a new model with this name. For example, you can download the model and register it from a local file by using the following commands:

```
az ml model download -i "mymodel:1" -t .
az ml model register -n mymodel -p "sklearn_mnist_model.pkl"
```

The first command downloads the registered model to the current directory. The file name is `sklearn_mnist_model.pkl`, which is the file referenced when you registered the model. The second command registers the local model (`-p "sklearn_mnist_model.pkl"`) with the same name as the previous registration (`mymodel`). This time, the JSON data returned lists the version as 2.

Deploy the model

To deploy a model, use the following command:

```
az ml model deploy -n myservice -m "mymodel:1" --ic inferenceConfig.json --dc aciDeploymentConfig.yml
```

> **NOTE**
> You may receive a warning about "Failed to check LocalWebservice existence" or "Failed to create Docker client". You can safely ignore this, as you are not deploying a local web service.

This command deploys a new service named `myservice`, using version 1 of the model that you registered previously.

The `inferenceConfig.yml` file provides information on how to use the model for inference. For example, it references the entry script (`score.py`) and software dependencies.

For more information on the structure of this file, see the Inference configuration schema. For more information on entry scripts, see Deploy models with the Azure Machine Learning.

The `aciDeploymentConfig.yml` describes the deployment environment used to host the service. The deployment configuration is specific to the compute type that you use for the deployment. In this case, an Azure Container Instance is used. For more information, see the Deployment configuration schema.

It will take several minutes before the deployment process completes.

TIP

In this example, Azure Container Instances is used. Deployments to ACI automatically create the needed ACI resource. If you were to instead deploy to Azure Kubernetes Service, you must create an AKS cluster ahead of time and specify it as part of the `az ml model deploy` command. For an example of deploying to AKS, see Deploy a model to an Azure Kubernetes Service cluster.

After several minutes, information similar to the following JSON is returned:

```
ACI service creation operation finished, operation "Succeeded"
{
  "computeType": "ACI",
  {...ommitted for space...}
  "runtimeType": null,
  "scoringUri": "http://6c061467-4e44-4f05-9db5-9f9a22ef7a5d.eastus2.azurecontainer.io/score",
  "state": "Healthy",
  "tags": "",
  "updatedAt": "2019-09-19T18:22:32.227401+00:00"
}
```

The scoring URI

The `scoringUri` returned from the deployment is the REST endpoint for a model deployed as a web service. You can also get this URI by using the following command:

```
az ml service show -n myservice
```

This command returns the same JSON document, including the `scoringUri`.

The REST endpoint can be used to send data to the service. For information on creating a client application that sends data to the service, see Consume an Azure Machine Learning model deployed as a web service

Send data to the service

While you can create a client application to call the endpoint, the machine learning CLI provides a utility that can act as a test client. Use the following command to send data in the `testdata.json` file to the service:

```
az ml service run -n myservice -d @testdata.json
```

TIP

If you use PowerShell, use the following command instead:

```
az ml service run -n myservice -d `@testdata.json
```

The response from the command is similar to `[ 3 ]`.

Clean up resources

IMPORTANT

The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

Delete deployed service

If you plan on continuing to use the Azure Machine Learning workspace, but want to get rid of the deployed service to reduce costs, use the following command:

```
az ml service delete -n myservice
```

This command returns a JSON document that contains the name of the deleted service. It may take several minutes before the service is deleted.

Delete the training compute

If you plan on continuing to use the Azure Machine Learning workspace, but want to get rid of the `cpu-cluster` compute target created for training, use the following command:

```
az ml computetarget delete -n cpu-cluster
```

This command returns a JSON document that contains the ID of the deleted compute target. It may take several minutes before the compute target has been deleted.

Delete everything

If you don't plan to use the resources you created, delete them so you don't incur additional charges.

To delete the resource group, and all the Azure resources created in this document, use the following command. Replace `<resource-group-name>` with the name of the resource group you created earlier:

```
az group delete -g <resource-group-name> -y
```

Next steps

In this Azure Machine Learning tutorial, you used the machine learning CLI for the following tasks:

- Install the machine learning extension
- Create an Azure Machine Learning workspace
- Create the compute resource used to train the model
- Define and register the dataset used to train the model
- Start a training run
- Register and download a model
- Deploy the model as a web service
- Score data using the web service

For more information on using the CLI, see Use the CLI extension for Azure Machine Learning.

Set up Azure Machine Learning Visual Studio Code extension

4/13/2020 • 3 minutes to read • Edit Online

Learn how to install and run scripts using the Azure Machine Learning Visual Studio Code extension.

In this tutorial, you learn the following tasks:

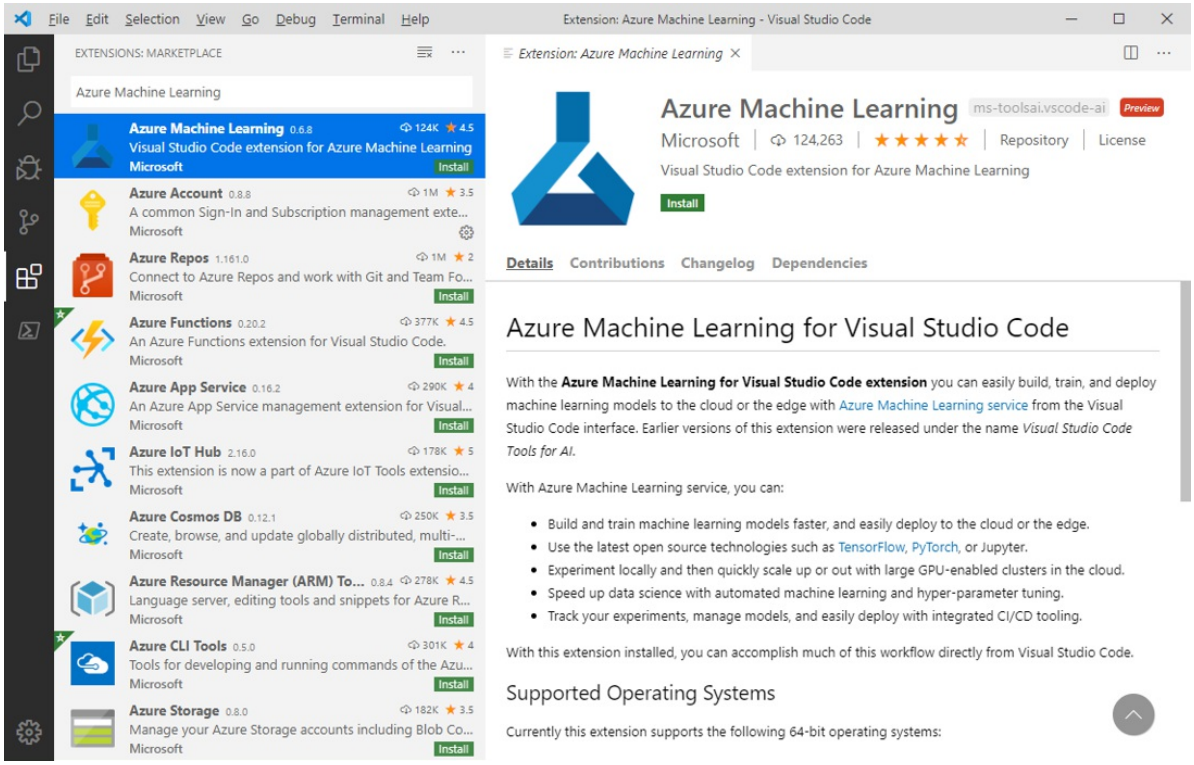
- Install the Azure Machine Learning Visual Studio Code extension
- Sign into your Azure account from Visual Studio Code
- Use the Azure Machine Learning extension to run a sample script

Prerequisites

- Azure subscription. If you don't have one, sign up to try the free or paid version of Azure Machine Learning.
- Visual Studio Code. If you don't have it, install it.
- Python 3

Install the extension

1. Open Visual Studio Code.
2. Select **Extensions** icon from the **Activity Bar** to open the Extensions view.
3. In the Extensions view, search for "Azure Machine Learning".
4. Select **Install**.



> **NOTE**
> Alternatively, you can install the Azure Machine Learning extension via the Visual Studio Marketplace by downloading the installer directly.

The rest of the steps in this tutorial have been tested with **version 0.6.8** of the extension.

Sign in to your Azure Account

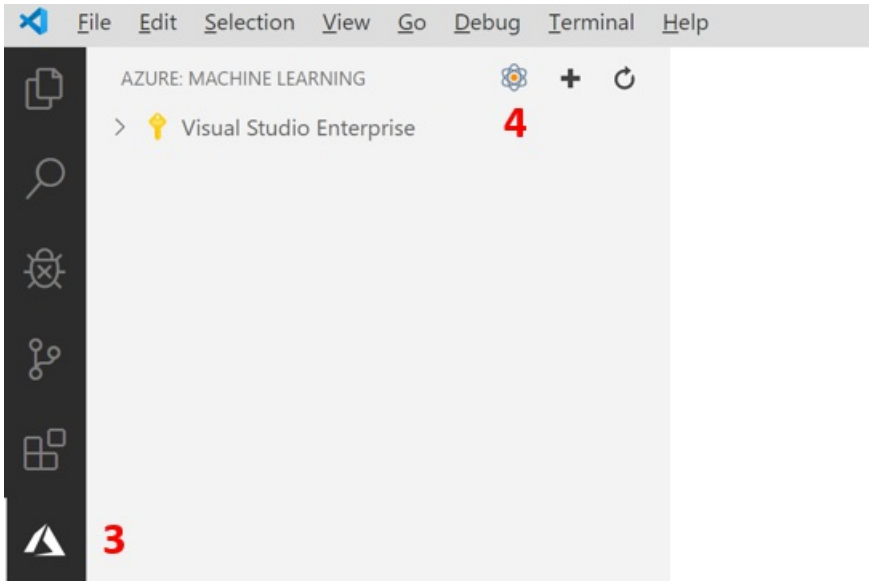
In order to provision resources and run workloads on Azure, you have to sign in with your Azure account credentials. To assist with account management, Azure Machine Learning automatically installs the Azure Account extension. Visit the following site to learn more about the Azure Account extension.

1. Open the command palette by selecting **View** > **Command Palette** from the menu bar.
2. Enter the command "Azure: Sign In" into the command palette to start the sign in process.

Run a machine learning model training script in Azure

Now that you have signed into Azure with your account credentials, Use the steps in this section to learn how to use the extension to train a machine learning model.

1. Download and unzip the VS Code Tools for AI repository anywhere on your computer.
2. Open the `mnist-vscode-docs-sample` directory in Visual Studio Code.
3. Select the **Azure** icon in the Activity Bar.
4. Select the **Run Experiment** icon at the top of the Azure Machine Learning View.



5. When the command palette expands, follow the prompts.
   a. Select your Azure subscription.
   b. From the list of environments, select **Conda dependencies file**.
   c. Press **Enter** to browse the Conda dependencies file. This file contains the dependencies required to run your script. In this case, the dependencies file is the `env.yml` file inside the `mnist-vscode-docs-sample` directory.
   d. Press **Enter** to browse the training script file. This is the file that contains code to a machine learning model that categorize images of handwritten digits. In this case, the script to train the model is the `train.py` file inside the `mnist-vscode-docs-sample` directory.

6. At this point, a configuration file similar to the one below appears in the text editor. The configuration contains the information required to run the training job like the file that contains the code to train the model and any Python dependencies specified in the previous step.

```
{
    "workspace": "WS04131142",
    "resourceGroup": "WS04131142-rg1",
    "location": "South Central US",
    "experiment": "WS04131142-exp1",
    "compute": {
        "name": "WS04131142-com1",
        "vmSize": "Standard_D1_v2, Cores: 1; RAM: 3.5GB;"
    },
    "runConfiguration": {
        "filename": "WS04131142-com1-rc1",
        "environment": {
            "name": "WS04131142-env1",
            "conda_dependencies": [
                "python=3.6.2",
                "tensorflow=1.15.0",
                "pip"
            ],
            "pip_dependencies": [
                "azureml-defaults"
            ],
            "environment_variables": {}
        }
    }
}
```

7. Once you're satisfied with your configuration, submit your experiment by opening the command palette and entering the following command:

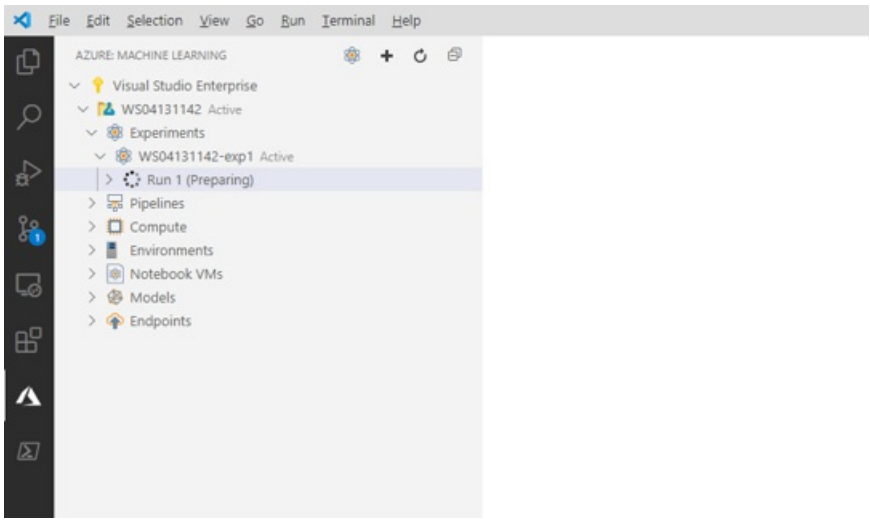
```
Azure ML: Submit Experiment
```

   This sends the `train.py` and configuration file to your Azure Machine Learning workspace. The training job is then started on a compute resource in Azure.

Track the progress of the training script

Running your script can take several minutes. To track its progress:

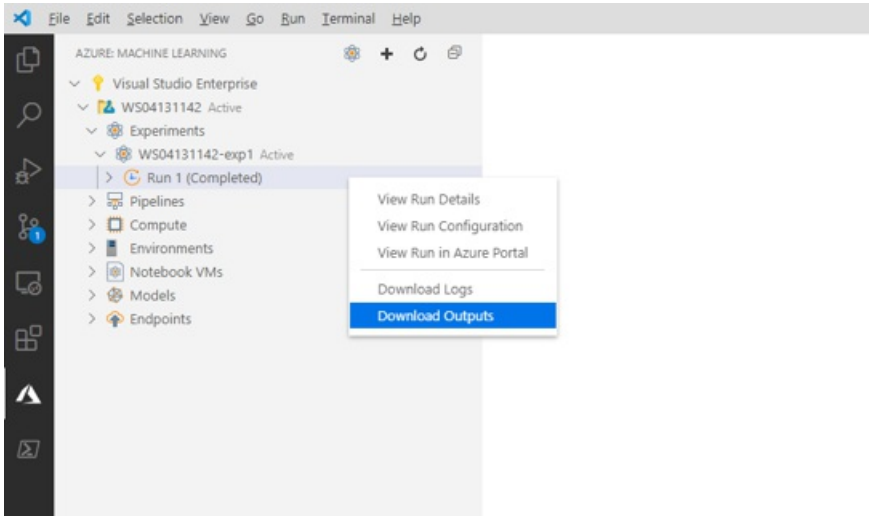
1. Select the **Azure** icon from the activity bar.
2. Expand your subscription node.
3. Expand your currently running experiment's node. This is located inside the `{workspace}/Experiments/{experiment}` node where the values for your workspace and experiment are the same as the properties defined in the configuration file.
4. All of the runs for the experiment are listed, as well as their status. To get the most recent status, click the refresh icon at the top of the Azure Machine Learning View.



Download the trained model

When the experiment run is complete, the output is a trained model. To download the outputs locally:

1. Right-click the most recent run and select **Download Outputs**.



2. Select a location where to save the outputs to.
3. A folder with the name of your run is downloaded locally. Navigate to it.
4. The model files are inside the `outputs/outputs/model` directory.

Next steps

- Tutorial: Train and deploy an image classification TensorFlow model using the Azure Machine Learning Visual Studio Code Extension.

Train and deploy an image classification TensorFlow model using the Azure Machine Learning Visual Studio Code Extension

4/13/2020 • 9 minutes to read • Edit Online

Learn how to train and deploy an image classification model to recognize hand-written numbers using TensorFlow and the Azure Machine Learning Visual Studio Code Extension.

In this tutorial, you learn the following tasks:

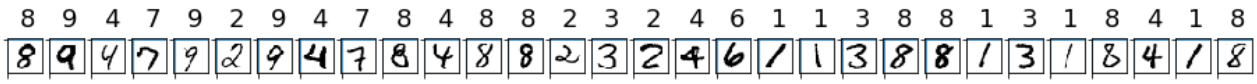
- Understand the code
- Create a workspace
- Create an experiment
- Configure Computer Targets
- Run a configuration file
- Train a model
- Register a model
- Deploy a model

Prerequisites

- Azure subscription. If you don't have one, sign up to try the free or paid version of Azure Machine Learning.
- Install Visual Studio Code, a lightweight, cross-platform code editor.
- Azure Machine Learning Studio Visual Studio Code extension. For install instructions see the Setup Azure Machine Learning Visual Studio Code extension tutorial

Understand the code

The code for this tutorial uses TensorFlow to train an image classification machine learning model that categorizes handwritten digits from 0-9. It does so by creating a neural network that takes the pixel values of 28 px x 28 px image as input and outputs a list of 10 probabilities, one for each of the digits being classified. Below is a sample of what the data looks like.

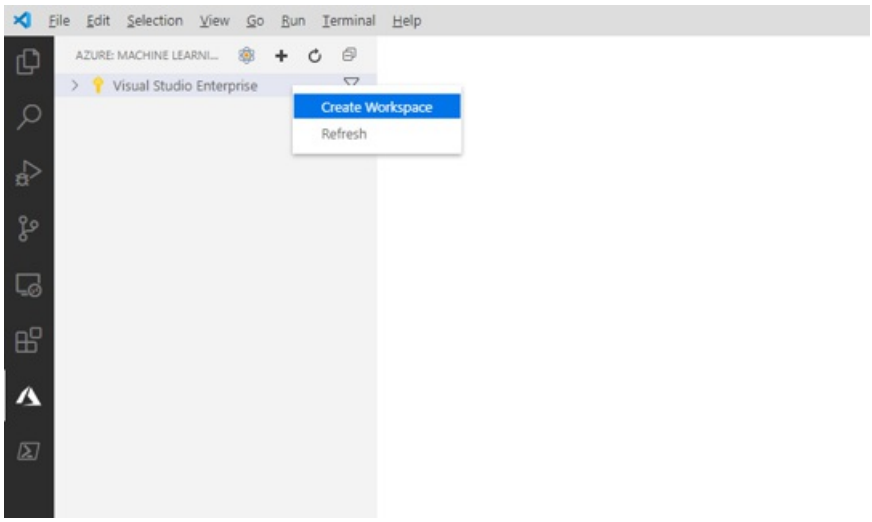


Get the code for this tutorial by downloading and unzipping the VS Code Tools for AI repository anywhere on your computer.

Create a workspace

The first thing you have to do to build an application in Azure Machine Learning is to create a workspace. A workspace contains the resources to train models as well as the trained models themselves. For more information, see what is a workspace.

1. On the Visual Studio Code activity bar, select the **Azure** icon to open the Azure Machine Learning view.
2. Right-click your Azure subscription and select **Create Workspace**.



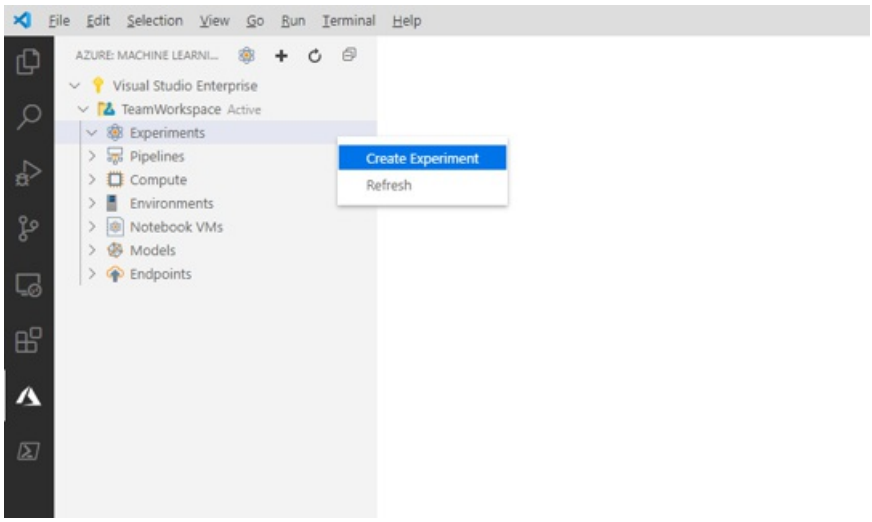
3. By default a name is generated containing the date and time of creation. In the text input box, change the name to "TeamWorkspace" and press **Enter**.
4. Select **Create a new resource group**.
5. Name your resource group "TeamWorkspace-rg" and press **Enter**.
6. Choose a location for your workspace. It's recommended to choose a location that is closest to the location you plan to deploy your model. For example, "West US 2".
7. When prompted to select the type of workspace, select **Basic** to create a basic workspace. For more information on different workspace offerings, see Azure Machine Learning overview.

At this point, a request to Azure is made to create a new workspace in your account. After a few minutes, the new workspace appears in your subscription node.

Create an experiment

One or more experiments can be created in your workspace to track and analyze individual model training runs. Runs can be done in the Azure cloud or on your local machine.

1. On the Visual Studio Code activity bar, select the **Azure** icon. The Azure Machine Learning view appears.
2. Expand your subscription node.
3. Expand the **TeamWorkspace** node.
4. Right-click the **Experiments** node.
5. Select **Create Experiment** from the context menu.



6. Name your experiment "MNIST" and press **Enter** to create the new experiment.

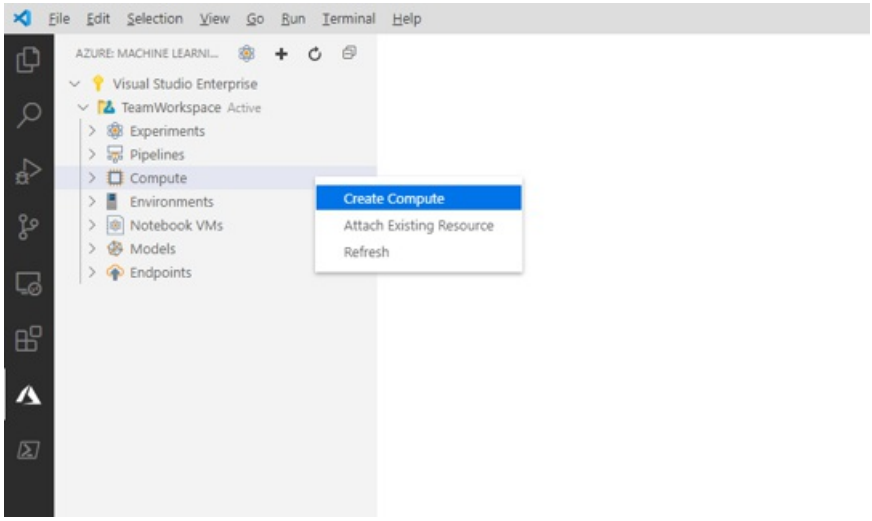
Like workspaces, a request is sent to Azure to create an experiment with the provided configurations. After a few minutes, the new experiment appears in the *Experiments* node of your workspace.

Configure Compute Targets

A compute target is the computing resource or environment where you run scripts and deploy trained models. For more information, see the Azure Machine Learning compute targets documentation.

To create a compute target:

1. On the Visual Studio Code activity bar, select the **Azure** icon. The Azure Machine Learning view appears.
2. Expand your subscription node.
3. Expand the **TeamWorkspace** node.
4. Under the workspace node, right-click the **Compute** node and choose **Create Compute**.



5. Select **Azure Machine Learning Compute (AmlCompute)**. Azure Machine Learning Compute is a managed-compute infrastructure that allows the user to easily create a single or multi-node compute that can be used with other users in your workspace.
6. Choose a VM size. Select **Standard_F2s_v2** from the list of options. The size of your VM has an impact on the amount of time it takes to train your models. For more information on VM sizes, see sizes for Linux virtual machines in Azure.
7. Name your compute "TeamWkspc-com" and press **Enter** to create your compute.

A file appears in VS Code with content similar to the one below:

```
{
    "location": "westus2",
    "tags": {},
    "properties": {
        "computeType": "AmlCompute",
        "description": "",
        "properties": {
            "vmSize": "Standard_F2s_v2",
            "vmPriority": "dedicated",
            "scaleSettings": {
                "maxNodeCount": 4,
                "minNodeCount": 0,
                "nodeIdleTimeBeforeScaleDown": 120
            },
            "userAccountCredentials": {
                "adminUserName": "",
                "adminUserPassword": "",
                "adminUserSshPublicKey": ""
            },
            "subnetName": "",
            "vnetName": "",
            "vnetResourceGroupName": "",
            "remoteLoginPortPublicAccess": ""
        }
    }
}
```

8. When satisfied with the configuration, open the command palette by selecting **View > Command Palette**.
9. Enter the following command into the command palette to save your run configuration file.

```
Azure ML: Save and Continue
```

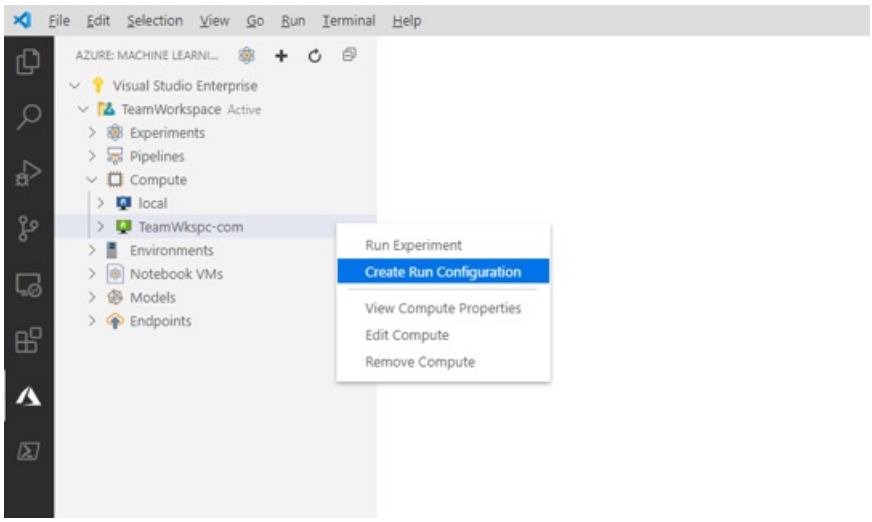
After a few minutes, the new compute target appears in the *Compute* node of your workspace.

Create a run configuration

When you submit a training run to a compute target, you also submit the configuration needed to run the training job. For example, the script that contains the training code and the Python dependencies needed to run it.

To create a run configuration:

1. On the Visual Studio Code activity bar, select the **Azure** icon. The Azure Machine Learning view appears.
2. Expand your subscription node.
3. Expand the **TeamWorkspace > Compute** node.
4. Under the compute node, right-click the **TeamWkspc-com** compute node and choose **Create Run Configuration**.



5. Name your run configuration "MNIST-rc" and press **Enter** to create your run configuration.

6. Then, select **Create new Azure ML Environment**. Environments define the dependencies required to run your scripts.

7. Name your environment "MNIST-env" and press **Enter**.

8. Select **Conda dependencies file** from the list.

9. Press **Enter** to browse the Conda dependencies file. In this case, the dependencies file is the `env.yml` file inside the `vscode-tools-for-ai/mnist-vscode-docs-sample` directory.

   A file appears in VS Code with content similar to the one below:

```
{
    "name": "MNIST-env",
    "version": "1",
    "python": {
        "interpreterPath": "python",
        "userManagedDependencies": false,
        "condaDependencies": {
            "name": "vs-code-azure-ml-tutorial",
            "channels": [
                "defaults"
            ],
            "dependencies": [
                "python=3.6.2",
                "tensorflow=1.15.0",
                "pip",
                {
                    "pip": [
                        "azureml-defaults"
                    ]
                }
            ]
        },
        "baseCondaEnvironment": null
    },
    "environmentVariables": {},
    "docker": {
        "baseImage": "mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04",
        "baseDockerfile": null,
        "baseImageRegistry": {
            "address": null,
            "username": null,
            "password": null
        },
        "enabled": false,
        "arguments": []
    },
    "spark": {
        "repositories": [],
        "packages": [],
        "precachePackages": true
    },
    "inferencingStackVersion": null
}
```

10. Once you're satisfied with your configuration, save it by opening the command palette and entering the following command:

    ```
    Azure ML: Save and Continue
    ```

11. Press **Enter** to browse the script file to run on the compute. In this case, the script to train the model is the `train.py` file inside the `vscode-tools-for-ai/mnist-vscode-docs-sample` directory.

    A file called `MNIST-rc.runconfig` appears in VS Code with content similar to the one below:

```
{
    "script": "train.py",
    "framework": "Python",
    "communicator": "None",
    "target": "TeamWkspc-com",
    "environment": {
        "name": "MNIST-env",
        "version": "1",
        "python": {
            "interpreterPath": "python",
            "userManagedDependencies": false,
            "condaDependencies": {
                "name": "vs-code-azure-ml-tutorial",
                "channels": [
                    "defaults"
                ],
                "dependencies": [
                    "python=3.6.2",
                    "tensorflow=1.15.0",
                    "pip",
                    {
                        "pip": [
                            "azureml-defaults"
                        ]
                    }
                ]
            },
            "baseCondaEnvironment": null
        },
        "environmentVariables": {},
        "docker": {
            "baseImage": "mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04",
            "baseDockerfile": null,
            "baseImageRegistry": {
                "address": null,
                "username": null,
                "password": null
            },
            "enabled": false,
            "arguments": []
        },
        "spark": {
            "repositories": [],
            "packages": [],
            "precachePackages": true
        },
        "inferencingStackVersion": null
    },
    "history": {
        "outputCollection": true,
        "snapshotProject": false,
        "directoriesToWatch": [
            "logs"
        ]
    }
}
```


12. Once you're satisfied with your configuration, save it by opening the command palette and entering the following command:

```
Azure ML: Save and Continue
```

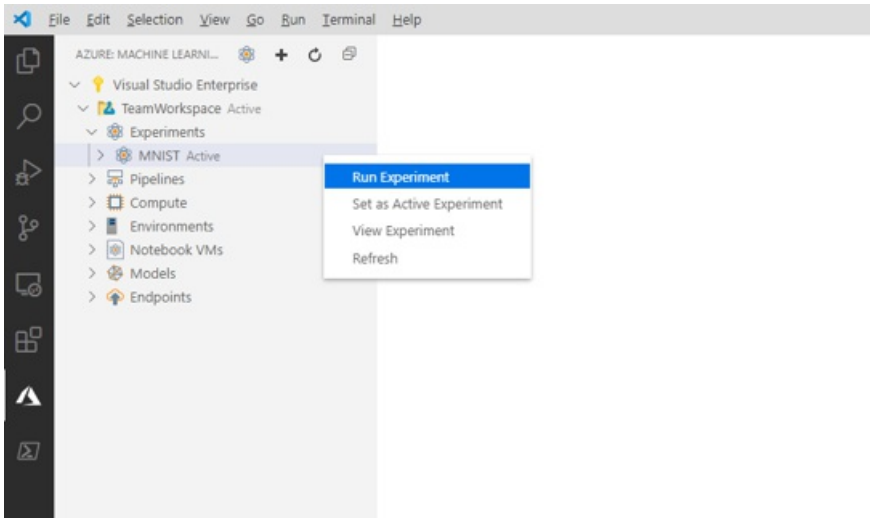
The `MNIST-rc` run configuration is added under the *TeamWkspc-com* compute node and the `MNIST-env` environment configuration is added under the *Environments* node.

Train the model

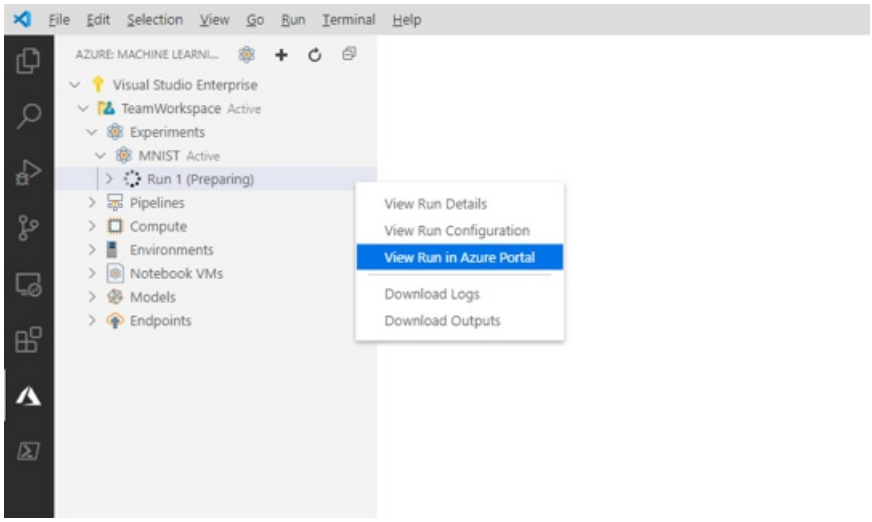
During the training process, a TensorFlow model is created by processing the training data and learning patterns embedded within it for each of the respective digits being classified.

To run an Azure Machine Learning experiment:

1. On the Visual Studio Code activity bar, select the **Azure** icon. The Azure Machine Learning view appears.
2. Expand your subscription node.
3. Expand the **TeamWorkspace > Experiments** node.
4. Right-click the **MNIST** experiment.
5. Select **Run Experiment**.



6. From the list of compute target options, select the **TeamWkspc-com** compute target.
7. Then, select the **MNIST-rc** run configuration.
8. At this point, a request is sent to Azure to run your experiment on the selected compute target in your workspace. This process takes several minutes. The amount of time to run the training job is impacted by several factors like the compute type and training data size. To track the progress of your experiment, right-click the current run node and select **View Run in Azure portal**.
9. When the dialog requesting to open an external website appears, select **Open**.



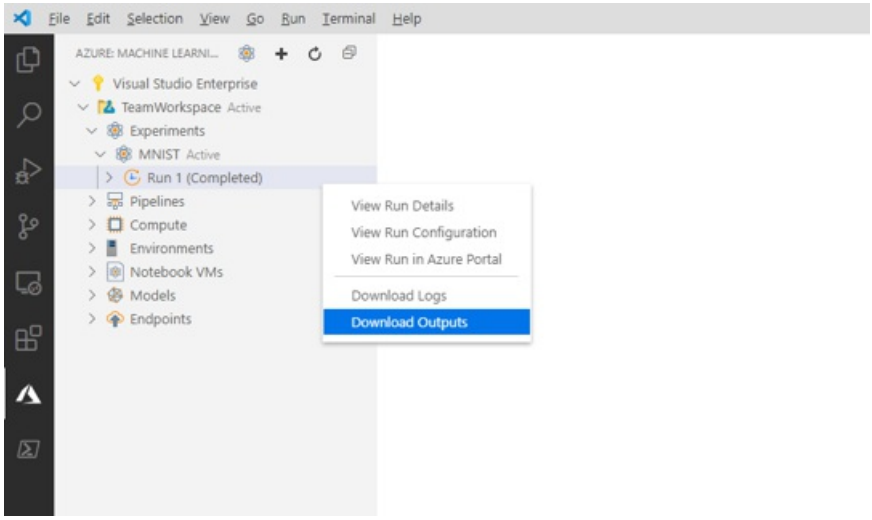
When the model is done training, the status label next to the run node updates to "Completed".

Register the model

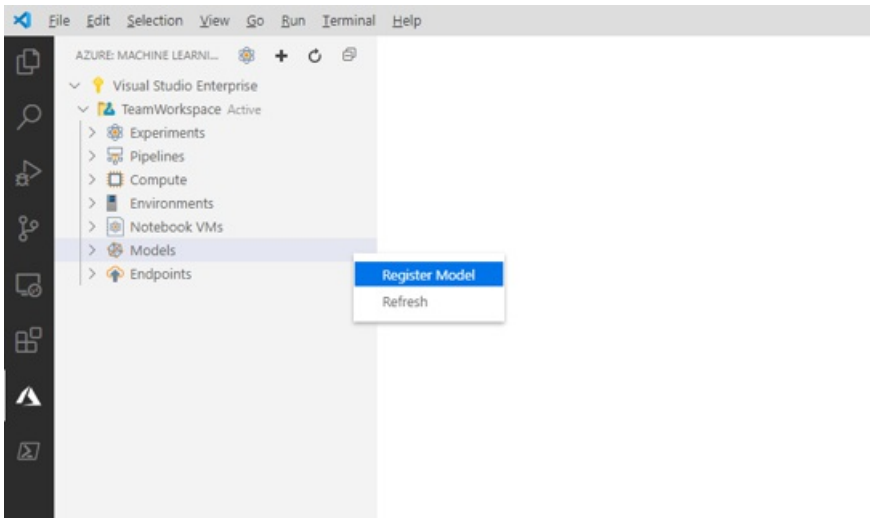
Now that you've trained your model, you can register it in your workspace.

To register your model:

1. On the Visual Studio Code activity bar, select the **Azure** icon. The Azure Machine Learning view appears.
2. Expand your subscription node.
3. Expand the **TeamWorkspace > Experiments > MNIST** node.
4. Get the model outputs generated from training the model. Right-click the **Run 1** run node and select **Download outputs**.



5. Choose the directory to save the downloaded outputs to. By default, the outputs are placed in the directory currently opened in Visual Studio Code.
6. Right-click the **Models** node and choose **Register Model**.



7. Name your model "MNIST-TensorFlow-model" and press **Enter**.
8. A TensorFlow model is made up of several files. Select **Model folder** as the model path format from the list of options.
9. Select the `azureml_outputs/Run_1/outputs/outputs/model` directory.

   A file containing your model configurations appears in Visual Studio Code with similar content to the one below:

```
{
    "modelName": "MNIST-TensorFlow-model",
    "tags": {
        "": ""
    },
    "modelPath": "c:\\Dev\\vscode-tools-for-ai\\mnist-vscode-docs-
sample\\azureml_outputs\\Run_1\\outputs\\outputs\\model",
    "description": ""
}
```

10. Once you're satisfied with your configuration, save it by opening the command palette and entering the following command:

```
Azure ML: Save and Continue
```

After a few minutes, the model appears under the *Models* node.

Deploy the model

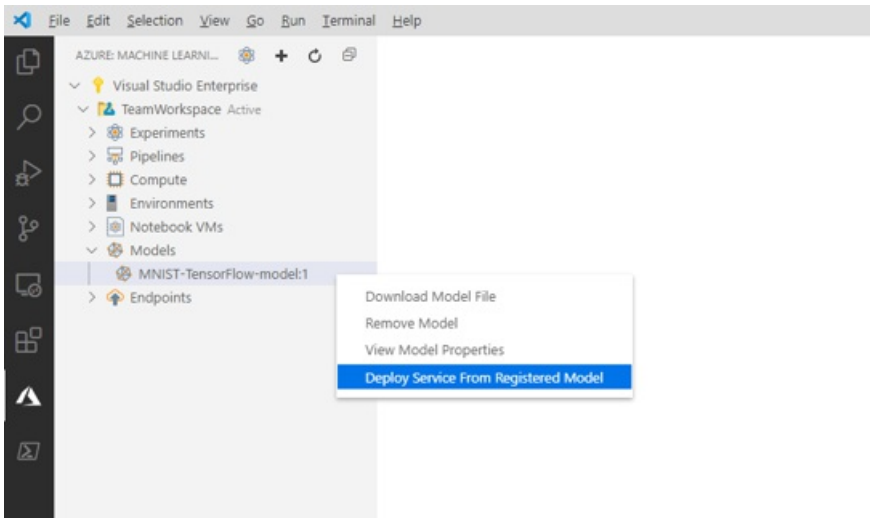
In Visual Studio Code, you can deploy your model as a web service to:

- Azure Container Instances (ACI).
- Azure Kubernetes Service (AKS).

You don't need to create an ACI container to test in advance, because ACI containers are created as needed. However, you do need to configure AKS clusters in advance. For more information on deployment options, see deploy models with Azure Machine Learning .

To deploy a web service as an ACI :

1. On the Visual Studio Code activity bar, select the **Azure** icon. The Azure Machine Learning view appears.
2. Expand your subscription node.
3. Expand the **TeamWorkspace > Models** node.
4. Right-click the **MNIST-TensorFlow-model** and select **Deploy Service from Registered Model**.



5. Select **Azure Container Instances**.
6. Name your service "mnist-tensorflow-svc" and press **Enter**.
7. Choose the script to run in the container by pressing **Enter** in the input box and browsing for the `score.py`

file in the `mnist-vscode-docs-sample` directory.

8. Provide the dependencies needed to run the script by pressing **Enter** in the input box and browsing for the `env.yml` file in the `mnist-vscode-docs-sample` directory.

   A file containing your model configurations appears in Visual Studio Code with similar content to the one below:

```
{
    "name": "mnist-tensorflow-svc",
    "imageConfig": {
        "runtime": "python",
        "executionScript": "score.py",
        "dockerFile": null,
        "condaFile": "env.yml",
        "dependencies": [],
        "schemaFile": null,
        "enableGpu": false,
        "description": ""
    },
    "deploymentConfig": {
        "cpu_cores": 1,
        "memory_gb": 10,
        "tags": {
            "": ""
        },
        "description": ""
    },
    "deploymentType": "ACI",
    "modelIds": [
        "MNIST-TensorFlow-model:1"
    ]
}
```

9. Once you're satisfied with your configuration, save it by opening the command palette and entering the following command:

```
Azure ML: Save and Continue
```

At this point, a request is sent to Azure to deploy your web service. This process takes several minutes. Once deployed, the new service appears under the *Endpoints* node.

Next steps

- For a walkthrough of how to train with Azure Machine Learning outside of Visual Studio Code, see Tutorial: Train models with Azure Machine Learning.
- For a walkthrough of how to edit, run, and debug code locally, see the Python hello-world tutorial.

Explore Azure Machine Learning with Jupyter notebooks

3/6/2020 • 2 minutes to read • Edit Online

The example Azure Machine Learning Notebooks repository includes the latest Azure Machine Learning Python SDK samples. These Juypter notebooks are designed to help you explore the SDK and serve as models for your own machine learning projects.

This article shows you how to access the repository from the following environments:

- Azure Machine Learning compute instance
- Bring your own notebook server
- Data Science Virtual Machine

> **NOTE**
> Once you've cloned the repository, you'll find tutorial notebooks in the **tutorials** folder and feature-specific notebooks in the **how-to-use-azureml** folder.

Get samples on Azure Machine Learning compute instance

The easiest way to get started with the samples is to complete the Tutorial: Setup environment and workspace. Once completed, you'll have a dedicated notebook server pre-loaded with the SDK and the sample repository. No downloads or installation necessary.

Get samples on your notebook server

If you'd like to bring your own notebook server for local development, follow these steps:

1. Use the instructions at Azure Machine Learning SDK to install the Azure Machine Learning SDK for Python
2. Create an Azure Machine Learning workspace.
3. Write a configuration file file (**aml_config/config.json**).
4. Clone the GitHub repository.

```
git clone https://github.com/Azure/MachineLearningNotebooks.git
```

5. Start the notebook server from your cloned directory.

```
jupyter notebook
```

These instructions install the base SDK packages necessary for the quickstart and tutorial notebooks. Other sample notebooks may require you to install extra components. For more information, see Install the Azure Machine Learning SDK for Python.

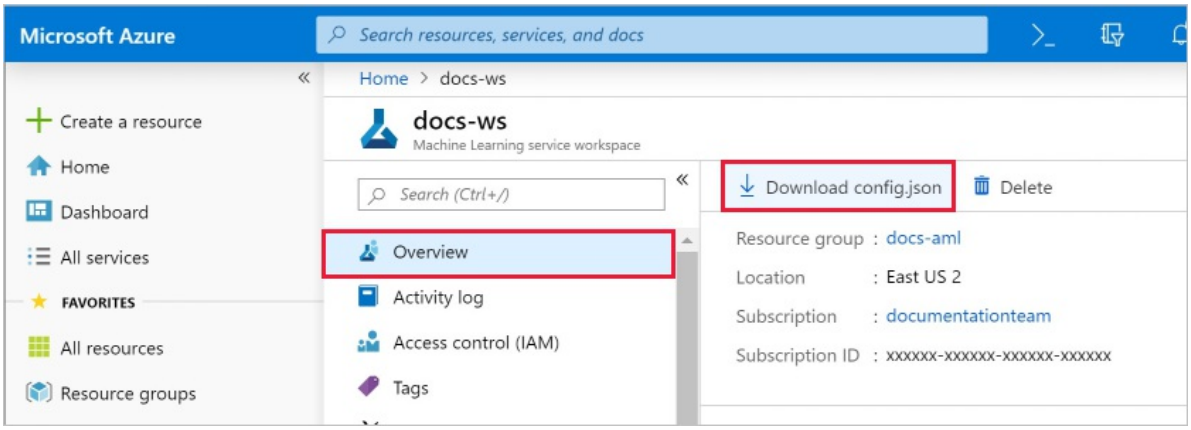
Get samples on DSVM

The Data Science Virtual Machine (DSVM) is a customized VM image built specifically for doing data science. If you create a DSVM, the SDK and notebook server are installed and configured for you. However, you'll still need to create a workspace and clone the sample repository.

1. Create an Azure Machine Learning workspace.
2. Clone the GitHub repository.

```
git clone https://github.com/Azure/MachineLearningNotebooks.git
```

3. Add a workspace configuration file to the cloned directory using either of these methods:
   - In the Azure portal, select **Download config.json** from the **Overview** section of your workspace.



   - Create a new workspace using code in the configuration.ipynb notebook in your cloned directory.
4. Start the notebook server from your cloned directory.

```
jupyter notebook
```

Next steps

Explore the sample notebooks to discover what Azure Machine Learning can do, or try these tutorials:

- Train and deploy an image classification model with MNIST
- Prepare data and use automated machine learning to train a regression model with the NYC taxi data set

Sample datasets in Azure Machine Learning designer (preview)

4/10/2020 • 4 minutes to read • Edit Online

When you create a new pipeline in Azure Machine Learning designer (preview), a number of sample datasets are included by default. These sample datasets are used by the sample pipelines in the designer homepage.

The sample datasets are available under **Datasets-Samples** category. You can find this in the module palette to the left of the canvas in the designer. You can use any of these datasets in your own pipeline by dragging it to the canvas.

Datasets

DATASET NAME	DATASET DESCRIPTION
Adult Census Income Binary Classification dataset	A subset of the 1994 Census database, using working adults over the age of 16 with an adjusted income index of > 100. **Usage**: Classify people using demographics to predict whether a person earns over 50K a year. **Related Research**: Kohavi, R., Becker, B., (1996). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science
Automobile price data (Raw)	Information about automobiles by make and model, including the price, features such as the number of cylinders and MPG, as well as an insurance risk score. The risk score is initially associated with auto price. It is then adjusted for actual risk in a process known to actuaries as symboling. A value of +3 indicates that the auto is risky, and a value of -3 that it is probably safe. **Usage**: Predict the risk score by features, using regression or multivariate classification. **Related Research**: Schlimmer, J.C. (1987). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.
CRM Appetency Labels Shared	Labels from the KDD Cup 2009 customer relationship prediction challenge (orange_small_train_appetency.labels).
CRM Churn Labels Shared	Labels from the KDD Cup 2009 customer relationship prediction challenge (orange_small_train_churn.labels).
CRM Dataset Shared	This data comes from the KDD Cup 2009 customer relationship prediction challenge (orange_small_train.data.zip). The dataset contains 50K customers from the French Telecom company Orange. Each customer has 230 anonymized features, 190 of which are numeric and 40 are categorical. The features are very sparse.
CRM Upselling Labels Shared	Labels from the KDD Cup 2009 customer relationship prediction challenge (orange_large_train_upselling.labels

DATASET NAME	DATASET DESCRIPTION
Flight Delays Data	Passenger flight on-time performance data taken from the TranStats data collection of the U.S. Department of Transportation (On-Time). The dataset covers the time period April-October 2013. Before uploading to the designer, the dataset was processed as follows: - The dataset was filtered to cover only the 70 busiest airports in the continental US - Canceled flights were labeled as delayed by more than 15 minutes - Diverted flights were filtered out - The following columns were selected: Year, Month, DayofMonth, DayOfWeek, Carrier, OriginAirportID, DestAirportID, CRSDepTime, DepDelay, DepDel15, CRSArrTime, ArrDelay, ArrDel15, Canceled
German Credit Card UCI dataset	The UCI Statlog (German Credit Card) dataset (Statlog+German+Credit+Data), using the german.data file. The dataset classifies people, described by a set of attributes, as low or high credit risks. Each example represents a person. There are 20 features, both numerical and categorical, and a binary label (the credit risk value). High credit risk entries have label = 2, low credit risk entries have label = 1. The cost of misclassifying a low risk example as high is 1, whereas the cost of misclassifying a high risk example as low is 5.
IMDB Movie Titles	The dataset contains information about movies that were rated in Twitter tweets: IMDB movie ID, movie name, genre, and production year. There are 17K movies in the dataset. The dataset was introduced in the paper "S. Dooms, T. De Pessemier and L. Martens. MovieTweetings: a Movie Rating Dataset Collected From Twitter. Workshop on Crowdsourcing and Human Computation for Recommender Systems, CrowdRec at RecSys 2013."
Movie Ratings	The dataset is an extended version of the Movie Tweetings dataset. The dataset has 170K ratings for movies, extracted from well-structured tweets on Twitter. Each instance represents a tweet and is a tuple: user ID, IMDB movie ID, rating, timestamp, number of favorites for this tweet, and number of retweets of this tweet. The dataset was made available by A. Said, S. Dooms, B. Loni and D. Tikk for Recommender Systems Challenge 2014.

DATASET NAME	DATASET DESCRIPTION
Weather Dataset	Hourly land-based weather observations from NOAA (merged data from 201304 to 201310). The weather data covers observations made from airport weather stations, covering the time period April-October 2013. Before uploading to the designer, the dataset was processed as follows: - Weather station IDs were mapped to corresponding airport IDs - Weather stations not associated with the 70 busiest airports were filtered out - The Date column was split into separate Year, Month, and Day columns - The following columns were selected: AirportID, Year, Month, Day, Time, TimeZone, SkyCondition, Visibility, WeatherType, DryBulbFarenheit, DryBulbCelsius, WetBulbFarenheit, WetBulbCelsius, DewPointFarenheit, DewPointCelsius, RelativeHumidity, WindSpeed, WindDirection, ValueForWindCharacter, StationPressure, PressureTendency, PressureChange, SeaLevelPressure, RecordType, HourlyPrecip, Altimeter
Wikipedia SP 500 Dataset	Data is derived from Wikipedia (https://www.wikipedia.org/) based on articles of each S&P 500 company, stored as XML data. Before uploading to the designer, the dataset was processed as follows: - Extract text content for each specific company - Remove wiki formatting - Remove non-alphanumeric characters - Convert all text to lowercase - Known company categories were added Note that for some companies an article could not be found, so the number of records is less than 500.

Next steps

- Learn the basics of predictive analytics and machine learning with Tutorial: Predict automobile price with the designer
- Learn how to modify existing designer samples to adapt them to your needs.

Designer sample pipelines

3/30/2020 • 4 minutes to read • Edit Online

Use the built-in examples in Azure Machine Learning designer to quickly get started building your own machine learning pipelines. The Azure Machine Learning designer GitHub repository contains detailed documentation to help you understand some common machine learning scenarios.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account.
- An Azure Machine Learning workspace with the Enterprise SKU.

How to use sample pipelines

The designer saves a copy of the sample pipelines to your studio workspace. You can edit the pipeline to adapt it to your needs and save it as your own. Use them as a starting point to jumpstart your projects.

Open a sample pipeline

1. Sign in to ml.azure.com, and select the workspace you want to work with.
2. Select **Designer**.
3. Select a sample pipeline under the **New pipeline** section.

   Select **Show more samples** for a complete list of samples.

Submit a pipeline run

To run a pipeline, you first have to set default compute target to run the pipeline on.

1. In the **Settings** pane to the right of the canvas, select **Select compute target**.
2. In the dialog that appears, select an existing compute target or create a new one. Select **Save**.
3. Select **Submit** at the top of the canvas to submit a pipeline run.

Depending on the sample pipeline and compute settings, runs may take some time to complete. The default compute settings have a minimum node size of 0, which means that the designer must allocate resources after being idle. Repeated pipeline runs will take less time since the compute resources are already allocated. Additionally, the designer uses cached results for each module to further improve efficiency.

Review the results

After the pipeline finishes running, you can review the pipeline and view the output for each module to learn more.

Use the following steps to view module outputs:

1. Select a module in the canvas.
2. In the module details pane to the right of the canvas, select **Outputs + logs**. Select the graph icon  to see the results of each module.

Use the samples as starting points for some of the most common machine learning scenarios.

Regression samples

Learn more about the built-in regression samples.

SAMPLE TITLE	DESCRIPTION
Sample 1: Regression - Automobile Price Prediction (Basic)	Predict car prices using linear regression.
Sample 2: Regression - Automobile Price Prediction (Advanced)	Predict car prices using decision forest and boosted decision tree regressors. Compare models to find the best algorithm.

Classification samples

Learn more about the built-in classification samples. You can learn more about the samples without documentation links by opening the samples and viewing the module comments instead.

SAMPLE TITLE	DESCRIPTION
Sample 3: Binary Classification with Feature Selection - Income Prediction	Predict income as high or low, using a two-class boosted decision tree. Use Pearson correlation to select features.
Sample 4: Binary Classification with custom Python script - Credit Risk Prediction	Classify credit applications as high or low risk. Use the Execute Python Script module to weight your data.
Sample 5: Binary Classification - Customer Relationship Prediction	Predict customer churn using two-class boosted decision trees. Use SMOTE to sample biased data.
Sample 7: Text Classification - Wikipedia SP 500 Dataset	Classify company types from Wikipedia articles with multiclass logistic regression.
Sample 12: Multiclass Classification - Letter Recognition	Create an ensemble of binary classifiers to classify written letters.

Recommender samples

Learn more about the built-in recommender samples. You can learn more about the samples without documentation links by opening the samples and viewing the module comments instead.

SAMPLE TITLE	DESCRIPTION
Sample 10: Recommendation - Movie Rating Tweets	Build a movie recommender engine from movie titles and rating.

Utility samples

Learn more about the samples that demonstrate machine learning utilities and features. You can learn more about the samples without documentation links by opening the samples and viewing the module comments instead.

SAMPLE TITLE	DESCRIPTION
Sample 6: Use custom R script - Flight Delay Prediction	
Sample 8: Cross Validation for Binary Classification - Adult Income Prediction	Use cross validation to build a binary classifier for adult income.

SAMPLE TITLE	DESCRIPTION
Sample 9: Permutation Feature Importance	Use permutation feature importance to compute importance scores for the test dataset.
Sample 11: Tune Parameters for Binary Classification - Adult Income Prediction	Use Tune Model Hyperparameters to find optimal hyperparameters to build a binary classifier.

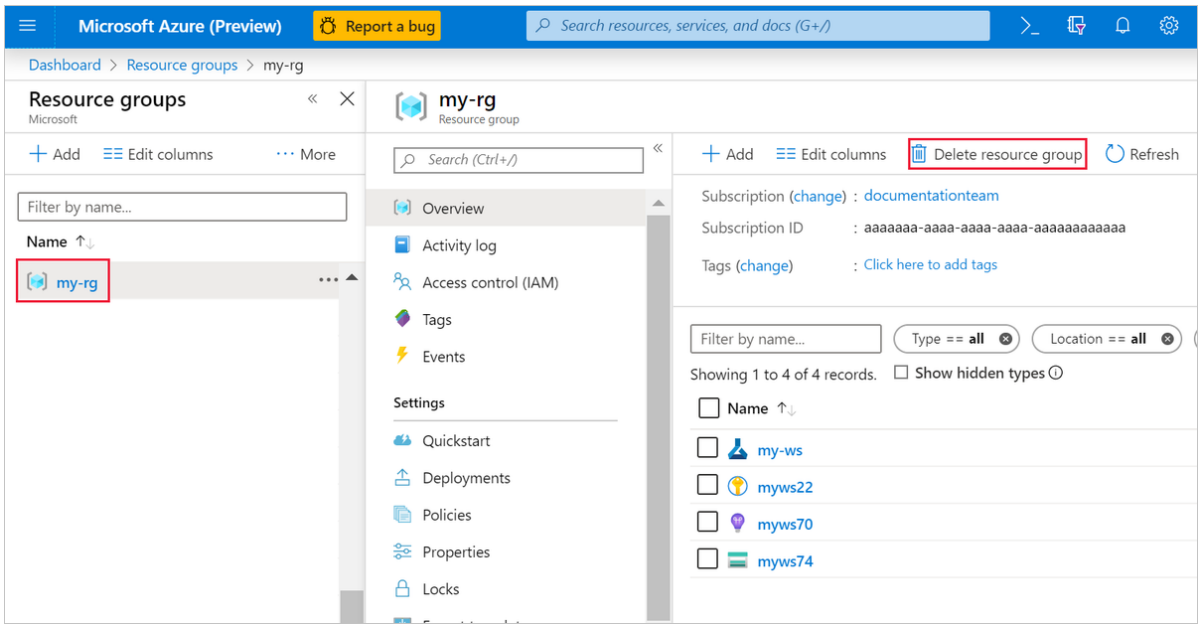
Clean up resources

> **IMPORTANT**
>
> You can use the resources that you created as prerequisites for other Azure Machine Learning tutorials and how-to articles.

Delete everything

If you don't plan to use anything that you created, delete the entire resource group so you don't incur any charges.

1. In the Azure portal, select **Resource groups** on the left side of the window.



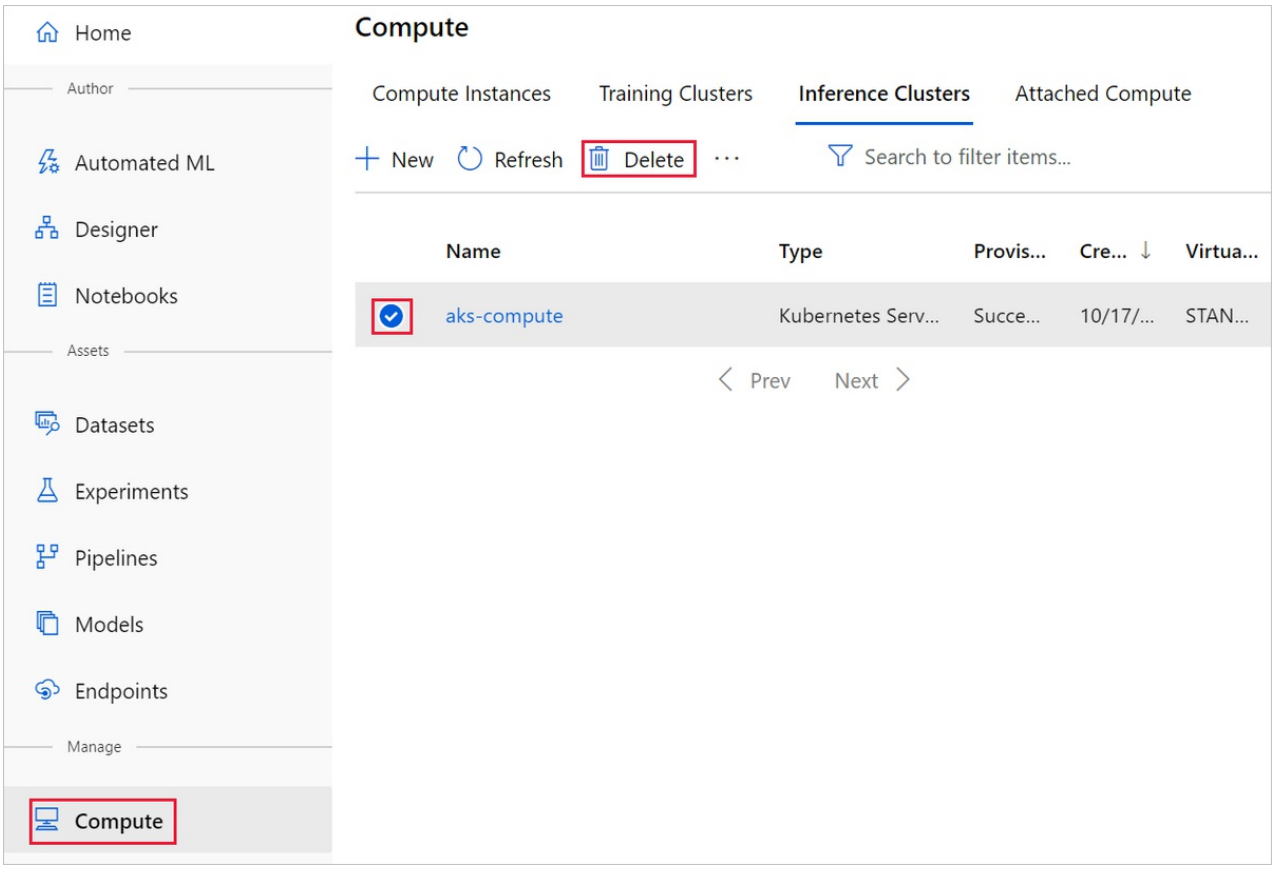
2. In the list, select the resource group that you created.
3. Select **Delete resource group**.

Deleting the resource group also deletes all resources that you created in the designer.

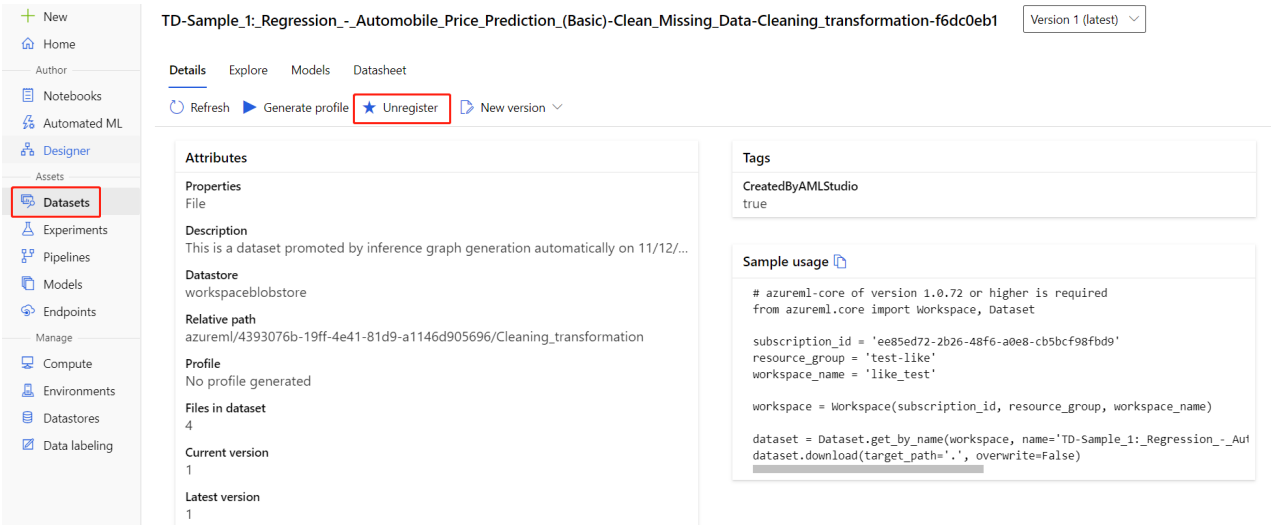
Delete individual assets

In the designer where you created your experiment, delete individual assets by selecting them and then selecting the **Delete** button.

The compute target that you created here *automatically autoscales* to zero nodes when it's not being used. This action is taken to minimize charges. If you want to delete the compute target, take these steps:



You can unregister datasets from your workspace by selecting each dataset and selecting **Unregister**.



To delete a dataset, go to the storage account by using the Azure portal or Azure Storage Explorer and manually delete those assets.

Next steps

Learn how to build and deploy machine learning models with Tutorial: Predict automobile price with the designer

What is an Azure Machine Learning workspace?

2/20/2020 • 4 minutes to read • Edit Online

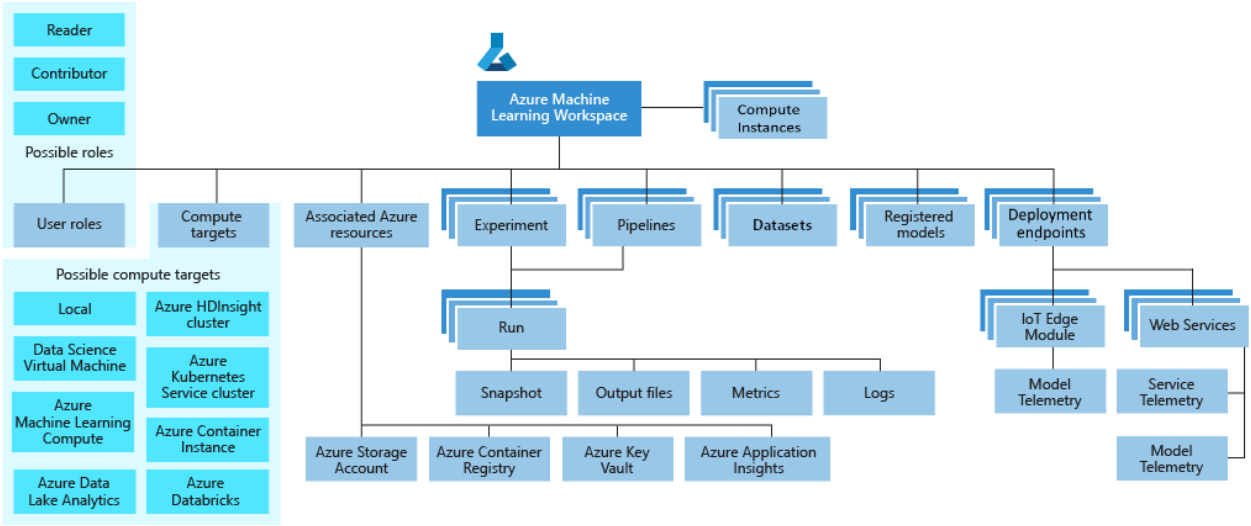
The workspace is the top-level resource for Azure Machine Learning, providing a centralized place to work with all the artifacts you create when you use Azure Machine Learning. The workspace keeps a history of all training runs, including logs, metrics, output, and a snapshot of your scripts. You use this information to determine which training run produces the best model.

Once you have a model you like, you register it with the workspace. You then use the registered model and scoring scripts to deploy to Azure Container Instances, Azure Kubernetes Service, or to a field-programmable gate array (FPGA) as a REST-based HTTP endpoint. You can also deploy the model to an Azure IoT Edge device as a module.

Pricing and features available depend on whether Basic or Enterprise edition is selected for the workspace. You select the edition when you create the workspace. You can also upgrade from Basic to Enterprise edition.

Taxonomy

A taxonomy of the workspace is illustrated in the following diagram:



The diagram shows the following components of a workspace:

- A workspace can contain Azure Machine Learning compute instances, cloud resources configured with the Python environment necessary to run Azure Machine Learning.
- User roles enable you to share your workspace with other users, teams or projects.
- Compute targets are used to run your experiments.
- When you create the workspace, associated resources are also created for you.
- Experiments are training runs you use to build your models.
- Pipelines are reusable workflows for training and retraining your model.
- Datasets aid in management of the data you use for model training and pipeline creation.
- Once you have a model you want to deploy, you create a registered model.
- Use the registered model and a scoring script to create a deployment endpoint.

Tools for workspace interaction

You can interact with your workspace in the following ways:

- On the web:
  - Azure Machine Learning studio
  - Azure Machine Learning designer (preview) - Available only in Enterprise edition workspaces.
- In any Python environment with the Azure Machine Learning SDK for Python.
- In any R environment with the Azure Machine Learning SDK for R.
- On the command line using the Azure Machine Learning CLI extension

Machine learning with a workspace

Machine learning tasks read and/or write artifacts to your workspace.

- Run an experiment to train a model - writes experiment run results to the workspace.
- Use automated ML to train a model - writes training results to the workspace.
- Register a model in the workspace.
- Deploy a model - uses the registered model to create a deployment.
- Create and run reusable workflows.
- View machine learning artifacts such as experiments, pipelines, models, deployments.
- Track and monitor models.

Workspace management

You can also perform the following workspace management tasks:

WORKSPACE MANAGEMENT TASK	PORTAL	STUDIO	PYTHON SDK / R SDK	CLI
Create a workspace	✓		✓	✓
Manage workspace access	✓			✓
Upgrade to Enterprise edition	✓	✓		
Create and manage compute resources	✓	✓	✓	✓
Create a Notebook VM		✓		

WARNING

Moving your Azure Machine Learning workspace to a different subscription, or moving the owning subscription to a new tenant, is not supported. Doing so may cause errors.

Create a workspace

When you create a workspace, you decide whether to create it with Basic or Enterprise edition. The edition determines the features available in the workspace. Among other features, Enterprise edition gives you access to

Azure Machine Learning designer and the studio version of building automated machine learning experiments. For more details and pricing information, see Azure Machine Learning pricing.

There are multiple ways to create a workspace:

- Use the Azure portal for a point-and-click interface to walk you through each step.
- Use the Azure Machine Learning SDK for Python to create a workspace on the fly from Python scripts or Jupiter notebooks
- Use an Azure Resource Manager template or the Azure Machine Learning CLI when you need to automate or customize the creation with corporate security standards.
- If you work in Visual Studio Code, use the VS Code extension.

> **NOTE**
> The workspace name is case-insensitive.

Upgrade to Enterprise edition

You can upgrade your workspace from Basic to Enterprise edition using Azure portal. You cannot downgrade an Enterprise edition workspace to a Basic edition workspace.

Associated resources

When you create a new workspace, it automatically creates several Azure resources that are used by the workspace:

- Azure Container Registry: Registers docker containers that you use during training and when you deploy a model. To minimize costs, ACR is **lazy-loaded** until deployment images are created.
- Azure Storage account: Is used as the default datastore for the workspace. Jupyter notebooks that are used with your Azure Machine Learning compute instances are stored here as well.
- Azure Application Insights: Stores monitoring information about your models.
- Azure Key Vault: Stores secrets that are used by compute targets and other sensitive information that's needed by the workspace.

> **NOTE**
> In addition to creating new versions, you can also use existing Azure services.

Next steps

To get started with Azure Machine Learning, see:

- Azure Machine Learning overview
- Create a workspace
- Manage a workspace
- Tutorial: Get started creating your first ML experiment with the Python SDK
- Tutorial: Get started with Azure Machine Learning with the R SDK
- Tutorial: Create your first classification model with automated machine learning (Available only in Enterprise edition workspaces)
- Tutorial: Predict automobile price with the designer (Available only in Enterprise edition workspaces)

What are Azure Machine Learning environments?

3/20/2020 • 4 minutes to read • Edit Online

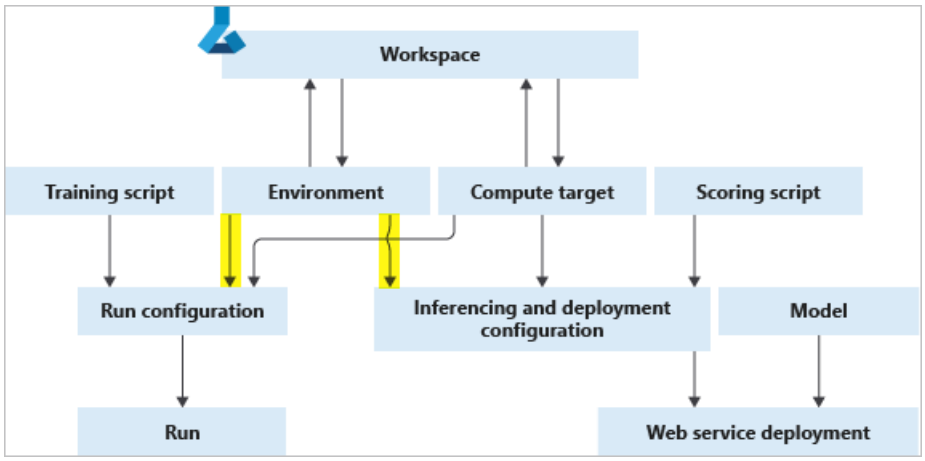
APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Azure Machine Learning environments specify the Python packages, environment variables, and software settings around your training and scoring scripts. They also specify run times (Python, Spark, or Docker). The environments are managed and versioned entities within your Machine Learning workspace that enable reproducible, auditable, and portable machine learning workflows across a variety of compute targets.

You can use an `Environment` object on your local compute to:

- Develop your training script.
- Reuse the same environment on Azure Machine Learning Compute for model training at scale.
- Deploy your model with that same environment.

The following diagram illustrates how you can use a single `Environment` object in both your run configuration, for training, and your inference and deployment configuration, for web service deployments.



Types of environments

Environments can broadly be divided into three categories: *curated*, *user-managed*, and *system-managed*.

Curated environments are provided by Azure Machine Learning and are available in your workspace by default. They contain collections of Python packages and settings to help you get started with various machine learning frameworks.

In user-managed environments, you're responsible for setting up your environment and installing every package that your training script needs on the compute target. Conda doesn't check your environment or install anything for you. If you're defining your own environment, you must list `azureml-defaults` with version `>= 1.0.45` as a pip dependency. This package contains the functionality that's needed to host the model as a web service.

You use system-managed environments when you want Conda to manage the Python environment and the script dependencies for you. The service assumes this type of environment by default, because of its usefulness on remote compute targets that are not manually configurable.

Create and manage environments

You can create environments by:

- Defining new `Environment` objects, either by using a curated environment or by defining your own dependencies.
- Using existing `Environment` objects from your workspace. This approach allows for consistency and reproducibility with your dependencies.
- Importing from an existing Anaconda environment definition.
- Using the Azure Machine Learning CLI

For specific code samples, see the "Create an environment" section of Reuse environments for training and deployment. Environments are also easily managed through your workspace. They include the following functionality:

- Environments are automatically registered to your workspace when you submit an experiment. They can also be manually registered.
- You can fetch environments from your workspace to use for training or deployment, or to make edits to the environment definition.
- With versioning, you can see changes to your environments over time, which ensures reproducibility.
- You can build Docker images automatically from your environments.

For code samples, see the "Manage environments" section of Reuse environments for training and deployment.

Environment building, caching, and reuse

The Azure Machine Learning service builds environment definitions into Docker images and conda environments. It also caches the environments so they can be reused in subsequent training runs and service endpoint deployments.

Building environments as Docker images

Typically, when you first submit a run using an environment, the Azure Machine Learning service invokes an ACR Build Task on the Azure Container Registry (ACR) associated with the Workspace. The built Docker image is then cached on the Workspace ACR. At the start of the run execution, the image is retrieved by the compute target.

The image build consists of two steps:

1. Downloading a base image, and executing any Docker steps
2. Building a conda environment according to conda dependencies specified in the environment definition.

The second step is omitted if you specify user-managed dependencies. In this case you're responsible for installing any Python packages, by including them in your base image, or specifying custom Docker steps within the first step. You're also responsible for specifying the correct location for the Python executable.

Image caching and reuse

If you use the same environment definition for another run, the Azure Machine Learning service reuses the cached image from the Workspace ACR.

To view the details of a cached image, use Environment.get_image_details method.

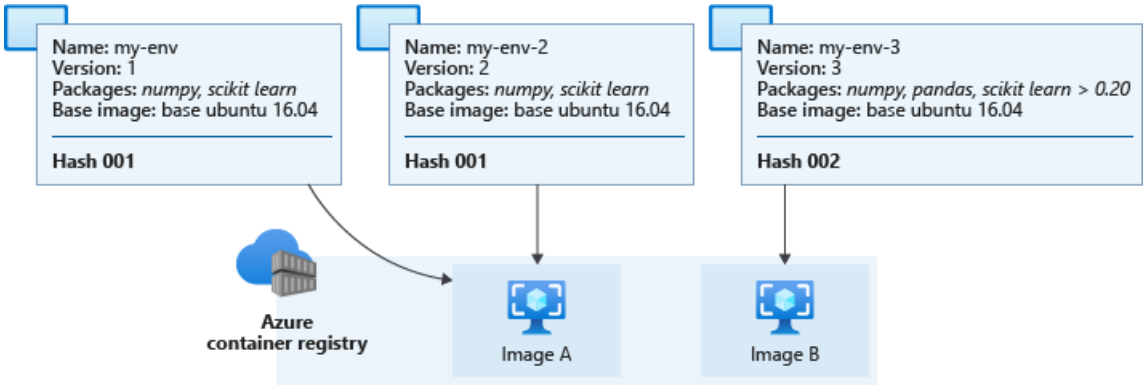
To determine whether to reuse a cached image or build a new one, the service computes a hash value from the environment definition and compares it to the hashes of existing environments. The hash is based on:

- Base image property value
- Custom docker steps property value
- List of Python packages in Conda definition
- List of packages in Spark definition

The hash doesn't depend on environment name or version. Environment definition changes, such as adding or

removing a Python package or changing the package version, causes the hash value to change and triggers an image rebuild. However, if you simply rename your environment or create a new environment with the exact properties and packages of an existing one, then the hash value remains the same and the cached image is used.

See the following diagram that shows three environment definitions. Two of them have different name and version, but identical base image and Python packages. They have the same hash and therefore correspond to the same cached image. The third environment has different Python packages and versions, and therefore corresponds to a different cached image.



> **IMPORTANT**
> If you create an environment with an unpinned package dependency, for example `numpy`, that environment will keep using the package version installed *at the time of environment creation*. Also, any future environment with matching definition will keep using the old version.

To update the package, specify a version number to force image rebuild, for example `numpy==1.18.1`. Note that new dependencies, including nested ones will be installed that might break a previously working scenario.

> **WARNING**
> The Environment.build method will rebuild the cached image, with possible side-effect of updating unpinned packages and breaking reproducibility for all environment definitions corresponding to that cached image.

Next steps

- Learn how to create and use environments in Azure Machine Learning.
- See the Python SDK reference documentation for the environment class.
- See the R SDK reference documentation for environments.

Data ingestion in Azure Machine Learning

3/11/2020 • 2 minutes to read • Edit Online

In this article, you learn the pros and cons of the following data ingestion options available with Azure Machine Learning.

1. Azure Data Factory pipelines
2. Azure Machine Learning Python SDK

Data ingestion is the process in which unstructured data is extracted from one or multiple sources and then prepared for training machine learning models. It's also time intensive, especially if done manually, and if you have large amounts of data from multiple sources. Automating this effort frees up resources and ensures your models use the most recent and applicable data.

Azure Data Factory (ADF) is specifically built to extract, load, and transform data, however the Python SDK let's you develop a custom code solution for basic data ingestion tasks. If neither are quite what you need, you can also use ADF and the Python SDK together to create an overall data ingestion workflow that meets your needs.

Use Azure Data Factory

Azure Data Factory offers native support for data source monitoring and triggers for data ingestion pipelines.

The following table summarizes the pros and cons for using Azure Data Factory for your data ingestion workflows.

PROS	CONS
Specifically built to extract, load, and transform data.	Currently offers a limited set of Azure Data Factory pipeline tasks
Allows you to create data-driven workflows for orchestrating data movement and transformations at scale.	Expensive to construct and maintain. See Azure Data Factory's pricing page for more information.
Integrated with various Azure tools like Azure Databricks and Azure Functions	Doesn't natively run scripts, instead relies on separate compute for script runs
Natively supports data source triggered data ingestion	
Data preparation and model training processes are separate.	
Embedded data lineage capability for Azure Data Factory dataflows	
Provides a low code experience user interface for non-scripting approaches	

These steps and the following diagram illustrate Azure Data Factory's data ingestion workflow.

1. Pull the data from its sources
2. Transform and save the data to an output blob container, which serves as data storage for Azure Machine Learning
3. With prepared data stored, the Azure Data Factory pipeline invokes a training Machine Learning pipeline that

receives the prepared data for model training

Learn how to build a data ingestion pipeline for Machine Learning with Azure Data Factory.

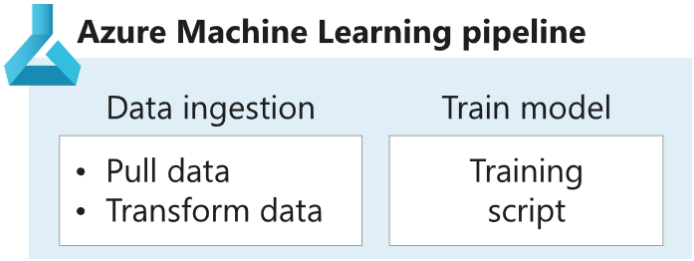
Use the Python SDK

With the Python SDK, you can incorporate data ingestion tasks into an Azure Machine Learning pipeline step.

The following table summarizes the pros and con for using the SDK and an ML pipelines step for data ingestion tasks.

PROS	CONS
Configure your own Python scripts	Does not natively support data source change triggering. Requires Logic App or Azure Function implementations
Data preparation as part of every model training execution	Requires development skills to create a data ingestion script
Supports data preparation scripts on various compute targets, including Azure Machine Learning compute	Does not provide a user interface for creating the ingestion mechanism

In the following diagram, the Azure Machine Learning pipeline consists of two steps: data ingestion and model training. The data ingestion step encompasses tasks that can be accomplished using Python libraries and the Python SDK, such as extracting data from local/web sources, and basic data transformations, like missing value imputation. The training step then uses the prepared data as input to your training script to train your machine learning model.



Next steps

- Learn how to build a data ingestion pipeline for Machine Learning with Azure Data Factory
- Learn how to automate and manage the development life cycles of your data ingestion pipelines with Azure Pipelines.

Secure data access in Azure Machine Learning

4/24/2020 • 4 minutes to read • Edit Online

Azure Machine Learning makes it easy to connect to your data in the cloud. It provides an abstraction layer over the underlying storage service, so you can securely access and work with your data without having to write code specific to your storage type. Azure Machine Learning also provides the following data capabilities:

- Versioning and tracking of data lineage
- Data labeling
- Data drift monitoring
- Interoperability with Pandas and Spark DataFrames

Data workflow

When you're ready to use the data in your cloud-based storage solution, we recommend the following data delivery workflow. This workflow assumes you have an Azure storage account and data in a cloud-based storage service in Azure.

1. Create an Azure Machine Learning datastore to store connection information to your Azure storage.
2. From that datastore, create an Azure Machine Learning dataset to point to a specific file(s) in your underlying storage.
3. To use that dataset in your machine learning experiment you can either
   a. Mount it to your experiment's compute target for model training.

      OR

   b. Consume it directly in Azure Machine Learning solutions like, automated machine learning (automated ML) experiment runs, machine learning pipelines, or the Azure Machine Learning designer.
4. Create dataset monitors for your model output dataset to detect for data drift.
5. If data drift is detected, update your input dataset and retrain your model accordingly.

The following diagram provides a visual demonstration of this recommended workflow.

Datastores

Azure Machine Learning datastores securely keep the connection information to your Azure storage, so you don't have to code it in your scripts. Register and create a datastore to easily connect to your storage account, and access the data in your underlying Azure storage service.

Supported cloud-based storage services in Azure that can be registered as datastores:

- Azure Blob Container
- Azure File Share
- Azure Data Lake
- Azure Data Lake Gen2
- Azure SQL Database

- Azure Database for PostgreSQL
- Databricks File System
- Azure Database for MySQL

Datasets

Azure Machine Learning datasets are references that point to the data in your storage service. They aren't copies of your data, so no extra storage cost is incurred. To interact with your data in storage, create a dataset to package your data into a consumable object for machine learning tasks. Register the dataset to your workspace to share and reuse it across different experiments without data ingestion complexities.

Datasets can be created from local files, public urls, Azure Open Datasets, or Azure storage services via datastores. To create a dataset from an in memory pandas dataframe, write the data to a local file, like a parquet, and create your dataset from that file.

We support 2 types of datasets:

- A TabularDataset represents data in a tabular format by parsing the provided file or list of files. You can load a TabularDataset into a Pandas or Spark DataFrame for further manipulation and cleansing. For a complete list of data formats you can create TabularDatasets from, see the TabularDatasetFactory class.
- A FileDataset references single or multiple files in your datastores or public URLs. You can download or mount files referenced by FileDatasets to your compute target.

Additional datasets capabilities can be found in the following documentation:

- Version and track dataset lineage.
- Monitor your dataset to help with data drift detection.

Work with your data

With datasets, you can accomplish a number of machine learning tasks through seamless integration with Azure Machine Learning features.

- Create a data labeling project.
- Train machine learning models:
  - automated ML experiments
  - the designer
  - notebooks
  - Azure Machine Learning pipelines
- Access datasets for scoring with batch inference in machine learning pipelines.
- Set up a dataset monitor for data drift detection.

Data labeling

Labeling large amounts of data has often been a headache in machine learning projects. Those with a computer vision component, such as image classification or object detection, generally require thousands of images and corresponding labels.

Azure Machine Learning gives you a central location to create, manage, and monitor labeling projects. Labeling projects help coordinate the data, labels, and team members, allowing you to more efficiently manage the labeling tasks. Currently supported tasks are image classification, either multi-label or multi-class, and object identification using bounded boxes.

Create a data labeling project, and output a dataset for use in machine learning experiments.

Data drift

In the context of machine learning, data drift is the change in model input data that leads to model performance degradation. It is one of the top reasons model accuracy degrades over time, thus monitoring data drift helps detect model performance issues.

See the Create a dataset monitor article, to learn more about how to detect and alert to data drift on new data in a dataset.

Next steps

- Create a dataset in Azure Machine Learning studio or with the Python SDK using these steps.
- Try out dataset training examples with our sample notebooks.
- For data drift examples, see this data drift tutorial.

Train models with Azure Machine Learning

3/11/2020 • 6 minutes to read • Edit Online

Azure Machine Learning provides several ways to train your models, from code first solutions using the SDK to low code solutions such as automated machine learning and the visual designer. Use the following list to determine which training method is right for you:

- Azure Machine Learning SDK for Python: The Python SDK provides several ways to train models, each with different capabilities.

TRAINING METHOD	DESCRIPTION
Run configuration	A **generic way to train models** is to use a training script and run configuration. The run configuration provides the information needed to configure the training environment used to train your model. You can take a run configuration, your training script, and a compute target (the training environment) and run a training job.
Automated machine learning	Automated machine learning allows you to **train models without extensive data science or programming knowledge**. For people with a data science and programming background, it provides a way to save time and resources by automating algorithm selection and hyperparameter tuning. You don't have to worry about defining a run configuration when using automated machine learning.
Estimators	Estimator classes **make it easy to train models based on popular machine learning frameworks**. There are estimator classes for **Scikit-learn**, **PyTorch**, **TensorFlow**, and **Chainer**. There is also a generic estimator that can be used with frameworks that do not already have a dedicated estimator class. You don't have to worry about defining a run configuration when using estimators.

TRAINING METHOD	DESCRIPTION
Machine learning pipeline	Pipelines are not a different training method, but a **way of defining a workflow using modular, reusable steps**, that can include training as part of the workflow. Machine learning pipelines support using automated machine learning, estimators, and run configuration to train models. Since pipelines are not focused specifically on training, the reasons for using a pipeline are more varied than the other training methods. Generally, you might use a pipeline when: * You want to **schedule unattended processes** such as long running training jobs or data preparation. * Use **multiple steps** that are coordinated across heterogeneous compute resources and storage locations. * Use the pipeline as a **reusable template** for specific scenarios, such as retraining or batch scoring. * **Track and version data sources, inputs, and outputs** for your workflow. * Your workflow is **implemented by different teams that work on specific steps independently**. Steps can then be joined together in a pipeline to implement the workflow.

- Azure Machine Learning SDK for Python: The SDK uses the reticulate package to bind to Azure Machine Learning's Python SDK. This allows you access to core objects and methods implemented in the Python SDK from any R environment.
- **Designer**: Azure Machine Learning designer (preview) provides an easy entry-point into machine learning for building proof of concepts, or for users with little coding experience. It allows you to train models using a drag and drop web-based UI. You can use Python code as part of the design, or train models without writing any code.
- **CLI**: The machine learning CLI provides commands for common tasks with Azure Machine Learning, and is often used for **scripting and automating tasks**. For example, once you've created a training script or pipeline, you might use the CLI to start a training run on a schedule or when the data files used for training are updated. For training models, it provides commands that submit training jobs. It can submit jobs using run configurations or pipelines.

Each of these training methods can use different types of compute resources for training. Collectively, these resources are referred to as **compute targets**. A compute target can be a local machine or a cloud resource, such as an Azure Machine Learning Compute, Azure HDInsight, or a remote virtual machine.

Python SDK

The Azure Machine Learning SDK for Python allows you to build and run machine learning workflows with Azure Machine Learning. You can interact with the service from an interactive Python session, Jupyter Notebooks, Visual Studio Code, or other IDE.

- What is the Azure Machine Learning SDK for Python
- Install/update the SDK
- Configure a development environment for Azure Machine Learning

Run configuration

A generic training job with Azure Machine Learning can be defined using the RunConfiguration. The run configuration is then used, along with your training script(s) to train a model on a compute target.

You may start with a run configuration for your local computer, and then switch to one for a cloud-based compute

target as needed. When changing the compute target, you only change the run configuration you use. A run also logs information about the training job, such as the inputs, outputs, and logs.

- What is a run configuration?
- Tutorial: Train your first ML model
- Examples: Jupyter Notebook examples of training models
- How to: Set up and use compute targets for model training

Automated Machine Learning

Define the iterations, hyperparameter settings, featurization, and other settings. During training, Azure Machine Learning tries different algorithms and parameters in parallel. Training stops once it hits the exit criteria you defined. You don't have to worry about defining a run configuration when using estimators.

> **TIP**
> In addition to the Python SDK, you can also use Automated ML through Azure Machine Learning studio.

- What is automated machine learning?
- Tutorial: Create your first classification model with automated machine learning
- Tutorial: Use automated machine learning to predict taxi fares
- Examples: Jupyter Notebook examples for automated machine learning
- How to: Configure automated ML experiments in Python
- How to: Autotrain a time-series forecast model
- How to: Create, explore, and deploy automated machine learning experiments with Azure Machine Learning studio

Estimators

Estimators make it easy to train models using popular ML frameworks. If you're using **Scikit-learn**, **PyTorch**, **TensorFlow**, or **Chainer**, you should consider using an estimator for training. There is also a generic estimator that can be used with frameworks that do not already have a dedicated estimator class. You don't have to worry about defining a run configuration when using estimators.

- What are estimators?
- Tutorial: Train image classification models with MNIST data and scikit-learn using Azure Machine Learning
- Examples: Jupyter Notebook examples of using estimators
- How to: Create estimators in training

Machine learning pipeline

Machine learning pipelines can use the previously mentioned training methods (run configuration, estimators, and automated machine learning). Pipelines are more about creating a workflow, so they encompass more than just the training of models. In a pipeline, you can train a model using automated machine learning, estimators, or run configurations.

- What are ML pipelines in Azure Machine Learning?
- Create and run machine learning pipelines with Azure Machine Learning SDK
- Tutorial: Use Azure Machine Learning Pipelines for batch scoring
- Examples: Jupyter Notebook examples for machine learning pipelines
- Examples: Pipeline with automated machine learning
- Examples: Pipeline with estimators

R SDK

The R SDK enables you to use the R language with Azure Machine Learning. The SDK uses the reticulate package to bind to Azure Machine Learning's Python SDK. This allows you access to core objects and methods implemented in the Python SDK from any R environment.

For more information, see the following articles:

- Tutorial: Create a logistic regression model
- Azure Machine Learning SDK for R reference

Azure Machine Learning designer

The designer lets you to train models using a drag and drop interface in your web browser.

- What is the designer?
- Tutorial : Predict automobile price
- Regression: Predict price
- Classification: Predict income
- Classification: Predict churn, appetency, and up-selling
- Classification with custom R script: Predict flight delays
- Text Classification: Wikipedia SP 500 Dataset

CLI

The machine learning CLI is an extension for the Azure CLI. It provides cross-platform CLI commands for working with Azure Machine Learning. Typically, you use the CLI to automate tasks, such as training a machine learning model.

- Use the CLI extension for Azure Machine Learning
- MLOps on Azure

Next steps

Learn how to Set up training environments.

Distributed training with Azure Machine Learning

3/29/2020 • 2 minutes to read • Edit Online

In this article, you learn about distributed training and how Azure Machine Learning supports it for deep learning models.

In distributed training the workload to train a model is split up and shared among multiple mini processors, called worker nodes. These worker nodes work in parallel to speed up model training. Distributed training can be used for traditional ML models, but is better suited for compute and time intensive tasks, like deep learning for training deep neural networks.

Deep learning and distributed training

There are two main types of distributed training: data parallelism and model parallelism. For distributed training on deep learning models, the Azure Machine Learning SDK in Python supports integrations with popular frameworks, PyTorch and TensorFlow. Both frameworks employ data parallelism for distributed training, and can leverage horovod for optimizing compute speeds.

- Distributed training with PyTorch
- Distributed training with TensorFlow

For ML models that don't require distributed training, see train models with Azure Machine Learning for the different ways to train models using the Python SDK.

Data parallelism

Data parallelism is the easiest to implement of the two distributed training approaches, and is sufficient for most use cases.

In this approach, the data is divided into partitions, where the number of partitions is equal to the total number of available nodes, in the compute cluster. The model is copied in each of these worker nodes, and each worker operates on its own subset of the data. Keep in mind that each node has to have the capacity to support the model that's being trained, that is the model has to entirely fit on each node.

Each node independently computes the errors between its predictions for its training samples and the labeled outputs. In turn, each node updates its model based on the errors and must communicate all of its changes to the other nodes to update their corresponding models. This means that the worker nodes need to synchronize the model parameters, or gradients, at the end of the batch computation to ensure they are training a consistent model.

Model parallelism

In model parallelism, also known as network parallelism, the model is segmented into different parts that can run concurrently in different nodes, and each one will run on the same data. The scalability of this method depends on the degree of task parallelization of the algorithm, and it is more complex to implement than data parallelism.

In model parallelism, worker nodes only need to synchronize the shared parameters, usually once for each forward or backward-propagation step. Also, larger models aren't a concern since each node operates on a subsection of the model on the same training data.

Next steps

- Learn how to set up training environments with the Python SDK.
- For a technical example, see the reference architecture scenario.
- Train ML models with TensorFlow.
- Train ML models with PyTorch.

MLOps: Model management, deployment, and monitoring with Azure Machine Learning

4/14/2020 • 9 minutes to read • Edit Online

In this article, learn about how to use Azure Machine Learning to manage the lifecycle of your models. Azure Machine Learning uses a Machine Learning Operations (MLOps) approach. MLOps improves the quality and consistency of your machine learning solutions.

What is MLOps?

Machine Learning Operations (MLOps) is based on DevOps principles and practices that increase the efficiency of workflows. For example, continuous integration, delivery, and deployment. MLOps applies these principles to the machine learning process, with the goal of:

- Faster experimentation and development of models
- Faster deployment of models into production
- Quality assurance

Azure Machine Learning provides the following MLOps capabilities:

- **Create reproducible ML pipelines**. Machine Learning pipelines allow you to define repeatable and reusable steps for your data preparation, training, and scoring processes.
- **Create reusable software environments** for training and deploying models.
- **Register, package, and deploy models from anywhere**. You can also track associated metadata required to use the model.
- **Capture the governance data for the end-to-end ML lifecycle**. The logged information can include who is publishing models, why changes were made, and when models were deployed or used in production.
- **Notify and alert on events in the ML lifecycle**. For example, experiment completion, model registration, model deployment, and data drift detection.
- **Monitor ML applications for operational and ML-related issues**. Compare model inputs between training and inference, explore model-specific metrics, and provide monitoring and alerts on your ML infrastructure.
- **Automate the end-to-end ML lifecycle with Azure Machine Learning and Azure Pipelines**. Using pipelines allows you to frequently update models, test new models, and continuously roll out new ML models alongside your other applications and services.

Create reproducible ML pipelines

Use ML pipelines from Azure Machine Learning to stitch together all of the steps involved in your model training process.

An ML pipeline can contain steps from data preparation to feature extraction to hyperparameter tuning to model evaluation. For more information, see ML pipelines.

If you use the Designer to create your ML pipelines, you may at any time click the **"..."** at the top-right of the Designer page and then select **Clone**. Cloning your pipeline allows you to iterate your pipeline design without losing your old versions.

Create reusable software environments

Azure Machine Learning environments allow you to track and reproduce your projects' software dependencies as they evolve. Environments allow you to ensure that builds are reproducible without manual software configurations.

Environments describe the pip and Conda dependencies for your projects, and can be used for both training and deployment of models. For more information, see What are Azure Machine Learning environments.

Register, package, and deploy models from anywhere

Register and track ML models

Model registration allows you to store and version your models in the Azure cloud, in your workspace. The model registry makes it easy to organize and keep track of your trained models.

> **TIP**
> A registered model is a logical container for one or more files that make up your model. For example, if you have a model that is stored in multiple files, you can register them as a single model in your Azure Machine Learning workspace. After registration, you can then download or deploy the registered model and receive all the files that were registered.

Registered models are identified by name and version. Each time you register a model with the same name as an existing one, the registry increments the version. Additional metadata tags can be provided during registration. These tags are then used when searching for a model. Azure Machine Learning supports any model that can be loaded using Python 3.5.2 or higher.

> **TIP**
> You can also register models trained outside Azure Machine Learning.

You can't delete a registered model that is being used in an active deployment. For more information, see the register model section of Deploy models.

Profile models

Azure Machine Learning can help you understand the CPU and memory requirements of the service that will be created when you deploy your model. Profiling tests the service that runs your model and returns information such as the CPU usage, memory usage, and response latency. It also provides a CPU and memory recommendation based on the resource usage. For more information, see the profiling section of Deploy models.

Package and debug models

Before deploying a model into production, it is packaged into a Docker image. In most cases, image creation happens automatically in the background during deployment. You can manually specify the image.

If you run into problems with the deployment, you can deploy on your local development environment for troubleshooting and debugging.

For more information, see Deploy models and Troubleshooting deployments.

Convert and optimize models

Converting your model to Open Neural Network Exchange (ONNX) may improve performance. On average, converting to ONNX can yield a 2x performance increase.

For more information on ONNX with Azure Machine Learning, see the Create and accelerate ML models article.

Use models

Trained machine learning models are deployed as web services in the cloud or locally. You can also deploy

models to Azure IoT Edge devices. Deployments use CPU, GPU, or field-programmable gate arrays (FPGA) for inferencing. You can also use models from Power BI.

When using a model as a web service or IoT Edge device, you provide the following items:

- The model(s) that are used to score data submitted to the service/device.
- An entry script. This script accepts requests, uses the model(s) to score the data, and return a response.
- An Azure Machine Learning environment that describes the pip and Conda dependencies required by the model(s) and entry script.
- Any additional assets such as text, data, etc. that are required by the model(s) and entry script.

You also provide the configuration of the target deployment platform. For example, the VM family type, available memory, and number of cores when deploying to Azure Kubernetes Service.

When the image is created, components required by Azure Machine Learning are also added. For example, assets needed to run the web service and interact with IoT Edge.

Batch scoring

Batch scoring is supported through ML pipelines. For more information, see Batch predictions on big data.

Real-time web services

You can use your models in **web services** with the following compute targets:

- Azure Container Instance
- Azure Kubernetes Service
- Local development environment

To deploy the model as a web service, you must provide the following items:

- The model or ensemble of models.
- Dependencies required to use the model. For example, a script that accepts requests and invokes the model, conda dependencies, etc.
- Deployment configuration that describes how and where to deploy the model.

For more information, see Deploy models.

Controlled rollout

When deploying to Azure Kubernetes Service, you can use controlled rollout to enable the following scenarios:

- Create multiple versions of an endpoint for a deployment
- Perform A/B testing by routing traffic to different versions of the endpoint.
- Switch between endpoint versions by updating the traffic percentage in endpoint configuration.

For more information, see Controlled rollout of ML models.

IoT Edge devices

You can use models with IoT devices through **Azure IoT Edge modules**. IoT Edge modules are deployed to a hardware device, which enables inference, or model scoring, on the device.

For more information, see Deploy models.

Analytics

Microsoft Power BI supports using machine learning models for data analytics. For more information, see Azure Machine Learning integration in Power BI (preview).

Capture the governance data required for capturing the end-to-end ML lifecycle

Azure ML gives you the capability to track the end-to-end audit trail of all of your ML assets by using metadata.

- Azure ML integrates with Git to track information on which repository / branch / commit your code came from.
- Azure ML Datasets help you track, profile, and version data.
- Interpretability allows you to explain your models, meet regulatory compliance, and understand how models arrive at a result for given input.
- Azure ML Run history stores a snapshot of the code, data, and computes used to train a model.
- The Azure ML Model Registry captures all of the metadata associated with your model (which experiment trained it, where it is being deployed, if its deployments are healthy).
- Integration with Azure Event Grid allows you to act on events in the ML lifecycle. For example, model registration, deployment, data drift, and training (run) events.

> **TIP**
> While some information on models and datasets is automatically captured, you can add additional information by using **tags**. When looking for registered models and datasets in your workspace, you can use tags as a filter.
>
> Associating a dataset with a registered model is an optional step. For information on referencing a dataset when registering a model, see the Model class reference.

Notify, automate, and alert on events in the ML lifecycle

Azure ML publishes key events to Azure EventGrid, which can be used to notify and automate on events in the ML lifecycle. For more information, please see this document.

Monitor for operational & ML issues

Monitoring enables you to understand what data is being sent to your model, and the predictions that it returns.

This information helps you understand how your model is being used. The collected input data may also be useful in training future versions of the model.

For more information, see How to enable model data collection.

Retrain your model on new data

Often, you'll want to validate your model, update it, or even retrain it from scratch, as you receive new information. Sometimes, receiving new data is an expected part of the domain. Other times, as discussed in Detect data drift (preview) on datasets, model performance can degrade in the face of such things as changes to a particular sensor, natural data changes such as seasonal effects, or features shifting in their relation to other features.

There is no universal answer to "How do I know if I should retrain?" but Azure ML event and monitoring tools previously discussed are good starting points for automation. Once you have decided to retrain, you should:

- Preprocess your data using a repeatable, automated process
- Train your new model
- Compare the outputs of your new model to those of your old model
- Use predefined criteria to choose whether to replace your old model

A theme of the above steps is that your retraining should be automated, not ad hoc. Azure Machine Learning pipelines are a good answer for creating workflows relating to data preparation, training, validation, and deployment. Read Retrain models with Azure Machine Learning designer (preview) to see how pipelines and the

Azure Machine Learning designer fit into a retraining scenario.

Automate the ML lifecycle

You can use GitHub and Azure Pipelines to create a continuous integration process that trains a model. In a typical scenario, when a Data Scientist checks a change into the Git repo for a project, the Azure Pipeline will start a training run. The results of the run can then be inspected to see the performance characteristics of the trained model. You can also create a pipeline that deploys the model as a web service.

The Azure Machine Learning extension makes it easier to work with Azure Pipelines. It provides the following enhancements to Azure Pipelines:

- Enables workspace selection when defining a service connection.
- Enables release pipelines to be triggered by trained models created in a training pipeline.

For more information on using Azure Pipelines with Azure Machine Learning, see the following links:

- Continuous integration and deployment of ML models with Azure Pipelines
- Azure Machine Learning MLOps repository.
- Azure Machine Learning MLOpsPython repository.

You can also use Azure Data Factory to create a data ingestion pipeline that prepares data for use with training. For more information, see Data ingestion pipeline.

Next steps

Learn more by reading and exploring the following resources:

- How & where to deploy models with Azure Machine Learning
- Tutorial: Deploy an image classification model in ACI.
- End-to-end MLOps examples repo
- CI/CD of ML models with Azure Pipelines
- Create clients that consume a deployed model
- Machine learning at scale
- Azure AI reference architectures & best practices rep

Model interpretability in Azure Machine Learning

4/3/2020 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Overview of model interpretability

Interpretability is critical for data scientists, auditors, and business decision makers alike to ensure compliance with company policies, industry standards, and government regulations:

- Data scientists need the ability to explain their models to executives and stakeholders, so they can understand the value and accuracy of their findings. They also require interpretability to debug their models and make informed decisions about how to improve them.
- Legal auditors require tools to validate models with respect to regulatory compliance and monitor how models' decisions are impacting humans.
- Business decision makers need peace-of-mind by having the ability to provide transparency for end users. This allows them to earn and maintain trust.

Enabling the capability of explaining a machine learning model is important during two main phases of model development:

- During the training phase, as model designers and evaluators can use interpretability output of a model to verify hypotheses and build trust with stakeholders. They also use the insights into the model for debugging, validating model behavior matches their objectives, and to check for model unfairness or insignificant features.
- During the inferencing phase, as having transparency around deployed models empowers executives to understand "when deployed" how the model is working and how its decisions are treating and impacting people in real life.

Interpretability with Azure Machine Learning

The interpretability classes are made available through multiple SDK packages: (Learn how to install SDK packages for Azure Machine Learning)

- `azureml.interpret`, the main package, containing functionalities supported by Microsoft.
- `azureml.contrib.interpret`, preview, and experimental functionalities that you can try.
- `azureml.train.automl.automlexplainer` package for interpreting automated machine learning models.

Use `pip install azureml-interpret` and `pip install azureml-interpret-contrib` for general use, and `pip install azureml-interpret-contrib` for AutoML use to get the interpretability packages.

> **IMPORTANT**
>
> Content in the `contrib` namespace is not fully supported. As the experimental functionalities become mature, they will gradually be moved to the main namespace. .

How to interpret your model

Using the classes and methods in the SDK, you can:

- Explain model prediction by generating feature importance values for the entire model and/or individual datapoints.
- Achieve model interpretability on real-world datasets at scale, during training and inference.
- Use an interactive visualization dashboard to discover patterns in data and explanations at training time

In machine learning, **features** are the data fields used to predict a target data point. For example, to predict credit risk, data fields for age, account size, and account age might be used. In this case, age, account size, and account age are **features**. Feature importance tells you how each data field affected the model's predictions. For example, age may be heavily used in the prediction while account size and age do not affect the prediction values significantly. This process allows data scientists to explain resulting predictions, so that stakeholders have visibility into what features are most important in the model.

Learn about supported interpretability techniques, supported machine learning models, and supported run environments here.

Supported interpretability techniques

`azureml-interpret` uses the interpretability techniques developed in Interpret-Community, an open source python package for training interpretable models and helping to explain blackbox AI systems. Interpret-Community serves as the host for this SDK's supported explainers, and currently supports the following interpretability techniques:

INTERPRETABILITY TECHNIQUE	DESCRIPTION	TYPE
1. SHAP Tree Explainer	SHAP's tree explainer, which focuses on polynomial time fast SHAP value estimation algorithm specific to **trees and ensembles of trees**.	Model-specific
2. SHAP Deep Explainer	Based on the explanation from SHAP, Deep Explainer "is a high-speed approximation algorithm for SHAP values in deep learning models that builds on a connection with DeepLIFT described in the SHAP NIPS paper. **TensorFlow** models and **Keras** models using the TensorFlow backend are supported (there is also preliminary support for PyTorch)".	Model-specific
3. SHAP Linear Explainer	SHAP's Linear explainer computes SHAP values for a **linear model**, optionally accounting for inter-feature correlations.	Model-specific
4. SHAP Kernel Explainer	SHAP's Kernel explainer uses a specially weighted local linear regression to estimate SHAP values for **any model**.	Model-agnostic

INTERPRETABILITY TECHNIQUE	DESCRIPTION	TYPE
5. Mimic Explainer (Global Surrogate)	Mimic explainer is based on the idea of training global surrogate models to mimic blackbox models. A global surrogate model is an intrinsically interpretable model that is trained to approximate the predictions of **any black box model** as accurately as possible. Data scientists can interpret the surrogate model to draw conclusions about the black box model. You can use one of the following interpretable models as your surrogate model: LightGBM (LGBMExplainableModel), Linear Regression (LinearExplainableModel), Stochastic Gradient Descent explainable model (SGDExplainableModel), and Decision Tree (DecisionTreeExplainableModel).	Model-agnostic
6. Permutation Feature Importance Explainer (PFI)	Permutation Feature Importance is a technique used to explain classification and regression models that is inspired by Breiman's Random Forests paper (see section 10). At a high level, the way it works is by randomly shuffling data one feature at a time for the entire dataset and calculating how much the performance metric of interest changes. The larger the change, the more important that feature is. PFI can explain the overall behavior of **any underlying model** but does not explain individual predictions.	Model-agnostic

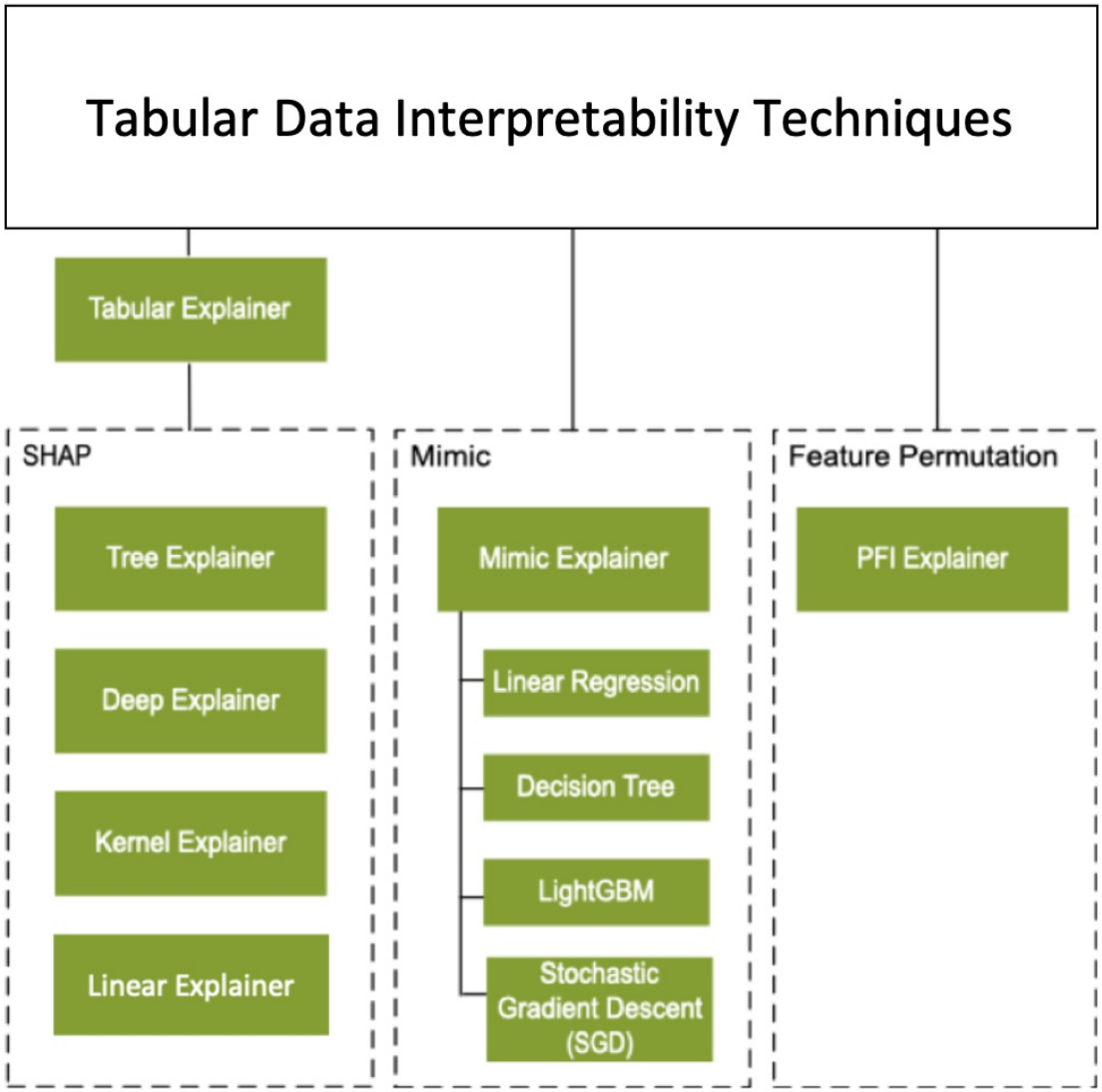
Besides the interpretability techniques described above, we support another SHAP-based explainer, called `TabularExplainer`. Depending on the model, `TabularExplainer` uses one of the supported SHAP explainers:

- TreeExplainer for all tree-based models
- DeepExplainer for DNN models
- LinearExplainer for linear models
- KernelExplainer for all other models

`TabularExplainer` has also made significant feature and performance enhancements over the direct SHAP Explainers:

- **Summarization of the initialization dataset**. In cases where speed of explanation is most important, we summarize the initialization dataset and generate a small set of representative samples, which speeds up the generation of overall and individual feature importance values.
- **Sampling the evaluation data set**. If the user passes in a large set of evaluation samples but does not actually need all of them to be evaluated, the sampling parameter can be set to true to speed up the calculation of overall model explanations.

The following diagram shows the current structure of supported explainers.



Supported machine learning models

The `azureml.interpret` package of the SDK supports models trained with the following dataset formats:

- `numpy.array`
- `pandas.DataFrame`
- `iml.datatypes.DenseData`
- `scipy.sparse.csr_matrix`

The explanation functions accept both models and pipelines as input. If a model is provided, the model must implement the prediction function `predict` or `predict_proba` that conforms to the Scikit convention. If your model does not support this, you can wrap your model in a function that generates the same outcome as `predict` or `predict_proba` in Scikit and use that wrapper function with the selected explainer. If a pipeline is provided, the explanation function assumes that the running pipeline script returns a prediction. Using this wrapping technique, `azureml.interpret` can support models trained via PyTorch, TensorFlow, and Keras deep learning frameworks as well as classic machine learning models.

Local and remote compute target

The `azureml.interpret` package is designed to work with both local and remote compute targets. If run locally, The SDK functions will not contact any Azure services.

You can run explanation remotely on Azure Machine Learning Compute and log the explanation info into the Azure Machine Learning Run History Service. Once this information is logged, reports and visualizations from the explanation are readily available on Azure Machine Learning studio for user analysis.

Next steps

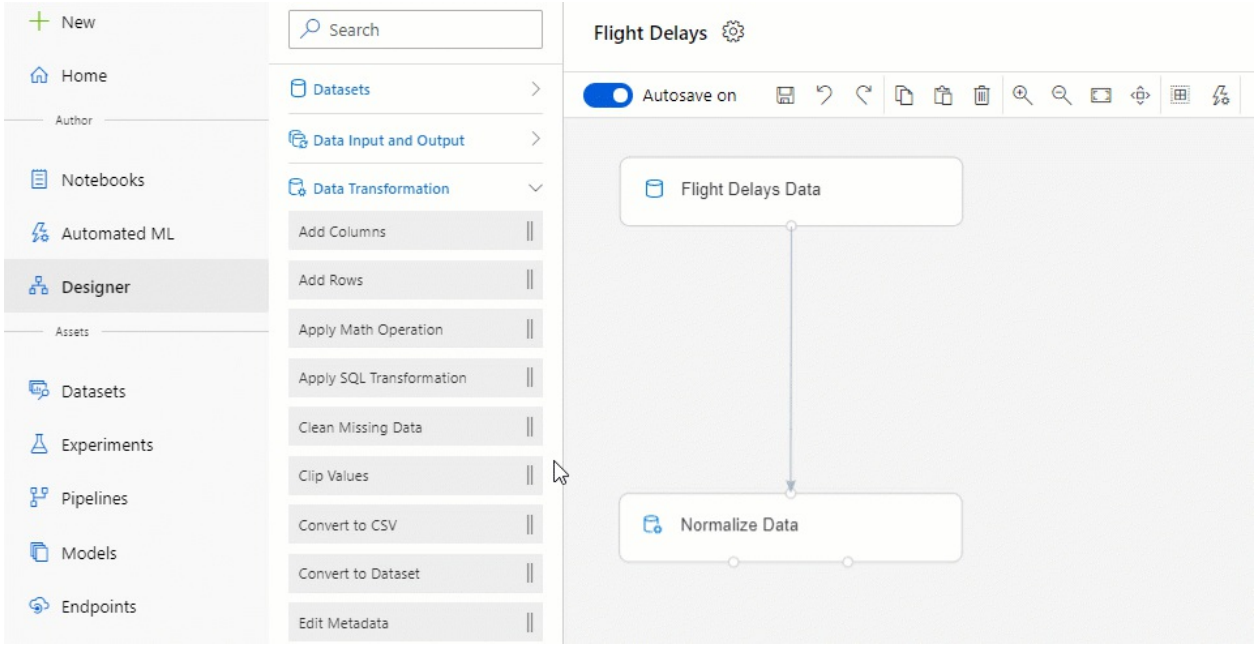
See the how-to for enabling interpretability for models training both locally and on Azure Machine Learning remote compute resources. See the sample notebooks for additional scenarios.

What is Azure Machine Learning designer (preview)?

3/10/2020 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

Azure Machine Learning designer lets you visually connect datasets and modules on an interactive canvas to create machine learning models. To learn how to get started with the designer, see Tutorial: Predict automobile price with the designer



The designer uses your Azure Machine Learning workspace to organize shared resources such as:

- Pipelines
- Datasets
- Compute resources
- Registered models
- Published pipelines
- Real-time endpoints

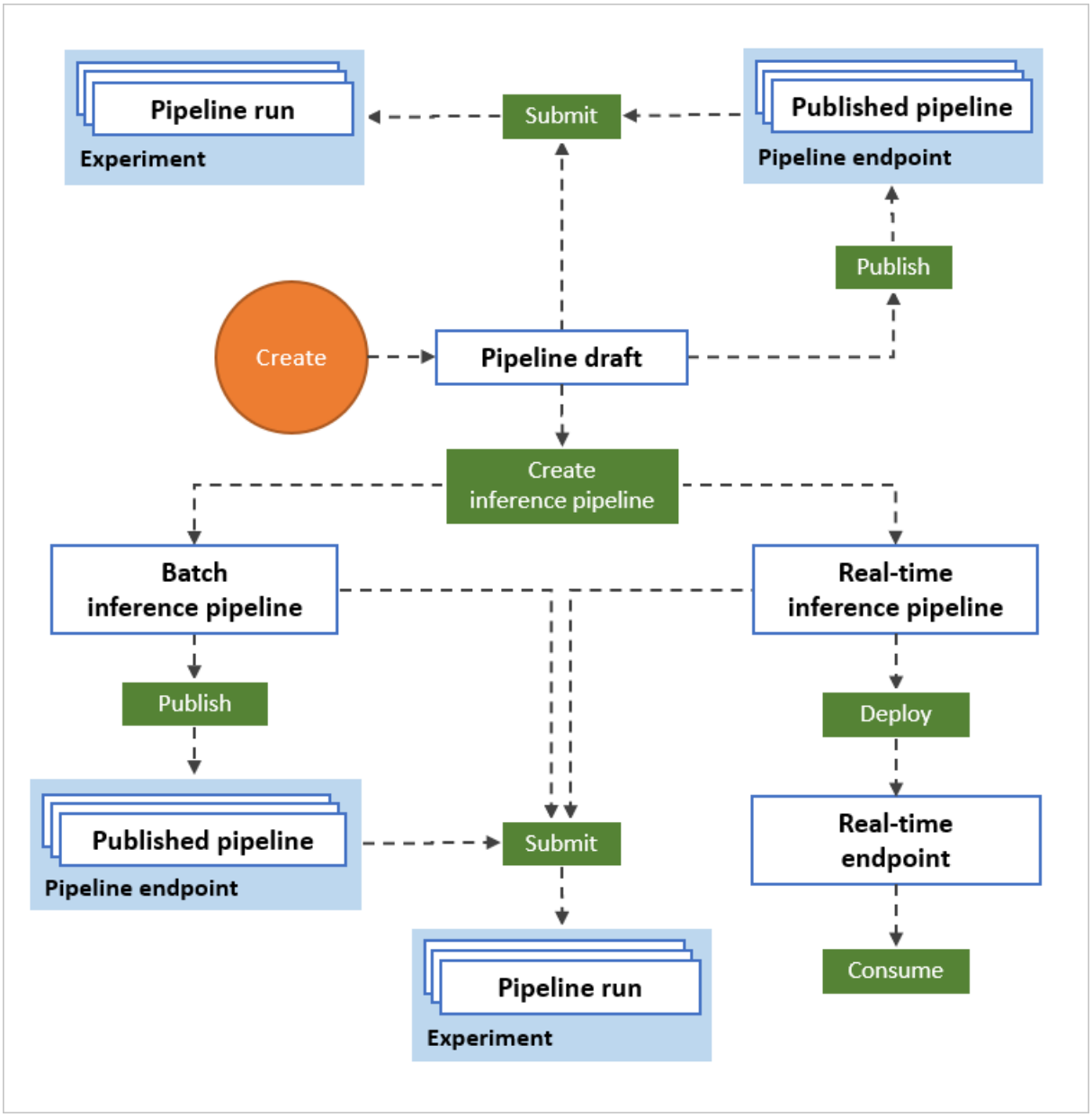
Model training and deployment

The designer gives you a visual canvas to build, test, and deploy machine learning models. With the designer you can:

- Drag-and-drop datasets and modules onto the canvas.
- Connect the modules together to create a pipeline draft.
- Submit a pipeline run using the compute resources in your Azure Machine Learning workspace.
- Convert your **training pipelines** to **inference pipelines**.
- Publish your pipelines to a REST **pipeline endpoint** to submit new pipeline runs with different parameters and datasets.
  - Publish a **training pipeline** to reuse a single pipeline to train multiple models while changing parameters and datasets.
  - Publish a **batch inference pipeline** to make predictions on new data by using a previously trained

model.

- Deploy a **real-time inference pipeline** to a real-time endpoint to make predictions on new data in real time.



Pipeline

A pipeline consists of datasets and analytical modules, which you connect together. Pipelines have many uses: you can make a pipeline that trains a single model, or one that trains multiple models. You can create a pipeline that makes predictions in real time or in batch, or make a pipeline that only cleans data. Pipelines let you reuse your work and organize your projects.

Pipeline draft

As you edit a pipeline in the designer, your progress is saved as a **pipeline draft**. You can edit a pipeline draft at any point by adding or removing modules, configuring compute targets, creating parameters, and so on.

A valid pipeline has these characteristics:

- Datasets can only connect to modules.
- Modules can only connect to either datasets or other modules.
- All input ports for modules must have some connection to the data flow.
- All required parameters for each module must be set.

When you're ready to run your pipeline draft, you submit a pipeline run.

Pipeline run

Each time you run a pipeline, the configuration of the pipeline and its results are stored in your workspace as a **pipeline run**. You can go back to any pipeline run to inspect it for troubleshooting or auditing purposes. **Clone** a pipeline run to create a new pipeline draft for you to edit.

Pipeline runs are grouped into experiments to organize run history. You can set the experiment for every pipeline run.

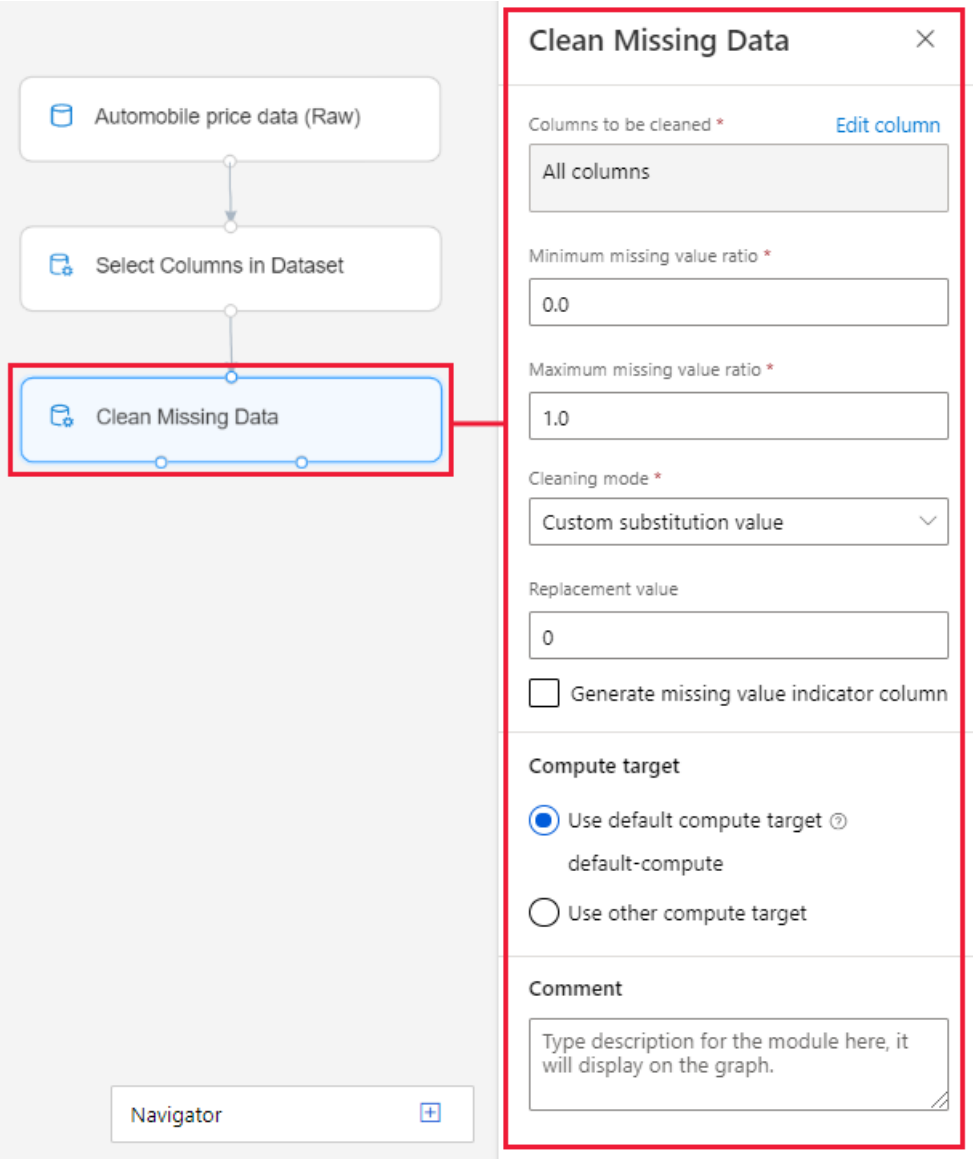
Datasets

A machine learning dataset makes it easy to access and work with your data. A number of sample datasets are included in the designer for you to experiment with. You can register more datasets as you need them.

Module

A module is an algorithm that you can perform on your data. The designer has a number of modules ranging from data ingress functions to training, scoring, and validation processes.

A module may have a set of parameters that you can use to configure the module's internal algorithms. When you select a module on the canvas, the module's parameters are displayed in the Properties pane to the right of the canvas. You can modify the parameters in that pane to tune your model. You can set the compute resources for individual modules in the designer.



For some help navigating through the library of machine learning algorithms available, see Algorithm & module reference overview

Compute resources

Use compute resources from your workspace to run your pipeline and host your deployed models as real-time endpoints or pipeline endpoints (for batch inference). The supported compute targets are:

COMPUTE TARGET	TRAINING	DEPLOYMENT
Azure Machine Learning compute	✓	
Azure Kubernetes Service		✓

Compute targets are attached to your Azure Machine Learning workspace. You manage your compute targets in your workspace in Azure Machine Learning Studio (classic).

Deploy

To perform real-time inferencing, you must deploy a pipeline as a **real-time endpoint**. The real-time endpoint creates an interface between an external application and your scoring model. A call to a real-time endpoint returns prediction results to the application in real time. To make a call to a real-time endpoint, you pass the API key that was created when you deployed the endpoint. The endpoint is based on REST, a popular architecture choice for web programming projects.

Real-time endpoints must be deployed to an Azure Kubernetes Service cluster.

To learn how to deploy your model, see Tutorial: Deploy a machine learning model with the designer.

Publish

You can also publish a pipeline to a **pipeline endpoint**. Similar to a real-time endpoint, a pipeline endpoint lets you submit new pipeline runs from external applications using REST calls. However, you cannot send or receive data in real-time using a pipeline endpoint.

Published pipelines are flexible, they can be used to train or retrain models, perform batch inferencing, process new data, and much more. You can publish multiple pipelines to a single pipeline endpoint and specify which pipeline version to run.

A published pipeline runs on the compute resources you define in the pipeline draft for each module.

The designer creates the same PublishedPipeline object as the SDK.

Moving from the visual interface to the designer

The visual interface (preview) has been updated and is now Azure Machine Learning designer (preview). The designer has been rearchitected to use a pipeline-based backend that fully integrates with the other features of Azure Machine Learning.

As a result of these updates, some concepts and terms for the visual interface have been changed or renamed. See the table below for the most important conceptual changes.

CONCEPT IN THE DESIGNER	PREVIOUSLY IN THE VISUAL INTERFACE
Pipeline draft	Experiment

CONCEPT IN THE DESIGNER	PREVIOUSLY IN THE VISUAL INTERFACE
Real-time endpoint	Web service

Migrating to the designer

You can convert existing visual interface experiments and web services to pipelines and real-time endpoints in the designer. Use the following steps to migrate your visual interface assets:

1. Sign in to Azure Machine Learning studio.
2. Upgrade your workspace to Enterprise edition.

   After upgrading, all of your visual interface experiments will convert to pipeline drafts in the designer.

   > **NOTE**
   > You don't need to upgrade to the Enterprise edition to convert visual interface web services to real-time endpoints.

3. Go to the designer section of the workspace to view your list of pipeline drafts.

   Converted web services can be found by navigating to **Endpoints** > **Real-time endpoints**.

4. Select a pipeline draft to open it.

   If there was an error during the conversion process, an error message will appear with instructions to resolve the issue.

Known issues

Below are known migration issues that need to be addressed manually:

- **Import Data** or **Export Data** modules

  If you have an **Import Data** or **Export Data** module in the experiment, you need to update the data source to use a datastores. To learn how to create a datastore, see How to Access Data in Azure storage services. Your cloud storage account information have been added in the comments of the **Import Data** or **Export Data** module for your convenience.

Next steps

- Learn the basics of predictive analytics and machine learning with Tutorial: Predict automobile price with the designer
- Learn how to modify existing designer samples to adapt them to your needs.

What is automated machine learning (AutoML)?

4/22/2020 • 9 minutes to read • Edit Online

Automated machine learning, also referred to as automated ML or AutoML, is the process of automating the time consuming, iterative tasks of machine learning model development. It allows data scientists, analysts, and developers to build ML models with high scale, efficiency, and productivity all while sustaining model quality. Automated ML is based on a breakthrough from our Microsoft Research division.

Traditional machine learning model development is resource-intensive, requiring significant domain knowledge and time to produce and compare dozens of models. With automated machine learning, you'll accelerate the time it takes to get production-ready ML models with great ease and efficiency.

When to use AutoML: classify, regression, & forecast

Apply automated ML when you want Azure Machine Learning to train and tune a model for you using the target metric you specify. Automated ML democratizes the machine learning model development process, and empowers its users, no matter their data science expertise, to identify an end-to-end machine learning pipeline for any problem.

Data scientists, analysts, and developers across industries can use automated ML to:

- Implement ML solutions without extensive programming knowledge
- Save time and resources
- Leverage data science best practices
- Provide agile problem-solving

Classification

Classification is a common machine learning task. Classification is a type of supervised learning in which models learn using training data, and apply those learnings to new data. Azure Machine Learning offers featurizations specifically for these tasks, such as deep neural network text featurizers for classification. Learn more about featurization options.

The main goal of classification models is to predict which categories new data will fall into based on learnings from its training data. Common classification examples include fraud detection, handwriting recognition, and object detection. Learn more and see an example of classification with automated machine learning.

See examples of classification and automated machine learning in these Python notebooks: Fraud Detection, Marketing Prediction, and Newsgroup Data Classification

Regression

Similar to classification, regression tasks are also a common supervised learning task. Azure Machine Learning offers featurizations specifically for these tasks.

Different from classification where predicted output values are categorical, regression models predict numerical output values based on independent predictors. In regression, the objective is to help establish the relationship among those independent predictor variables by estimating how one variable impacts the others. For example, automobile price based on features like, gas mileage, safety rating, etc. Learn more and see an example of regression with automated machine learning.

See examples of regression and automated machine learning for predictions in these Python notebooks: CPU Performance Prediction,

Time-series forecasting

Building forecasts is an integral part of any business, whether it's revenue, inventory, sales, or customer demand. You can use automated ML to combine techniques and approaches and get a recommended, high-quality time-series forecast. Learn more with this how-to: automated machine learning for time series forecasting.

An automated time-series experiment is treated as a multivariate regression problem. Past time-series values are "pivoted" to become additional dimensions for the regressor together with other predictors. This approach, unlike classical time series methods, has an advantage of naturally incorporating multiple contextual variables and their relationship to one another during training. Automated ML learns a single, but often internally branched model for all items in the dataset and prediction horizons. More data is thus available to estimate model parameters and generalization to unseen series becomes possible.

Advanced forecasting configuration includes:

- holiday detection and featurization
- time-series and DNN learners (Auto-ARIMA, Prophet, ForecastTCN)
- many models support through grouping
- rolling-origin cross validation
- configurable lags
- rolling window aggregate features

See examples of regression and automated machine learning for predictions in these Python notebooks: Sales Forecasting, Demand Forecasting, and Beverage Production Forecast.

How AutoML works

During training, Azure Machine Learning creates a number of pipelines in parallel that try different algorithms and parameters for you. The service iterates through ML algorithms paired with feature selections, where each iteration produces a model with a training score. The higher the score, the better the model is considered to "fit" your data. It will stop once it hits the exit criteria defined in the experiment.

Using **Azure Machine Learning**, you can design and run your automated ML training experiments with these steps:

1. **Identify the ML problem** to be solved: classification, forecasting, or regression
2. **Choose whether you want to use the Python SDK or the studio web experience**: Learn about the parity between the Python SDK and studio web experience.
   - For limited or no code experience, try the Azure Machine Learning studio web experience at https://ml.azure.com
   - For Python developers, check out the Azure Machine Learning Python SDK

   > **IMPORTANT**
   > The functionality in this studio, https://ml.azure.com, is **accessible from Enterprise workspaces only**. Learn more about editions and upgrading.

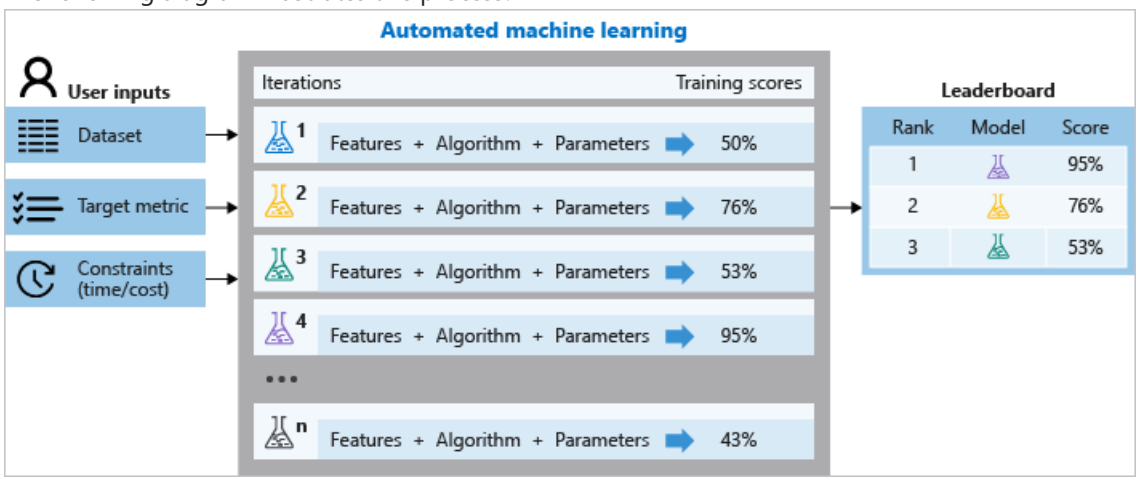
3. **Specify the source and format of the labeled training data**: Numpy arrays or Pandas dataframe
4. **Configure the compute target for model training**, such as your local computer, Azure Machine Learning Computes, remote VMs, or Azure Databricks. Learn about automated training on a remote resource.
5. **Configure the automated machine learning parameters** that determine how many iterations over different models, hyperparameter settings, advanced preprocessing/featurization, and what metrics to look

at when determining the best model.

6. **Submit the training run.**

7. **Review the results**

The following diagram illustrates this process.



You can also inspect the logged run information, which contains metrics gathered during the run. The training run produces a Python serialized object (`.pkl` file) that contains the model and data preprocessing.

While model building is automated, you can also learn how important or relevant features are to the generated models.

Preprocessing

In every automated machine learning experiment, your data is preprocessed using the default methods and optionally through advanced preprocessing.

> **NOTE**
> Automated machine learning pre-processing steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same pre-processing steps applied during training are applied to your input data automatically.

Automatic preprocessing (standard)

In every automated machine learning experiment, your data is automatically scaled or normalized to help algorithms perform well. During model training, one of the following scaling or normalization techniques will be applied to each model.

SCALING & NORMALIZATION	DESCRIPTION
StandardScaleWrapper	Standardize features by removing the mean and scaling to unit variance
MinMaxScalar	Transforms features by scaling each feature by that column's minimum and maximum
MaxAbsScaler	Scale each feature by its maximum absolute value
RobustScalar	This Scaler features by their quantile range

SCALING & NORMALIZATION	DESCRIPTION
PCA	Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space
TruncatedSVDWrapper	This transformer performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). Contrary to PCA, this estimator does not center the data before computing the singular value decomposition, which means it can work with scipy.sparse matrices efficiently
SparseNormalizer	Each sample (that is, each row of the data matrix) with at least one non-zero component is rescaled independently of other samples so that its norm (l1 or l2) equals one

Advanced preprocessing & featurization

Additional advanced preprocessing and featurization are also available, such as data guardrails, encoding, and transforms. Learn more about what featurization is included. Enable this setting with:

- Azure Machine Learning studio: Enable **Automatic featurization** in the **View additional configuration** section with these steps.
- Python SDK: Specifying `"featurization": 'auto' / 'off' / 'FeaturizationConfig'` for the `AutoMLConfig` class.

The studio vs SDK

Learn about the parity and differences between the high-level automated ML capabilities available through the Python SDK and the studio in Azure Machine Learning.

Experiment settings

The following settings allow you to configure your automated ML experiment.

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Split data into train/validation sets	✓	✓
Supports ML tasks: classification, regression, and forecasting	✓	✓
Optimizes based on primary metric	✓	✓
Supports AML compute as compute target	✓	✓
Configure forecast horizon, target lags & rolling window	✓	✓
Set exit criteria	✓	✓
Set concurrent iterations	✓	✓
Drop columns	✓	✓

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Block algorithms	✓	✓
Cross validation	✓	✓
Supports training on Azure Databricks clusters	✓	
View engineered feature names	✓	
Featurization summary	✓	
Featurization for holidays	✓	
Log file verbosity levels	✓	

Model settings

These settings can be applied to the best model as a result of your automated ML experiment.

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Best model registration, deployment, explainability	✓	✓
Enable voting ensemble & stack ensemble models	✓	✓
Show best model based on non-primary metric	✓	
Enable/disable ONNX model compatibility	✓	
Test the model	✓	

Run control settings

These settings allow you to review and control your experiment runs and its child runs.

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Run summary table	✓	✓
Cancel runs & child runs	✓	✓
Get guardrails	✓	✓
Pause & resume runs	✓	

Ensemble models

Automated machine learning supports ensemble models, which are enabled by default. Ensemble learning improves machine learning results and predictive performance by combining multiple models as opposed to

using single models. The ensemble iterations appear as the final iterations of your run. Automated machine learning uses both voting and stacking ensemble methods for combining models:

- **Voting**: predicts based on the weighted average of predicted class probabilities (for classification tasks) or predicted regression targets (for regression tasks).
- **Stacking**: stacking combines heterogenous models and trains a meta-model based on the output from the individual models. The current default meta-models are LogisticRegression for classification tasks and ElasticNet for regression/forecasting tasks.

The Caruana ensemble selection algorithm with sorted ensemble initialization is used to decide which models to use within the ensemble. At a high level, this algorithm initializes the ensemble with up to five models with the best individual scores, and verifies that these models are within 5% threshold of the best score to avoid a poor initial ensemble. Then for each ensemble iteration, a new model is added to the existing ensemble and the resulting score is calculated. If a new model improved the existing ensemble score, the ensemble is updated to include the new model.

See the how-to for changing default ensemble settings in automated machine learning.

AutoML & ONNX

With Azure Machine Learning, you can use automated ML to build a Python model and have it converted to the ONNX format. Once the models are in the ONNX format, they can be run on a variety of platforms and devices. Learn more about accelerating ML models with ONNX.

See how to convert to ONNX format in this Jupyter notebook example. Learn which algorithms are supported in ONNX.

The ONNX runtime also supports C#, so you can use the model built automatically in your C# apps without any need for recoding or any of the network latencies that REST endpoints introduce. Learn more about inferencing ONNX models with the ONNX runtime C# API.

Next steps

See examples and learn how to build models using automated machine learning:

- Follow the Tutorial: Automatically train a regression model with Azure Machine Learning
- Configure the settings for automatic training experiment:
  - In Azure Machine Learning studio, use these steps.
  - With the Python SDK, use these steps.
- Learn how to auto train using time series data, use these steps.
- Try out Jupyter Notebook samples for automated machine learning
- Automated ML is also available in other Microsoft solutions such as, ML.NET, HDInsight, Power BI and SQL Server

Prevent overfitting and imbalanced data with automated machine learning

4/7/2020 • 6 minutes to read • Edit Online

Over-fitting and imbalanced data are common pitfalls when you build machine learning models. By default, Azure Machine Learning's automated machine learning provides charts and metrics to help you identify these risks, and implements best practices to help mitigate them.

Identify over-fitting

Over-fitting in machine learning occurs when a model fits the training data too well, and as a result can't accurately predict on unseen test data. In other words, the model has simply memorized specific patterns and noise in the training data, but is not flexible enough to make predictions on real data.

Consider the following trained models and their corresponding train and test accuracies.

MODEL	TRAIN ACCURACY	TEST ACCURACY
A	99.9%	95%
B	87%	87%
C	99.9%	45%

Considering model **A**, there is a common misconception that if test accuracy on unseen data is lower than training accuracy, the model is over-fitted. However, test accuracy should always be less than training accuracy, and the distinction for over-fit vs. appropriately fit comes down to *how much* less accurate.

When comparing models **A** and **B**, model **A** is a better model because it has higher test accuracy, and although the test accuracy is slightly lower at 95%, it is not a significant difference that suggests over-fitting is present. You wouldn't choose model **B** simply because the train and test accuracies are closer together.

Model **C** represents a clear case of over-fitting; the training accuracy is very high but the test accuracy isn't anywhere near as high. This distinction is subjective, but comes from knowledge of your problem and data, and what magnitudes of error are acceptable.

Prevent over-fitting

In the most egregious cases, an over-fitted model will assume that the feature value combinations seen during training will always result in the exact same output for the target.

The best way to prevent over-fitting is to follow ML best-practices including:

- Using more training data, and eliminating statistical bias
- Preventing target leakage
- Using fewer features
- **Regularization and hyperparameter optimization**
- **Model complexity limitations**
- **Cross-validation**

In the context of automated ML, the first three items above are **best-practices you implement**. The last three bolded items are **best-practices automated ML implements** by default to protect against over-fitting. In settings other than automated ML, all six best-practices are worth following to avoid over-fitting models.

Best practices you implement

Using **more data** is the simplest and best possible way to prevent over-fitting, and as an added bonus typically increases accuracy. When you use more data, it becomes harder for the model to memorize exact patterns, and it is forced to reach solutions that are more flexible to accommodate more conditions. It's also important to recognize **statistical bias**, to ensure your training data doesn't include isolated patterns that won't exist in live-prediction data. This scenario can be difficult to solve, because there may not be over-fitting between your train and test sets, but there may be over-fitting present when compared to live test data.

Target leakage is a similar issue, where you may not see over-fitting between train/test sets, but rather it appears at prediction-time. Target leakage occurs when your model "cheats" during training by having access to data that it shouldn't normally have at prediction-time. For example, if your problem is to predict on Monday what a commodity price will be on Friday, but one of your features accidentally included data from Thursdays, that would be data the model won't have at prediction-time since it cannot see into the future. Target leakage is an easy mistake to miss, but is often characterized by abnormally high accuracy for your problem. If you are attempting to predict stock price and trained a model at 95% accuracy, there is likely target leakage somewhere in your features.

Removing features can also help with over-fitting by preventing the model from having too many fields to use to memorize specific patterns, thus causing it to be more flexible. It can be difficult to measure quantitatively, but if you can remove features and retain the same accuracy, you have likely made the model more flexible and have reduced the risk of over-fitting.

Best practices automated ML implements

Regularization is the process of minimizing a cost function to penalize complex and over-fitted models. There are different types of regularization functions, but in general they all penalize model coefficient size, variance, and complexity. Automated ML uses L1 (Lasso), L2 (Ridge), and ElasticNet (L1 and L2 simultaneously) in different combinations with different model hyperparameter settings that control over-fitting. In simple terms, automated ML will vary how much a model is regulated and choose the best result.

Automated ML also implements explicit model complexity limitations to prevent over-fitting. In most cases this implementation is specifically for decision tree or forest algorithms, where individual tree max-depth is limited, and the total number of trees used in forest or ensemble techniques are limited.

Cross-validation (CV) is the process of taking many subsets of your full training data and training a model on each subset. The idea is that a model could get "lucky" and have great accuracy with one subset, but by using many subsets the model won't achieve this high accuracy every time. When doing CV, you provide a validation holdout dataset, specify your CV folds (number of subsets) and automated ML will train your model and tune hyperparameters to minimize error on your validation set. One CV fold could be over-fit, but by using many of them it reduces the probability that your final model is over-fit. The tradeoff is that CV does result in longer training times and thus greater cost, because instead of training a model once, you train it once for each *n* CV subsets.

> **NOTE**
> Cross-validation is not enabled by default; it must be configured in automated ML settings. However, after cross-validation is configured and a validation data set has been provided, the process is automated for you. See

Identify models with imbalanced data

Imbalanced data is commonly found in data for machine learning classification scenarios, and refers to data that contains a disproportionate ratio of observations in each class. This imbalance can lead to a falsely perceived positive effect of a model's accuracy, because the input data has bias towards one class, which results in the trained

model to mimic that bias.

As classification algorithms are commonly evaluated by accuracy, checking a model's accuracy score is a good way to identify if it was impacted by imbalanced data. Did it have really high accuracy or really low accuracy for certain classes?

In addition, automated ML runs generate the following charts automatically, which can help you understand the correctness of the classifications of your model, and identify models potentially impacted by imbalanced data.

CHART	DESCRIPTION
Confusion Matrix	Evaluates the correctly classified labels against the actual labels of the data.
Precision-recall	Evaluates the ratio of correct labels against the ratio of found label instances of the data
ROC Curves	Evaluates the ratio of correct labels against the ratio of false-positive labels.

Handle imbalanced data

As part of its goal of simplifying the machine learning workflow, automated ML has built in capabilities to help deal with imbalanced data such as,

- A **weight column**: automated ML supports a weighted column as input, causing rows in the data to be weighted up or down, which can make a class more or less "important".
- The algorithms used by automated ML can properly handle imbalance of up to 20:1, meaning the most common class can have 20 times more rows in the data than the least common class.

The following techniques are additional options to handle imbalanced data outside of automated ML.

- Resampling to even the class imbalance, either by up-sampling the smaller classes or down-sampling the larger classes. These methods require expertise to process and analyze.
- Use a performance metric that deals better with imbalanced data. For example, the F1 score is a weighted average of precision and recall. Precision measures a classifier's exactness-- low precision indicates a high number of false positives--, while recall measures a classifier's completeness-- low recall indicates a high number of false negatives.

Next steps

See examples and learn how to build models using automated machine learning:

- Follow the Tutorial: Automatically train a regression model with Azure Machine Learning
- Configure the settings for automatic training experiment:
  - In Azure Machine Learning studio, use these steps.
  - With the Python SDK, use these steps.

What is an Azure Machine Learning compute instance?

2/14/2020 • 6 minutes to read • Edit Online

An Azure Machine Learning compute instance (preview) is a fully-managed cloud-based workstation for data scientists.

Compute instances make it easy to get started with Azure Machine Learning development as well as provide management and enterprise readiness capabilities for IT administrators.

Use a compute instance as your fully configured and managed development environment in the cloud.

Compute instances are typically used as development environments. They can also be used as a compute target for training and inferencing for development and testing. For large tasks, an Azure Machine Learning compute cluster with multi-node scaling capabilities is a better compute target choice.

Why use a compute instance?

A compute instance is a fully-managed cloud-based workstation optimized for your machine learning development environment. It provides the following benefits:

KEY BENEFITS	
Productivity	Data scientists can build and deploy models using integrated notebooks and the following tools in their web browser: - Jupyter - JupyterLab - RStudio
Managed & secure	Reduce your security footprint and add compliance with enterprise security requirements. Compute instances provide robust management policies and secure networking configurations such as: - Auto-provisioning from Resource Manager templates or Azure Machine Learning SDK - Role-based access control (RBAC) - Virtual network support - SSH policy to enable/disable SSH access
Preconfigured or ML	Save time on setup tasks with pre-configured and up-to-date ML packages, deep learning frameworks, GPU drivers.
Fully customizable	Broad support for Azure VM types including GPUs and persisted low-level customization such as installing packages and drivers makes advanced scenarios a breeze.

Tools and environments

Azure Machine Learning compute instance enables you to author, train, and deploy models in a fully integrated notebook experience in your workspace.

These tools and environments are installed on the compute instance:

GENERAL TOOLS & ENVIRONMENTS	DETAILS

Drivers	`CUDA`
	`cuDNN`
	`NVIDIA`
	`Blob FUSE`

Intel MPI library	
Azure CLI	
Azure Machine Learning samples	
Azure Machine Learning EDAT engine	
Docker	
Nginx	
NCCL 2.0	
Protobuf	

R TOOLS & ENVIRONMENTS	DETAILS
RStudio Server Open Source Edition	
R kernel	
Azure Machine Learning SDK for R	azuremlsdk SDK samples

PYTHON TOOLS & ENVIRONMENTS	DETAILS
Anaconda Python	
Jupyter and extensions	
Jupyterlab and extensions	
Visual Studio Code	

Azure Machine Learning SDK for Python
from PyPI
	`azureml-sdk[notebooks,contrib,automl,explain]`
	`azureml-contrib-datadrift`
	`azureml-telemetry`
	`azureml-tensorboard`
	`azureml-contrib-opendatasets`
	`azureml-opendatasets`
	`azureml-contrib-reinforcementlearning`
	`azureml-mlflow`
	`azureml-contrib-interpret`

PYTHON TOOLS & ENVIRONMENTS	DETAILS

Other PyPI packages	`jupytext`
	`jupyterlab-git`
	`tensorboard`
	`nbconvert`
	`notebook`
	`Pillow`

Conda packages	`cython`
	`numpy`
	`ipykernel`
	`scikit-learn`
	`matplotlib`
	`tqdm`
	`joblib`
	`nodejs`
	`nb_conda_kernels`

Deep learning packages	`PyTorch`
	`TensorFlow`
	`Keras`
	`Horovod`
	`MLFlow`
	`pandas-ml`
	`scrapbook`

ONNX packages	`keras2onnx`
	`onnx`
	`onnxconverter-common`
	`skl2onnx`
	`onnxmltools`

Azure Machine Learning Python & R SDK samples	

Python packages are all installed in the **Python 3.6 - AzureML** environment.

Compute instances are typically used as development environments. They can also be used as a compute target for training and inferencing for development and testing. For large tasks, an Azure Machine Learning compute cluster with multi-node scaling capabilities is a better compute target choice.

Installing packages

You can install packages directly in a Jupyter notebook or Rstudio:

- RStudio Use the **Packages** tab on the bottom right, or the **Console** tab on the top left.
- Python: Add install code and execute in a Jupyter notebook cell.

Or you can access a terminal window in any of these ways:

- RStudio: Select the **Terminal** tab on top left.
- Jupyter Lab: Select the **Terminal** tile under the **Other** heading in the Launcher tab.
- Jupyter: Select **New > Terminal** on top right in the Files tab.
- SSH to the machine. Then install Python packages into the **Python 3.6 - AzureML** environment. Install R packages into the **R** environment.

Accessing files

Notebooks and R scripts are stored in the default storage account of your workspace in Azure file share. These files are located under your "User files" directory. This storage makes it easy to share notebooks between compute instances. The storage account also keeps your notebooks safely preserved when you stop or delete a compute instance.

The Azure file share account of your workspace is mounted as a drive on the compute instance. This drive is the default working directory for Jupyter, Jupyter Labs, and RStudio.

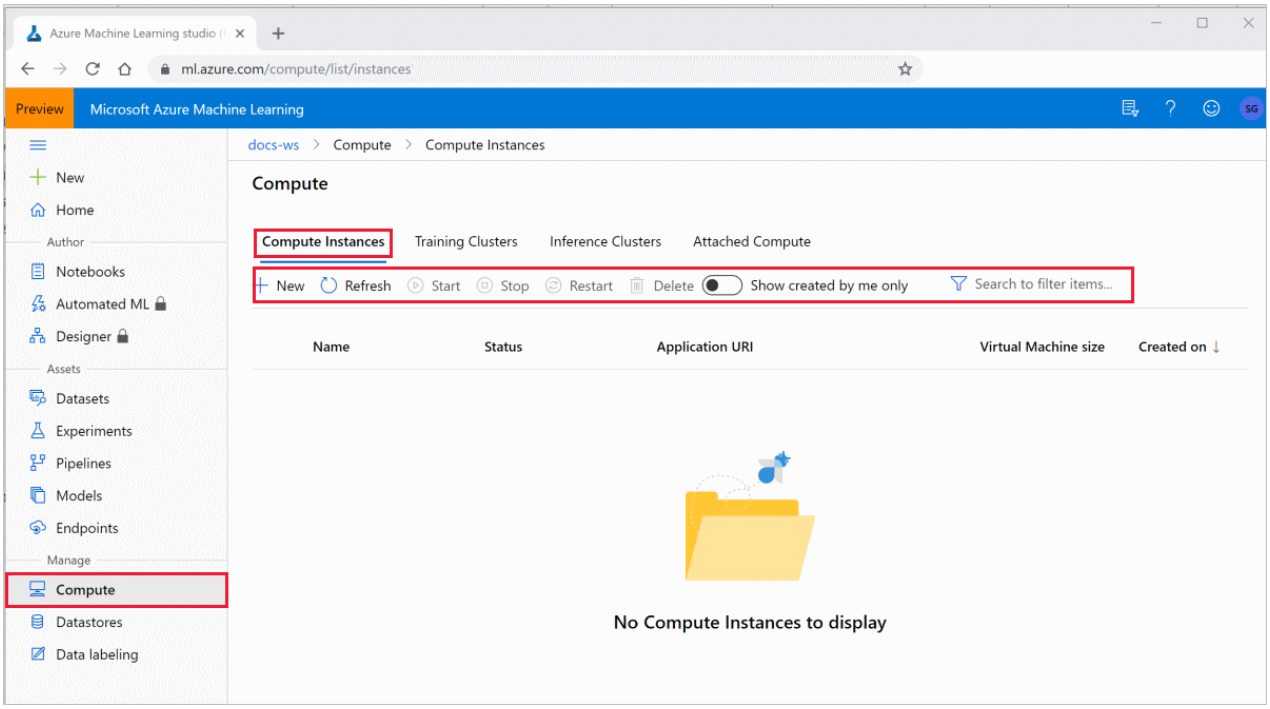
The files in the file share are accessible from all compute instances in the same workspace. Any changes to these files on the compute instance will be reliably persisted back to the file share.

You can also clone the latest Azure Machine Learning samples to your folder under the user files directory in the workspace file share.

Writing small files can be slower on network drives than writing to the VM itself. If you are writing many small files, try using a directory directly on the compute instance, such as a `/tmp` directory. Please note these files will not be accessible from other compute instances in the workspace.

Managing a compute instance

In your workspace in Azure Machine Learning studio, select **Compute**, then select **Compute Instance** on the top.



You can perform the following actions:

- Create a compute instance. Specify the name, Azure VM type including GPUs (please note VM type can not be changed after creation), enable/disable SSH access, and configure virtual network settings optionally. You can also create an instance directly from integrated notebooks, Azure portal, Resource Manager template, or Azure Machine Learning SDK. The dedicated cores per region quota which applies to compute instance creation is unified and shared with Azure Machine Learning compute cluster quota.
- Refresh the compute instances tab
- Start, stop and restart a compute instance
- Delete a compute instance

For each compute instance in your workspace you can:

- Access Jupyter, JupyterLab, RStudio on the compute instance
- SSH into compute instance. SSH access is disabled by default but can be enabled at compute instance creation time. SSH access is through public/private key mechanism. The tab will give you details for SSH connection such as IP address, username, and port number.
- Get details about a specific compute instance such as IP address, and region.

RBAC allows you to control which users in the workspace can create, delete, start, stop, restart a compute instance. All users in the workspace contributor and owner role can create, delete, start, stop, and restart compute instances across the workspace. However, only the creator of a specific compute instance is allowed to access Jupyter, JupyterLab, and RStudio on that compute instance. The creator of the compute instance has the compute instance dedicated to them, have root access, and can terminal in through Jupyter. Compute instance will have single-user login of creator user and all actions will use that user's identity for RBAC and attribution of experiment runs. SSH access is controlled through public/private key mechanism.

You can also create an instance

- Directly from the integrated notebooks experience
- In Azure portal
- From Azure Resource Manager template
- With Azure Machine Learning SDK

The dedicated cores per region quota, which applies to compute instance creation is unified and shared with Azure Machine Learning training cluster quota.

Compute Target

Compute instances can be used as a training compute target similar to Azure Machine Learning compute training clusters. Provision a multi-GPU VM to run distributed training jobs using TensorFlow/PyTorch estimators. You can also create a run configuration and use it to run your experiment on compute instance. You can use compute instance as a local inferencing deployment target for testing/debugging scenarios.

What happened to Notebook VM?

Compute instances are replacing the Notebook VM.

Any notebook files stored in the workspace file share and data in workspace data stores will be accessible from a compute instance. However, any custom packages previously installed on a Notebook VM will need to be re-installed on the compute instance. Quota limitations which apply to compute clusters creation will apply to compute instance creation as well.

New Notebook VMs cannot be created. However, you can still access and use Notebook VMs you have created, with full functionality. Compute instances can be created in same workspace as the existing Notebook VMs.

Next steps

- Tutorial: Train your first ML model shows how to use a compute instance with an integrated notebook.

What are compute targets in Azure Machine Learning?

3/30/2020 • 3 minutes to read • Edit Online

A **compute target** is a designated compute resource/environment where you run your training script or host your service deployment. This location may be your local machine or a cloud-based compute resource. Using compute targets make it easy for you to later change your compute environment without having to change your code.

In a typical model development lifecycle, you might:

1. Start by developing and experimenting on a small amount of data. At this stage, we recommend your local environment (local computer or cloud-based VM) as your compute target.
2. Scale up to larger data, or do distributed training using one of these training compute targets.
3. Once your model is ready, deploy it to a web hosting environment or IoT device with one of these deployment compute targets.

The compute resources you use for your compute targets are attached to a workspace. Compute resources other than the local machine are shared by users of the workspace.

Training compute targets

Azure Machine Learning has varying support across different compute resources. You can also attach your own compute resource, although support for various scenarios may vary.

Compute targets can be reused from one training job to the next. For example, once you attach a remote VM to your workspace, you can reuse it for multiple jobs. For machine learning pipelines, use the appropriate pipeline step for each compute target.

TRAINING TARGETS	AUTOMATED ML	ML PIPELINES	AZURE MACHINE LEARNING DESIGNER
Local computer	yes		
Azure Machine Learning compute cluster	yes & hyperparameter tuning	yes	yes
Remote VM	yes & hyperparameter tuning	yes	
Azure Databricks	yes (SDK local mode only)	yes	
Azure Data Lake Analytics		yes	
Azure HDInsight		yes	
Azure Batch		yes	

Learn more about setting up and using a compute target for model training.

Deployment targets

The following compute resources can be used to host your model deployment.

COMPUTE TARGET	USED FOR	GPU SUPPORT	FPGA SUPPORT	DESCRIPTION
Local web service	Testing/debugging			Use for limited testing and troubleshooting. Hardware acceleration depends on use of libraries in the local system.
Azure Machine Learning compute instance web service	Testing/debugging			Use for limited testing and troubleshooting.
Azure Kubernetes Service (AKS)	Real-time inference	Yes (web service deployment)	Yes	Use for high-scale production deployments. Provides fast response time and autoscaling of the deployed service. Cluster autoscaling isn't supported through the Azure Machine Learning SDK. To change the nodes in the AKS cluster, use the UI for your AKS cluster in the Azure portal. AKS is the only option available for the designer.
Azure Container Instances	Testing or development			Use for low-scale CPU-based workloads that require less than 48 GB of RAM.
Azure Machine Learning compute clusters	(Preview) Batch inference	Yes (machine learning pipeline)		Run batch scoring on serverless compute. Supports normal and low-priority VMs.
Azure Functions	(Preview) Real-time inference			
Azure IoT Edge	(Preview) IoT module			Deploy and serve ML models on IoT devices.
Azure Data Box Edge	Via IoT Edge		Yes	Deploy and serve ML models on IoT devices.

> **NOTE**
>
> Although compute targets like local, Azure Machine Learning compute instance, and Azure Machine Learning compute clusters support GPU for training and experimentation, using GPU for inference **when deployed as a web service** is supported only on Azure Kubernetes Service.
>
> Using a GPU for inference **when scoring with a machine learning pipeline** is supported only on Azure Machine Learning Compute.

Learn where and how to deploy your model to a compute target.

Azure Machine Learning compute (managed)

A managed compute resource is created and managed by Azure Machine Learning. This compute is optimized for machine learning workloads. Azure Machine Learning compute clusters and compute instances are the only managed computes. Additional managed compute resources may be added in the future.

You can create Azure Machine Learning compute instances (preview) or compute clusters from:

- Azure Machine Learning studio
- Azure portal
- Python SDK ComputeInstance and AmlCompute classes
- R SDK
- Resource Manager template

You can also create compute clusters using the machine learning extension for the Azure CLI.

When created these compute resources are automatically part of your workspace unlike other kinds of compute targets.

Compute clusters

You can use Azure Machine Learning compute clusters for training and for batch inferencing (preview). With this compute resource, you have:

- Single- or multi-node cluster
- Autoscales each time you submit a run
- Automatic cluster management and job scheduling
- Support for both CPU and GPU resources

Unmanaged compute

An unmanaged compute target is *not* managed by Azure Machine Learning. You create this type of compute target outside Azure Machine Learning, then attach it to your workspace. Unmanaged compute resources can require additional steps for you to maintain or to improve performance for machine learning workloads.

Next steps

Learn how to:

- Set up a compute target to train your model
- Deploy your model to a compute target

What are Azure Machine Learning pipelines?

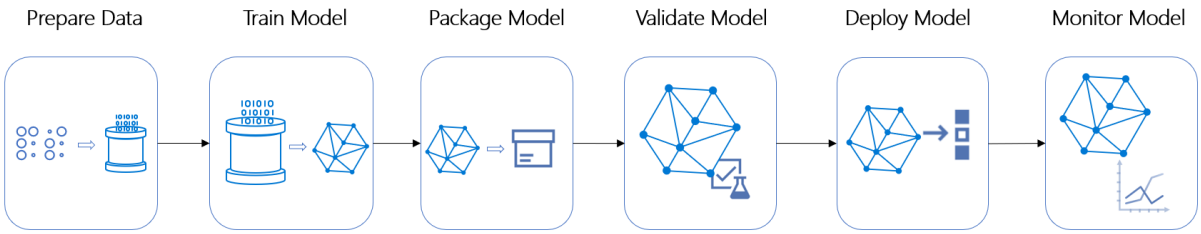
4/7/2020 • 15 minutes to read • Edit Online

Azure Machine Learning pipelines allow you to create workflows in your machine learning projects. These workflows have a number of benefits:

- Simplicity
- Speed
- Repeatability
- Flexibility
- Versioning and tracking
- Modularity
- Quality assurance
- Cost control

These benefits become significant as soon as your machine learning project moves beyond pure exploration and into iteration. Even simple one-step pipelines can be valuable. Machine learning projects are often in a complex state, and it can be a relief to make the precise accomplishment of a single workflow a trivial process.

Learn how to create your first pipeline.



Which Azure pipeline technology should I use?

The Azure cloud provides several other pipelines, each with a different purpose. The following table lists the different pipelines and what they are used for:

SCENARIO	PRIMARY PERSONA	AZURE OFFERING	OSS OFFERING	CANONICAL PIPE	STRENGTHS
Model orchestration (Machine learning)	Data scientist	Azure Machine Learning Pipelines	Kubeflow Pipelines	Data -> Model	Distribution, caching, code-first, reuse
Data orchestration (Data prep)	Data engineer	Azure Data Factory pipelines	Apache Airflow	Data -> Data	Strongly-typed movement. Data-centric activities.
Code & app orchestration (CI/CD)	App Developer / Ops	Azure DevOps Pipelines	Jenkins	Code + Model -> App/Service	Most open and flexible activity support, approval queues, phases with gating

What can Azure ML pipelines do?

An Azure Machine Learning pipeline is an independently executable workflow of a complete machine learning task. Subtasks are encapsulated as a series of steps within the pipeline. An Azure Machine Learning pipeline can be as simple as one that calls a Python script, so *may* do just about anything. Pipelines *should* focus on machine learning tasks such as:

- Data preparation including importing, validating and cleaning, munging and transformation, normalization, and staging
- Training configuration including parameterizing arguments, filepaths, and logging / reporting configurations
- Training and validating efficiently and repeatedly. Efficiency might come from specifying specific data subsets, different hardware compute resources, distributed processing, and progress monitoring
- Deployment, including versioning, scaling, provisioning, and access control

Independent steps allow multiple data scientists to work on the same pipeline at the same time without over-taxing compute resources. Separate steps also make it easy to use different compute types/sizes for each step.

After the pipeline is designed, there is often more fine-tuning around the training loop of the pipeline. When you rerun a pipeline, the run jumps to the steps that need to be rerun, such as an updated training script. Steps that do not need to be rerun are skipped. The same analysis applies to unchanged scripts used for the accomplishment of the step. This reuse functionality helps to avoid running costly and time-intensive steps like data ingestion and transformation if the underlying data hasn't changed.

With Azure Machine Learning, you can use various toolkits and frameworks, such as PyTorch or TensorFlow, for each step in your pipeline. Azure coordinates the various compute targets you use, so your intermediate data can be shared with the downstream compute targets.

You can track the metrics for your pipeline experiments directly in Azure portal or your workspace landing page (preview). After a pipeline has been published, you can configure a REST endpoint, which allows you to rerun the pipeline from any platform or stack.

In short, all of the complex tasks of the machine learning lifecycle can be helped with pipelines. Other Azure pipeline technologies have their own strengths. Azure Data Factory pipelines excels at working with data and Azure Pipelines is the right tool for continuous integration and deployment. But if your focus is machine learning, Azure Machine Learning pipelines are likely to be the best choice for your workflow needs.

What are Azure ML pipelines?

An Azure ML pipeline performs a complete logical workflow with an ordered sequence of steps. Each step is a discrete processing action. Pipelines run in the context of an Azure Machine Learning Experiment.

In the early stages of an ML project, it's fine to have a single Jupyter notebook or Python script that does all the work of Azure workspace and resource configuration, data preparation, run configuration, training, and validation. But just as functions and classes quickly become preferable to a single imperative block of code, ML workflows quickly become preferable to a monolithic notebook or script.

By modularizing ML tasks, pipelines support the Computer Science imperative that a component should "do (only) one thing well." Modularity is clearly vital to project success when programming in teams, but even when working alone, even a small ML project involves separate tasks, each with a good amount of complexity. Tasks include: workspace configuration and data access, data preparation, model definition and configuration, and deployment. While the outputs of one or more tasks form the inputs to another, the exact implementation details of any one task are, at best, irrelevant distractions in the next. At worst, the computational state of one task can cause a bug in another.

Analyzing dependencies

Many programming ecosystems have tools that orchestrate resource, library, or compilation dependencies.

Generally, these tools use file timestamps to calculate dependencies. When a file is changed, only it and its dependents are updated (downloaded, recompiled, or packaged). Azure ML pipelines extend this concept dramatically. Like traditional build tools, pipelines calculate dependencies between steps and only perform the necessary recalculations.

The dependency analysis in Azure ML pipelines is more sophisticated than simple timestamps though. Every step may run in a different hardware and software environment. Data preparation might be a time-consuming process but not need to run on hardware with powerful GPUs, certain steps might require OS-specific software, you might want to use distributed training, and so forth. While the cost savings for optimizing resources may be significant, it can be overwhelming to manually juggle all the different variations in hardware and software resources. It's even harder to do all that without ever making a mistake in the data you transfer between steps.

Pipelines solve this problem. Azure Machine Learning automatically orchestrates all of the dependencies between pipeline steps. This orchestration might include spinning up and down Docker images, attaching and detaching compute resources, and moving data between the steps in a consistent and automatic manner.

Reusing results

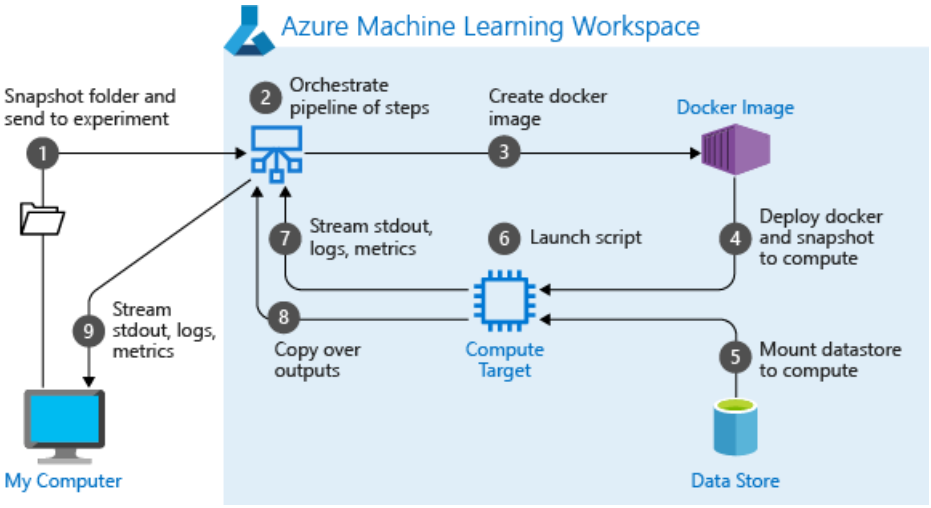
Additionally, the output of a step may, if you choose, be reused. If you specify reuse as a possibility and there are no upstream dependencies triggering recalculation, the pipeline service will use a cached version of the step's results. Such reuse can dramatically decrease development time. If you have a complex data preparation task, you probably rerun it more often than is strictly necessary. Pipelines relieve you of that worry: if necessary, the step will run, if not, it won't.

All of this dependency analysis, orchestration, and activation are handled by Azure Machine Learning when you instantiate a Pipeline object, pass it to an `Experiment`, and call `submit()`.

Coordinating the steps involved

When you create and run a `Pipeline` object, the following high-level steps occur:

- For each step, the service calculates requirements for:
  - Hardware compute resources
  - OS resources (Docker image(s))
  - Software resources (Conda / virtualenv dependencies)
  - Data inputs
- The service determines the dependencies between steps, resulting in a dynamic execution graph
- When each node in the execution graph runs:
  - The service configures the necessary hardware and software environment (perhaps reusing existing resources)
  - The step runs, providing logging and monitoring information to its containing `Experiment` object
  - When the step completes, its outputs are prepared as inputs to the next step and/or written to storage
  - Resources that are no longer needed are finalized and detached



Building pipelines with the Python SDK

In the Azure Machine Learning Python SDK, a pipeline is a Python object defined in the `azureml.pipeline.core` module. A Pipeline object contains an ordered sequence of one or more PipelineStep objects. The `PipelineStep` class is abstract and the actual steps will be of subclasses such as EstimatorStep, PythonScriptStep, or DataTransferStep. The ModuleStep class holds a reusable sequence of steps that can be shared among pipelines. A `Pipeline` runs as part of an `Experiment`.

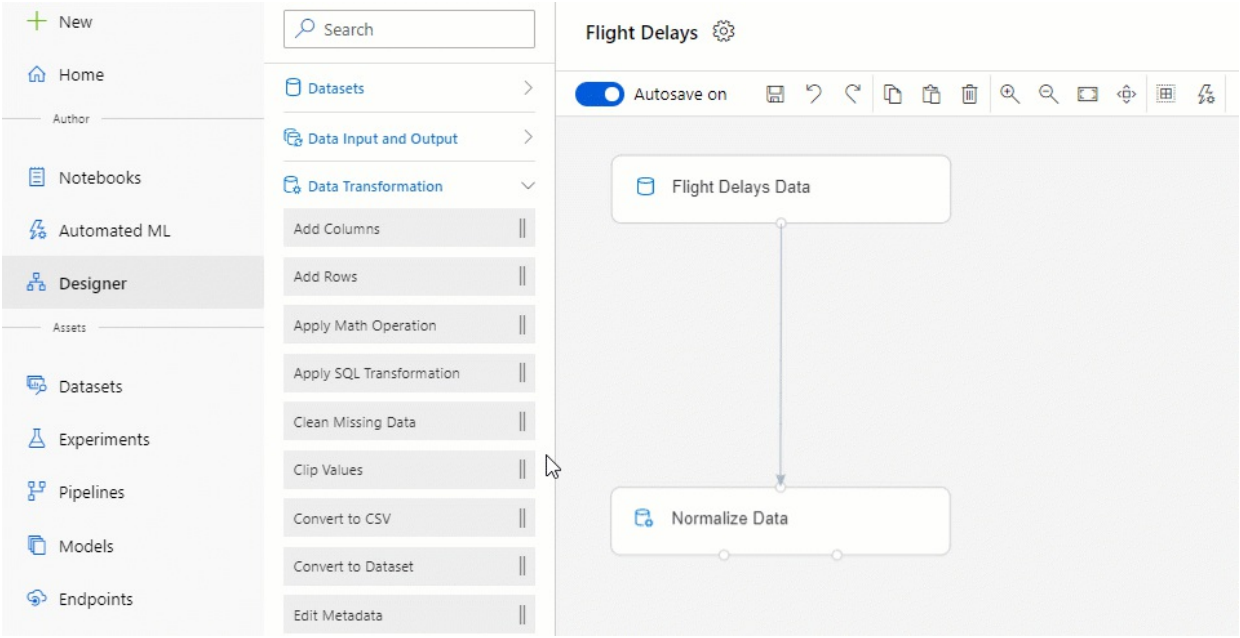
An Azure ML pipeline is associated with an Azure Machine Learning workspace and a pipeline step is associated with a compute target available within that workspace. For more information, see Create and manage Azure Machine Learning workspaces in the Azure portal or What are compute targets in Azure Machine Learning?.

In Azure Machine Learning, a compute target is the environment in which an ML phase occurs. The software environment may be a Remote VM, Azure Machine Learning Compute, Azure Databricks, Azure Batch, and so on. The hardware environment can also vary greatly, depending on GPU support, memory, storage, and so forth. You may specify the compute target for each step, which gives you fine-grained control over costs. You can use more- or less- powerful resources for the specific action, data volume, and performance needs of your project.

Building pipelines with the designer

Developers who prefer a visual design surface can use the Azure Machine Learning designer to create pipelines. You can access this tool from the **Designer** selection on the homepage of your workspace. The designer allows you to drag and drop steps onto the design surface. For rapid development, you can use existing modules across the spectrum of ML tasks; existing modules cover everything from data transformation to algorithm selection to training to deployment. Or you can create a fully custom pipeline by combining your own steps defined in Python scripts.

When you visually design pipelines, the inputs and outputs of a step are displayed visibly. You can drag and drop data connections, allowing you to quickly understand and modify the dataflow of your pipeline.



Understanding the execution graph

The steps within a pipeline may have dependencies on other steps. The Azure ML pipeline service does the work of analyzing and orchestrating these dependencies. The nodes in the resulting "execution graph" are processing steps. Each step may involve creating or reusing a particular combination of hardware and software, reusing cached results, and so on. The service's orchestration and optimization of this execution graph can significantly speed up an ML phase and reduce costs.

Because steps run independently, objects to hold the input and output data that flows between steps must be defined externally. This is the role of DataSetand PipelineData, objects. These data objects are associated with a Datastore object that encapsulates their storage configuration. The `PipelineStep` base class is always created with a `name` string, a list of `inputs`, and a list of `outputs`. Usually, it also has a list of `arguments` and often it will have a list of `resource_inputs`. Subclasses will generally have additional arguments as well (for instance, `PythonScriptStep` requires the filename and path of the script to run).

The execution graph is acyclic, but pipelines can be run on a recurring schedule and can run Python scripts that can write state information to the file system, making it possible to create complex profiles. If you design your pipeline so that certain steps may run in parallel or asynchronously, Azure Machine Learning transparently handles the dependency analysis and coordination of fan-out and fan-in. You generally don't have to concern yourself with the details of the execution graph, but it's available via the Pipeline.graph attribute.

A simple Python Pipeline

This snippet shows the objects and calls needed to create and run a basic `Pipeline`:

```
ws = Workspace.from_config()
blob_store = Datastore(ws, "workspaceblobstore")
compute_target = ws.compute_targets["STANDARD_NC6"]
experiment = Experiment(ws, 'MyExperiment')

input_data = Dataset.File.from_files(
    DataPath(datastore, '20newsgroups/20news.pkl'))

output_data = PipelineData("output_data", datastore=blob_store)

input_named = input_data.as_named_input('input')

steps = [ PythonScriptStep(
    script_name="train.py",
    arguments=["--input", input_named.as_download(), "--output", output_data],
    inputs=[input_data],
    outputs=[output_data],
    compute_target=compute_target,
    source_directory="myfolder"
) ]

pipeline = Pipeline(workspace=ws, steps=steps)

pipeline_run = experiment.submit(pipeline)
pipeline_run.wait_for_completion()
```

The snippet starts with common Azure Machine Learning objects, a `Workspace`, a `Datastore`, a ComputeTarget, and an `Experiment`. Then, the code creates the objects to hold `input_data` and `output_data`. The array `steps` holds a single element, a `PythonScriptStep` that will use the data objects and run on the `compute_target`. Then, the code instantiates the `Pipeline` object itself, passing in the workspace and steps array. The call to `experiment.submit(pipeline)` begins the Azure ML pipeline run. The call to `wait_for_completion()` blocks until the pipeline is finished.

To learn more about connecting your pipeline to your data, see the articles Data access in Azure Machine Learning and Moving data into and between ML pipeline steps (Python).

Best practices when using pipelines

As you can see, creating an Azure ML pipeline is a little more complex than starting a script. Pipelines require a few Python objects be configured and created.

Some situations that suggest using a pipeline:

- In a team environment: divide ML tasks into multiple independent steps so that developers can work and evolve their programs independently.
- When in or near deployment: nail down the configuration and use scheduled and event-driven operations to stay on top of changing data.
- In the early stages of an ML project or working alone: use pipelines to automate the build. If you've started worrying about recreating the configuration and computational state before implementing a new idea, that's a signal that you might consider using a pipeline to automate the workflow.

It's easy to become enthusiastic about reusing cached results, fine-grained control over compute costs, and process isolation, but pipelines do have costs. Some anti-patterns include:

- Using pipelines as the sole means to separate concerns. Python's built-in functions, objects, and modules go a long way to avoid confusing programmatic state! A pipeline step is much more expensive than a function call.

- Heavy coupling between pipeline steps. If refactoring a dependent step frequently requires modifying the outputs of a previous step, it's likely that separate steps are currently more of a cost than a benefit. Another clue that steps are overly coupled is arguments to a step that are not data but flags to control processing.
- Prematurely optimizing compute resources. For instance, there are often several stages to data preparation and one can often see "Oh, here's a place where I could use an `MpiStep` for parallel-programming but here's a place where I could use a `PythonScriptStep` with a less-powerful compute target," and so forth. And maybe, in the long run, creating fine-grained steps like that might prove worthwhile, especially if there's a possibility to use cached results rather than always recalculating. But pipelines are not intended to be a substitute for Python's native `multiprocessing` module.

Until a project gets large or nears deployment, your pipelines should be coarser rather than fine-grained. If you think of your ML project as involving *stages* and a pipeline as providing a complete workflow to move you through a particular stage, you're on the right path.

Key advantages

The key advantages of using pipelines for your machine learning workflows are:

KEY ADVANTAGE	DESCRIPTION
Unattended runs	Schedule steps to run in parallel or in sequence in a reliable and unattended manner. Data preparation and modeling can last days or weeks, and pipelines allow you to focus on other tasks while the process is running.
Heterogenous compute	Use multiple pipelines that are reliably coordinated across heterogeneous and scalable compute resources and storage locations. Make efficient use of available compute resources by running individual pipeline steps on different compute targets, such as HDInsight, GPU Data Science VMs, and Databricks.
Reusability	Create pipeline templates for specific scenarios, such as retraining and batch-scoring. Trigger published pipelines from external systems via simple REST calls.
Tracking and versioning	Instead of manually tracking data and result paths as you iterate, use the pipelines SDK to explicitly name and version your data sources, inputs, and outputs. You can also manage scripts and data separately for increased productivity.
Modularity	Separating areas of concerns and isolating changes allows software to evolve at a faster rate with higher quality.
Collaboration	Pipelines allow data scientists to collaborate across all areas of the machine learning design process, while being able to concurrently work on pipeline steps.

Choosing the proper PipelineStep subclass

The `PythonScriptStep` is the most flexible subclass of the abstract `PipelineStep`. Other subclasses, such as `EstimatorStep` subclasses and `DataTransferStep` can accomplish specific tasks with less code. For instance, an `EstimatorStep` can be created just by passing in a name for the step, an `Estimator`, and a compute target. Or, you can override inputs and outputs, pipeline parameters, and arguments. For more information, see Train models with Azure Machine Learning using estimator.

The `DataTransferStep` makes it easy to move data between data sources and sinks. The code to do this transfer manually is straightforward but repetitive. Instead, you can just create a `DataTransferStep` with a name, references to a data source and a data sink, and a compute target. The notebook Azure Machine Learning Pipeline with DataTransferStep demonstrates this flexibility.

Modules

While pipeline steps allow the reuse of the results of a previous run, in many cases the construction of the step assumes that the scripts and dependent files required must be locally available. If a data scientist wants to build on top of existing code, the scripts and dependencies often must be cloned from a separate repository.

Modules are similar in usage to pipeline steps, but provide versioning facilitated through the workspace, which enables collaboration and reusability at scale. Modules are designed to be reused in multiple pipelines and can evolve to adapt a specific computation to different use-cases. Users can do the following tasks through the workspace, without using external repositories:

- Create new modules, and publish new versions of existing modules
- Deprecate existing versions
- Mark versions disabled to prevent consumers from using that version
- Designate default versions
- Retrieve modules by version from the workspace, to ensure teams are using the same code

See the notebook for code examples on how to create, connect, and use modules in Azure Machine Learning pipelines.

Next steps

Azure ML pipelines are a powerful facility that begins delivering value in the early development stages. The value increases as the team and project grows. This article has explained how pipelines are specified with the Azure Machine Learning Python SDK and orchestrated on Azure. You've seen some basic source code and been introduced to a few of the `PipelineStep` classes that are available. You should have a sense of when to use Azure ML pipelines and how Azure runs them.

- Learn how to create your first pipeline.
- Learn how to run batch predictions on large data.
- See the SDK reference docs for pipeline core and pipeline steps.
- Try out example Jupyter notebooks showcasing Azure Machine Learning pipelines. Learn how to run notebooks to explore this service.

ONNX and Azure Machine Learning: Create and accelerate ML models

12/27/2019 • 3 minutes to read • Edit Online

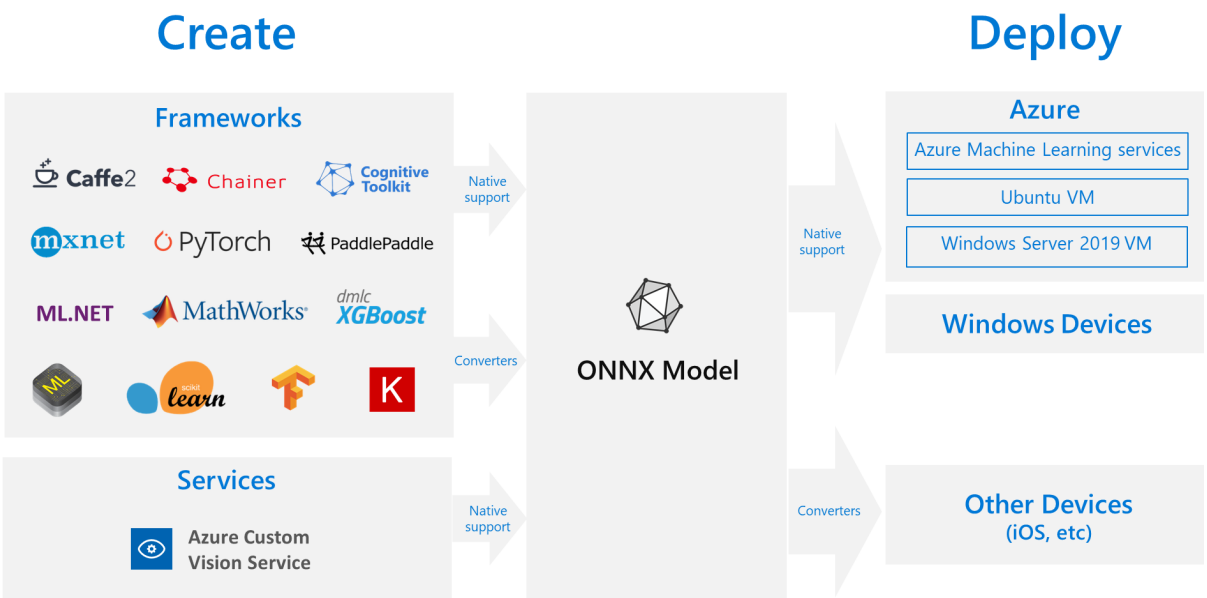
Learn how using the Open Neural Network Exchange (ONNX) can help optimize the inference of your machine learning model. Inference, or model scoring, is the phase where the deployed model is used for prediction, most commonly on production data.

Optimizing machine learning models for inference (or model scoring) is difficult since you need to tune the model and the inference library to make the most of the hardware capabilities. The problem becomes extremely hard if you want to get optimal performance on different kinds of platforms (cloud/edge, CPU/GPU, etc.), since each one has different capabilities and characteristics. The complexity increases if you have models from a variety of frameworks that need to run on a variety of platforms. It's very time consuming to optimize all the different combinations of frameworks and hardware. A solution to train once in your preferred framework and run anywhere on the cloud or edge is needed. This is where ONNX comes in.

Microsoft and a community of partners created ONNX as an open standard for representing machine learning models. Models from many frameworks including TensorFlow, PyTorch, SciKit-Learn, Keras, Chainer, MXNet, and MATLAB can be exported or converted to the standard ONNX format. Once the models are in the ONNX format, they can be run on a variety of platforms and devices.

ONNX Runtime is a high-performance inference engine for deploying ONNX models to production. It's optimized for both cloud and edge and works on Linux, Windows, and Mac. Written in C++, it also has C, Python, and C# APIs. ONNX Runtime provides support for all of the ONNX-ML specification and also integrates with accelerators on different hardware such as TensorRT on NVidia GPUs.

The ONNX Runtime is used in high scale Microsoft services such as Bing, Office, and Cognitive Services. Performance gains are dependent on a number of factors but these Microsoft services have seen an **average 2x performance gain on CPU**. ONNX Runtime is also used as part of Windows ML on hundreds of millions of devices. You can use the runtime with Azure Machine Learning. By using ONNX Runtime, you can benefit from the extensive production-grade optimizations, testing, and ongoing improvements.



Get ONNX models

You can obtain ONNX models in several ways:

- Train a new ONNX model in Azure Machine Learning (see examples at the bottom of this article)
- Convert existing model from another format to ONNX (see the tutorials)
- Get a pre-trained ONNX model from the ONNX Model Zoo (see examples at the bottom of this article)
- Generate a customized ONNX model from Azure Custom Vision service

Many models including image classification, object detection, and text processing can be represented as ONNX models. However some models may not be able to be converted successfully. If you run into this situation, please file an issue in the GitHub of the respective converter that you used. You can continue using your existing format model until the issue is addressed.

Deploy ONNX models in Azure

With Azure Machine Learning, you can deploy, manage, and monitor your ONNX models. Using the standard deployment workflow and ONNX Runtime, you can create a REST endpoint hosted in the cloud. See example Jupyter notebooks at the end of this article to try it out for yourself.

Install and use ONNX Runtime with Python

Python packages for ONNX Runtime are available on PyPi.org (CPU, GPU). Please read system requirements before installation.

To install ONNX Runtime for Python, use one of the following commands:

```
pip install onnxruntime       # CPU build
pip install onnxruntime-gpu   # GPU build
```

To call ONNX Runtime in your Python script, use:

```
import onnxruntime
session = onnxruntime.InferenceSession("path to model")
```

The documentation accompanying the model usually tells you the inputs and outputs for using the model. You can also use a visualization tool such as Netron to view the model. ONNX Runtime also lets you query the model metadata, inputs, and outputs:

```
session.get_modelmeta()
first_input_name = session.get_inputs()[0].name
first_output_name = session.get_outputs()[0].name
```

To inference your model, use `run` and pass in the list of outputs you want returned (leave empty if you want all of them) and a map of the input values. The result is a list of the outputs.

```
results = session.run(["output1", "output2"], {
                      "input1": indata1, "input2": indata2})
results = session.run([], {"input1": indata1, "input2": indata2})
```

For the complete Python API reference, see the ONNX Runtime reference docs.

Examples

See how-to-use-azureml/deployment/onnx for example notebooks that create and deploy ONNX models.

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

More info

Learn more about ONNX or contribute to the project:

- ONNX project website
- ONNX code on GitHub

Learn more about ONNX Runtime or contribute to the project:

- ONNX Runtime GitHub Repo

Enterprise security for Azure Machine Learning

4/24/2020 • 17 minutes to read • Edit Online

In this article, you'll learn about security features available for Azure Machine Learning.

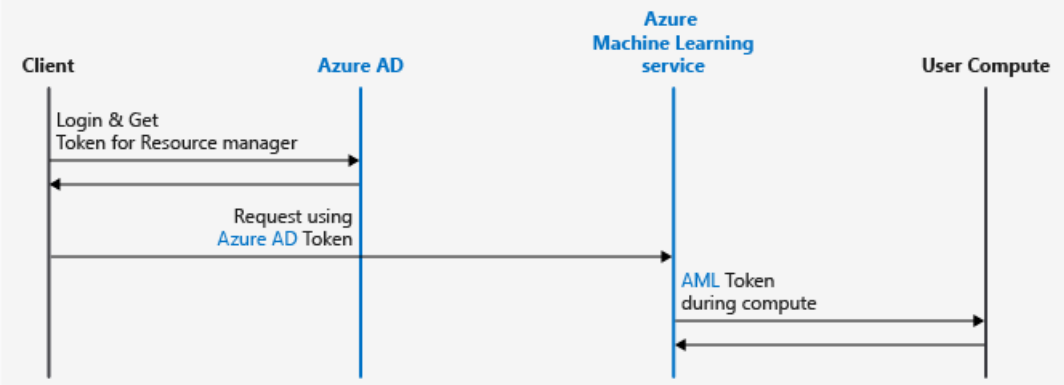
When you use a cloud service, a best practice is to restrict access to only the users who need it. Start by understanding the authentication and authorization model used by the service. You might also want to restrict network access or securely join resources in your on-premises network with the cloud. Data encryption is also vital, both at rest and while data moves between services. Finally, you need to be able to monitor the service and produce an audit log of all activity.

> **NOTE**
> The information in this article works with the Azure Machine Learning Python SDK version 1.0.83.1 or higher.

Authentication

Multi-factor authentication is supported if Azure Active Directory (Azure AD) is configured to use it. Here's the authentication process:

1. The client signs in to Azure AD and gets an Azure Resource Manager token. Users and service principals are fully supported.
2. The client presents the token to Azure Resource Manager and to all Azure Machine Learning.
3. The Machine Learning service provides a Machine Learning service token to the user compute target (for example, Machine Learning Compute). This token is used by the user compute target to call back into the Machine Learning service after the run is complete. Scope is limited to the workspace.



For more information, see Set up authentication for Azure Machine Learning resources and workflows. This article provides information and examples on authentication, including using service principals and automated workflows.

Authentication for web service deployment

Azure Machine Learning supports two forms of authentication for web services: key and token. Each web service can enable only one form of authentication at a time.

AUTHENTICATION METHOD	DESCRIPTION	AZURE CONTAINER INSTANCES	AKS

AUTHENTICATION METHOD	DESCRIPTION	AZURE CONTAINER INSTANCES	AKS
Key	Keys are static and do not need to be refreshed. Keys can be regenerated manually.	Disabled by default	Enabled by default
Token	Tokens expire after a specified time period and need to be refreshed.	Not available	Disabled by default

For code examples, see the web-service authentication section.

Authorization

You can create multiple workspaces, and each workspace can be shared by multiple people. When you share a workspace, you can control access to it by assigning these roles to users:

- Owner
- Contributor
- Reader

The following table lists some of the major Azure Machine Learning operations and the roles that can perform them:

AZURE MACHINE LEARNING OPERATION	OWNER	CONTRIBUTOR	READER
Create workspace	✓	✓	
Share workspace	✓		
Upgrade workspace to Enterprise edition	✓		
Create compute target	✓	✓	
Attach compute target	✓	✓	
Attach data stores	✓	✓	
Run experiment	✓	✓	
View runs/metrics	✓	✓	✓
Register model	✓	✓	
Create image	✓	✓	
Deploy web service	✓	✓	
View models/images	✓	✓	✓
Call web service	✓	✓	✓

If the built-in roles don't meet your needs, you can create custom roles. Custom roles are supported only for operations on the workspace and Machine Learning Compute. Custom roles can have read, write, or delete permissions on the workspace and on the compute resource in that workspace. You can make the role available at a specific workspace level, a specific resource-group level, or a specific subscription level. For more information, see Manage users and roles in an Azure Machine Learning workspace.

> **WARNING**
> Azure Machine Learning is not currently supported with Azure Active Directory business-to-business collaboration.

Securing compute targets and data

Owners and contributors can use all compute targets and data stores that are attached to the workspace.

Each workspace also has an associated system-assigned managed identity that has the same name as the workspace. The managed identity has the following permissions on attached resources used in the workspace.

For more information on managed identities, see Managed identities for Azure resources.

RESOURCE	PERMISSIONS
Workspace	Contributor
Storage account	Storage Blob Data Contributor
Key vault	Access to all keys, secrets, certificates
Azure Container Registry	Contributor
Resource group that contains the workspace	Contributor
Resource group that contains the key vault (if different from the one that contains the workspace)	Contributor

We don't recommend that admins revoke the access of the managed identity to the resources mentioned in the preceding table. You can restore access by using the resync keys operation.

Azure Machine Learning creates an additional application (the name starts with `aml-` or `Microsoft-AzureML-Support-App-`) with contributor-level access in your subscription for every workspace region. For example, if you have one workspace in East US and one in North Europe in the same subscription, you'll see two of these applications. These applications enable Azure Machine Learning to help you manage compute resources.

Network security

Azure Machine Learning relies on other Azure services for compute resources. Compute resources (compute targets) are used to train and deploy models. You can create these compute targets in a virtual network. For example, you can use Azure Data Science Virtual Machine to train a model and then deploy the model to AKS.

For more information, see How to run experiments and inference in a virtual network.

You can also enable Azure Private Link for your workspace. Private Link allows you to restrict communications to your workspace from an Azure Virtual Network. For more information, see How to configure Private Link.

TIP

You can combine virtual network and Private Link together to protect communication between your workspace and other Azure resources. However, some combinations require an Enterprise edition workspace. Use the following table to understand what scenarios require Enterprise edition:

SCENARIO	ENTERPRISE EDITION	BASIC EDITION
No virtual network or Private Link	✔	✔
Workspace without Private Link. Other resources (except Azure Container Registry) in a virtual network	✔	✔
Workspace without Private Link. Other resources with Private Link	✔	
Workspace with Private Link. Other resources (except Azure Container Registry) in a virtual network	✔	✔
Workspace and any other resource with Private Link	✔	
Workspace with Private Link. Other resources without Private Link or virtual network	✔	✔
Azure Container Registry in a virtual network	✔	
Customer Managed Keys for workspace	✔	

WARNING

Azure Machine Learning compute instances preview is not supported in a workspace where Private Link is enabled.

Azure Machine Learning does not support using an Azure Kubernetes Service that has private link enabled. Instead, you can use Azure Kubernetes Service in a virtual network. For more information, see Secure Azure ML experimentation and inference jobs within an Azure Virtual Network.

Data encryption

Encryption at rest

IMPORTANT

If your workspace contains sensitive data we recommend setting the hbi_workspace flag while creating your workspace.

The `hbi_workspace` flag controls the amount of data Microsoft collects for diagnostic purposes and enables additional encryption in Microsoft managed environments. In addition it enables the following:

- Starts encrypting the local scratch disk in your Amlcompute cluster provided you have not created any previous clusters in that subscription. Else, you need to raise a support ticket to enable encryption of the scratch disk of your compute clusters
- Cleans up your local scratch disk between runs

- Securely passes credentials for your storage account, container registry and SSH account from the execution layer to your compute clusters using your key vault
- Enables IP filtering to ensure the underlying batch pools cannot be called by any external services other than AzureMachineLearningService

For more information on how encryption at rest works in Azure, see Azure data encryption at rest.

Azure Blob storage

Azure Machine Learning stores snapshots, output, and logs in the Azure Blob storage account that's tied to the Azure Machine Learning workspace and your subscription. All the data stored in Azure Blob storage is encrypted at rest with Microsoft-managed keys.

For information on how to use your own keys for data stored in Azure Blob storage, see Azure Storage encryption with customer-managed keys in Azure Key Vault.

Training data is typically also stored in Azure Blob storage so that it's accessible to training compute targets. This storage isn't managed by Azure Machine Learning but mounted to compute targets as a remote file system.

If you need to **rotate or revoke** your key, you can do so at any time. When rotating a key, the storage account will start using the new key (latest version) to encrypt data at rest. When revoking (disabling) a key, the storage account takes care of failing requests. It usually takes an hour for the rotation or revocation to be effective.

For information on regenerating the access keys, see Regenerate storage access keys.

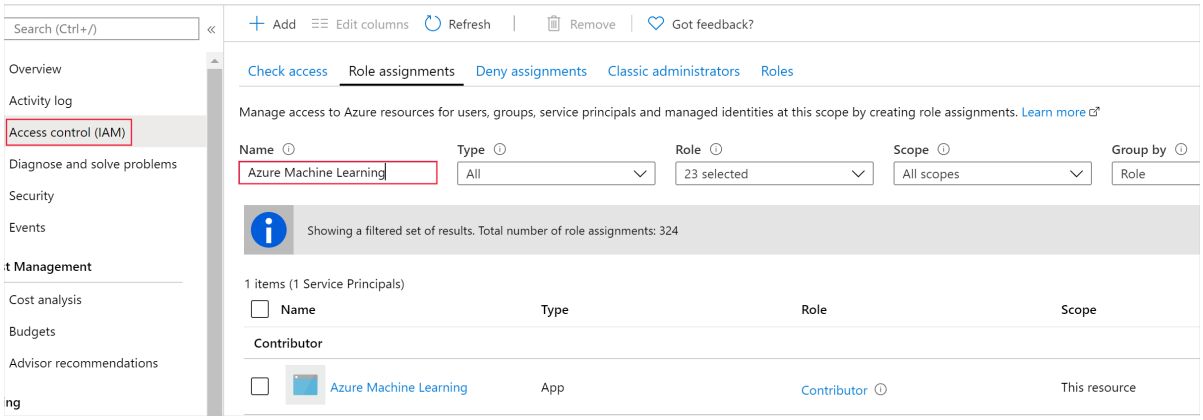
Azure Cosmos DB

Azure Machine Learning stores metrics and metadata in an Azure Cosmos DB instance. This instance is associated with a Microsoft subscription managed by Azure Machine Learning. All the data stored in Azure Cosmos DB is encrypted at rest with Microsoft-managed keys.

To use your own (customer-managed) keys to encrypt the Azure Cosmos DB instance, you can create a dedicated Cosmos DB instance for use with your workspace. We recommend this approach if you want to store your data, such as run history information, outside of the multi-tenant Cosmos DB instance hosted in our Microsoft subscription.

To enable provisioning a Cosmos DB instance in your subscription with customer-managed keys, perform the following actions:

- Enable customer-managed key capabilities for Cosmos DB. At this time, you must request access to use this capability. To do so, please contact cosmosdbpm@microsoft.com.
- Register the Azure Machine Learning and Azure Cosmos DB resource providers in your subscription, if not done already.
- Authorize the Machine Learning App (in Identity and Access Management) with contributor permissions on your subscription.



- Use the following parameters when creating the Azure Machine Learning workspace. Both parameters are

mandatory and supported in SDK, CLI, REST APIs, and Resource Manager templates.

- `resource_cmk_uri`: This parameter is the full resource URI of the customer managed key in your key vault, including the version information for the key.
- `cmk_keyvault`: This parameter is the resource ID of the key vault in your subscription. This key vault needs to be in the same region and subscription that you will use for the Azure Machine Learning workspace.

> **NOTE**
> This key vault instance can be different than the key vault that is created by Azure Machine Learning when you provision the workspace. If you want to use the same key vault instance for the workspace, pass the same key vault while provisioning the workspace by using the key_vault parameter.

This Cosmos DB instance is created in a Microsoft-managed resource group in your subscription. The managed resource group is named in the format `<AML Workspace Resource Group Name><GUID>`.

> **IMPORTANT**
> - If you need to delete this Cosmos DB instance, you must delete the Azure Machine Learning workspace that uses it.
> - The default **Request Units** for this Cosmos DB account is set at **8000**. Changing this value is unsupported.

If you need to **rotate or revoke** your key, you can do so at any time. When rotating a key, Cosmos DB will start using the new key (latest version) to encrypt data at rest. When revoking (disabling) a key, Cosmos DB takes care of failing requests. It usually takes an hour for the rotation or revocation to be effective.

For more information on customer-managed keys with Cosmos DB, see Configure customer-managed keys for your Azure Cosmos DB account.

Azure Container Registry

All container images in your registry (Azure Container Registry) are encrypted at rest. Azure automatically encrypts an image before storing it and decrypts it when Azure Machine Learning pulls the image.

To use your own (customer-managed) keys to encrypt your Azure Container Registry, you need to create your own ACR and attach it while provisioning the workspace or encrypt the default instance that gets created at the time of workspace provisioning.

For an example of creating a workspace using an existing Azure Container Registry, see the following articles:

- Create a workspace for Azure Machine Learning with Azure CLI.
- Use an Azure Resource Manager template to create a workspace for Azure Machine Learning

Azure Container Instance

You may encrypt a deployed Azure Container Instance (ACI) resource using customer-managed keys. The customer-managed key used for ACI can be stored in the Azure Key Vault for your workspace. For information on generating a key, see Encrypt data with a customer-managed key.

To use the key when deploying a model to Azure Container Instance, create a new deployment configuration using `AciWebservice.deploy_configuration()`. Provide the key information using the following parameters:

- `cmk_vault_base_url`: The URL of the key vault that contains the key.
- `cmk_key_name`: The name of the key.
- `cmk_key_version`: The version of the key.

For more information on creating and using a deployment configuration, see the following articles:

- AciWebservice.deploy_configuration() reference
- Where and how to deploy
- Deploy a model to Azure Container Instances

For more information on using a customer-managed key with ACI, see Encrypt data with a customer-managed key.

Azure Kubernetes Service

You may encrypt a deployed Azure Kubernetes Service resource using customer-managed keys at any time. For more information, see Bring your own keys with Azure Kubernetes Service.

This process allows you to encrypt both the Data and the OS Disk of the deployed virtual machines in the Kubernetes cluster.

> **IMPORTANT**
> This process only works with AKS K8s version 1.17 or higher. Azure Machine Learning added support for AKS 1.17 on Jan 13, 2020.

Machine Learning Compute

The OS disk for each compute node stored in Azure Storage is encrypted with Microsoft-managed keys in Azure Machine Learning storage accounts. This compute target is ephemeral, and clusters are typically scaled down when no runs are queued. The underlying virtual machine is de-provisioned, and the OS disk is deleted. Azure Disk Encryption isn't supported for the OS disk.

Each virtual machine also has a local temporary disk for OS operations. If you want, you can use the disk to stage training data. The disk is encrypted by default for workspaces with the `hbi_workspace` parameter set to `TRUE`. This environment is short-lived only for the duration of your run, and encryption support is limited to system-managed keys only.

Azure Databricks

Azure Databricks can be used in Azure Machine Learning pipelines. By default, the Databricks File System (DBFS) used by Azure Databricks is encrypted using a Microsoft-managed key. To configure Azure Databricks to use customer-managed keys, see Configure customer-managed keys on default (root) DBFS.

Encryption in transit

Azure Machine Learning uses TLS to secure internal communication between various Azure Machine Learning microservices. All Azure Storage access also occurs over a secure channel.

To secure external calls to the scoring endpoint Azure Machine Learning uses TLS. For more information, see Use TLS to secure a web service through Azure Machine Learning.

Using Azure Key Vault

Azure Machine Learning uses the Azure Key Vault instance associated with the workspace to store credentials of various kinds:

- The associated storage account connection string
- Passwords to Azure Container Repository instances
- Connection strings to data stores

SSH passwords and keys to compute targets like Azure HDInsight and VMs are stored in a separate key vault that's associated with the Microsoft subscription. Azure Machine Learning doesn't store any passwords or keys provided by users. Instead, it generates, authorizes, and stores its own SSH keys to connect to VMs and HDInsight to run the experiments.

Each workspace has an associated system-assigned managed identity that has the same name as the workspace. This managed identity has access to all keys, secrets, and certificates in the key vault.

Data collection and handling

Microsoft collected data

Microsoft may collect non-user identifying information like resource names (for example the dataset name, or the machine learning experiment name), or job environment variables for diagnostic purposes. All such data is stored using Microsoft-managed keys in storage hosted in Microsoft owned subscriptions and follows Microsoft's standard Privacy policy and data handling standards.

Microsoft also recommends not storing sensitive information (such as account key secrets) in environment variables. Environment variables are logged, encrypted, and stored by us. Similarly when naming runid, avoid including sensitive information such as user names or secret project names. This information may appear in telemetry logs accessible to Microsoft Support engineers.

You may opt out from diagnostic data being collected by setting the `hbi_workspace` parameter to `TRUE` while provisioning the workspace. This functionality is supported when using the AzureML Python SDK, CLI, REST APIs, or Azure Resource Manager templates.

Microsoft-generated data

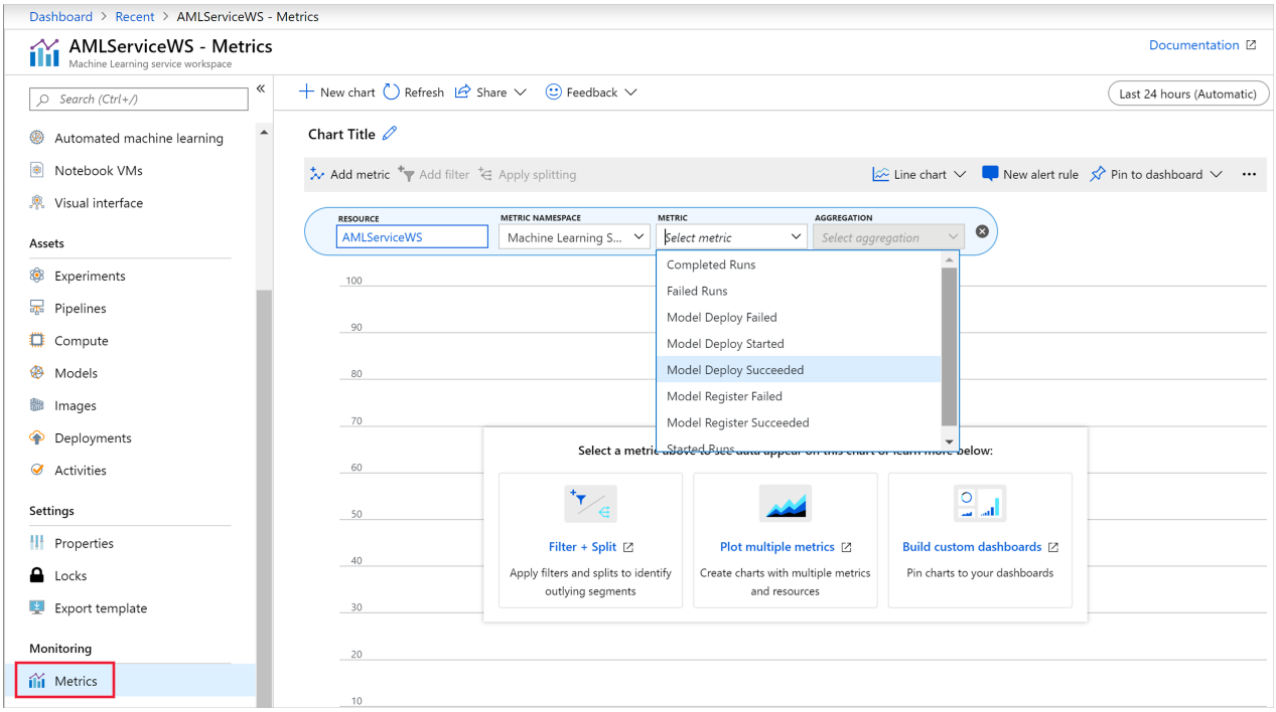
When using services such as Automated Machine Learning, Microsoft may generate a transient, pre-processed data for training multiple models. This data is stored in a datastore in your workspace, which allows you to enforce access controls and encryption appropriately.

You may also want to encrypt diagnostic information logged from your deployed endpoint into your Azure Application Insights instance.

Monitoring

Metrics

You can use Azure Monitor metrics to view and monitor metrics for your Azure Machine Learning workspace. In the Azure portal, select your workspace and then select **Metrics**:



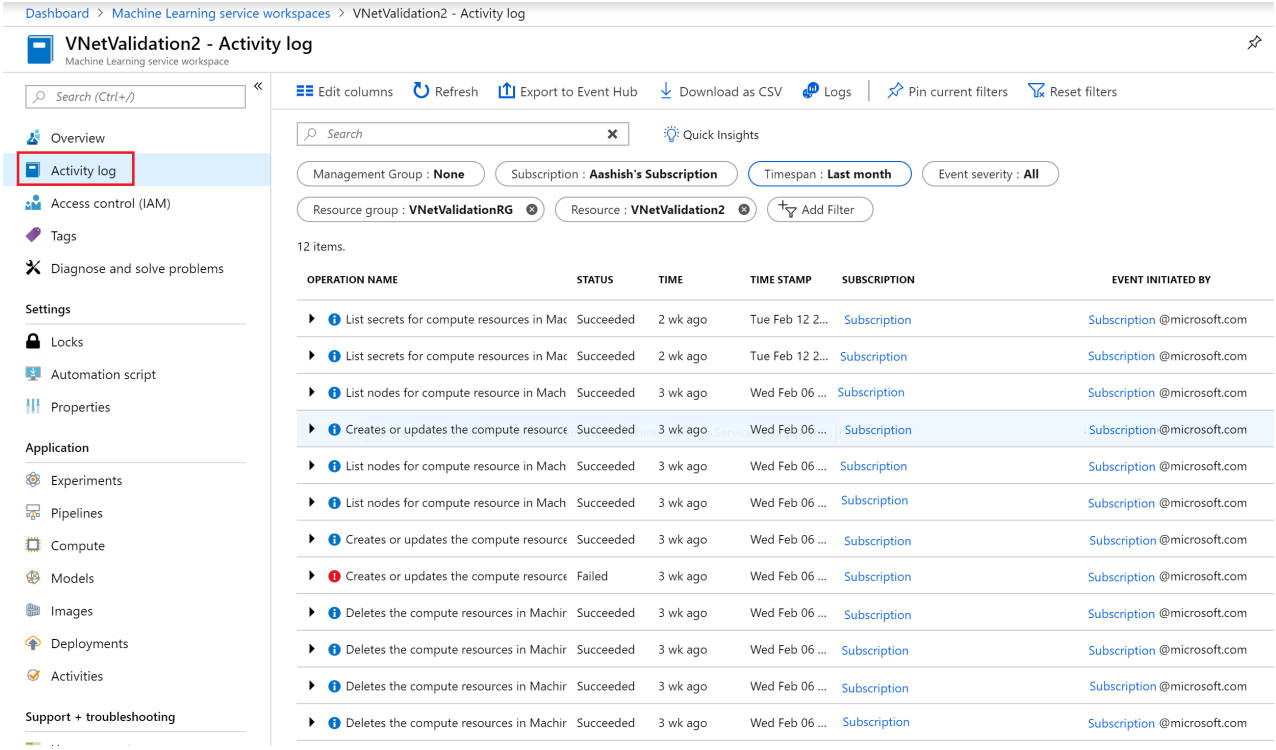
The metrics include information on runs, deployments, and registrations.

For more information, see Metrics in Azure Monitor.

Activity log

You can view the activity log of a workspace to see various operations that are performed on the workspace. The log includes basic information like the operation name, event initiator, and timestamp.

This screenshot shows the activity log of a workspace:



Scoring request details are stored in Application Insights. Application Insights is created in your subscription when you create a workspace. Logged information includes fields such as:

- HTTPMethod
- UserAgent
- ComputeType
- RequestUrl
- StatusCode
- RequestId
- Duration

> **IMPORTANT**
>
> Some actions in the Azure Machine Learning workspace don't log information to the activity log. For example, the start of a training run and the registration of a model aren't logged.
>
> Some of these actions appear in the **Activities** area of your workspace, but these notifications don't indicate who initiated the activity.

Data flow diagrams

Create workspace

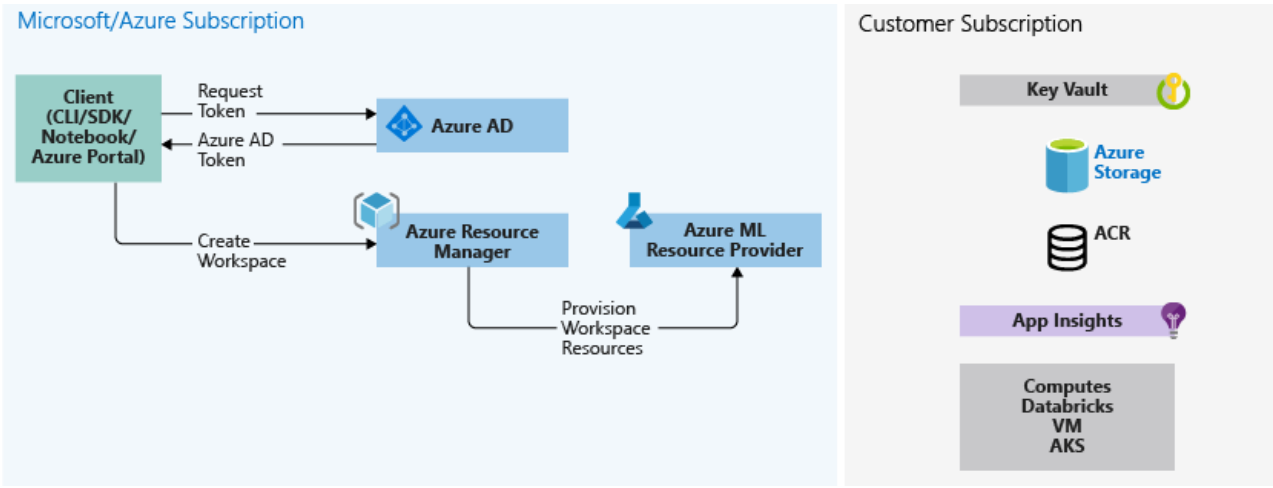
The following diagram shows the create workspace workflow.

- You sign in to Azure AD from one of the supported Azure Machine Learning clients (Azure CLI, Python SDK, Azure portal) and request the appropriate Azure Resource Manager token.
- You call Azure Resource Manager to create the workspace.
- Azure Resource Manager contacts the Azure Machine Learning resource provider to provision the workspace.

Additional resources are created in the user's subscription during workspace creation:

- Key Vault (to store secrets)
- An Azure storage account (including blob and file share)
- Azure Container Registry (to store Docker images for inference/scoring and experimentation)
- Application Insights (to store telemetry)

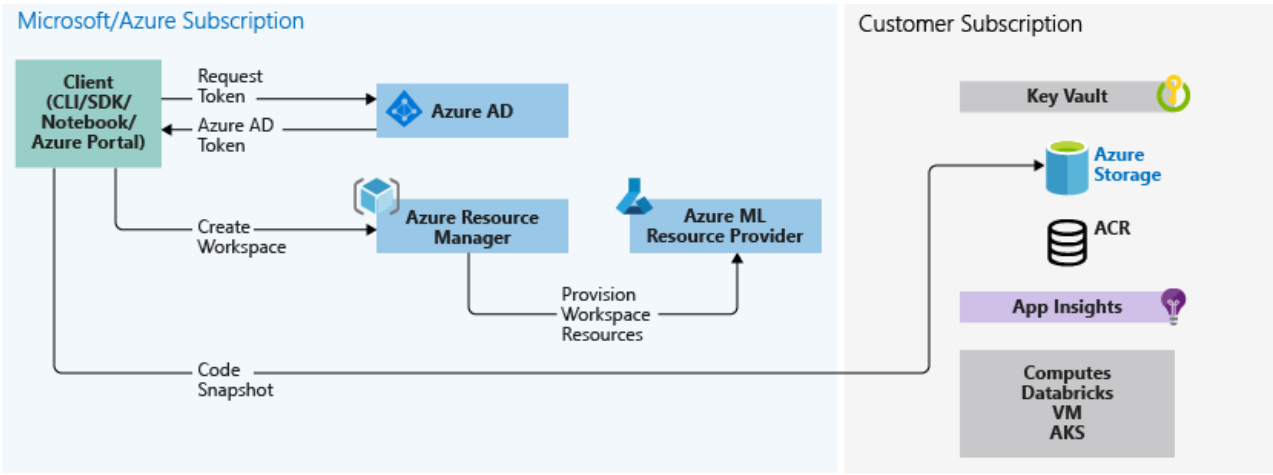
The user can also provision other compute targets that are attached to a workspace (like Azure Kubernetes Service or VMs) as needed.



Save source code (training scripts)

The following diagram shows the code snapshot workflow.

Associated with an Azure Machine Learning workspace are directories (experiments) that contain the source code (training scripts). These scripts are stored on your local machine and in the cloud (in the Azure Blob storage for your subscription). The code snapshots are used for execution or inspection for historical auditing.



Training

The following diagram shows the training workflow.

- Azure Machine Learning is called with the snapshot ID for the code snapshot saved in the previous section.
- Azure Machine Learning creates a run ID (optional) and a Machine Learning service token, which is later used by compute targets like Machine Learning Compute/VMs to communicate with the Machine Learning service.
- You can choose either a managed compute target (like Machine Learning Compute) or an unmanaged compute target (like VMs) to run training jobs. Here are the data flows for both scenarios:
    - VMs/HDInsight, accessed by SSH credentials in a key vault in the Microsoft subscription. Azure Machine

Learning runs management code on the compute target that:

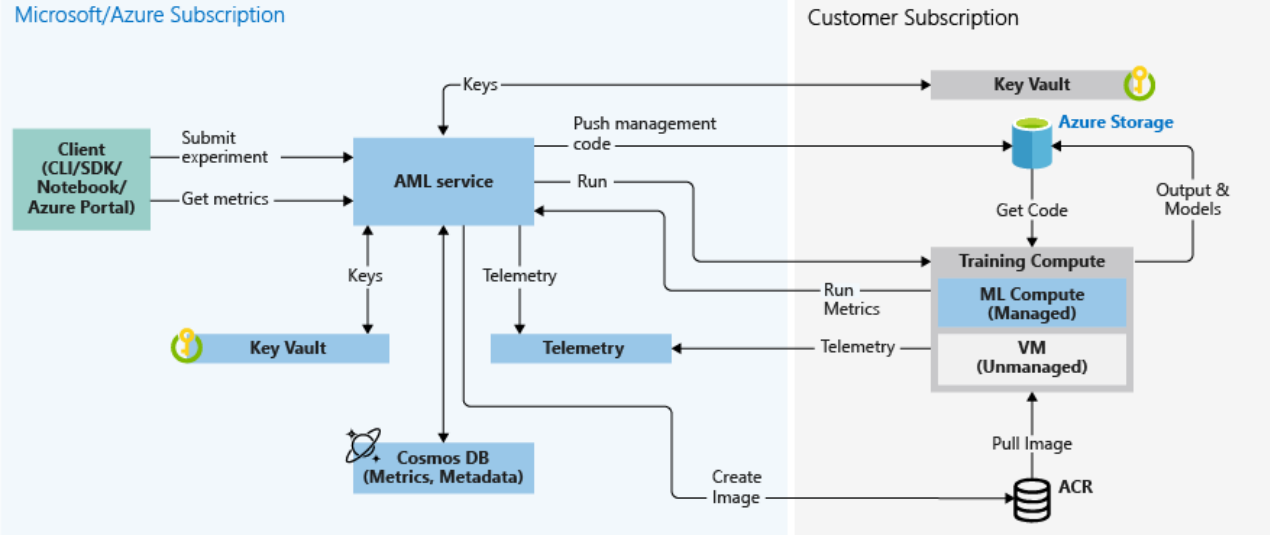
1. Prepares the environment. (Docker is an option for VMs and local computers. See the following steps for Machine Learning Compute to understand how running experiments on Docker containers works.)
2. Downloads the code.
3. Sets up environment variables and configurations.
4. Runs user scripts (the code snapshot mentioned in the previous section).

- Machine Learning Compute, accessed through a workspace-managed identity. Because Machine Learning Compute is a managed compute target (that is, it's managed by Microsoft) it runs under your Microsoft subscription.

1. Remote Docker construction is kicked off, if needed.
2. Management code is written to the user's Azure Files share.
3. The container is started with an initial command. That is, management code as described in the previous step.

Querying runs and metrics

In the flow diagram below, this step occurs when the training compute target writes the run metrics back to Azure Machine Learning from storage in the Cosmos DB database. Clients can call Azure Machine Learning. Machine Learning will in turn pull metrics from the Cosmos DB database and return them back to the client.

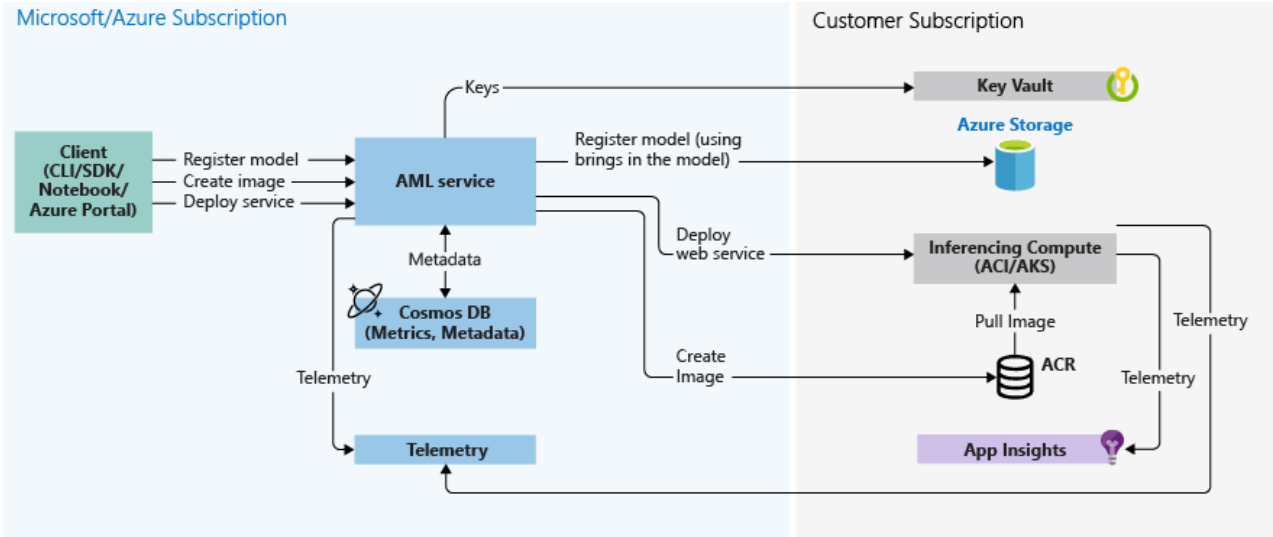


Creating web services

The following diagram shows the inference workflow. Inference, or model scoring, is the phase in which the deployed model is used for prediction, most commonly on production data.

Here are the details:

- The user registers a model by using a client like the Azure Machine Learning SDK.
- The user creates an image by using a model, a score file, and other model dependencies.
- The Docker image is created and stored in Azure Container Registry.
- The web service is deployed to the compute target (Container Instances/AKS) using the image created in the previous step.
- Scoring request details are stored in Application Insights, which is in the user's subscription.
- Telemetry is also pushed to the Microsoft/Azure subscription.



Next steps

- Secure Azure Machine Learning web services with TLS
- Consume a Machine Learning model deployed as a web service
- How to run batch predictions
- Monitor your Azure Machine Learning models with Application Insights
- Collect data for models in production
- Azure Machine Learning SDK
- Use Azure Machine Learning with Azure Virtual Network
- Best practices for building recommendation systems
- Build a real-time recommendation API on Azure

Manage access to an Azure Machine Learning workspace

3/8/2020 • 5 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to manage access to an Azure Machine Learning workspace. Role-based access control (RBAC) is used to manage access to Azure resources. Users in your Azure Active Directory are assigned specific roles, which grant access to resources. Azure provides both built-in roles and the ability to create custom roles.

Default roles

An Azure Machine Learning workspace is an Azure resource. Like other Azure resources, when a new Azure Machine Learning workspace is created, it comes with three default roles. You can add users to the workspace and assign them to one of these built-in roles.

ROLE	ACCESS LEVEL
Reader	Read-only actions in the workspace. Readers can list and view assets in a workspace, but can't create or update these assets.
Contributor	View, create, edit, or delete (where applicable) assets in a workspace. For example, contributors can create an experiment, create or attach a compute cluster, submit a run, and deploy a web service.
Owner	Full access to the workspace, including the ability to view, create, edit, or delete (where applicable) assets in a workspace. Additionally, you can change role assignments.

IMPORTANT

Role access can be scoped to multiple levels in Azure. For example, someone with owner access to a workspace may not have owner access to the resource group that contains the workspace. For more information, see How RBAC works.

For more information on specific built-in roles, see Built-in roles for Azure.

Manage workspace access

If you're an owner of a workspace, you can add and remove roles for the workspace. You can also assign roles to users. Use the following links to discover how to manage access:

- Azure portal UI
- PowerShell
- Azure CLI
- REST API
- Azure Resource Manager templates

If you have installed the Azure Machine Learning CLI, you can also use a CLI command to assign roles to users.

```
az ml workspace share -w <workspace_name> -g <resource_group_name> --role <role_name> --user
<user_corp_email_address>
```

The `user` field is the email address of an existing user in the instance of Azure Active Directory where the workspace parent subscription lives. Here is an example of how to use this command:

```
az ml workspace share -w my_workspace -g my_resource_group --role Contributor --user jdoe@contoson.com
```

Create custom role

If the built-in roles are insufficient, you can create custom roles. Custom roles might have read, write, delete, and compute resource permissions in that workspace. You can make the role available at a specific workspace level, a specific resource group level, or a specific subscription level.

> **NOTE**
> You must be an owner of the resource at that level to create custom roles within that resource.

To create a custom role, first construct a role definition JSON file that specifies the permission and scope for the role. The following example defines a custom role named "Data Scientist" scoped at a specific workspace level:

`data_scientist_role.json` :

```
{
    "Name": "Data Scientist",
    "IsCustom": true,
    "Description": "Can run experiment but can't create or delete compute.",
    "Actions": ["*"],
    "NotActions": [
        "Microsoft.MachineLearningServices/workspaces/*/delete",
        "Microsoft.MachineLearningServices/workspaces/computes/*/write",
        "Microsoft.MachineLearningServices/workspaces/computes/*/delete",
        "Microsoft.Authorization/*/write"
    ],
    "AssignableScopes": [

"/subscriptions/<subscription_id>/resourceGroups/<resource_group_name>/providers/Microsoft.MachineLearningServ
ices/workspaces/<workspace_name>"
    ]
}
```

You can change the `AssignableScopes` field to set the scope of this custom role at the subscription level, the resource group level, or a specific workspace level.

This custom role can do everything in the workspace except for the following actions:

- It can't create or update a compute resource.
- It can't delete a compute resource.
- It can't add, delete, or alter role assignments.
- It can't delete the workspace.

To deploy this custom role, use the following Azure CLI command:

```
az role definition create --role-definition data_scientist_role.json
```

After deployment, this role becomes available in the specified workspace. Now you can add and assign this role in the Azure portal. Or, you can assign this role to a user by using the `az ml workspace share` CLI command:

```
az ml workspace share -w my_workspace -g my_resource_group --role "Data Scientist" --user jdoe@contoson.com
```

For more information on custom roles, see Custom roles for Azure resources.

For more information on the operations (actions) usable with custom roles, see Resource provider operations.

Frequently asked questions

Q. What are the permissions needed to perform various actions in the Azure Machine Learning service?

The following table is a summary of Azure Machine Learning activities and the permissions required to perform them at the least scope. As an example if an activity can be performed with a workspace scope (Column 4), then all higher scope with that permission will also work automatically. All paths in this table are **relative paths** to `Microsoft.MachineLearningServices/`.

ACTIVITY	SUBSCRIPTION-LEVEL SCOPE	RESOURCE GROUP-LEVEL SCOPE	WORKSPACE-LEVEL SCOPE
Create new workspace	Not required	Owner or contributor	N/A (becomes Owner or inherits higher scope role after creation)
Create new compute cluster	Not required	Not required	Owner, contributor, or custom role allowing: `workspaces/computes/write`
Create new Notebook VM	Not required	Owner or contributor	Not possible
Create new compute instance	Not required	Not required	Owner, contributor, or custom role allowing: `workspaces/computes/write`
Data plane activity like submitting run, accessing data, deploying model or publishing pipeline	Not required	Not required	Owner, contributor, or custom role allowing: `workspaces/*/write` Note that you also need a datastore registered to the workspace to allow MSI to access data in your storage account.

Q. How do I list all the custom roles in my subscription?

In the Azure CLI, run the following command.

```
az role definition list --subscription <sub-id> --custom-role-only true
```

Q. How do I find the role definition for a role in my subscription?

In the Azure CLI, run the following command. Note that `<role-name>` should be in the same format returned by the command above.

```
az role definition list -n <role-name> --subscription <sub-id>
```

Q. How do I update a role definition?

In the Azure CLI, run the following command.

```
az role definition update --role-definition update_def.json --subscription <sub-id>
```

Note that you need to have permissions on the entire scope of your new role definition. For example if this new role has a scope across three subscriptions, you need to have permissions on all three subscriptions.

> **NOTE**
> Role updates can take 15 minutes to an hour to apply across all role assignments in that scope.

Q. Can I define a role that prevents updating the workspace Edition?

Yes, you can define a role that prevents updating the workspace Edition. Since the workspace update is a PATCH call on the workspace object, you do this by putting the following action in the `"NotActions"` array in your JSON definition:

```
"Microsoft.MachineLearningServices/workspaces/write"
```

Q. What permissions are needed to perform quota operations in a workspace?

You need subscription level permissions to perform any quota related operation in the workspace. This means setting either subscription level quota or workspace level quota for your managed compute resources can only happen if you have write permissions at the subscription scope.

Next steps

- Enterprise security overview
- Securely run experiments and inference/score inside a virtual network
- Tutorial: Train models
- Resource provider operations

Secure Azure ML experimentation and inference jobs within an Azure Virtual Network

4/24/2020 • 21 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you'll learn how to secure experimentation/training jobs and inference/scoring jobs in Azure Machine Learning within an Azure Virtual Network (vnet).

A **virtual network** acts as a security boundary, isolating your Azure resources from the public internet. You can also join an Azure virtual network to your on-premises network. By joining networks, you can securely train your models and access your deployed models for inference.

Azure Machine Learning relies on other Azure services for compute resources. Compute resources, or compute targets, are used to train and deploy models. The targets can be created within a virtual network. For example, you can use Microsoft Data Science Virtual Machine to train a model and then deploy the model to Azure Kubernetes Service (AKS). For more information about virtual networks, see Azure Virtual Network overview.

This article also provides detailed information about *advanced security settings*, information that isn't necessary for basic or experimental use cases. Certain sections of this article provide configuration information for a variety of scenarios. You don't need to complete the instructions in order or in their entirety.

> **TIP**
> Unless specifically called out, using resources such as storage accounts or compute targets inside a virtual network will work with both machine learning pipelines, and non-pipeline workflows such as script runs.

> **WARNING**
> Microsoft does not support using the Azure Machine Learning Studio features such as Automated ML, Datasets, Datalabeling, Designer, and Notebooks if the underlying storage has virtual network enabled.

Prerequisites

- An Azure Machine Learning workspace.
- General working knowledge of both the Azure Virtual Network service and IP networking.
- A pre-existing virtual network and subnet to use with your compute resources.

Use a storage account for your workspace

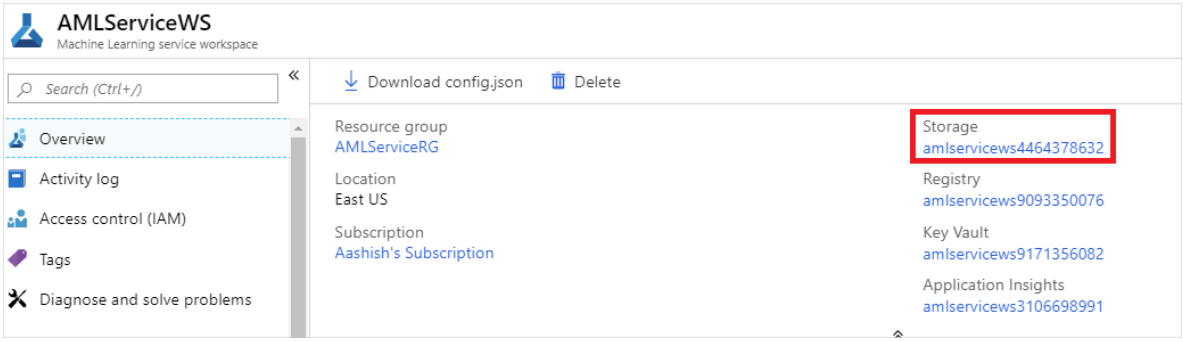
> **WARNING**
> If you have data scientists that use the Azure Machine Learning designer, they will receive an error when visualizing data from a storage account inside a virtual network. The following text is the error that they receive:
>
> **Error: Unable to profile this dataset. This might be because your data is stored behind a virtual network or your data does not support profile.**

To use an Azure storage account for the workspace in a virtual network, use the following steps:

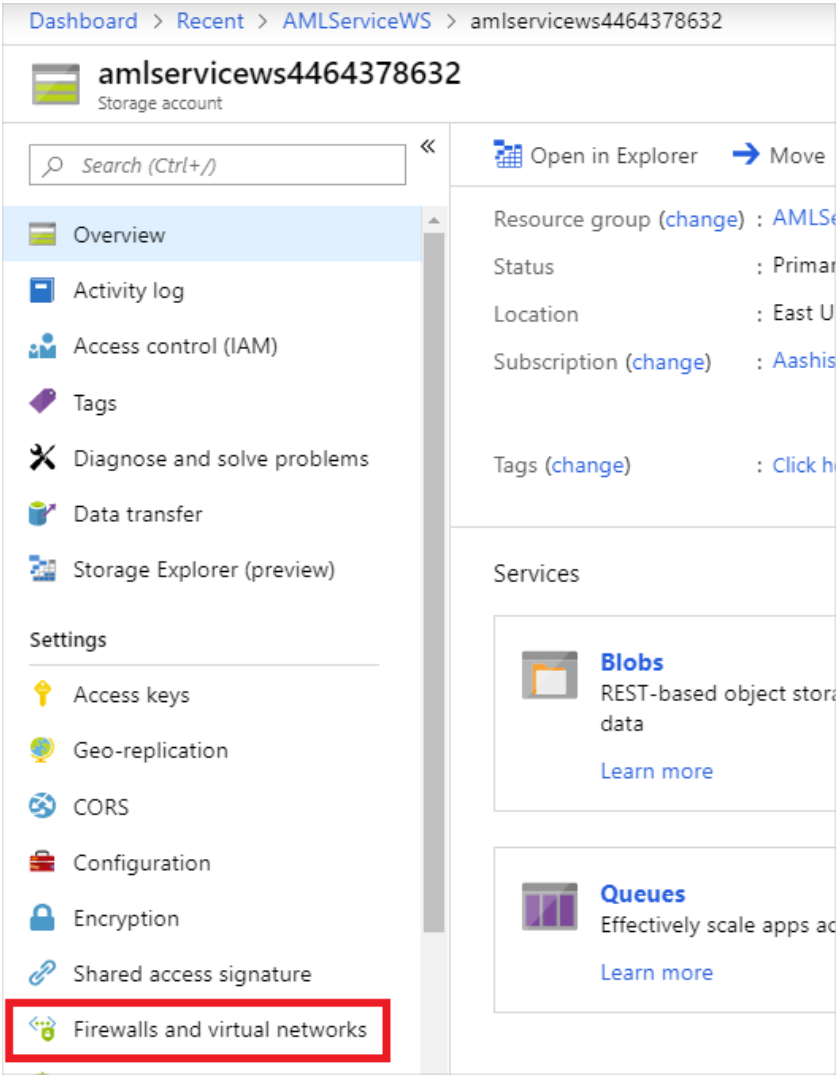
1. Create a compute resource (for example, a Machine Learning compute instance or cluster) behind a virtual network, or attach a compute resource to the workspace (for example, an HDInsight cluster, virtual machine, or Azure Kubernetes Service cluster). The compute resource can be for experimentation or model deployment.

   For more information, see the Use a Machine Learning compute, Use a virtual machine or HDInsight cluster, and Use Azure Kubernetes Service sections in this article.

2. In the Azure portal, go to the storage that's attached to your workspace.



3. On the **Azure Storage** page, select **Firewalls and virtual networks**.



4. On the **Firewalls and virtual networks** page, do the following actions:

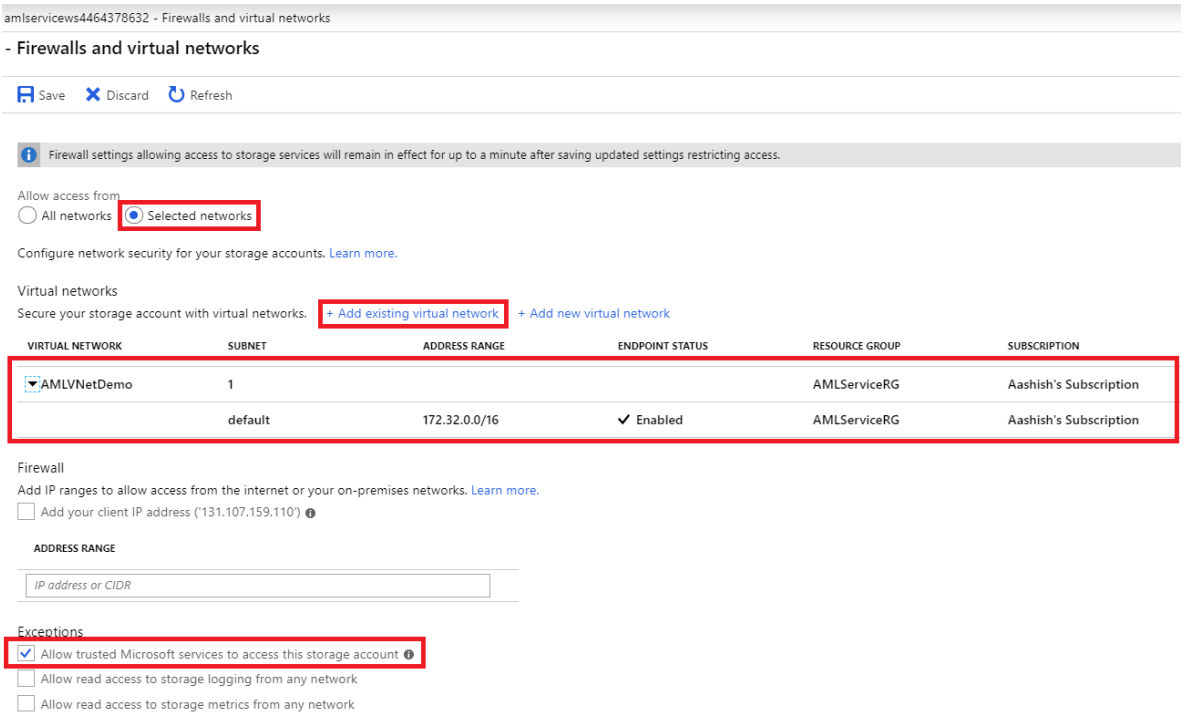
   - Select **Selected networks**.

- Under **Virtual networks**, select the **Add existing virtual network** link. This action adds the virtual network where your compute resides (see step 1).

> **IMPORTANT**
>
> The storage account must be in the same virtual network and subnet as the compute instances or clusters used for training or inference.

- Select the **Allow trusted Microsoft services to access this storage account** check box.

> **IMPORTANT**
>
> When working with the Azure Machine Learning SDK, your development environment must be able to connect to the Azure Storage Account. When the storage account is inside a virtual network, the firewall must allow access from the development environment's IP address.
>
> To enable access to the storage account, visit the **Firewalls and virtual networks** for the storage account *from a web browser on the development client*. Then use the **Add your client IP address** check box to add the client's IP address to the **ADDRESS RANGE**. You can also use the **ADDRESS RANGE** field to manually enter the IP address of the development environment. Once the IP address for the client has been added, it can access the storage account using the SDK.



> **IMPORTANT**
>
> You can place the both the *default storage account* for Azure Machine Learning, or *non-default storage accounts* in a virtual network.
>
> The default storage account is automatically provisioned when you create a workspace.
>
> For non-default storage accounts, the `storage_account` parameter in the `Workspace.create()` function allows you to specify a custom storage account by Azure resource ID.

Use Azure Data Lake Storage Gen 2

Azure Data Lake Storage Gen 2 is a set of capabilities for big data analytics, built on Azure Blob storage. It can be

used to store data used to train models with Azure Machine Learning.

To use Data Lake Storage Gen 2 inside the virtual network of your Azure Machine Learning workspace, use the following steps:

1. Create an Azure Data Lake Storage gen 2 account. For more information, see Create an Azure Data Lake Storage Gen2 storage account.
2. Use the steps 2-4 in the previous section, Use a storage account for your workspace, to put the account in the virtual network.

When using Azure Machine Learning with Data Lake Storage Gen 2 inside a virtual network, use the following guidance:

- If you use the **SDK to create a dataset**, and the system running the code **is not in the virtual network**, use the `validate=False` parameter. This parameter skips validation, which fails if the system is not in the same virtual network as the storage account. For more information, see the from_files() method.
- When using Azure Machine Learning Compute Instance or compute cluster to train a model using the dataset, it must be in the same virtual network as the storage account.

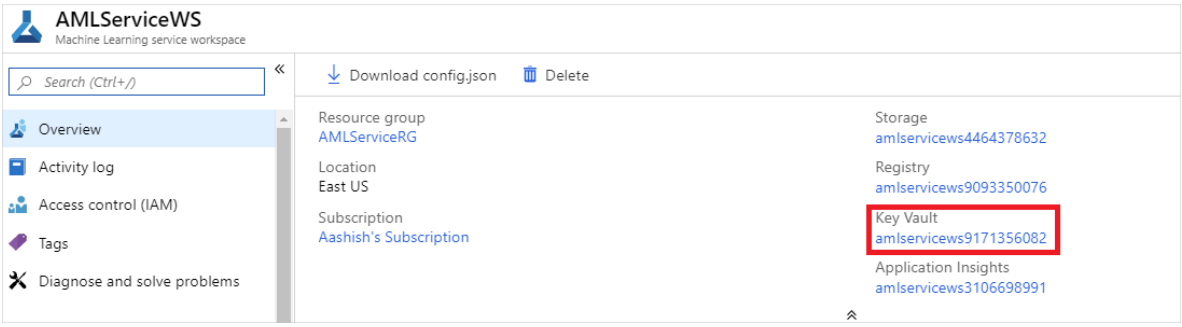
Use a key vault instance with your workspace

The key vault instance that's associated with the workspace is used by Azure Machine Learning to store the following credentials:

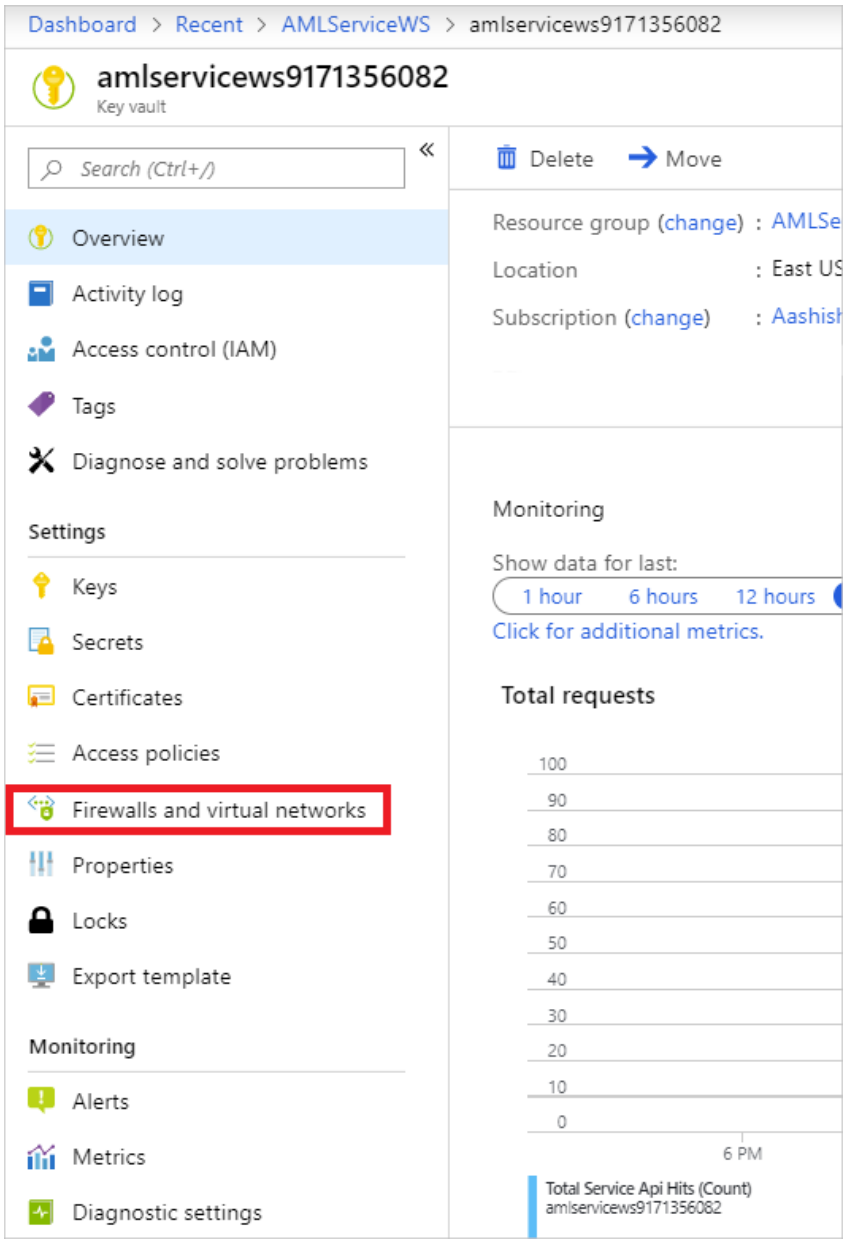
- The associated storage account connection string
- Passwords to Azure Container Repository instances
- Connection strings to data stores

To use Azure Machine Learning experimentation capabilities with Azure Key Vault behind a virtual network, use the following steps:

1. Go to the key vault that's associated with the workspace.

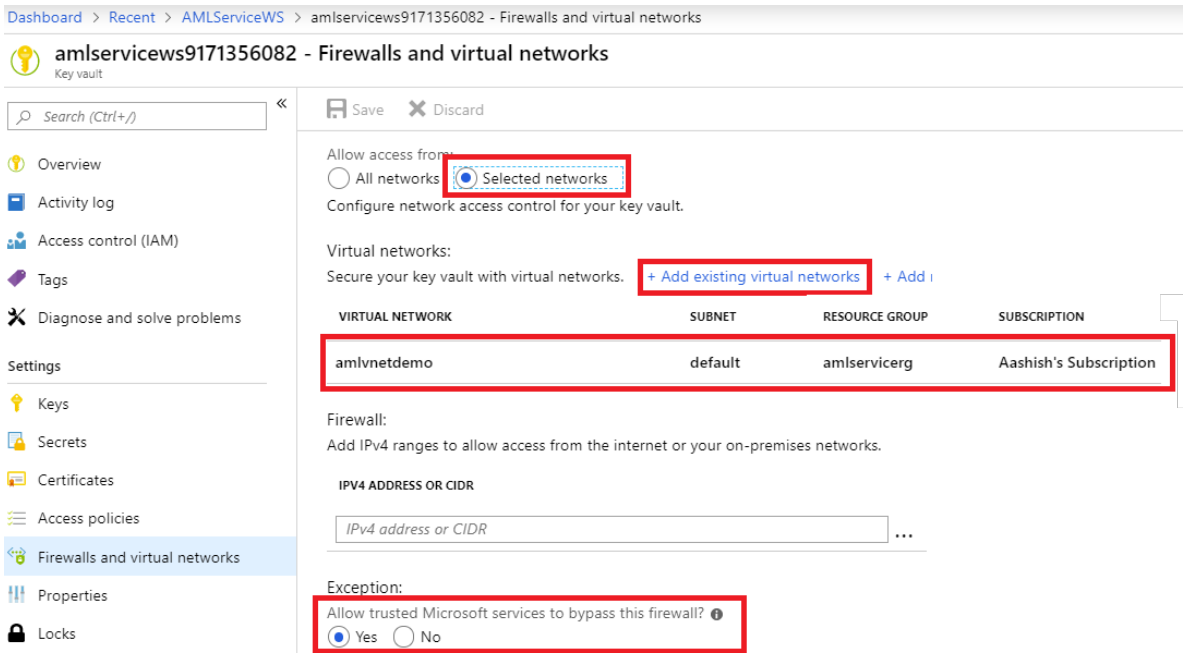


2. On the **Key Vault** page, in the left pane, select **Firewalls and virtual networks**.



3. On the **Firewalls and virtual networks** page, do the following actions:

   - Under **Allow access from**, select **Selected networks**.
   - Under **Virtual networks**, select **Add existing virtual networks** to add the virtual network where your experimentation compute resides.
   - Under **Allow trusted Microsoft services to bypass this firewall**, select **Yes**.



Use a Machine Learning Compute

To use an Azure Machine Learning compute instance or compute cluster in a virtual network, the following network requirements must be met:

- The virtual network must be in the same subscription and region as the Azure Machine Learning workspace.
- The subnet that's specified for the compute instance or cluster must have enough unassigned IP addresses to accommodate the number of VMs that are targeted. If the subnet doesn't have enough unassigned IP addresses, a compute cluster will be partially allocated.
- Check to see whether your security policies or locks on the virtual network's subscription or resource group restrict permissions to manage the virtual network. If you plan to secure the virtual network by restricting traffic, leave some ports open for the compute service. For more information, see the Required ports section.
- If you're going to put multiple compute instances or clusters in one virtual network, you might need to request a quota increase for one or more of your resources.
- If the Azure Storage Account(s) for the workspace are also secured in a virtual network, they must be in the same virtual network as the Azure Machine Learning compute instance or cluster.

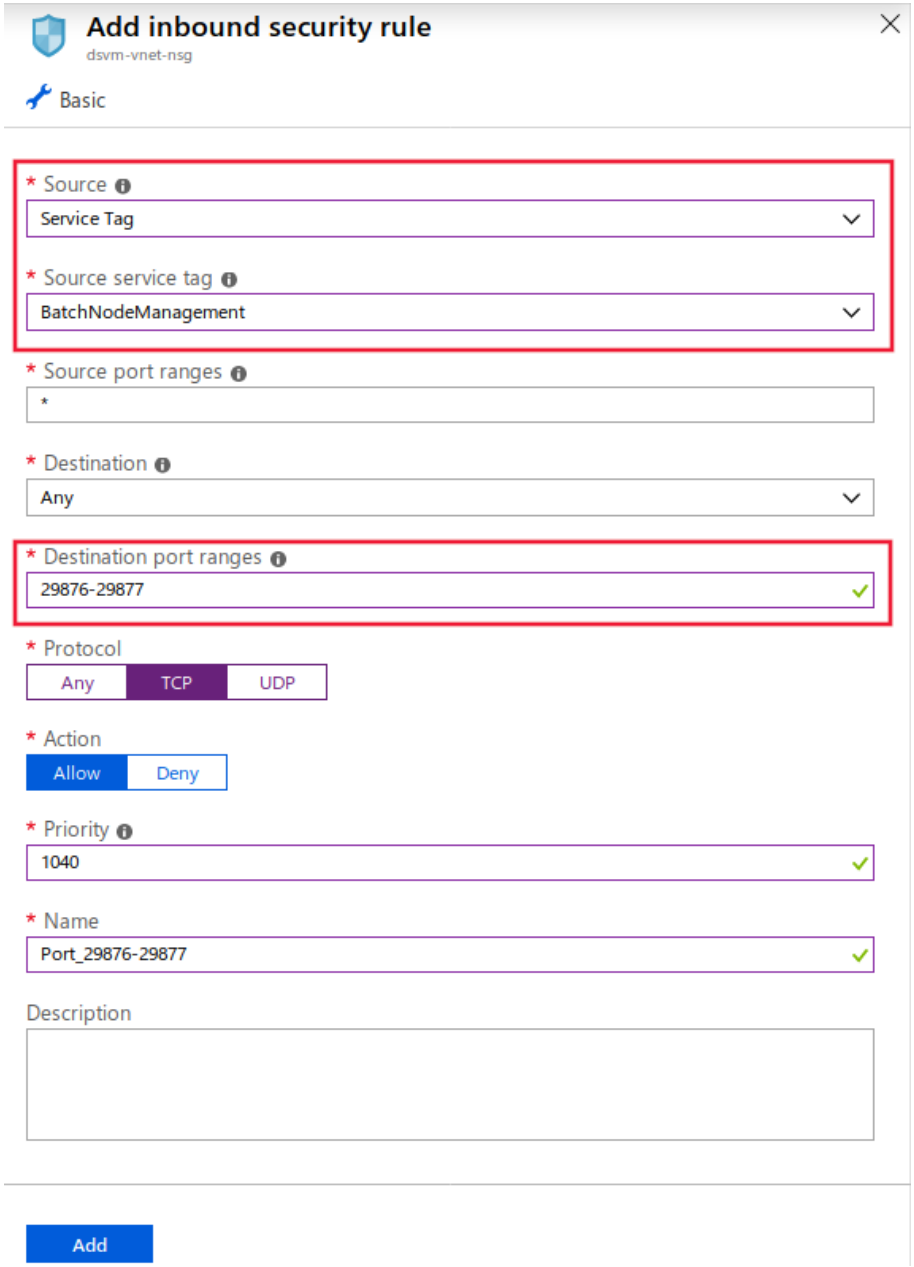
> **TIP**
>
> The Machine Learning compute instance or cluster automatically allocates additional networking resources **in the resource group that contains the virtual network**. For each compute instance or cluster, the service allocates the following resources:
>
> - One network security group
> - One public IP address
> - One load balancer
>
> In the case of clusters these resources are deleted (and recreated) every time the cluster scales down to 0 nodes, however for an instance the resources are held onto till the instance is completely deleted (stopping does not remove the resources). These resources are limited by the subscription's resource quotas.

Required ports

Machine Learning Compute currently uses the Azure Batch service to provision VMs in the specified virtual network. The subnet must allow inbound communication from the Batch service. You use this communication to schedule runs on the Machine Learning Compute nodes and to communicate with Azure Storage and other resources. The Batch service adds network security groups (NSGs) at the level of network interfaces (NICs) that are

attached to VMs. These NSGs automatically configure inbound and outbound rules to allow the following traffic:

- Inbound TCP traffic on ports 29876 and 29877 from a **Service Tag** of **BatchNodeManagement**.

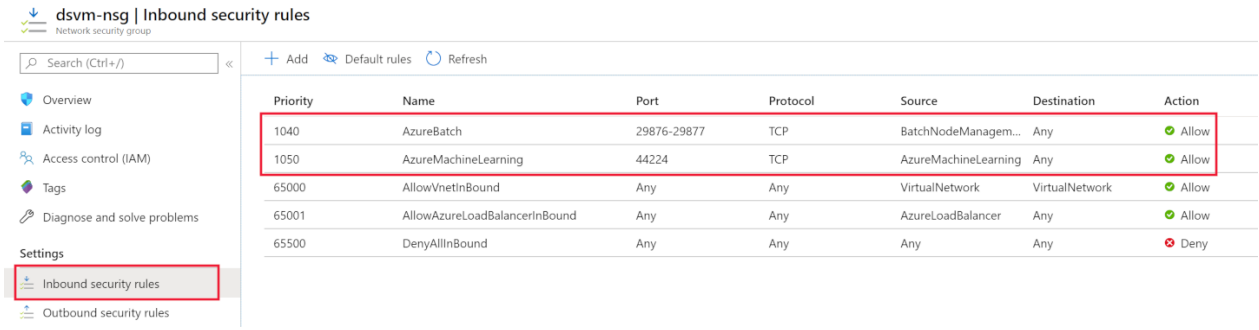


- (Optional) Inbound TCP traffic on port 22 to permit remote access. Use this port only if you want to connect by using SSH on the public IP.
- Outbound traffic on any port to the virtual network.
- Outbound traffic on any port to the internet.
- For compute instance inbound TCP traffic on port 44224 from a **Service Tag** of **AzureMachineLearning**.

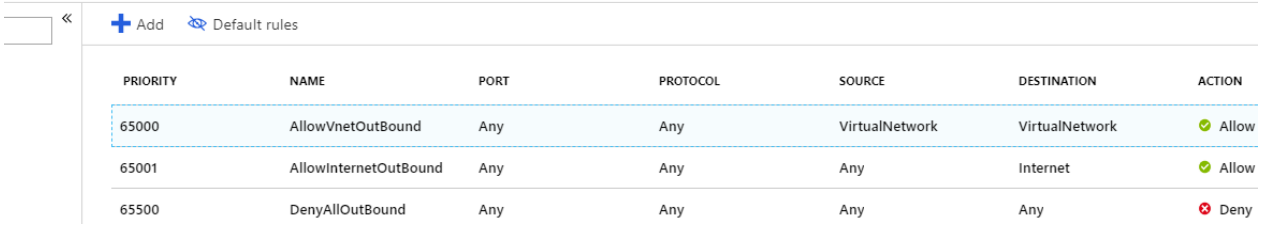
Exercise caution if you modify or add inbound or outbound rules in Batch-configured NSGs. If an NSG blocks communication to the compute nodes, the compute service sets the state of the compute nodes to unusable.

You don't need to specify NSGs at the subnet level, because the Azure Batch service configures its own NSGs. However, if the specified subnet has associated NSGs or a firewall, configure the inbound and outbound security rules as mentioned earlier.

The NSG rule configuration in the Azure portal is shown in the following images:



Outbound security rules

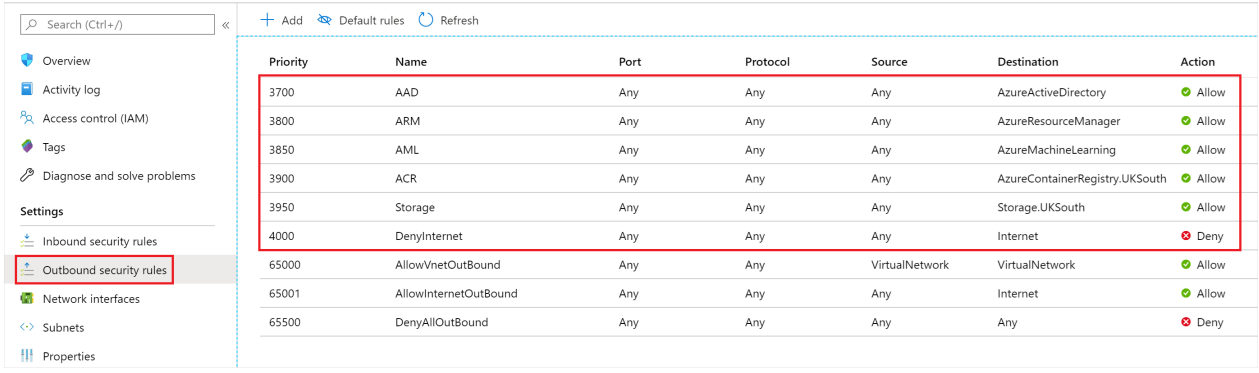


Limit outbound connectivity from the virtual network

If you don't want to use the default outbound rules and you do want to limit the outbound access of your virtual network, use the following steps:

- Deny outbound internet connection by using the NSG rules.
- For a **compute instance** or a **compute cluster**, limit outbound traffic to the following items:
  - Azure Storage, by using **Service Tag** of **Storage.RegionName**. Where `{RegionName}` is the name of an Azure region.
  - Azure Container Registry, by using **Service Tag** of **AzureContainerRegistry.RegionName**. Where `{RegionName}` is the name of an Azure region.
  - Azure Machine Learning, by using **Service Tag** of **AzureMachineLearning**
  - Azure Resource Manager, by using **Service Tag** of **AzureResourceManager**
  - Azure Active Directory, by using **Service Tag** of **AzureActiveDirectory**

The NSG rule configuration in the Azure portal is shown in the following image:



NOTE

If you plan on using default Docker images provided by Microsoft, and enabling user managed dependencies, you must also use a **Service Tag** of **MicrosoftContainerRegistry.Region_Name** (For example, MicrosoftContainerRegistry.EastUS).

This configuration is needed when you have code similar to the following snippets as part of your training scripts:

RunConfig training

```
# create a new runconfig object
run_config = RunConfiguration()

# configure Docker
run_config.environment.docker.enabled = True
# For GPU, use DEFAULT_GPU_IMAGE
run_config.environment.docker.base_image = DEFAULT_CPU_IMAGE
run_config.environment.python.user_managed_dependencies = True
```

Estimator training

```
est = Estimator(source_directory='.',
                script_params=script_params,
                compute_target='local',
                entry_script='dummy_train.py',
                user_managed=True)
run = exp.submit(est)
```

User-defined routes for forced tunneling

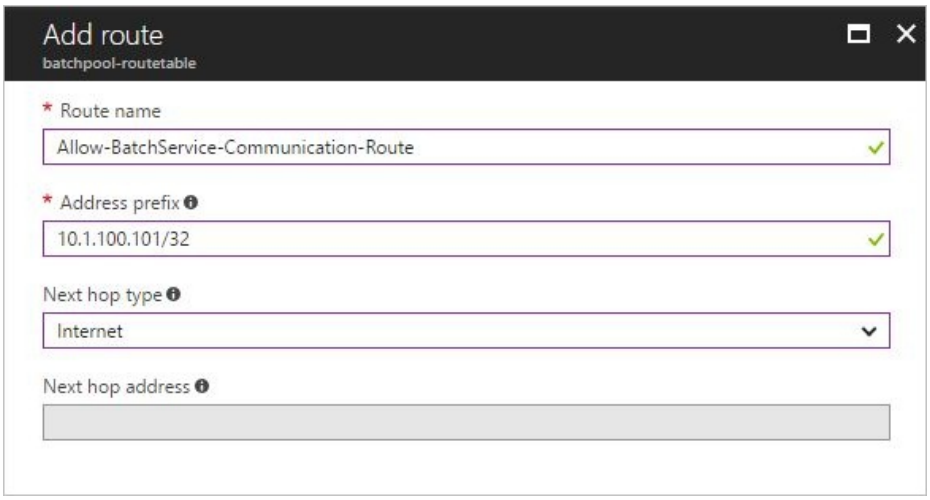
If you're using forced tunneling with the Machine Learning Compute, add user-defined routes (UDRs) to the subnet that contains the compute resource.

- Establish a UDR for each IP address that's used by the Azure Batch service in the region where your resources exist. These UDRs enable the Batch service to communicate with compute nodes for task scheduling. Also add the IP address for the Azure Machine Learning service where the resources exist, as this is required for access to Compute Instances. To get a list of IP addresses of the Batch service and Azure Machine Learning service, use one of the following methods:
  - Download the Azure IP Ranges and Service Tags and search the file for `BatchNodeManagement.<region>` and `AzureMachineLearning.<region>`, where `<region>` is your Azure region.
  - Use the Azure CLI to download the information. The following example downloads the IP address information and filters out the information for the East US 2 region:

    ```
    az network list-service-tags -l "East US 2" --query "values[?starts_with(id, 'Batch')] | [?
    properties.region=='eastus2']"
    az network list-service-tags -l "East US 2" --query "values[?starts_with(id,
    'AzureMachineLearning')] | [?properties.region=='eastus2']"
    ```

- Outbound traffic to Azure Storage must not be blocked by your on-premises network appliance. Specifically, the URLs are in the form `<account>.table.core.windows.net`, `<account>.queue.core.windows.net`, and `<account>.blob.core.windows.net`.

When you add the UDRs, define the route for each related Batch IP address prefix and set **Next hop type** to **Internet**. The following image shows an example of this UDR in the Azure portal:

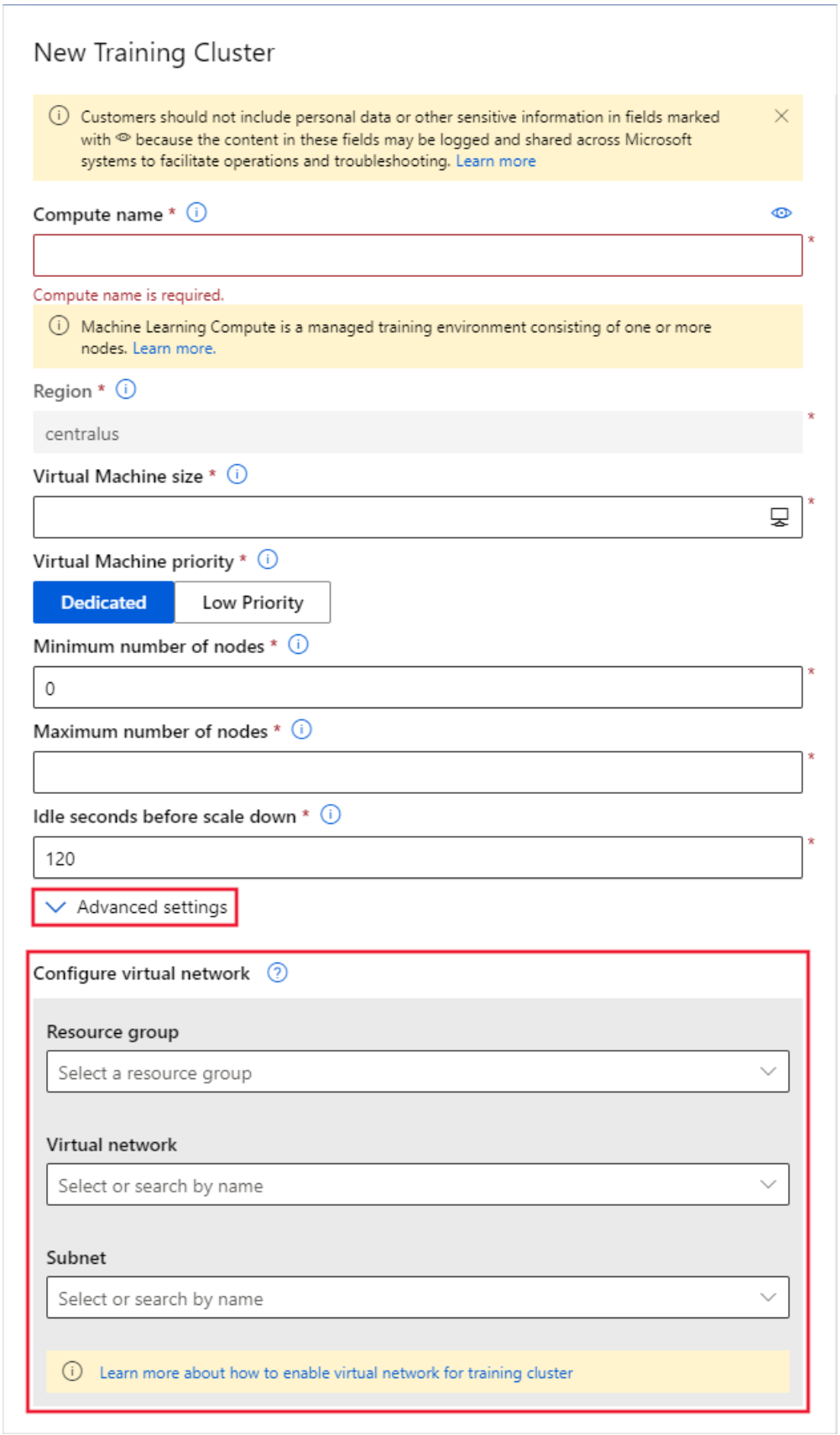


For more information, see Create an Azure Batch pool in a virtual network.

Create a compute cluster in a virtual network

To create a Machine Learning Compute cluster, use the following steps:

1. Sign in to Azure Machine Learning studio, and then select your subscription and workspace.
2. Select **Compute** on the left.
3. Select **Training clusters** from the center, and then select **+**.
4. In the **New Training Cluster** dialog, expand the **Advanced settings** section.
5. To configure this compute resource to use a virtual network, perform the following actions in the **Configure virtual network** section:
   a. In the **Resource group** drop-down list, select the resource group that contains the virtual network.
   b. In the **Virtual network** drop-down list, select the virtual network that contains the subnet.
   c. In the **Subnet** drop-down list, select the subnet to use.



You can also create a Machine Learning Compute cluster by using the Azure Machine Learning SDK. The following code creates a new Machine Learning Compute cluster in the `default` subnet of a virtual network named `mynetwork`:

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# The Azure virtual network name, subnet, and resource group
vnet_name = 'mynetwork'
subnet_name = 'default'
vnet_resourcegroup_name = 'mygroup'

# Choose a name for your CPU cluster
cpu_cluster_name = "cpucluster"

# Verify that cluster does not exist already
try:
    cpu_cluster = ComputeTarget(workspace=ws, name=cpu_cluster_name)
    print("Found existing cpucluster")
except ComputeTargetException:
    print("Creating new cpucluster")

    # Specify the configuration for the new cluster
    compute_config = AmlCompute.provisioning_configuration(vm_size="STANDARD_D2_V2",
                                                           min_nodes=0,
                                                           max_nodes=4,
                                                           vnet_resourcegroup_name=vnet_resourcegroup_name,
                                                           vnet_name=vnet_name,
                                                           subnet_name=subnet_name)

    # Create the cluster with the specified name and configuration
    cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)

    # Wait for the cluster to be completed, show the output log
    cpu_cluster.wait_for_completion(show_output=True)
```

When the creation process finishes, you train your model by using the cluster in an experiment. For more information, see Select and use a compute target for training.

Use Azure Databricks

To use Azure Databricks in a virtual network with your workspace, the following requirements must be met:

- The virtual network must be in the same subscription and region as the Azure Machine Learning workspace.
- If the Azure Storage Account(s) for the workspace are also secured in a virtual network, they must be in the same virtual network as the Azure Databricks cluster.
- In addition to the **databricks-private** and **databricks-public** subnets used by Azure Databricks, the **default** subnet created for the virtual network is also required.

For specific information on using Azure Databricks with a virtual network, see Deploy Azure Databricks in your Azure Virtual Network.

Use a virtual machine or HDInsight cluster

> **IMPORTANT**
> Azure Machine Learning supports only virtual machines that are running Ubuntu.

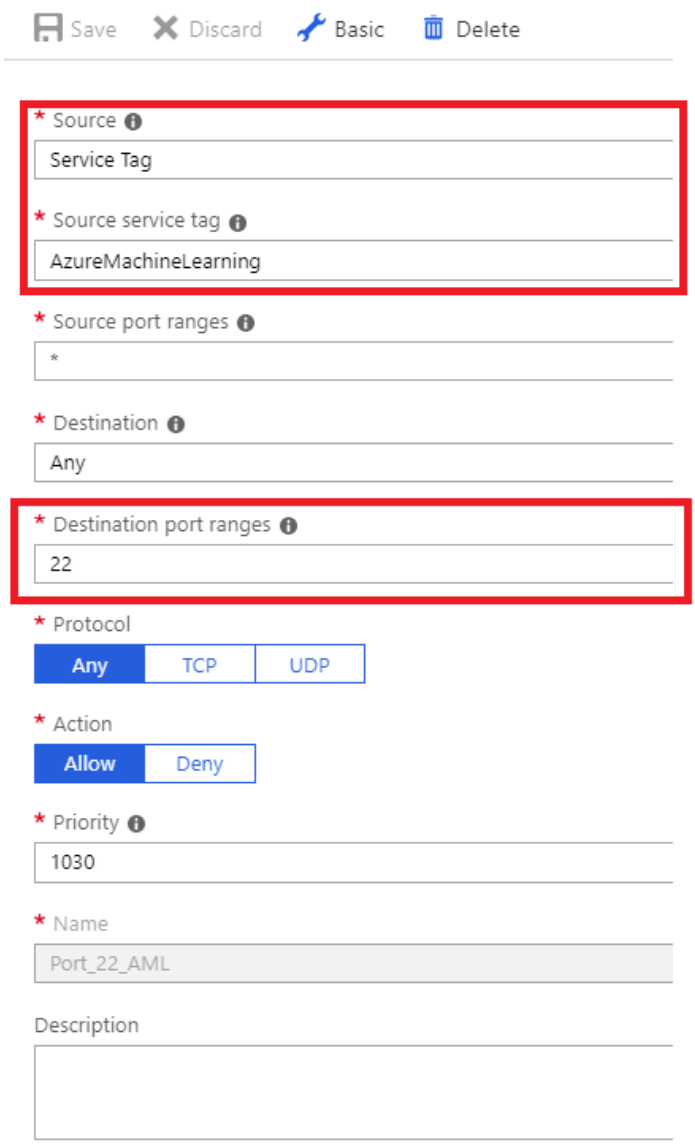
To use a virtual machine or Azure HDInsight cluster in a virtual network with your workspace, use the following steps:

1. Create a VM or HDInsight cluster by using the Azure portal or the Azure CLI, and put the cluster in an Azure virtual network. For more information, see the following articles:

- Create and manage Azure virtual networks for Linux VMs
- Extend HDInsight using an Azure virtual network

2. To allow Azure Machine Learning to communicate with the SSH port on the VM or cluster, configure a source entry for the network security group. The SSH port is usually port 22. To allow traffic from this source, do the following actions:

   - In the **Source** drop-down list, select **Service Tag**.
   - In the **Source service tag** drop-down list, select **AzureMachineLearning**.
   - In the **Source port ranges** drop-down list, select *****.
   - In the **Destination** drop-down list, select **Any**.
   - In the **Destination port ranges** drop-down list, select **22**.
   - Under **Protocol**, select **Any**.
   - Under **Action**, select **Allow**.



Keep the default outbound rules for the network security group. For more information, see the default security rules in Security groups.

If you don't want to use the default outbound rules and you do want to limit the outbound access of your virtual network, see the Limit outbound connectivity from the virtual network section.

3. Attach the VM or HDInsight cluster to your Azure Machine Learning workspace. For more information, see Set up compute targets for model training.

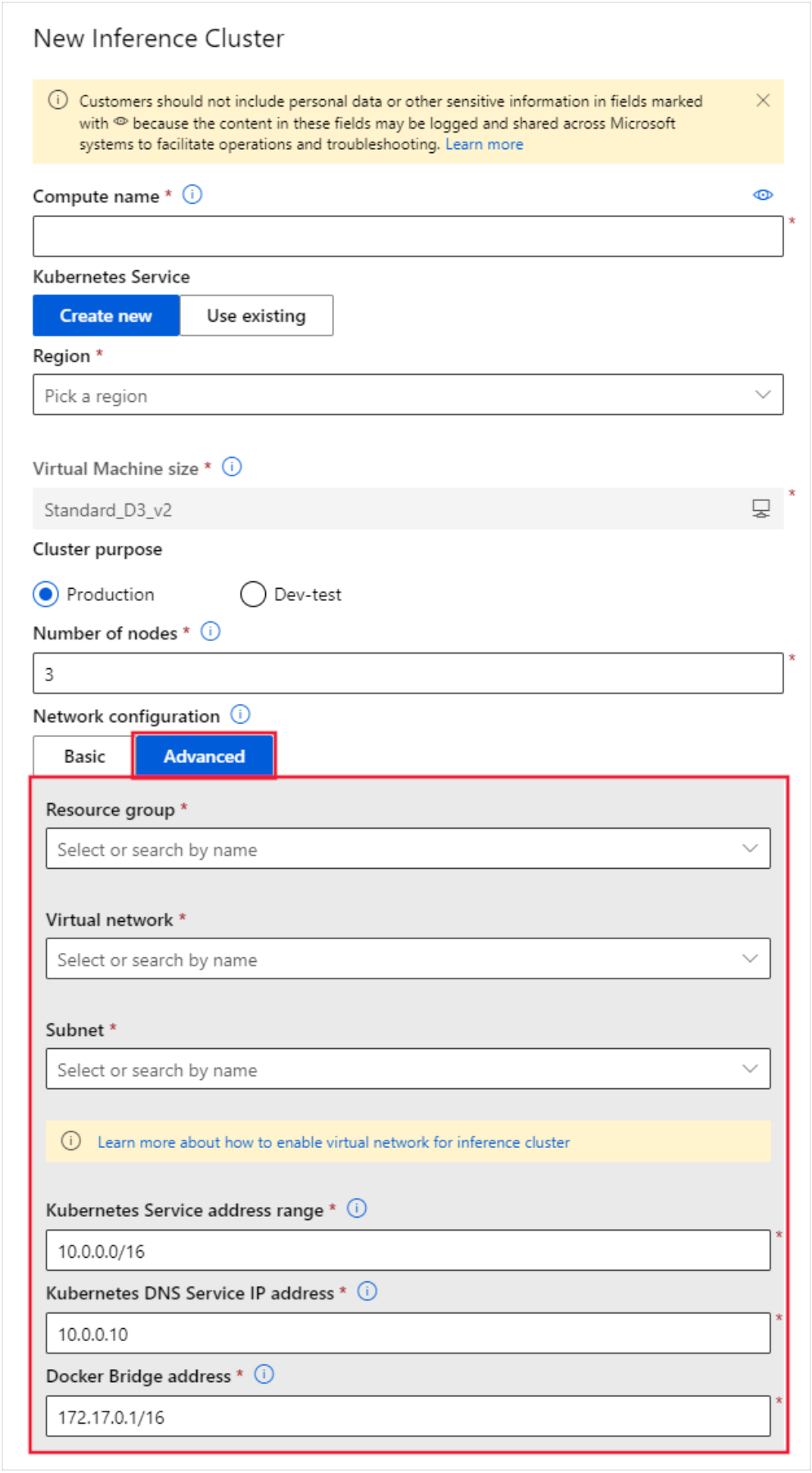
Use Azure Kubernetes Service (AKS)

To add AKS in a virtual network to your workspace, use the following steps:

> **IMPORTANT**
>
> Before you begin the following procedure, follow the prerequisites in the Configure advanced networking in Azure Kubernetes Service (AKS) how-to and plan the IP addressing for your cluster.
>
> The AKS instance and the Azure virtual network must be in the same region. If you secure the Azure Storage Account(s) used by the workspace in a virtual network, they must be in the same virtual network as the AKS instance.

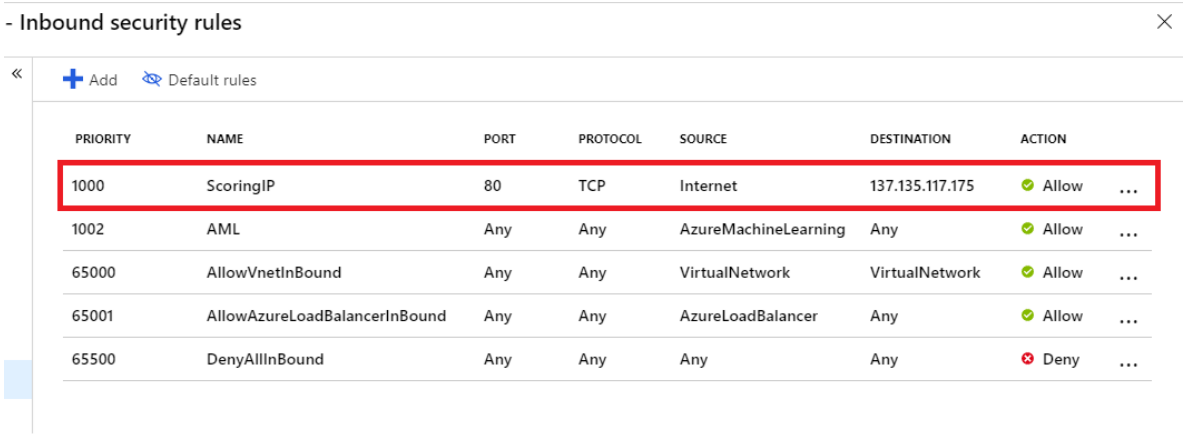
> **WARNING**
>
> Azure Machine Learning does not support using an Azure Kubernetes Service that has private link enabled.

1. Sign in to Azure Machine Learning studio, and then select your subscription and workspace.
2. Select **Compute** on the left.
3. Select **Inference clusters** from the center, and then select **+**.
4. In the **New Inference Cluster** dialog, select **Advanced** under **Network configuration**.
5. To configure this compute resource to use a virtual network, perform the following actions:
    a. In the **Resource group** drop-down list, select the resource group that contains the virtual network.
    b. In the **Virtual network** drop-down list, select the virtual network that contains the subnet.
    c. In the **Subnet** drop-down list, select the subnet.
    d. In the **Kubernetes Service address range** box, enter the Kubernetes service address range. This address range uses a Classless Inter-Domain Routing (CIDR) notation IP range to define the IP addresses that are available for the cluster. It must not overlap with any subnet IP ranges (for example, 10.0.0.0/16).
    e. In the **Kubernetes DNS service IP address** box, enter the Kubernetes DNS service IP address. This IP address is assigned to the Kubernetes DNS service. It must be within the Kubernetes service address range (for example, 10.0.0.10).
    f. In the **Docker bridge address** box, enter the Docker bridge address. This IP address is assigned to Docker Bridge. It must not be in any subnet IP ranges, or the Kubernetes service address range (for example, 172.17.0.1/16).



6. Make sure that the NSG group that controls the virtual network has an inbound security rule enabled for the scoring endpoint so that it can be called from outside the virtual network.

> **IMPORTANT**
>
> Keep the default outbound rules for the NSG. For more information, see the default security rules in Security groups.



You can also use the Azure Machine Learning SDK to add Azure Kubernetes Service in a virtual network. If you already have an AKS cluster in a virtual network, attach it to the workspace as described in How to deploy to AKS. The following code creates a new AKS instance in the `default` subnet of a virtual network named `mynetwork`:

```
from azureml.core.compute import ComputeTarget, AksCompute

# Create the compute configuration and set virtual network information
config = AksCompute.provisioning_configuration(location="eastus2")
config.vnet_resourcegroup_name = "mygroup"
config.vnet_name = "mynetwork"
config.subnet_name = "default"
config.service_cidr = "10.0.0.0/16"
config.dns_service_ip = "10.0.0.10"
config.docker_bridge_cidr = "172.17.0.1/16"

# Create the compute target
aks_target = ComputeTarget.create(workspace=ws,
                                  name="myaks",
                                  provisioning_configuration=config)
```

When the creation process is completed, you can run inference, or model scoring, on an AKS cluster behind a virtual network. For more information, see How to deploy to AKS.

Use private IPs with Azure Kubernetes Service

By default, a public IP address is assigned to AKS deployments. When using AKS inside a virtual network, you can use a private IP address instead. Private IP addresses are only accessible from inside the virtual network or joined networks.

A private IP address is enabled by configuring AKS to use an *internal load balancer*.

> **IMPORTANT**
> You cannot enable private IP when creating the Azure Kubernetes Service cluster. It must be enabled as an update to an existing cluster.

The following code snippet demonstrates how to **create a new AKS cluster**, and then update it to use a private IP/internal load balancer:

```
import azureml.core
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute import AksCompute, ComputeTarget

# Verify that cluster does not exist already
try:
    aks_target = AksCompute(workspace=ws, name=aks_cluster_name)
    print("Found existing aks cluster")

except:
    print("Creating new aks cluster")

    # Subnet to use for AKS
    subnet_name = "default"
    # Create AKS configuration
    prov_config = AksCompute.provisioning_configuration(location = "eastus2")
    # Set info for existing virtual network to create the cluster in
    prov_config.vnet_resourcegroup_name = "myvnetresourcegroup"
    prov_config.vnet_name = "myvnetname"
    prov_config.service_cidr = "10.0.0.0/16"
    prov_config.dns_service_ip = "10.0.0.10"
    prov_config.subnet_name = subnet_name
    prov_config.docker_bridge_cidr = "172.17.0.1/16"

    # Create compute target
    aks_target = ComputeTarget.create(workspace = ws, name = "myaks", provisioning_configuration =
prov_config)
    # Wait for the operation to complete
    aks_target.wait_for_completion(show_output = True)

    # Update AKS configuration to use an internal load balancer
    update_config = AksUpdateConfiguration(None, "InternalLoadBalancer", subnet_name)
    aks_target.update(update_config)
    # Wait for the operation to complete
    aks_target.wait_for_completion(show_output = True)
```

Azure CLI

```
az rest --method put --uri https://management.azure.com"/subscriptions/<subscription-
id>/resourcegroups/<resource-group>/providers/Microsoft.ContainerService/managedClusters/<aks-resource-id>?
api-version=2018-11-19 --body @body.json
```

The contents of the `body.json` file referenced by the command are similar to the following JSON document:

```
{
    "location": "<region>",
    "properties": {
        "resourceId": "/subscriptions/<subscription-id>/resourcegroups/<resource-
group>/providers/Microsoft.ContainerService/managedClusters/<aks-resource-id>",
        "computeType": "AKS",
        "provisioningState": "Succeeded",
        "properties": {
            "loadBalancerType": "InternalLoadBalancer",
            "agentCount": <agent-count>,
            "agentVmSize": "vm-size",
            "clusterFqdn": "<cluster-fqdn>"
        }
    }
}
```

> **NOTE**
> Currently, you cannot configure the load balancer when performing an **attach** operation on an existing cluster. You must first attach the cluster, and then perform an update operation to change the load balancer.

For more information on using the internal load balancer with AKS, see Use internal load balancer with Azure Kubernetes Service.

Use Azure Container Instances (ACI)

Azure Container Instances are dynamically created when deploying a model. To enable Azure Machine Learning to create ACI inside the virtual network, you must enable **subnet delegation** for the subnet used by the deployment.

To use ACI in a virtual network to your workspace, use the following steps:

1. To enable subnet delegation on your virtual network, use the information in the Add or remove a subnet delegation article. You can enable delegation when creating a virtual network, or add it to an existing network.

   > **IMPORTANT**
   > When enabling delegation, use `Microsoft.ContainerInstance/containerGroups` as the **Delegate subnet to service** value.

2. Deploy the model using AciWebservice.deploy_configuration(), use the `vnet_name` and `subnet_name` parameters. Set these parameters to the virtual network name and subnet where you enabled delegation.

Use Azure Firewall

When using Azure Firewall, you must configure a network rule to allow traffic to and from the following addresses:

- `*.batchai.core.windows.net`
- `ml.azure.com`
- `*.azureml.ms`
- `*.experiments.azureml.net`
- `*.modelmanagement.azureml.net`
- `mlworkspace.azure.ai`
- `*.aether.ms`

When adding the rule, set the **Protocol** to any, and the ports to `*`.

For more information on configuring a network rule, see Deploy and configure Azure Firewall.

Use Azure Container Registry

IMPORTANT

Azure Container Registry (ACR) can be put inside a virtual network, however you must meet the following prerequisites:

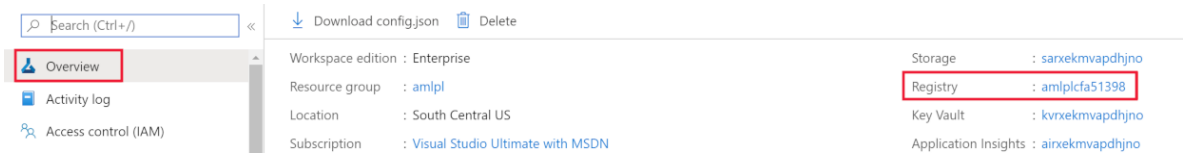
- Your Azure Machine Learning workspace must be Enterprise edition. For information on upgrading, see Upgrade to Enterprise edition.
- Your Azure Container Registry must be Premium version . For more information on upgrading, see Changing SKUs.
- Your Azure Container Registry must be in the same virtual network and subnet as the storage account and compute targets used for training or inference.
- Your Azure Machine Learning workspace must contain an Azure Machine Learning compute cluster.

  When ACR is behind a virtual network, Azure Machine Learning cannot use it to directly build Docker images. Instead, the compute cluster is used to build the images.

1. To find the name of the Azure Container Registry for your workspace, use one of the following methods:

   ### Azure portal

   From the overview section of your workspace, the **Registry** value links to the Azure Container Registry.

   ### Azure CLI

   If you have installed the Machine Learning extension for Azure CLI, you can use the `az ml workspace show` command to show the workspace information.

   ```
   az ml workspace show -w yourworkspacename -g resourcegroupname --query 'containerRegistry'
   ```

   This command returns a value similar to `"/subscriptions/{GUID}/resourceGroups/{resourcegroupname}/providers/Microsoft.ContainerRegistry/registries/{ACRname}"`. The last part of the string is the name of the Azure Container Registry for the workspace.

2. To limit access to your virtual network, use the steps in Configure network access for registry. When adding the virtual network, select the virtual network and subnet for your Azure Machine Learning resources.

3. Use the Azure Machine Learning Python SDK to configure a compute cluster to build docker images. The following code snippet demonstrates how to do this:

   ```
   from azureml.core import Workspace
   # Load workspace from an existing config file
   ws = Workspace.from_config()
   # Update the workspace to use an existing compute cluster
   ws.update(image_build_compute = 'mycomputecluster')
   ```

   IMPORTANT

   Your storage account, compute cluster, and Azure Container Registry must all be in the same subnet of the virtual network.

   For more information, see the update() method reference.

4. If you are using Private Link for your Azure Machine Learning workspace, and put the Azure Container Registry for your workspace in a virtual network, you must also apply the following Azure Resource Manager template. This template enables your workspace to communicate with ACR over the Private Link.

```
{
"$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
"contentVersion": "1.0.0.0",
"parameters": {
    "keyVaultArmId": {
    "type": "string"
    },
    "workspaceName": {
    "type": "string"
    },
    "containerRegistryArmId": {
    "type": "string"
    },
    "applicationInsightsArmId": {
    "type": "string"
    },
    "storageAccountArmId": {
    "type": "string"
    },
    "location": {
    "type": "string"
    }
},
"resources": [
    {
    "type": "Microsoft.MachineLearningServices/workspaces",
    "apiVersion": "2019-11-01",
    "name": "[parameters('workspaceName')]",
    "location": "[parameters('location')]",
    "identity": {
        "type": "SystemAssigned"
    },
    "sku": {
        "tier": "enterprise",
        "name": "enterprise"
    },
    "properties": {
        "sharedPrivateLinkResources":
[{"Name":"Acr","Properties":{"PrivateLinkResourceId":"[concat(parameters('containerRegistryArmId'),
'/privateLinkResources/registry')]","GroupId":"registry","RequestMessage":"Approve","Status":"Pending"}
}],
        "keyVault": "[parameters('keyVaultArmId')]",
        "containerRegistry": "[parameters('containerRegistryArmId')]",
        "applicationInsights": "[parameters('applicationInsightsArmId')]",
        "storageAccount": "[parameters('storageAccountArmId')]"
    }
    }
]
}
```

Next steps

- Set up training environments
- Where to deploy models
- Use TLS to secure a web service through Azure Machine Learning

Configure Azure Private Link for an Azure Machine Learning workspace (Preview)

4/15/2020 • 13 minutes to read • Edit Online

In this document, you learn how to use Azure Private Link with your Azure Machine Learning workspace. This capability is currently in preview, and is available in the US East, US West 2, US South Central regions.

Azure Private Link enables you to connect to your workspace using a private endpoint. The private endpoint is a set of private IP addresses within your virtual network. You can then limit access to your workspace to only occur over the private IP addresses. Private Link helps reduce the risk of data exfiltration. To learn more about private endpoints, see the Azure Private Link article.

> **IMPORTANT**
>
> Azure Private Link does not effect Azure control plane (management operations) such as deleting the workspace or managing compute resources. For example, creating, updating, or deleting a compute target. These operations are performed over the public Internet as normal.
>
> Azure Machine Learning compute instances preview is not supported in a workspace where Private Link is enabled.

Create a workspace that uses a private endpoint

Currently, we only support enabling a private endpoint when creating a new Azure Machine Learning workspace. The following templates are provided for several popular configurations:

> **TIP**
>
> Auto-approval controls the automated access to the Private Link enabled resource. For more information, see What is Azure Private Link service.

- Workspace with customer-managed keys and auto-approval for Private Link
- Workspace with customer-managed keys and manual approval for Private Link
- Workspace with Microsoft-managed keys and auto-approval for Private Link
- Workspace with Microsoft-managed keys and manual approval for Private Link

When deploying a template, you must provide the following information:

- Workspace name
- Azure region to create the resources in
- Workspace edition (Basic or Enterprise)
- If high confidentiality settings for the workspace should be enabled
- If encryption for the workspace with a customer-managed key should be enabled, and associated values for the key
- Virtual Network and Subnet name, template will create new virtual network and subnet

Once a template has been submitted and provisioning completes, the resource group that contains your workspace will contain three new artifact types related to Private Link:

- Private endpoint

- Network interface
- Private DNS zone

The workspace also contains an Azure Virtual Network that can communicate with the workspace over the private endpoint.

Deploy the template using the Azure portal

1. Follow the steps in Deploy resources from custom template. When you arrive at the **Edit template** screen, paste in one of the templates from the end of this document.

2. Select **Save** to use the template. Provide the following information and agree to the listed terms and conditions:

   - Subscription: Select the Azure subscription to use for these resources.
   - Resource group: Select or create a resource group to contain the services.
   - Workspace name: The name to use for the Azure Machine Learning workspace that will be created. The workspace name must be between 3 and 33 characters. It may only contain alphanumeric characters and '-'.
   - Location: Select the location where the resources will be created.

For more information, see Deploy resources from custom template.

Deploy the template using Azure PowerShell

This example assumes that you have saved one of the templates from the end of this document to a file named `azuredeploy.json` in the current directory:

```
New-AzResourceGroup -Name examplegroup -Location "East US"
new-azresourcegroupdeployment -name exampledeployment `
  -resourcegroupname examplegroup -location "East US" `
  -templatefile .\azuredeploy.json -workspaceName "exampleworkspace" -sku "basic"
```

For more information, see Deploy resources with Resource Manager templates and Azure PowerShell and Deploy private Resource Manager template with SAS token and Azure PowerShell.

Deploy the template using the Azure CLI

This example assumes that you have saved one of the templates from the end of this document to a file named `azuredeploy.json` in the current directory:

```
az group create --name examplegroup --location "East US"
az group deployment create \
  --name exampledeployment \
  --resource-group examplegroup \
  --template-file azuredeploy.json \
  --parameters workspaceName=exampleworkspace location=eastus sku=basic
```

For more information, see Deploy resources with Resource Manager templates and Azure CLI and Deploy private Resource Manager template with SAS token and Azure CLI.

Using a workspace over a private endpoint

Since communication to the workspace is only allowed from the virtual network, any development environments that use the workspace must be members of the virtual network. For example, a virtual machine in the virtual network or a machine connected to the virtual network using a VPN gateway.

> **IMPORTANT**
> To avoid temporary disruption of connectivity, Microsoft recommends flushing the DNS cache on machines connecting to the workspace after enabling Private Link.

For information on Azure Virtual Machines, see the Virtual Machines documentation.

For information on VPN gateways, see What is VPN gateway.

Using Azure Storage

To secure the Azure Storage account used by your workspace, put it inside the virtual network.

For information on putting the storage account in the virtual network, see Use a storage account for your workspace.

Using Azure Key Vault

To secure the Azure Key Vault used by your workspace, you can either put it inside the virtual network or enable Private Link for it.

For information on putting the key vault in the virtual network, see Use a key vault instance with your workspace.

For information on enabling Private Link for the key vault, see Integrate Key Vault with Azure Private Link.

Using Azure Kubernetes Services

To secure the Azure Kubernetes services used by your workspace, put it inside a virtual network. For more information, see Use Azure Kubernetes Services with your workspace.

> **WARNING**
> Azure Machine Learning does not support using an Azure Kubernetes Service that has private link enabled.

Azure Container Registry

For information on securing Azure Container Registry inside the virtual network, see Use Azure Container Registry.

> **IMPORTANT**
> If you are using Private Link for your Azure Machine Learning workspace, and put the Azure Container Registry for your workspace in a virtual network, you must also apply the following Azure Resource Manager template. This template enables your workspace to communicate with ACR over the Private Link.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
      "keyVaultArmId": {
      "type": "string"
      },
      "workspaceName": {
      "type": "string"
      },
      "containerRegistryArmId": {
      "type": "string"
      },
      "applicationInsightsArmId": {
      "type": "string"
      },
      "storageAccountArmId": {
      "type": "string"
      },
      "location": {
      "type": "string"
      }
  },
  "resources": [
      {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2019-11-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "identity": {
          "type": "SystemAssigned"
      },
      "sku": {
          "tier": "enterprise",
          "name": "enterprise"
      },
      "properties": {
          "sharedPrivateLinkResources":
  [{"Name":"Acr","Properties":{"PrivateLinkResourceId":"[concat(parameters('containerRegistryArmId'),
'/privateLinkResources/registry')]","GroupId":"registry","RequestMessage":"Approve","Status":"Pending"}}],
          "keyVault": "[parameters('keyVaultArmId')]",
          "containerRegistry": "[parameters('containerRegistryArmId')]",
          "applicationInsights": "[parameters('applicationInsightsArmId')]",
          "storageAccount": "[parameters('storageAccountArmId')]"
      }
      }
  ]
}
```

Azure Resource Manager templates

Workspace with customer-managed keys and auto-approval for Private Link

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "workspaceName": {
      "type": "string",
      "metadata": {
        "description": "Specifies the name of the Azure Machine Learning workspace."
      }
    },
    "location": {
      "type": "string",
```

```
      "allowedValues": [
        "eastus",
        "southcentralus",
        "westus2"
      ],
      "metadata": {
        "description": "Specifies the location for all resources."
      }
    },
    "sku":{
      "type": "string",
      "defaultValue": "basic",
      "allowedValues": [
        "basic",
        "enterprise"
      ],
      "metadata": {
        "description": "Specifies the sku, also referred as 'edition' of the Azure Machine Learning
workspace."
      }
    },
    "hbi_workspace":{
      "type": "string",
      "defaultValue": "false",
      "allowedValues": [
        "false",
        "true"
      ],
      "metadata": {
        "description": "Specifies that the Azure Machine Learning workspace holds highly confidential data."
      }
    },
    "encryption_status":{
      "type": "string",
      "defaultValue": "Disabled",
      "allowedValues": [
        "Enabled",
        "Disabled"
      ],
      "metadata": {
        "description": "Specifies if the Azure Machine Learning workspace should be encrypted with customer
managed key."
      }
    },
    "cmk_keyvault":{
      "type": "string",
      "metadata": {
        "description": "Specifies the customer managed keyVault id."
      }
    },
    "resource_cmk_uri":{
      "type": "string",
      "metadata": {
        "description": "Specifies if the customer managed keyvault key uri."
      }
    },
    "subnetName": {
        "type": "string"
      },
      "vnetName": {
        "type": "string"
      }
  },
  "variables": {
    "storageAccountName": "[concat('sa',uniqueString(resourceGroup().id))]",
    "storageAccountType": "Standard_LRS",
    "keyVaultName": "[concat('kv',uniqueString(resourceGroup().id))]",
    "tenantId": "[subscription().tenantId]",
    "applicationInsightsName": "[concat('ai',uniqueString(resourceGroup().id))]",
```

```
    "privateDnsGuid": "[guid(resourceGroup().id, deployment().name)]"
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-04-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[variables('storageAccountType')]"
      },
      "kind": "StorageV2",
      "properties": {
        "encryption": {
          "services": {
            "blob": {
              "enabled": true
            },
            "file": {
              "enabled": true
            }
          },
          "keySource": "Microsoft.Storage"
        },
        "supportsHttpsTrafficOnly": true
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2018-02-14",
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": []
      }
    },
    {
      "type": "Microsoft.Insights/components",
      "apiVersion": "2018-05-01-preview",
      "name": "[variables('applicationInsightsName')]",
      "location": "
[if(or(equals(parameters('location'),'eastus2'),equals(parameters('location'),'westcentralus')),'southcentralu
s',parameters('location'))]",
      "kind": "web",
      "properties": {
        "Application_Type": "web"
      }
    },
    {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2020-01-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
        "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]"
      ],
      "identity": {
        "type": "systemAssigned"
      },
      "sku": {
            "tier": "[parameters('sku')]",
            "name": "[parameters('sku')]"
```

```
      },
      "properties": {
        "friendlyName": "[parameters('workspaceName')]",
        "keyVault": "[resourceId('Microsoft.KeyVault/vaults',variables('keyVaultName'))]",
        "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
        "storageAccount": "
[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]",
         "encryption": {
                "status": "[parameters('encryption_status')]",
                "keyVaultProperties": {
                    "keyVaultArmId": "[parameters('cmk_keyvault')]",
                    "keyIdentifier": "[parameters('resource_cmk_uri')]"
                  }
            },
        "hbi_workspace": "[parameters('hbi_workspace')]"
      }
    },
    {
        "type": "Microsoft.Network/virtualNetworks",
        "apiVersion": "2019-09-01",
        "name": "[parameters('vnetName')]",
        "location": "[parameters('location')]",
        "properties": {
            "addressSpace": {
                "addressPrefixes": [
                    "10.0.0.0/27"
                ]
            },
            "virtualNetworkPeerings": [],
            "enableDdosProtection": false,
            "enableVmProtection": false
        }
      },
    {
        "type": "Microsoft.Network/virtualNetworks/subnets",
        "apiVersion": "2019-09-01",
        "name": "[concat(parameters('vnetName'), '/', parameters('subnetName'))]",
        "dependsOn": [
            "[resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))]"
        ],
        "properties": {
            "addressPrefix": "10.0.0.0/27",
            "delegations": [],
            "privateEndpointNetworkPolicies": "Disabled",
            "privateLinkServiceNetworkPolicies": "Enabled"
        }
    },
    {
      "apiVersion": "2019-04-01",
      "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
      "type": "Microsoft.Network/privateEndpoints",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.MachineLearningServices/workspaces', parameters('workspaceName'))]",
        "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
      ],
      "properties": {
        "privateLinkServiceConnections": [
          {
            "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
            "properties": {
              "privateLinkServiceId": "[resourceId('Microsoft.MachineLearningServices/workspaces',
parameters('workspaceName'))]",
              "groupIds": [
                "amlworkspace"
              ]
            }
```

```
          }
        ],
        "manualPrivateLinkServiceConnections": [],
        "subnet": {
          "id": "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
        }
      }
    },
    {
      "type": "Microsoft.Resources/deployments",
      "apiVersion": "2017-05-10",
      "name": "[concat('PrivateDns-', variables('privateDnsGuid'))]",
      "dependsOn": [
        "[resourceId('Microsoft.Network/privateEndpoints', concat(parameters('workspaceName'), '-
PrivateEndpoint'))]"
      ],
      "properties": {
          "mode": "Incremental",
          "template": {
              "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
              "contentVersion": "1.0.0.0",
              "resources": [
                  {
                      "type": "Microsoft.Network/privateDnsZones",
                      "apiVersion": "2018-09-01",
                      "name": "privatelink.api.azureml.ms",
                      "location": "global",
                      "tags": {},
                      "properties": {}
                  },
                  {
                      "type": "Microsoft.Network/privateDnsZones/virtualNetworkLinks",
                      "apiVersion": "2018-09-01",
                      "name": "[concat('privatelink.api.azureml.ms', '/',
uniqueString(resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))))]",
                      "location": "global",
                      "dependsOn": [
                          "privatelink.api.azureml.ms"
                      ],
                      "properties": {
                          "virtualNetwork": {
                              "id": "[resourceId('Microsoft.Network/virtualNetworks',
parameters('vnetName'))]"
                          },
                          "registrationEnabled": false
                      }
                  },
                  {
                      "apiVersion": "2017-05-10",
                      "name": "[concat('EndpointDnsRecords-', variables('privateDnsGuid'))]",
                      "type": "Microsoft.Resources/deployments",
                      "dependsOn": [
                          "privatelink.api.azureml.ms"
                      ],
                      "properties": {
                          "mode": "Incremental",
                          "templatelink": {
                              "contentVersion": "1.0.0.0",
                              "uri":
"https://network.hosting.portal.azure.net/network/Content/4.13.392.925/DeploymentTemplates/PrivateDnsForPrivat
eEndpoint.json"
                          },
                          "parameters": {
                              "privateDnsName": {
                                  "value": "privatelink.api.azureml.ms"
                              },
                              "privateEndpointNicResourceId": {
                                  "value": "
```

```
                                "value":
[reference(resourceId('Microsoft.Network/privateEndpoints',concat(parameters('workspaceName'), '-
PrivateEndpoint'))).networkInterfaces[0].id]"
                            },
                            "nicRecordsTemplateUri": {
                                "value":
"https://network.hosting.portal.azure.net/network/Content/4.13.392.925/DeploymentTemplates/PrivateDnsForPrivat
eEndpointNic.json"
                            },
                            "ipConfigRecordsTemplateUri": {
                                "value":
"https://network.hosting.portal.azure.net/network/Content/4.13.392.925/DeploymentTemplates/PrivateDnsForPrivat
eEndpointIpConfig.json"
                            },
                            "uniqueId": {
                                "value": "[variables('privateDnsGuid')]"
                            },
                            "existingRecords": {
                                "value": {}
                            }
                        }
                    }
                }
            ]
        }
      },
      "resourceGroup": "[resourceGroup().name]"
  }
  ]
}
```

Workspace with customer-managed keys and manual approval for Private Link

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "workspaceName": {
      "type": "string",
      "metadata": {
        "description": "Specifies the name of the Azure Machine Learning workspace."
      }
    },
    "location": {
      "type": "string",
      "allowedValues": [
        "eastus",
        "southcentralus",
        "westus2"
      ],
      "metadata": {
        "description": "Specifies the location for all resources."
      }
    },
    "sku":{
      "type": "string",
      "defaultValue": "basic",
      "allowedValues": [
        "basic",
        "enterprise"
      ],
      "metadata": {
        "description": "Specifies the sku, also referred as 'edition' of the Azure Machine Learning
workspace."
      }
    },
    "hbi_workspace":{
      "type": "string",
```

```
      "defaultValue": "false",
      "allowedValues": [
        "false",
        "true"
      ],
      "metadata": {
        "description": "Specifies that the Azure Machine Learning workspace holds highly confidential data."
      }
    },
    "encryption_status":{
      "type": "string",
      "defaultValue": "Disabled",
      "allowedValues": [
        "Enabled",
        "Disabled"
      ],
      "metadata": {
        "description": "Specifies if the Azure Machine Learning workspace should be encrypted with customer
managed key."
      }
    },
    "cmk_keyvault":{
      "type": "string",
      "metadata": {
        "description": "Specifies the customer managed keyVault id."
      }
    },
    "resource_cmk_uri":{
      "type": "string",
      "metadata": {
        "description": "Specifies if the customer managed keyvault key uri."
      }
    },
    "subnetName": {
        "type": "string"
      },
      "vnetName": {
        "type": "string"
      }
  },
  "variables": {
    "storageAccountName": "[concat('sa',uniqueString(resourceGroup().id))]",
    "storageAccountType": "Standard_LRS",
    "keyVaultName": "[concat('kv',uniqueString(resourceGroup().id))]",
    "tenantId": "[subscription().tenantId]",
    "applicationInsightsName": "[concat('ai',uniqueString(resourceGroup().id))]"
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-04-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[variables('storageAccountType')]"
      },
      "kind": "StorageV2",
      "properties": {
        "encryption": {
          "services": {
            "blob": {
              "enabled": true
            },
            "file": {
              "enabled": true
            }
          },
          "keySource": "Microsoft.Storage"
        }
```

```
        },
        "supportsHttpsTrafficOnly": true
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2018-02-14",
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": []
      }
    },
    {
      "type": "Microsoft.Insights/components",
      "apiVersion": "2018-05-01-preview",
      "name": "[variables('applicationInsightsName')]",
      "location": "
[if(or(equals(parameters('location'),'eastus2'),equals(parameters('location'),'westcentralus')),'southcentralu
s',parameters('location'))]",
      "kind": "web",
      "properties": {
        "Application_Type": "web"
      }
    },
    {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2020-01-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
        "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]"
      ],
      "identity": {
        "type": "systemAssigned"
      },
      "sku": {
            "tier": "[parameters('sku')]",
            "name": "[parameters('sku')]"
      },
      "properties": {
        "friendlyName": "[parameters('workspaceName')]",
        "keyVault": "[resourceId('Microsoft.KeyVault/vaults',variables('keyVaultName'))]",
        "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
        "storageAccount": "
[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]",
         "encryption": {
                "status": "[parameters('encryption_status')]",
                "keyVaultProperties": {
                    "keyVaultArmId": "[parameters('cmk_keyvault')]",
                    "keyIdentifier": "[parameters('resource_cmk_uri')]"
                  }
            },
        "hbi_workspace": "[parameters('hbi_workspace')]"
      }
    },
    {
        "type": "Microsoft.Network/virtualNetworks",
        "apiVersion": "2019-09-01",
        "name": "[parameters('vnetName')]",
        "location": "[parameters('location')]",
        "properties": {
```

```
        "properties": {
            "addressSpace": {
                "addressPrefixes": [
                    "10.0.0.0/27"
                ]
            },
            "virtualNetworkPeerings": [],
            "enableDdosProtection": false,
            "enableVmProtection": false
        }
      },
    {
        "type": "Microsoft.Network/virtualNetworks/subnets",
        "apiVersion": "2019-09-01",
        "name": "[concat(parameters('vnetName'), '/', parameters('subnetName'))]",
        "dependsOn": [
            "[resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))]"
        ],
        "properties": {
            "addressPrefix": "10.0.0.0/27",
            "delegations": [],
            "privateEndpointNetworkPolicies": "Disabled",
            "privateLinkServiceNetworkPolicies": "Enabled"
        }
    },
    {
      "apiVersion": "2019-04-01",
      "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
      "type": "Microsoft.Network/privateEndpoints",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.MachineLearningServices/workspaces', parameters('workspaceName'))]",
        "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
      ],
      "properties": {
        "privateLinkServiceConnections": [],
        "manualPrivateLinkServiceConnections": [
          {
            "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
            "properties": {
              "privateLinkServiceId": "[resourceId('Microsoft.MachineLearningServices/workspaces',
parameters('workspaceName'))]",
              "groupIds": [
                "amlworkspace"
              ]
            }
          }
        ],
        "subnet": {
          "id": "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
        }
      }
    }
  ]
}
```

Workspace with Microsoft-managed keys and auto-approval for Private Link

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "workspaceName": {
      "type": "string",
      "metadata": {
        "description": "Specifies the name of the Azure Machine Learning workspace."
```

```
        "description": "Specifies the name of the Azure Machine Learning workspace."
      }
    },
    "location": {
      "type": "string",
      "allowedValues": [
        "eastus",
        "southcentralus",
        "westus2"
      ],
      "metadata": {
        "description": "Specifies the location for all resources."
      }
    },
    "sku": {
      "type": "string",
      "defaultValue": "basic",
      "allowedValues": [
        "basic",
        "enterprise"
      ],
      "metadata": {
        "description": "Specifies the sku, also referred as 'edition' of the Azure Machine Learning
workspace."
      }
    },
    "subnetName": {
      "type": "string"
    },
    "vnetName": {
      "type": "string"
    }
  },
  "variables": {
    "storageAccountName": "[concat('sa',uniqueString(resourceGroup().id))]",
    "storageAccountType": "Standard_LRS",
    "keyVaultName": "[concat('kv',uniqueString(resourceGroup().id))]",
    "tenantId": "[subscription().tenantId]",
    "applicationInsightsName": "[concat('ai',uniqueString(resourceGroup().id))]",
    "privateDnsGuid": "[guid(resourceGroup().id, deployment().name)]"

  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-04-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[variables('storageAccountType')]"
      },
      "kind": "StorageV2",
      "properties": {
        "encryption": {
          "services": {
            "blob": {
              "enabled": true
            },
            "file": {
              "enabled": true
            }
          },
          "keySource": "Microsoft.Storage"
        },
        "supportsHttpsTrafficOnly": true
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2019-03-14"
```

```
      "apiVersion": "2018-02-14",
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": []
      }
    },
    {
      "type": "Microsoft.Insights/components",
      "apiVersion": "2018-05-01-preview",
      "name": "[variables('applicationInsightsName')]",
      "location": "
[if(or(equals(parameters('location'),'eastus2'),equals(parameters('location'),'westcentralus')),'southcentralu
s',parameters('location'))]",
      "kind": "web",
      "properties": {
        "Application_Type": "web"
      }
    },
    {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2019-11-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
        "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]"
      ],
      "identity": {
        "type": "systemAssigned"
      },
      "sku": {
        "tier": "[parameters('sku')]",
        "name": "[parameters('sku')]"
      },
      "properties": {
        "friendlyName": "[parameters('workspaceName')]",
        "keyVault": "[resourceId('Microsoft.KeyVault/vaults',variables('keyVaultName'))]",
        "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
        "storageAccount": "[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]"
      }
    },
    {
      "type": "Microsoft.Network/virtualNetworks",
      "apiVersion": "2019-09-01",
      "name": "[parameters('vnetName')]",
      "location": "[parameters('location')]",
      "properties": {
        "addressSpace": {
          "addressPrefixes": [
            "10.0.0.0/27"
          ]
        },
        "virtualNetworkPeerings": [],
        "enableDdosProtection": false,
        "enableVmProtection": false
      }
    },
    {
      "type": "Microsoft.Network/virtualNetworks/subnets",
      "apiVersion": "2019-09-01",
      "name": "[concat(parameters('vnetName'), '/', parameters('subnetName'))]",
```

```
      "dependsOn": [
        "[resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))]"
      ],
      "properties": {
        "addressPrefix": "10.0.0.0/27",
        "delegations": [],
        "privateEndpointNetworkPolicies": "Disabled",
        "privateLinkServiceNetworkPolicies": "Enabled"
      }
    },
    {
      "apiVersion": "2019-04-01",
      "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
      "type": "Microsoft.Network/privateEndpoints",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.MachineLearningServices/workspaces', parameters('workspaceName'))]",
        "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
      ],
      "properties": {
        "privateLinkServiceConnections": [
          {
            "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
            "properties": {
              "privateLinkServiceId": "[resourceId('Microsoft.MachineLearningServices/workspaces',
parameters('workspaceName'))]",
              "groupIds": [
                "amlworkspace"
              ]
            }
          }
        ],
        "subnet": {
          "id": "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
        }
      }
    },
    {
      "type": "Microsoft.Resources/deployments",
      "apiVersion": "2017-05-10",
      "name": "[concat('PrivateDns-', variables('privateDnsGuid'))]",
      "dependsOn": [
        "[resourceId('Microsoft.Network/privateEndpoints', concat(parameters('workspaceName'), '-
PrivateEndpoint'))]"
      ],
      "properties": {
        "mode": "Incremental",
        "template": {
          "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
          "contentVersion": "1.0.0.0",
          "resources": [
            {
              "type": "Microsoft.Network/privateDnsZones",
              "apiVersion": "2018-09-01",
              "name": "privatelink.api.azureml.ms",
              "location": "global",
              "tags": {},
              "properties": {}
            },
            {
              "type": "Microsoft.Network/privateDnsZones/virtualNetworkLinks",
              "apiVersion": "2018-09-01",
              "name": "[concat('privatelink.api.azureml.ms', '/',
uniqueString(resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))))]",
              "location": "global",
              "dependsOn": [
                "privatelink.api.azureml.ms"
```

```
              ],
              "properties": {
                "virtualNetwork": {
                  "id": "[resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))]"
                },
                "registrationEnabled": false
              }
            },
            {
              "apiVersion": "2017-05-10",
              "name": "[concat('EndpointDnsRecords-', variables('privateDnsGuid'))]",
              "type": "Microsoft.Resources/deployments",
              "dependsOn": [
                "privatelink.api.azureml.ms"
              ],
              "properties": {
                "mode": "Incremental",
                "templatelink": {
                  "contentVersion": "1.0.0.0",
                  "uri":
"https://network.hosting.portal.azure.net/network/Content/4.13.392.925/DeploymentTemplates/PrivateDnsForPrivat
eEndpoint.json"
                },
                "parameters": {
                  "privateDnsName": {
                    "value": "privatelink.api.azureml.ms"
                  },
                  "privateEndpointNicResourceId": {
                    "value": "
[reference(resourceId('Microsoft.Network/privateEndpoints',concat(parameters('workspaceName'), '-
PrivateEndpoint'))).networkInterfaces[0].id]"
                  },
                  "nicRecordsTemplateUri": {
                    "value":
"https://network.hosting.portal.azure.net/network/Content/4.13.392.925/DeploymentTemplates/PrivateDnsForPrivat
eEndpointNic.json"
                  },
                  "ipConfigRecordsTemplateUri": {
                    "value":
"https://network.hosting.portal.azure.net/network/Content/4.13.392.925/DeploymentTemplates/PrivateDnsForPrivat
eEndpointIpConfig.json"
                  },
                  "uniqueId": {
                    "value": "[variables('privateDnsGuid')]"
                  },
                  "existingRecords": {
                    "value": {}
                  }
                }
              }
            }
          ]
        }
      },
      "resourceGroup": "[resourceGroup().name]"
    }
  ]
}
```

Workspace with Microsoft-managed keys and manual approval for Private Link

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "workspaceName": {
      "type": "string",
```

```
      "metadata": {
        "description": "Specifies the name of the Azure Machine Learning workspace."
      }
    },
    "location": {
      "type": "string",
      "allowedValues": [
        "eastus",
        "southcentralus",
        "westus2"
      ],
      "metadata": {
        "description": "Specifies the location for all resources."
      }
    },
    "sku": {
      "type": "string",
      "defaultValue": "basic",
      "allowedValues": [
        "basic",
        "enterprise"
      ],
      "metadata": {
        "description": "Specifies the sku, also referred as 'edition' of the Azure Machine Learning
workspace."
      }
    },
    "subnetName": {
      "type": "string"
    },
    "vnetName": {
      "type": "string"
    }
  },
  "variables": {
    "storageAccountName": "[concat('sa',uniqueString(resourceGroup().id))]",
    "storageAccountType": "Standard_LRS",
    "keyVaultName": "[concat('kv',uniqueString(resourceGroup().id))]",
    "tenantId": "[subscription().tenantId]",
    "applicationInsightsName": "[concat('ai',uniqueString(resourceGroup().id))]"
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-04-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[variables('storageAccountType')]"
      },
      "kind": "StorageV2",
      "properties": {
        "encryption": {
          "services": {
            "blob": {
              "enabled": true
            },
            "file": {
              "enabled": true
            }
          },
          "keySource": "Microsoft.Storage"
        },
        "supportsHttpsTrafficOnly": true
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2018-02-14",
```

```
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": []
      }
    },
    {
      "type": "Microsoft.Insights/components",
      "apiVersion": "2018-05-01-preview",
      "name": "[variables('applicationInsightsName')]",
      "location": "
[if(or(equals(parameters('location'),'eastus2'),equals(parameters('location'),'westcentralus')),'southcentralu
s',parameters('location'))]",
      "kind": "web",
      "properties": {
        "Application_Type": "web"
      }
    },
    {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2019-11-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
        "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]"
      ],
      "identity": {
        "type": "systemAssigned"
      },
      "sku": {
        "tier": "[parameters('sku')]",
        "name": "[parameters('sku')]"
      },
      "properties": {
        "friendlyName": "[parameters('workspaceName')]",
        "keyVault": "[resourceId('Microsoft.KeyVault/vaults',variables('keyVaultName'))]",
        "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
        "storageAccount": "[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]"
      }
    },
    {
      "type": "Microsoft.Network/virtualNetworks",
      "apiVersion": "2019-09-01",
      "name": "[parameters('vnetName')]",
      "location": "[parameters('location')]",
      "properties": {
        "addressSpace": {
          "addressPrefixes": [
            "10.0.0.0/27"
          ]
        },
        "virtualNetworkPeerings": [],
        "enableDdosProtection": false,
        "enableVmProtection": false
      }
    },
    {
      "type": "Microsoft.Network/virtualNetworks/subnets",
      "apiVersion": "2019-09-01",
      "name": "[concat(parameters('vnetName'), '/', parameters('subnetName'))]",
      "dependsOn": [
```

```
        "[resourceId('Microsoft.Network/virtualNetworks', parameters('vnetName'))]"
      ],
      "properties": {
        "addressPrefix": "10.0.0.0/27",
        "delegations": [],
        "privateEndpointNetworkPolicies": "Disabled",
        "privateLinkServiceNetworkPolicies": "Enabled"
      }
    },
    {
      "apiVersion": "2019-04-01",
      "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
      "type": "Microsoft.Network/privateEndpoints",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.MachineLearningServices/workspaces', parameters('workspaceName'))]",
        "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
      ],
      "properties": {
        "privateLinkServiceConnections": [],
        "manualPrivateLinkServiceConnections": [
          {
            "name": "[concat(parameters('workspaceName'), '-PrivateEndpoint')]",
            "properties": {
              "privateLinkServiceId": "[resourceId('Microsoft.MachineLearningServices/workspaces',
parameters('workspaceName'))]",
              "groupIds": [
                "amlworkspace"
              ]
            }
          }
        ],
        "subnet": {
          "id": "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('vnetName'),
parameters('subnetName') )]"
        }
      }
    }
  ]
}
```

Next steps

For more information on securing your Azure Machine Learning workspace, see the Enterprise security article.

Use TLS to secure a web service through Azure Machine Learning

3/25/2020 • 7 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article shows you how to secure a web service that's deployed through Azure Machine Learning.

You use HTTPS to restrict access to web services and secure the data that clients submit. HTTPS helps secure communications between a client and a web service by encrypting communications between the two. Encryption uses Transport Layer Security (TLS). TLS is sometimes still referred to as *Secure Sockets Layer* (SSL), which was the predecessor of TLS.

> **TIP**
> The Azure Machine Learning SDK uses the term "SSL" for properties that are related to secure communications. This doesn't mean that your web service doesn't use *TLS*. SSL is just a more commonly recognized term.
>
> Specifically, web services deployed through Azure Machine Learning only support TLS version 1.2.

TLS and SSL both rely on *digital certificates*, which help with encryption and identity verification. For more information on how digital certificates work, see the Wikipedia topic Public key infrastructure.

> **WARNING**
> If you don't use HTTPS for your web service, data that's sent to and from the service might be visible to others on the internet.
>
> HTTPS also enables the client to verify the authenticity of the server that it's connecting to. This feature protects clients against man-in-the-middle attacks.

This is the general process to secure a web service:

1. Get a domain name.
2. Get a digital certificate.
3. Deploy or update the web service with TLS enabled.
4. Update your DNS to point to the web service.

> **IMPORTANT**
> If you're deploying to Azure Kubernetes Service (AKS), you can purchase your own certificate or use a certificate that's provided by Microsoft. If you use a certificate from Microsoft, you don't need to get a domain name or TLS/SSL certificate. For more information, see the Enable TLS and deploy section of this article.

There are slight differences when you secure s across deployment targets.

Get a domain name

If you don't already own a domain name, purchase one from a *domain name registrar*. The process and price differ among registrars. The registrar provides tools to manage the domain name. You use these tools to map a fully qualified domain name (FQDN) (such as www.contoso.com) to the IP address that hosts your web service.

Get a TLS/SSL certificate

There are many ways to get an TLS/SSL certificate (digital certificate). The most common is to purchase one from a *certificate authority* (CA). Regardless of where you get the certificate, you need the following files:

- A **certificate**. The certificate must contain the full certificate chain, and it must be "PEM-encoded."
- A **key**. The key must also be PEM-encoded.

When you request a certificate, you must provide the FQDN of the address that you plan to use for the web service (for example, www.contoso.com). The address that's stamped into the certificate and the address that the clients use are compared to verify the identity of the web service. If those addresses don't match, the client gets an error message.

> **TIP**
> If the certificate authority can't provide the certificate and key as PEM-encoded files, you can use a utility such as OpenSSL to change the format.

> **WARNING**
> Use *self-signed* certificates only for development. Don't use them in production environments. Self-signed certificates can cause problems in your client applications. For more information, see the documentation for the network libraries that your client application uses.

Enable TLS and deploy

To deploy (or redeploy) the service with TLS enabled, set the *ssl_enabled* parameter to "True" wherever it's applicable. Set the *ssl_certificate* parameter to the value of the *certificate* file. Set the *ssl_key* to the value of the *key* file.

Deploy on AKS and field-programmable gate array (FPGA)

> **NOTE**
> The information in this section also applies when you deploy a secure web service for the designer. If you aren't familiar with using the Python SDK, see What is the Azure Machine Learning SDK for Python?.

When you deploy to AKS, you can create a new AKS cluster or attach an existing one. For more information on creating or attaching a cluster, see Deploy a model to an Azure Kubernetes Service cluster.

- If you create a new cluster, you use **AksCompute.provisioning_configuration()**.
- If you attach an existing cluster, you use **AksCompute.attach_configuration()**. Both return a configuration object that has an **enable_ssl** method.

The **enable_ssl** method can use a certificate that's provided by Microsoft or a certificate that you purchase.

- When you use a certificate from Microsoft, you must use the *leaf_domain_label* parameter. This parameter generates the DNS name for the service. For example, a value of "contoso" creates a domain name of "contoso<six-random-characters>.<azureregion>.cloudapp.azure.com", where <azureregion> is the region that contains the service. Optionally, you can use the *overwrite_existing_domain* parameter to

overwrite the existing *leaf_domain_label*.

To deploy (or redeploy) the service with TLS enabled, set the *ssl_enabled* parameter to "True" wherever it's applicable. Set the *ssl_certificate* parameter to the value of the *certificate* file. Set the *ssl_key* to the value of the *key* file.

> **IMPORTANT**
> When you use a certificate from Microsoft, you don't need to purchase your own certificate or domain name.

The following example demonstrates how to create a configuration that enables an TLS/SSL certificate from Microsoft:

```
from azureml.core.compute import AksCompute
# Config used to create a new AKS cluster and enable TLS
provisioning_config = AksCompute.provisioning_configuration()
# Leaf domain label generates a name using the formula
#  "<leaf-domain-label>######.<azure-region>.cloudapp.azure.net"
#  where "######" is a random series of characters
provisioning_config.enable_ssl(leaf_domain_label = "contoso")


# Config used to attach an existing AKS cluster to your workspace and enable TLS
attach_config = AksCompute.attach_configuration(resource_group = resource_group,
                                     cluster_name = cluster_name)
# Leaf domain label generates a name using the formula
#  "<leaf-domain-label>######.<azure-region>.cloudapp.azure.net"
#  where "######" is a random series of characters
attach_config.enable_ssl(leaf_domain_label = "contoso")
```

- When you use *a certificate that you purchased*, you use the *ssl_cert_pem_file*, *ssl_key_pem_file*, and *ssl_cname* parameters. The following example demonstrates how to use *.pem* files to create a configuration that uses a TLS/SSL certificate that you purchased:

```
from azureml.core.compute import AksCompute
# Config used to create a new AKS cluster and enable TLS
provisioning_config = AksCompute.provisioning_configuration()
provisioning_config.enable_ssl(ssl_cert_pem_file="cert.pem",
                                    ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")
# Config used to attach an existing AKS cluster to your workspace and enable SSL
attach_config = AksCompute.attach_configuration(resource_group = resource_group,
                                     cluster_name = cluster_name)
attach_config.enable_ssl(ssl_cert_pem_file="cert.pem",
                                    ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")
```

For more information about *enable_ssl*, see AksProvisioningConfiguration.enable_ssl() and AksAttachConfiguration.enable_ssl().

Deploy on Azure Container Instances

When you deploy to Azure Container Instances, you provide values for TLS-related parameters, as the following code snippet shows:

```
from azureml.core.webservice import AciWebservice

aci_config = AciWebservice.deploy_configuration(
    ssl_enabled=True, ssl_cert_pem_file="cert.pem", ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")
```

For more information, see AciWebservice.deploy_configuration().

Update your DNS

Next, you must update your DNS to point to the web service.

- **For Container Instances:**

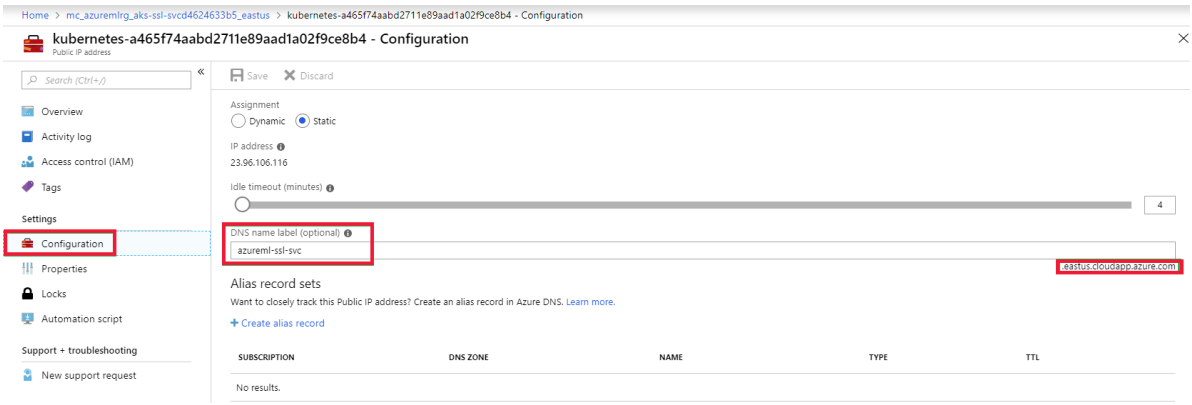
  Use the tools from your domain name registrar to update the DNS record for your domain name. The record must point to the IP address of the service.

  There can be a delay of minutes or hours before clients can resolve the domain name, depending on the registrar and the "time to live" (TTL) that's configured for the domain name.

- **For AKS:**

  > **WARNING**
  > If you used *leaf_domain_label* to create the service by using a certificate from Microsoft, don't manually update the DNS value for the cluster. The value should be set automatically.

  Update the DNS of the Public IP Address of the AKS cluster on the **Configuration** tab under **Settings** in the left pane. (See the following image.) The Public IP Address is a resource type that's created under the resource group that contains the AKS agent nodes and other networking resources.



Update the TLS/SSL certificate

TLS/SSL certificates expire and must be renewed. Typically this happens every year. Use the information in the following sections to update and renew your certificate for models deployed to Azure Kubernetes Service:

Update a Microsoft generated certificate

If the certificate was originally generated by Microsoft (when using the *leaf_domain_label* to create the service), use one of the following examples to update the certificate:

Use the SDK

```
from azureml.core.compute import AksCompute
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute.aks import SslConfiguration

# Get the existing cluster
aks_target = AksCompute(ws, clustername)

# Update the existing certificate by referencing the leaf domain label
ssl_configuration = SslConfiguration(leaf_domain_label="myaks", overwrite_existing_domain=True)
update_config = AksUpdateConfiguration(ssl_configuration)
aks_target.update(update_config)
```

Use the CLI

```
az ml computetarget update aks -g "myresourcegroup" -w "myresourceworkspace" -n "myaks" --ssl-leaf-domain-
label "myaks" --ssl-overwrite-domain True
```

For more information, see the following reference docs:

- SslConfiguration
- AksUpdateConfiguration

Update custom certificate

If the certificate was originally generated by a certificate authority, use the following steps:

1. Use the documentation provided by the certificate authority to renew the certificate. This process creates new certificate files.
2. Use either the SDK or CLI to update the service with the new certificate:

   #### Use the SDK

   ```
   from azureml.core.compute import AksCompute
   from azureml.core.compute.aks import AksUpdateConfiguration
   from azureml.core.compute.aks import SslConfiguration

   # Read the certificate file
   def get_content(file_name):
       with open(file_name, 'r') as f:
           return f.read()

   # Get the existing cluster
   aks_target = AksCompute(ws, clustername)

   # Update cluster with custom certificate
   ssl_configuration = SslConfiguration(cname="myaks", cert=get_content('cert.pem'),
   key=get_content('key.pem'))
   update_config = AksUpdateConfiguration(ssl_configuration)
   aks_target.update(update_config)
   ```

   #### Use the CLI

   ```
   az ml computetarget update aks -g "myresourcegroup" -w "myresourceworkspace" -n "myaks" --ssl-cname
   "myaks"--ssl-cert-file "cert.pem" --ssl-key-file "key.pem"
   ```

For more information, see the following reference docs:

- SslConfiguration
- AksUpdateConfiguration

Disable TLS

To disable TLS for a model deployed to Azure Kubernetes Service, create an `SslConfiguration` with `status="Disabled"`, then perform an update:

```
from azureml.core.compute import AksCompute
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute.aks import SslConfiguration

# Get the existing cluster
aks_target = AksCompute(ws, clustername)

# Disable TLS
ssl_configuration = SslConfiguration(status="Disabled")
update_config = AksUpdateConfiguration(ssl_configuration)
aks_target.update(update_config)
```

Next steps

Learn how to:

- Consume a machine learning model deployed as a web service
- Securely run experiments and inference inside an Azure virtual network

Use Azure AD identity with your machine learning web service in Azure Kubernetes Service

2/10/2020 • 4 minutes to read • Edit Online

In this how-to, you learn how to assign an Azure Active Directory (AAD) identity to your deployed machine learning model in Azure Kubernetes Service. The AAD Pod Identity project allows applications to access cloud resources securely with AAD by using a Managed Identity and Kubernetes primitives. This allows your web service to securely access your Azure resources without having to embed credentials or manage tokens directly inside your `score.py` script. This article explains the steps to create and install an Azure Identity in your Azure Kubernetes Service cluster and assign the identity to your deployed web service.

Prerequisites

- The Azure CLI extension for the Machine Learning service, the Azure Machine Learning SDK for Python, or the Azure Machine Learning Visual Studio Code extension.
- Access to your AKS cluster using the `kubectl` command. For more information, see Connect to the cluster
- An Azure Machine Learning web service deployed to your AKS cluster.

Create and install an Azure Identity in your AKS cluster

1. To determine if your AKS cluster is RBAC enabled, use the following command:

   ```
   az aks show --name <AKS cluster name> --resource-group <resource group name> --subscription
   <subscription id> --query enableRbac
   ```

   This command returns a value of `true` if RBAC is enabled. This value determines the command to use in the next step.

2. To install AAD Pod Identity in your AKS cluster, use one of the following commands:

   - If your AKS cluster has **RBAC enabled** use the following command:

     ```
     kubectl apply -f https://raw.githubusercontent.com/Azure/aad-pod-
     identity/master/deploy/infra/deployment-rbac.yaml
     ```

   - If your AKS cluster **does not have RBAC enabled**, use the following command:

     ```
     kubectl apply -f https://raw.githubusercontent.com/Azure/aad-pod-
     identity/master/deploy/infra/deployment.yaml
     ```

     The output of the command is similar to the following text:

```
customresourcedefinition.apiextensions.k8s.io/azureassignedidentities.aadpodidentity.k8s.io
created
customresourcedefinition.apiextensions.k8s.io/azureidentitybindings.aadpodidentity.k8s.io created
customresourcedefinition.apiextensions.k8s.io/azureidentities.aadpodidentity.k8s.io created
customresourcedefinition.apiextensions.k8s.io/azurepodidentityexceptions.aadpodidentity.k8s.io
created
daemonset.apps/nmi created
deployment.apps/mic created
```

3. Create an Azure Identity following the steps shown in AAD Pod Identity project page.
4. Install the Azure Identity following the steps shown in AAD Pod Identity project page.
5. Install the Azure Identity Binding following the steps shown in AAD Pod Identity project page.
6. If the Azure Identity created in the previous step is not in the same resource group as your AKS cluster, follow Set Permissions for MIC following the steps shown in AAD Pod Identity project page.

Assign Azure Identity to machine learning web service

The following steps use the Azure Identity created in the previous section, and assign it to your AKS web service through a **selector label**.

First, identify the name and namespace of your deployment in your AKS cluster that you want to assign the Azure Identity. You can get this information by running the following command. The namespaces should be your Azure Machine Learning workspace name and your deployment name should be your endpoint name as shown in the portal.

```
kubectl get deployment --selector=isazuremlapp=true --all-namespaces --show-labels
```

Add the Azure Identity selector label to your deployment by editing the deployment spec. The selector value should be the one that you defined in step 5 of Install the Azure Identity Binding.

```
apiVersion: "aadpodidentity.k8s.io/v1"
kind: AzureIdentityBinding
metadata:
  name: demo1-azure-identity-binding
spec:
  AzureIdentity: <a-idname>
  Selector: <label value to match>
```

Edit the deployment to add the Azure Identity selector label. Go to the following section under `/spec/template/metadata/labels`. You should see values such as `isazuremlapp: “true”`. Add the aad-pod-identity label like shown below.

```
    kubectl edit deployment/<name of deployment> -n azureml-<name of workspace>
```

```
spec:
  template:
    metadata:
      labels:
      - aadpodidbinding: "<value of Selector in AzureIdentityBinding>"
      ...
```

To verify that the label was correctly added, run the following command.

```
kubectl get deployment <name of deployment> -n azureml-<name of workspace> --show-labels
```

To see all pod statuses, run the following command.

```
kubectl get pods -n azureml-<name of workspace>
```

Once the pods are up and running, the web services for this deployment will now be able to access Azure resources through your Azure Identity without having to embed the credentials in your code.

Assign the appropriate roles to your Azure Identity

Assign your Azure Managed Identity with appropriate roles to access other Azure resources. Ensure that the roles you are assigning have the correct **Data Actions**. For example, the Storage Blob Data Reader Role will have read permissions to your Storage Blob while the generic Reader Role might not.

Use Azure Identity with your machine learning web service

Deploy a model to your AKS cluster. The `score.py` script can contain operations pointing to the Azure resources that your Azure Identity has access to. Ensure that you have installed your required client library dependencies for the resource that you are trying to access to. Below are a couple examples of how you can use your Azure Identity to access different Azure resources from your service.

Access Key Vault from your web service

If you have given your Azure Identity read access to a secret inside a **Key Vault**, your `score.py` can access it using the following code.

```
from azure.identity import DefaultAzureCredential
from azure.keyvault.secrets import SecretClient

my_vault_name = "yourkeyvaultname"
my_vault_url = "https://{}.vault.azure.net/".format(my_vault_name)
my_secret_name = "sample-secret"

# This will use your Azure Managed Identity
credential = DefaultAzureCredential()
secret_client = SecretClient(
    vault_url=my_vault_url,
    credential=credential)
secret = secret_client.get_secret(my_secret_name)
```

Access Blob from your web service

If you have given your Azure Identity read access to data inside a **Storage Blob**, your `score.py` can access it using the following code.

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

my_storage_account_name = "yourstorageaccountname"
my_storage_account_url = "https://{}.blob.core.windows.net/".format(my_storage_account_name)

# This will use your Azure Managed Identity
credential = DefaultAzureCredential()
blob_service_client = BlobServiceClient(
    account_url=my_storage_account_url,
    credential=credential
)
blob_client = blob_service_client.get_blob_client(container="some-container", blob="some_text.txt")
blob_data = blob_client.download_blob()
blob_data.readall()
```

Next steps

- For more information on how to use the Python Azure Identity client library, see the repository on GitHub.
- For a detailed guide on deploying models to Azure Kubernetes Service clusters, see the how-to.

Regenerate storage account access keys

3/26/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to change the access keys for Azure Storage accounts used by Azure Machine Learning. Azure Machine Learning can use storage accounts to store data or trained models.

For security purposes, you may need to change the access keys for an Azure Storage account. When you regenerate the access key, Azure Machine Learning must be updated to use the new key. Azure Machine Learning may be using the storage account for both model storage and as a datastore.

Prerequisites

- An Azure Machine Learning workspace. For more information, see the Create a workspace article.
- The Azure Machine Learning SDK.
- The Azure Machine Learning CLI extension.

> **NOTE**
> The code snippets in this document were tested with version 1.0.83 of the Python SDK.

What needs to be updated

Storage accounts can be used by the Azure Machine Learning workspace (storing logs, models, snapshots, etc.) and as a datastore. The process to update the workspace is a single Azure CLI command, and can be ran after updating the storage key. The process of updating datastores is more involved, and requires discovering what datastores are currently using the storage account and then re-registering them.

> **IMPORTANT**
> Update the workspace using the Azure CLI, and the datastores using Python, at the same time. Updating only one or the other is not sufficient, and may cause errors until both are updated.

To discover the storage accounts that are used by your datastores, use the following code:

```
import azureml.core
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()

default_ds = ws.get_default_datastore()
print("Default datstore: " + default_ds.name + ", storage account name: " +
      default_ds.account_name + ", container name: " + default_ds.container_name)

datastores = ws.datastores
for name, ds in datastores.items():
    if ds.datastore_type == "AzureBlob":
        print("Blob store - datastore name: " + name + ", storage account name: " +
              ds.account_name + ", container name: " + ds.container_name)
    if ds.datastore_type == "AzureFile":
        print("File share - datastore name: " + name + ", storage account name: " +
              ds.account_name + ", container name: " + ds.container_name)
```

This code looks for any registered datastores that use Azure Storage and lists the following information:

- Datastore name: The name of the datastore that the storage account is registered under.
- Storage account name: The name of the Azure Storage account.
- Container: The container in the storage account that is used by this registration.

It also indicates whether the datastore is for an Azure Blob or an Azure File share, as there are different methods to re-register each type of datastore.

If an entry exists for the storage account that you plan on regenerating access keys for, save the datastore name, storage account name, and container name.

Update the access key

To update Azure Machine Learning to use the new key, use the following steps:

> **IMPORTANT**
> Perform all steps, updating both the workspace using the CLI, and datastores using Python. Updating only one or the other may cause errors until both are updated.

1. Regenerate the key. For information on regenerating an access key, see Manage storage account access keys. Save the new key.
2. To update the workspace to use the new key, use the following steps:
    a. To sign in to the Azure subscription that contains your workspace by using the following Azure CLI command:

    ```
    az login
    ```

> **TIP**
>
> After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.
>
> To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

b. To update the workspace to use the new key, use the following command. Replace `myworkspace` with your Azure Machine Learning workspace name, and replace `myresourcegroup` with the name of the Azure resource group that contains the workspace.

```
az ml workspace sync-keys -w myworkspace -g myresourcegroup
```

> **TIP**
>
> If you get an error message stating that the ml extension isn't installed, use the following command to install it:
>
> ```
> az extension add -n azure-cli-ml
> ```

This command automatically syncs the new keys for the Azure storage account used by the workspace.

3. To re-register datastore(s) that use the storage account, use the values from the What needs to be updated section and the key from step 1 with the following code:

```
# Re-register the blob container
ds_blob = Datastore.register_azure_blob_container(workspace=ws,
                                          datastore_name='your datastore name',
                                          container_name='your container name',
                                          account_name='your storage account name',
                                          account_key='new storage account key',
                                          overwrite=True)
# Re-register file shares
ds_file = Datastore.register_azure_file_share(workspace=ws,
                                      datastore_name='your datastore name',
                                      file_share_name='your container name',
                                      account_name='your storage account name',
                                      account_key='new storage account key',
                                      overwrite=True)
```

Since `overwrite=True` is specified, this code overwrites the existing registration and updates it to use the new key.

Next steps

For more information on registering datastores, see the `Datastore` class reference.

Set up authentication for Azure Machine Learning resources and workflows

2/25/2020 • 9 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to set up and configure authentication for various resources and workflows in Azure Machine Learning. There are multiple ways to authenticate to the service, ranging from simple UI-based auth for development or testing purposes to full Azure Active Directory service principal authentication. This article also explains the differences in how web-service authentication works, as well as how to authenticate to the Azure Machine Learning REST API.

This how-to shows you how to do the following tasks:

- Use interactive UI authentication for testing/development
- Set up service principal authentication
- Authenticating to your workspace
- Get OAuth2.0 bearer-type tokens for Azure Machine Learning REST API
- Understand web-service authentication

See the concept article for a general overview of security and authentication within Azure Machine Learning.

Prerequisites

- Create an Azure Machine Learning workspace.
- Configure your development environment to install the Azure Machine Learning SDK, or use a Azure Machine Learning Notebook VM with the SDK already installed.

Interactive authentication

Most examples in the documentation for this service use interactive authentication in Jupyter notebooks as a simple method for testing and demonstration. This is a lightweight way to test what you're building. There are two function calls that will automatically prompt you with a UI-based authentication flow.

Calling the `from_config()` function will issue the prompt.

```
from azureml.core import Workspace
ws = Workspace.from_config()
```

The `from_config()` function looks for a JSON file containing your workspace connection information. You can also specify the connection details explicitly by using the `Workspace` constructor, which will also prompt for interactive authentication. Both calls are equivalent.

```
ws = Workspace(subscription_id="your-sub-id",
               resource_group="your-resource-group-id",
               workspace_name="your-workspace-name"
              )
```

If you have access to multiple tenants, you may need to import the class and explicitly define what tenant you are

targeting. Calling the constructor for `InteractiveLoginAuthentication` will also prompt you to login similar to the calls above.

```
from azureml.core.authentication import InteractiveLoginAuthentication
interactive_auth = InteractiveLoginAuthentication(tenant_id="your-tenant-id")
```

While useful for testing and learning, interactive authentication will not help you with building automated or headless workflows. Setting up service principal authentication is the best approach for automated processes that use the SDK.

Set up service principal authentication

This process is necessary for enabling authentication that is decoupled from a specific user login, which allows you to authenticate to the Azure Machine Learning Python SDK in automated workflows. Service principal authentication will also allow you to authenticate to the REST API.

To set up service principal authentication, you first create an app registration in Azure Active Directory, and then grant your app role-based access to your ML workspace. The easiest way to complete this setup is through the Azure Cloud Shell in the Azure portal. After you login to the portal, click the `>_` icon in the top right of the page near your name to open the shell.

If you haven't used the cloud shell before in your Azure account, you will need to create a storage account resource for storing any files that are written. In general this storage account will incur a negligible monthly cost. Additionally, install the machine learning extension if you haven't used it previously with the following command.

```
az extension add -n azure-cli-ml
```

> **NOTE**
> You must be an admin on the subscription to perform the following steps.

Next, run the following command to create the service principal. Give it a name, in this case **ml-auth**.

```
az ad sp create-for-rbac --sdk-auth --name ml-auth
```

The output will be a JSON similar to the following. Take note of the `clientId`, `clientSecret`, and `tenantId` fields, as you will need them for other steps in this article.

```
{
    "clientId": "your-client-id",
    "clientSecret": "your-client-secret",
    "subscriptionId": "your-sub-id",
    "tenantId": "your-tenant-id",
    "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
    "resourceManagerEndpointUrl": "https://management.azure.com",
    "activeDirectoryGraphResourceId": "https://graph.windows.net",
    "sqlManagementEndpointUrl": "https://management.core.windows.net:5555",
    "galleryEndpointUrl": "https://gallery.azure.com/",
    "managementEndpointUrl": "https://management.core.windows.net"
}
```

Next, run the following command to get the details on the service principal you just created, using the `clientId` value from above as the input to the `--id` parameter.

```
az ad sp show --id your-client-id
```

The following is a simplified example of the JSON output from the command. Take note of the `objectId` field, as you will need its value for the next step.

```
{
    "accountEnabled": "True",
    "addIns": [],
    "appDisplayName": "ml-auth",
    ...
    ...
    ...
    "objectId": "your-sp-object-id",
    "objectType": "ServicePrincipal"
}
```

Next, use the following command to assign your service principal access to your machine learning workspace. You will need your workspace name, and its resource group name for the `-w` and `-g` parameters, respectively. For the `--user` parameter, use the `objectId` value from the previous step. The `--role` parameter allows you to set the access role for the service principal, and in general you will use either **owner** or **contributor**. Both have write access to existing resources like compute clusters and datastores, but only **owner** can provision these resources.

```
az ml workspace share -w your-workspace-name -g your-resource-group-name --user your-sp-object-id --role owner
```

This call does not produce any output, but you now have service principal authentication set up for your workspace.

Authenticate to your workspace

Now that you have service principal auth enabled, you can authenticate to your workspace in the SDK without physically logging in as a user. Use the `ServicePrincipalAuthentication` class constructor, and use the values you got from the previous steps as the parameters. The `tenant_id` parameter maps to `tenantId` from above, `service_principal_id` maps to `clientId`, and `service_principal_password` maps to `clientSecret`.

```
from azureml.core.authentication import ServicePrincipalAuthentication

sp = ServicePrincipalAuthentication(tenant_id="your-tenant-id", # tenantID
                                    service_principal_id="your-client-id", # clientId
                                    service_principal_password="your-client-secret") # clientSecret
```

The `sp` variable now holds an authentication object that you use directly in the SDK. In general, it is a good idea to store the ids/secrets used above in environment variables as shown in the following code.

```
import os

sp = ServicePrincipalAuthentication(tenant_id=os.environ['AML_TENANT_ID'],
                                    service_principal_id=os.environ['AML_PRINCIPAL_ID'],
                                    service_principal_password=os.environ['AML_PRINCIPAL_PASS'])
```

For automated workflows that run in Python and use the SDK primarily, you can use this object as-is in most cases for your authentication. The following code authenticates to your workspace using the auth object you just created.

```
from azureml.core import Workspace

ws = Workspace.get(name="ml-example",
                   auth=sp,
                   subscription_id="your-sub-id")
ws.get_details()
```

Azure Machine Learning REST API auth

The service principal created in the steps above can also be used to authenticate to the Azure Machine Learning REST API. You use the Azure Active Directory client credentials grant flow, which allow service-to-service calls for headless authentication in automated workflows. The examples are implemented with the ADAL library in both Python and Node.js, but you can also use any open-source library that supports OpenID Connect 1.0.

> **NOTE**
> MSAL.js is a newer library than ADAL, but you cannot do service-to-service authentication using client credentials with MSAL.js, since it is primarily a client-side library intended for interactive/UI authentication tied to a specific user. We recommend using ADAL as shown below to build automated workflows with the REST API.

Node.js

Use the following steps to generate an auth token using Node.js. In your environment, run `npm install adal-node`. Then, use your `tenantId`, `clientId`, and `clientSecret` from the service principal you created in the steps above as values for the matching variables in the following script.

```
const adal = require('adal-node').AuthenticationContext;

const authorityHostUrl = 'https://login.microsoftonline.com/';
const tenantId = 'your-tenant-id';
const authorityUrl = authorityHostUrl + tenantId;
const clientId = 'your-client-id';
const clientSecret = 'your-client-secret';
const resource = 'https://management.azure.com/';

const context = new adal(authorityUrl);

context.acquireTokenWithClientCredentials(
  resource,
  clientId,
  clientSecret,
  (err, tokenResponse) => {
    if (err) {
      console.log(`Token generation failed due to ${err}`);
    } else {
      console.dir(tokenResponse, { depth: null, colors: true });
    }
  }
);
```

The variable `tokenResponse` is an object that includes the token and associated metadata such as expiration time. Tokens are valid for 1 hour, and can be refreshed by running the same call again to retrieve a new token. The following is a sample response.

```
{
    tokenType: 'Bearer',
    expiresIn: 3599,
    expiresOn: 2019-12-17T19:15:56.326Z,
    resource: 'https://management.azure.com/',
    accessToken: "random-oauth-token",
    isMRRT: true,
    _clientId: 'your-client-id',
    _authority: 'https://login.microsoftonline.com/your-tenant-id'
}
```

Use the `accessToken` property to fetch the auth token. See the REST API documentation for examples on how to use the token to make API calls.

Python

Use the following steps to generate an auth token using Python. In your environment, run `pip install adal`. Then, use your `tenantId`, `clientId`, and `clientSecret` from the service principal you created in the steps above as values for the appropriate variables in the following script.

```
from adal import AuthenticationContext

client_id = "your-client-id"
client_secret = "your-client-secret"
resource_url = "https://login.microsoftonline.com"
tenant_id = "your-tenant-id"
authority = "{}/{}".format(resource_url, tenant_id)

auth_context = AuthenticationContext(authority)
token_response = auth_context.acquire_token_with_client_credentials("https://management.azure.com/",
client_id, client_secret)
print(token_response)
```

The variable `token_response` is a dictionary that includes the token and associated metadata such as expiration time. Tokens are valid for 1 hour, and can be refreshed by running the same call again to retrieve a new token. The following is a sample response.

```
{
    'tokenType': 'Bearer',
    'expiresIn': 3599,
    'expiresOn': '2019-12-17 19:47:15.150205',
    'resource': 'https://management.azure.com/',
    'accessToken': 'random-oauth-token',
    'isMRRT': True,
    '_clientId': 'your-client-id',
    '_authority': 'https://login.microsoftonline.com/your-tenant-id'
}
```

Use `token_response["accessToken"]` to fetch the auth token. See the REST API documentation for examples on how to use the token to make API calls.

Web-service authentication

Web-services in Azure Machine Learning use a different authentication pattern than what is described above. The easiest way to authenticate to deployed web-services is to use **key-based authentication**, which generates static bearer-type authentication keys that do not need to be refreshed. If you only need to authenticate to a deployed web-service, you do not need to set up service principle authentication as shown above.

Web-services deployed on Azure Kubernetes Service have key-based auth *enabled* by default. Azure Container

Instances deployed services have key-based auth *disabled* by default, but you can enable it by setting `auth_enabled=True` when creating the ACI web-service. The following is an example of creating an ACI deployment configuration with key-based auth enabled.

```
from azureml.core.webservice import AciWebservice

aci_config = AciWebservice.deploy_configuration(cpu_cores = 1,
                                                memory_gb = 1,
                                                auth_enabled=True)
```

Then you can use the custom ACI configuration in deployment using the `Model` class.

```
from azureml.core.model import Model, InferenceConfig


inference_config = InferenceConfig(entry_script="score.py",
                                   environment=myenv)
aci_service = Model.deploy(workspace=ws,
                       name="aci_service_sample",
                       models=[model],
                       inference_config=inference_config,
                       deployment_config=aci_config)
aci_service.wait_for_deployment(True)
```

To fetch the auth keys, use `aci_service.get_keys()`. To regenerate a key, use the `regen_key()` function and pass either **Primary** or **Secondary**.

```
aci_service.regen_key("Primary")
# or
aci_service.regen_key("Secondary")
```

Web-services also support token-based authentication, but only for Azure Kubernetes Service deployments. See the how-to on consuming web-services for additional information on authenticating.

Token-based web-service authentication

When you enable token authentication for a web service, users must present an Azure Machine Learning JSON Web Token to the web service to access it. The token expires after a specified time-frame and needs to be refreshed to continue making calls.

- Token authentication is **disabled by default** when you deploy to Azure Kubernetes Service.
- Token authentication **isn't supported** when you deploy to Azure Container Instances.

To control token authentication, use the `token_auth_enabled` parameter when you create or update a deployment.

If token authentication is enabled, you can use the `get_token` method to retrieve a JSON Web Token (JWT) and that token's expiration time:

```
token, refresh_by = service.get_token()
print(token)
```

IMPORTANT

You'll need to request a new token after the token's `refresh_by` time. If you need to refresh tokens outside of the Python SDK, one option is to use the REST API with service-principal authentication to periodically make the `service.get_token()` call, as discussed previously.

We strongly recommend that you create your Azure Machine Learning workspace in the same region as your Azure Kubernetes Service cluster.

To authenticate with a token, the web service will make a call to the region in which your Azure Machine Learning workspace is created. If your workspace's region is unavailable, you won't be able to fetch a token for your web service, even if your cluster is in a different region from your workspace. The result is that Azure AD Authentication is unavailable until your workspace's region is available again.

Also, the greater the distance between your cluster's region and your workspace's region, the longer it will take to fetch a token.

Next steps

- Train and deploy an image classification model.
- Consume an Azure Machine Learning model deployed as a web service.

Monitoring Azure Machine Learning

3/5/2020 • 5 minutes to read • Edit Online

This article describes the monitoring data generated by Azure Machine Learning. It also describes how you can use the Azure Monitor to analyze your data and define alerts.

> **TIP**
> The information in this document is primarily for administrators, as it describes monitoring for the Azure Machine Learning. If you are a data scientist or developer, and want to monitor information specific to your model training runs, see the following documents:
>
> - Start, monitor, and cancel training runs
> - Log metrics for training runs
> - Track experiments with MLflow
> - Visualize runs with TensorBoard

Azure Monitor

Azure Machine Learning logs monitoring data using Azure Monitor, which is a full stack monitoring service in Azure. Azure Monitor provides a complete set of features to monitor your Azure resources. It can also monitor resources in other clouds and on-premises.

Start with the article Azure Monitor overview, which provides an overview of the monitoring capabilities. The following sections build on this information by providing specifics of using Azure Monitor with Azure Machine Learning.

To understand costs associated with Azure Monitor, see Usage and estimated costs. To understand the time it takes for your data to appear in Azure Monitor, see Log data ingestion time.

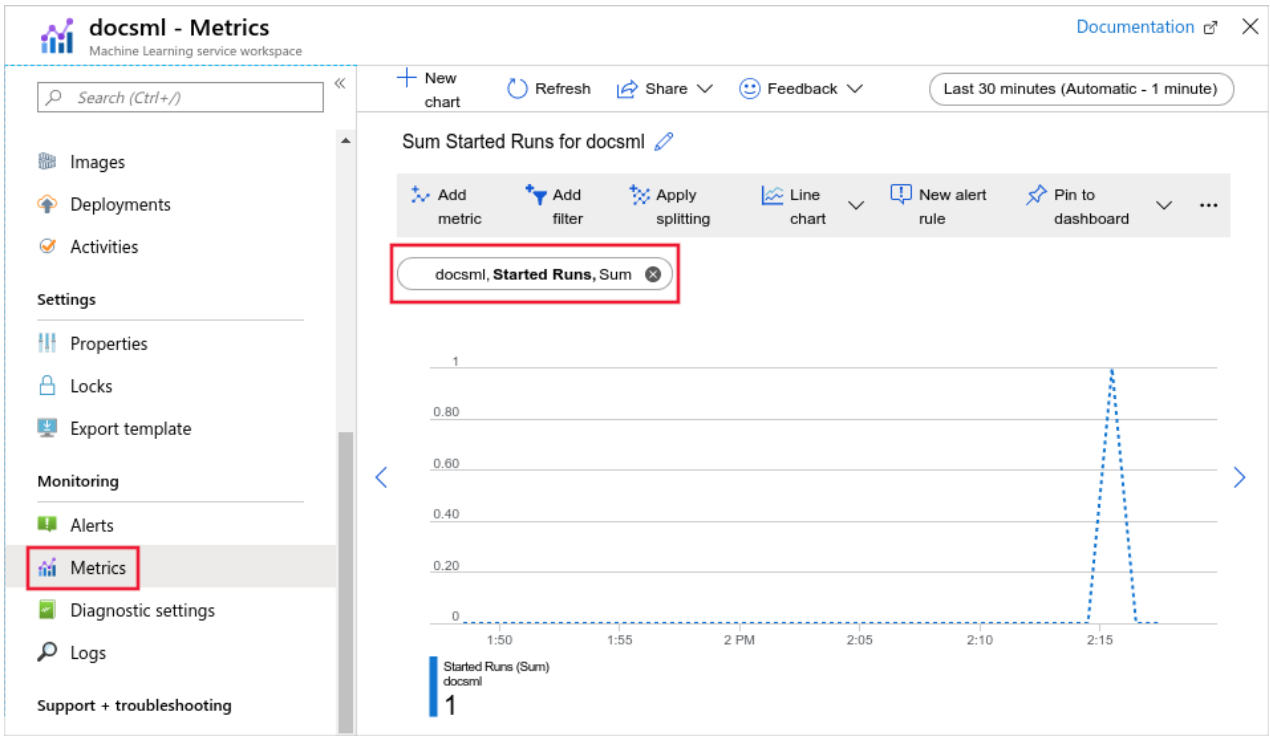
Monitoring data from Azure Machine Learning

Azure Machine Learning collects the same kinds of monitoring data as other Azure resources, which are described in Monitoring data from Azure resources. See Azure Machine Learning monitoring data reference for a detailed reference of the logs and metrics created by Azure Machine Learning.

Analyzing metric data

You can analyze metrics for Azure Machine Learning by opening **Metrics** from the **Azure Monitor** menu. See Getting started with Azure Metrics Explorer for details on using this tool.

All metrics for Azure Machine Learning are in the namespace **Machine Learning Service Workspace**.



Filtering and splitting

For metrics that support dimensions, you can apply filters using a dimension value. For example, filtering **Active Cores** for a **Cluster Name** of `cpu-cluster`.

You can also split a metric by dimension to visualize how different segments of the metric compare with each other. For example, splitting out the **Pipeline Step Type** to see a count of the types of steps used in the pipeline.

For more information of filtering and splitting, see Advanced features of Azure Monitor.

Alerts

You can access alerts for Azure Machine Learning by opening **Alerts** from the **Azure Monitor** menu. See Create, view, and manage metric alerts using Azure Monitor for details on creating alerts.

The following table lists common and recommended metric alert rules for Azure Machine Learning:

ALERT TYPE	CONDITION	DESCRIPTION
Model Deploy Failed	Aggregation type: Total, Operator: Greater than, Threshold value: 0	When one or more model deployments have failed
Quota Utilization Percentage	Aggregation type: Average, Operator: Greater than, Threshold value: 90	When the quota utilization percentage is greater than 90%
Unusable Nodes	Aggregation type: Total, Operator: Greater than, Threshold value: 0	When there are one or more unusable nodes

Configuration

> **IMPORTANT**
>
> **Metrics for Azure Machine Learning do not need to be configured**, they are collected automatically and are available in the Metrics Explorer for monitoring and alerting.

You can add a diagnostic setting to configure the following functionality:

- Archive log and metrics information to an Azure storage account.
- Stream log and metrics information to an Azure Event Hub.
- Send log and metrics information to Azure Monitor Log Analytics.

Enabling these settings requires additional Azure services (storage account, event hub, or Log Analytics), which may increase your cost. To calculate an estimated cost, visit the Azure pricing calculator.

For more information on creating a diagnostic setting, see Create diagnostic setting to collect platform logs and metrics in Azure.

You can configure the following logs for Azure Machine Learning:

CATEGORY	DESCRIPTION
AmlComputeClusterEvent	Events from Azure Machine Learning compute clusters.
AmlComputeClusterNodeEvent	Events from nodes within an Azure Machine Learning compute cluster.
AmlComputeJobEvent	Events from jobs running on Azure Machine Learning compute.

> **NOTE**
> When you enable metrics in a diagnostic setting, dimension information is not currently included as part of the information sent to a storage account, event hub, or log analytics.

Analyzing log data

Using Azure Monitor Log Analytics requires you to create a diagnostic configuration and enable **Send information to Log Analytics**. For more information, see the Configuration section.

Data in Azure Monitor Logs is stored in tables, with each table having its own set of unique properties. Azure Machine Learning stores data in the following tables:

TABLE	DESCRIPTION
AmlComputeClusterEvent	Events from Azure Machine Learning compute clusters.
AmlComputeClusterNodeEvent	Events from nodes within an Azure Machine Learning compute cluster.
AmlComputeJobEvent	Events from jobs running on Azure Machine Learning compute.

> **IMPORTANT**
> When you select **Logs** from the Azure Machine Learning menu, Log Analytics is opened with the query scope set to the current workspace. This means that log queries will only include data from that resource. If you want to run a query that includes data from other databases or data from other Azure services, select **Logs** from the **Azure Monitor** menu. See Log query scope and time range in Azure Monitor Log Analytics for details.

For a detailed reference of the logs and metrics, see Azure Machine Learning monitoring data reference.

Sample queries

Following are queries that you can use to help you monitor your Azure Machine Learning resources:

- Get failed jobs in the last five days:

```
AmlComputeJobEvent
| where TimeGenerated > ago(5d) and EventType == "JobFailed"
| project  TimeGenerated , ClusterId , EventType , ExecutionState , ToolType
```

- Get records for a specific job name:

```
AmlComputeJobEvent
| where JobName == "automl_a9940991-dedb-4262-9763-2fd08b79d8fb_setup"
| project  TimeGenerated , ClusterId , EventType , ExecutionState , ToolType
```

- Get cluster events in the last five days for clusters where the VM size is Standard_D1_V2:

```
AmlComputeClusterEvent
| where TimeGenerated > ago(4d) and VmSize == "STANDARD_D1_V2"
| project  ClusterName , InitialNodeCount , MaximumNodeCount , QuotaAllocated , QuotaUtilized
```

- Get nodes allocated in the last eight days:

```
AmlComputeClusterNodeEvent
| where TimeGenerated > ago(8d) and NodeAllocationTime  > ago(8d)
| distinct NodeId
```

Next steps

- For a reference of the logs and metrics, see Azure Machine Learning monitoring data reference.
- For information on working with quotas related to Azure Machine Learning, see Manage and request quotas for Azure resources.
- For details on monitoring Azure resources, see Monitoring Azure resources with Azure Monitor.

Consume Azure Machine Learning events (Preview)

3/12/2020 • 4 minutes to read • Edit Online

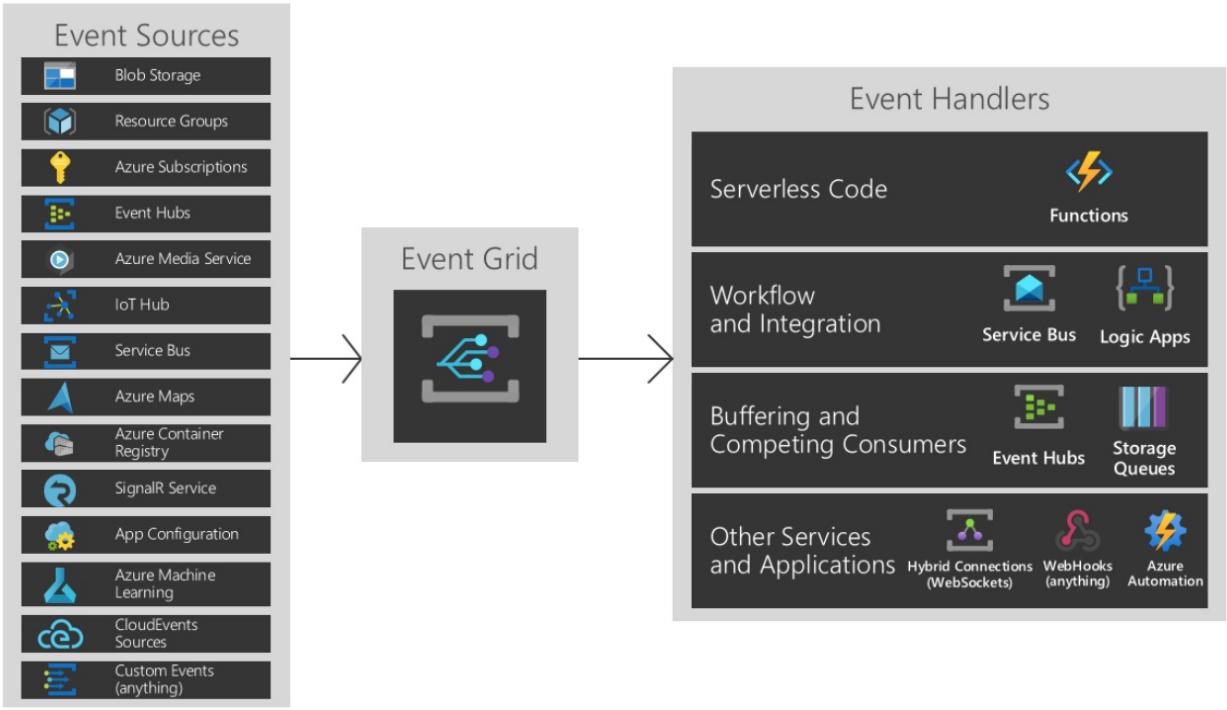
Azure Machine Learning manages the entire lifecycle of machine learning process, including model training, model deployment, and monitoring. Azure Machine Learning events allow applications to react to events during the machine learning lifecycle, such as the completion of training runs, the registration and deployment of models, and the detection of data drift, by using modern serverless architectures.

These events are published through Azure Event Grid. Using Azure portal, Powershell or Azure CLI, customers can easily subscribe events by specifying one or more event types, and filtering conditions. Customers also have choice to build a wide range of event handlers such as Azure Functions, Azure Logic Apps, or generic webhooks. Azure Machine Learning, along with Azure Event Grid, provides a high available, reliable, and fault-tolerant event delivery platform for you to build event driven applications.

For information on using Azure Machine Learning with Event Grid, see Create Event Driven Machine Learning Workflows (Preview).

The event model

Azure Event Grid reads events from sources, such as Azure Machine Learning and other Azure services. These events are then sent to event handlers such as Azure Event Hubs, Azure Functions, Logic Apps, and others. The following diagram shows how Event Grid connects sources and handlers, but is not a comprehensive list of supported integrations.



For more information on event sources and event handlers, see What is Event Grid?.

Azure Machine Learning event types

Azure Machine Learning provides events in the various points of machine learning lifecycle:

EVENT TYPE	DESCRIPTION
`Microsoft.MachineLearningServices.RunCompleted`	Raised when a machine learning experiment run is completed

EVENT TYPE	DESCRIPTION
`Microsoft.MachineLearningServices.ModelRegistered`	Raised when a machine learning model is registered in the workspace
`Microsoft.MachineLearningServices.ModelDeployed`	Raised when a deployment of inference service with one or more models is completed
`Microsoft.MachineLearningServices.DatasetDriftDetected`	Raised when a data drift detection job for two datasets is completed
`Microsoft.MachineLearningServices.RunStatusChanged`	Raised when a run status changed, currently only raised when a run status is 'failed'

Subscribe to Machine Learning events

Subscriptions for Azure Machine Learning events are protected by role-based access control (RBAC). Only contributor or owner of a workspace can create, update, and delete event subscriptions.

Event subscriptions can be filtered based on a variety of conditions. Filters can be applied to event subscriptions either during the creation of the event subscription or at a later time.

Filter by event type

An event subscription can specify one or more Azure Machine Learning event types.

Filter by event subject

Azure Event Grid supports subject filters based on **begins with** and **ends with** matches, so that events with a matching subject are delivered to the subscriber. Different machine learning events have different subject format.

EVENT TYPE	SUBJECT FORMAT	SAMPLE SUBJECT
`Microsoft.MachineLearningServices.RunComp`	`experiments/{ExperimentId}/runs/{RunId}`	`experiments/b1d7966c-f73a-4c68-b846-992ace89551f/runs/my_exp1_1554835758_38dbaa9`
`Microsoft.MachineLearningServices.ModelRe`	`models/{modelName}:{modelVersion}`	`models/sklearn_regression_model:3`
`Microsoft.MachineLearningServices.ModelDe`	`endpoints/{serviceId}`	`endpoints/my_sklearn_aks`
`Microsoft.MachineLearningServices.Dataset`	`datadrift/{data.DataDriftId}/run/{data.Ru`	`datadrift/4e694bf5-712e-4e40-b06a-d2a2755212d4/run/my_driftrun1_1550564444_fbb`
`Microsoft.MachineLearningServices.RunStat`	`experiments/{ExperimentId}/runs/{RunId}`	`experiments/b1d7966c-f73a-4c68-b846-992ace89551f/runs/my_exp1_1554835758_38dbaa9`

Advanced filtering

Azure Event Grid also supports advanced filtering based on published event schema. Azure Machine Learning event schema details can be found in Azure Event Grid event schema for Azure Machine Learning.

Some sample advanced filterings you can perform include:

- For `Microsoft.MachineLearningServices.ModelRegistered` event, to filter model's tag value:

```
--advanced-filter data.ModelTags.key1 StringIn ('value1')
```

To learn more about how to apply filters, see Filter events for Event Grid.

Consume Machine Learning events

Applications that handle Machine Learning events should follow a few recommended practices:

- As multiple subscriptions can be configured to route events to the same event handler, it is important not to assume events are from a particular source, but to check the topic of the message to ensure that it comes from the machine learning workspace you are expecting.
- Similarly, check that the eventType is one you are prepared to process, and do not assume that all events you receive will be the types you expect.
- As messages can arrive out of order and after some delay, use the etag fields to understand if your information about objects is still up-to-date. Also, use the sequencer fields to understand the order of events on any particular object.
- Ignore fields you don't understand. This practice will help keep you resilient to new features that might be added in the future.
- Failed or cancelled Azure Machine Learning operations will not trigger an event. For example, if a model deployment fails Microsoft.MachineLearningServices.ModelDeployed won't be triggered. Consider such failure mode when design your applications. You can always use Azure Machine Learning SDK, CLI or portal to check the status of an operation and understand the detailed failure reasons.

Azure Event Grid allows customers to build de-coupled message handlers, which can be triggered by Azure Machine Learning events. Some notable examples of message handlers are:

- Azure Functions
- Azure Logic Apps
- Azure Event Hubs
- Azure Data Factory Pipeline
- Generic webhooks, which may be hosted on the Azure platform or elsewhere

Next steps

Learn more about Event Grid and give Azure Machine Learning events a try:

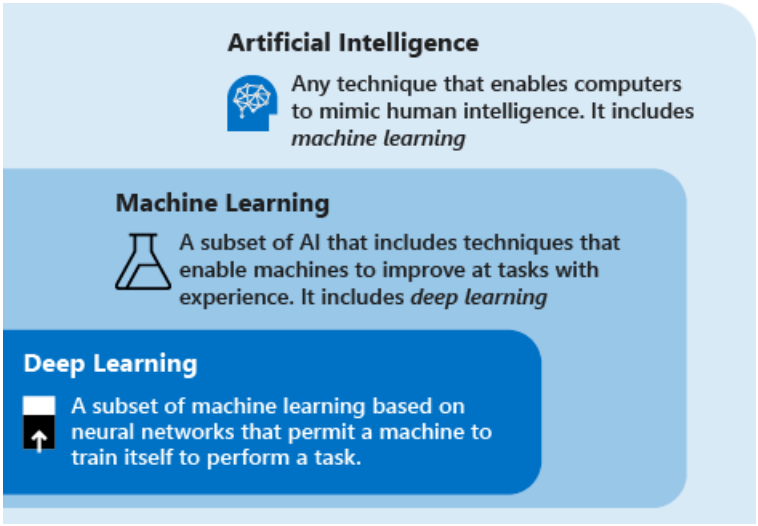
- About Event Grid
- Azure Event Grid event schema for Azure Machine Learning
- Create event driven workflows with Azure Machine Learning

Deep learning vs. machine learning

4/16/2020 • 6 minutes to read • Edit Online

This article helps you compare deep learning vs. machine learning. You'll learn how the two concepts compare and how they fit into the broader category of artificial intelligence. The article also describes how deep learning can be applied to real-world scenarios such as fraud detection, voice and facial recognition, sentiment analytics, and time series forecasting.

Deep learning, machine learning, and AI



Consider the following definitions to understand deep learning vs. machine learning vs. AI:

- **Deep learning** is a subset of machine learning that's based on artificial neural networks. The *learning process* is *deep* because the structure of artificial neural networks consists of multiple input, output, and hidden layers. Each layer contains units that transform the input data into information that the next layer can use for a certain predictive task. Thanks to this structure, a machine can learn through its own data processing.
- **Machine learning** is a subset of artificial intelligence that uses techniques (such as deep learning) that enable machines to use experience to improve at tasks. The *learning process* is based on the following steps:

  1. Feed data into an algorithm. (In this step you can provide additional information to the model, for example, by performing feature extraction.)
  2. Use this data to train a model.
  3. Test and deploy the model.
  4. Consume the deployed model to do an automated predictive task. (In other words, call and use the deployed model to receive the predictions returned by the model.)
- **Artificial intelligence (AI)** is a technique that enables computers to mimic human intelligence. It includes machine learning.

It's important to understand the relationship among AI, machine learning, and deep learning. Machine learning is a way to achieve artificial intelligence. By using machine learning and deep learning techniques, you can build computer systems and applications that do tasks that are commonly associated with human intelligence. These tasks include image recognition, speech recognition, and language translation.

Techniques of deep learning vs. machine learning

Now that you have the overview of machine learning vs. deep learning, let's compare the two techniques. In machine learning, the algorithm needs to be told how to make an accurate prediction by consuming more information (for example, by performing feature extraction). In deep learning, the algorithm can learn how to make an accurate prediction through its own data processing, thanks to the artificial neural network structure.

The following table compares the two techniques in more detail:

	ALL MACHINE LEARNING	ONLY DEEP LEARNING
Number of data points	Can use small amounts of data to make predictions.	Needs to use large amounts of training data to make predictions.
Hardware dependencies	Can work on low-end machines. It doesn't need a large amount of computational power.	Depends on high-end machines. It inherently does a large number of matrix multiplication operations. A GPU can efficiently optimize these operations.
Featurization process	Requires features to be accurately identified and created by users.	Learns high-level features from data and creates new features by itself.
Learning approach	Divides the learning process into smaller steps. It then combines the results from each step into one output.	Moves through the learning process by resolving the problem on an end-to-end basis.
Execution time	Takes comparatively little time to train, ranging from a few seconds to a few hours.	Usually takes a long time to train because a deep learning algorithm involves many layers.
Output	The output is usually a numerical value, like a score or a classification.	The output can have multiple formats, like a text, a score or a sound.

Deep learning use cases

Because of the artificial neural network structure, deep learning excels at identifying patterns in unstructured data such as images, sound, video, and text. For this reason, deep learning is rapidly transforming many industries, including healthcare, energy, finance, and transportation. These industries are now rethinking traditional business processes.

Some of the most common applications for deep learning are described in the following paragraphs.

Named-entity recognition

Named-entity recognition is a deep learning method that takes a piece of text as input and transforms it into a pre-specified class. This new information could be a postal code, a date, a product ID. The information can then be stored in a structured schema to build a list of addresses or serve as a benchmark for an identity validation engine.

Object detection

Deep learning has been applied in many object detection use cases. Object detection comprises two parts: image classification and then image localization. Image *classification* identifies the image's objects, such as cars or people. Image *localization* provides the specific location of these objects.

Object detection is already used in industries such as gaming, retail, tourism, and self-driving cars.

Image caption generation

Like image recognition, in image captioning, for a given image, the system must generate a caption that describes the contents of the image. When you can detect and label objects in photographs, the next step is to turn those labels into descriptive sentences.

Usually, image captioning applications use convolutional neural networks to identify objects in an image and then use a recurrent neural network to turn the labels into consistent sentences.

Machine translation

Machine translation takes words or sentences from one language and automatically translates them into another language. Machine translation has been around for a long time, but deep learning achieves impressive results in two specific areas: automatic translation of text (and translation of speech to text) and automatic translation of images.

With the appropriate data transformation, a neural network can understand text, audio, and visual signals. Machine translation can be used to identify snippets of sound in larger audio files and transcribe the spoken word or image as text.

Text analytics

Text analytics based on deep learning methods involves analyzing large quantities of text data (for example, medical documents or expenses receipts), recognizing patterns, and creating organized and concise information out of it.

Companies use deep learning to perform text analysis to detect insider trading and compliance with government regulations. Another common example is insurance fraud: text analytics has often been used to analyze large amounts of documents to recognize the chances of an insurance claim being fraud.

Artificial neural networks

Artificial neural networks are formed by layers of connected nodes. Deep learning models use neural networks that have a large number of layers.

The following sections explore most popular artificial neural network typologies.

Feedforward neural network

The feedforward neural network is the most basic type of artificial neural network. In a feedforward network, information moves in only one direction from input layer to output layer. Feedforward neural networks transform an input by putting it through a series of hidden layers. Every layer is made up of a set of neurons, and each layer is fully connected to all neurons in the layer before. The last fully connected layer (the output layer) represents the generated predictions.

Recurrent neural network

Recurrent neural networks are a widely used artificial neural network. These networks save the output of a layer and feed it back to the input layer to help predict the layer's outcome. Recurrent neural networks have great learning abilities. They're widely used for complex tasks such as time series forecasting, learning handwriting and recognizing language.

Convolutional neural networks

A convolutional neural network is a particularly effective artificial neural network, and it presents a unique architecture. Layers are organized in three dimensions: width, height, and depth. The neurons in one layer connect not to all the neurons in the next layer, but only to a small region of the layer's neurons. The final output is reduced to a single vector of probability scores, organized along the depth dimension.

Convolutional neural networks have been used in areas such as video recognition, image recognition and recommender systems.

Next steps

The following articles show you how to use deep learning technology in Azure Machine Learning:

- Classify handwritten digits by using a TensorFlow model
- Classify handwritten digits by using a TensorFlow estimator and Keras
- Classify images by using a Pytorch model
- Classify handwritten digits by using a Chainer model

Also, use the Machine Learning Algorithm Cheat Sheet to choose algorithms for your model.

Create and manage Azure Machine Learning workspaces in the Azure portal

4/13/2020 • 4 minutes to read • Edit Online

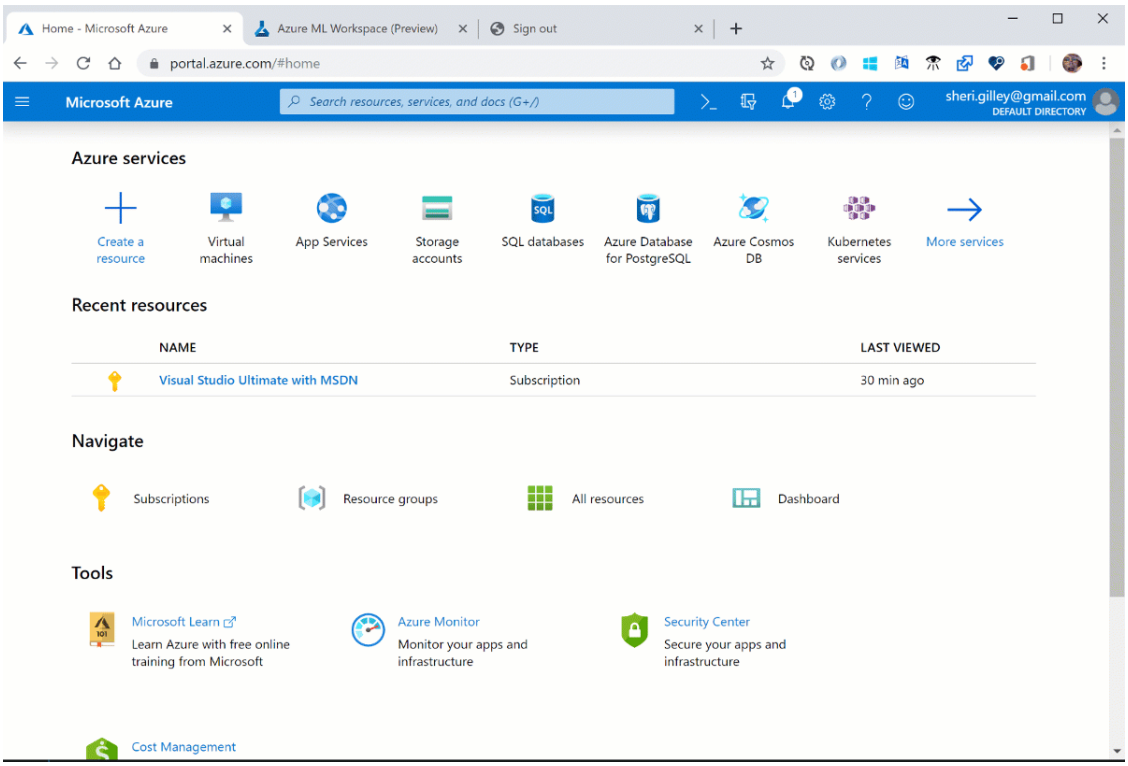
APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you'll create, view, and delete **Azure Machine Learning workspaces** in the Azure portal for Azure Machine Learning. The portal is the easiest way to get started with workspaces but as your needs change or requirements for automation increase you can also create and delete workspaces using the CLI, with Python code or via the VS Code extension.

Create a workspace

To create a workspace, you need an Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.

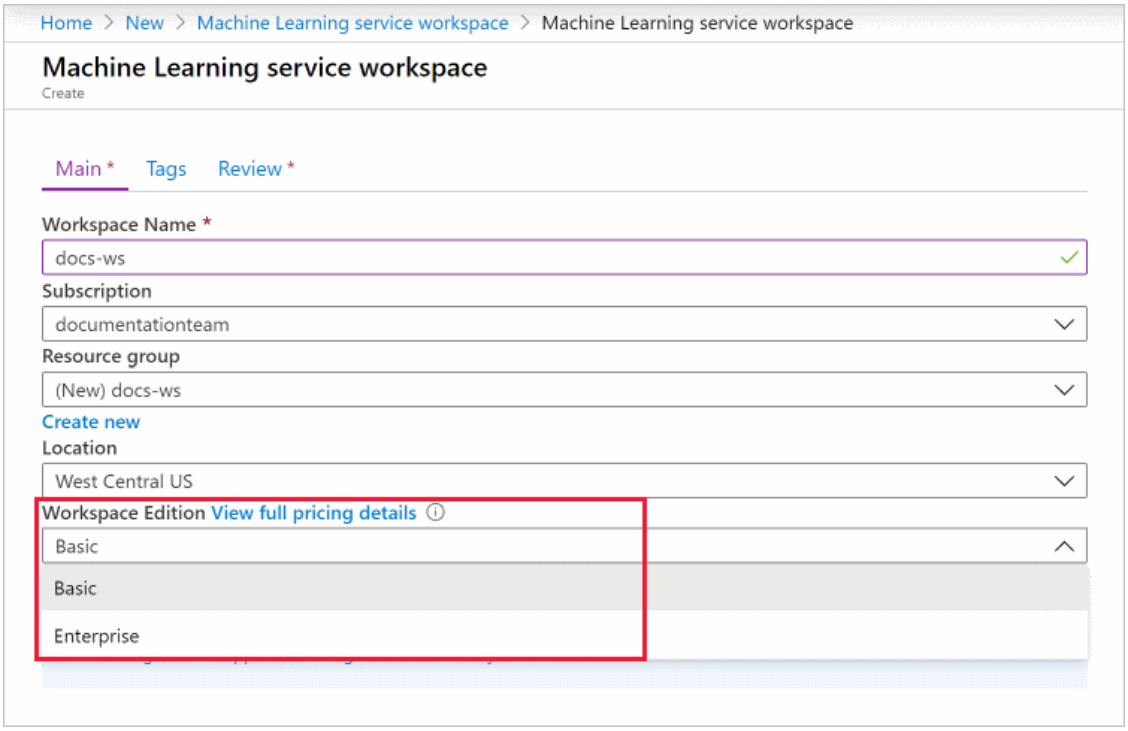
1. Sign in to the Azure portal by using the credentials for your Azure subscription.
2. In the upper-left corner of Azure portal, select **+ Create a resource**.



3. Use the search bar to find **Machine Learning**.
4. Select **Machine Learning**.
5. In the **Machine Learning** pane, select **Create** to begin.
6. Provide the following information to configure your new workspace:

FIELD	DESCRIPTION

FIELD	DESCRIPTION
Workspace name	Enter a unique name that identifies your workspace. In this example, we use **docs-ws**. Names must be unique across the resource group. Use a name that's easy to recall and to differentiate from workspaces created by others. The workspace name is case-insensitive.
Subscription	Select the Azure subscription that you want to use.
Resource group	Use an existing resource group in your subscription or enter a name to create a new resource group. A resource group holds related resources for an Azure solution. In this example, we use **docs-aml**.
Location	Select the location closest to your users and the data resources to create your workspace.
Workspace edition	Select **Basic** or **Enterprise**. This workspace edition determines the features to which you'll have access and pricing. Learn more about Basic and Enterprise edition offerings.



7. When you're finished configuring the workspace, select **Review + Create**.
8. Review the settings and make any additional changes or corrections. When you're satisfied with the settings, select **Create**.

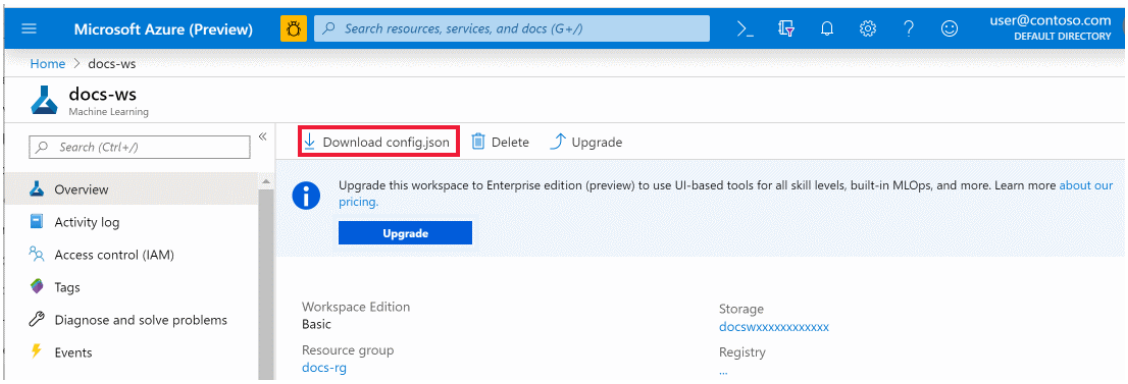
> **WARNING**
> It can take several minutes to create your workspace in the cloud.

When the process is finished, a deployment success message appears.

9. To view the new workspace, select **Go to resource**.

Download a configuration file

1. If you will be creating a compute instance, skip this step.
2. If you plan to use code on your local environment that references this workspace, select **Download config.json** from the **Overview** section of the workspace.

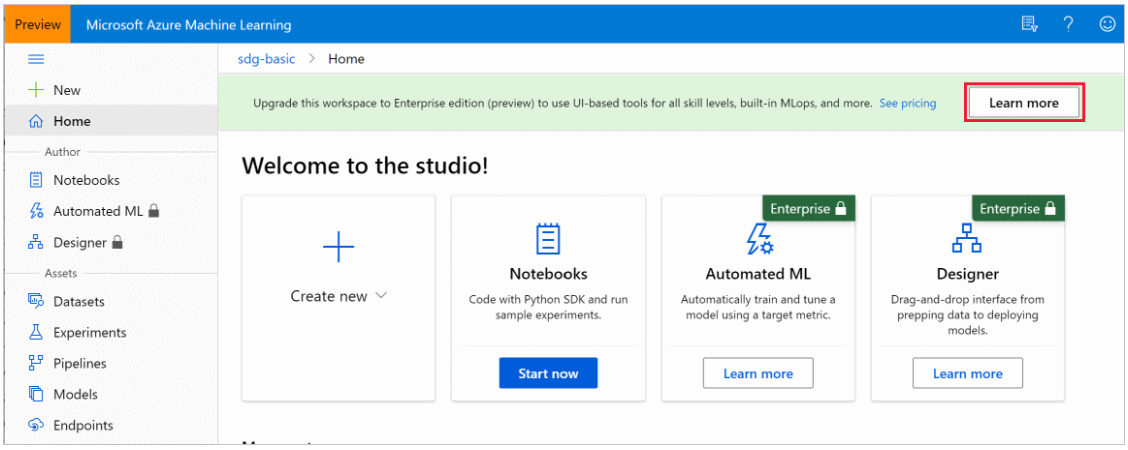


Place the file into the directory structure with your Python scripts or Jupyter Notebooks. It can be in the same directory, a subdirectory named *.azureml*, or in a parent directory. When you create a compute instance, this file is added to the correct directory on the VM for you.

Upgrade to Enterprise edition

You can upgrade your workspace from Basic edition to Enterprise edition to take advantage of the enhanced features such as low-code experiences and enhanced security features.

1. Sign in to Azure Machine Learning studio.
2. Select the workspace that you wish to upgrade.
3. Select **Learn more** at the top right of the page.



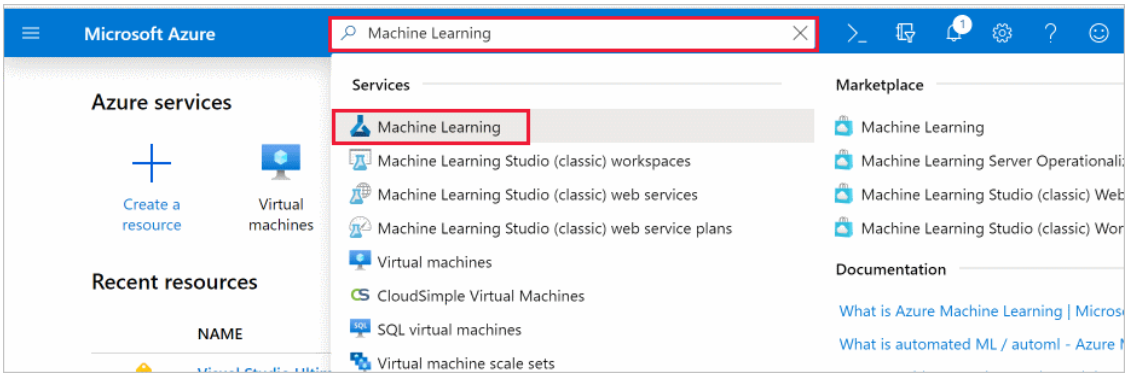
4. Select **Upgrade** in the window that appears.

> **IMPORTANT**
>
> You cannot downgrade an Enterprise edition workspace to a Basic edition workspace.

Find a workspace

1. Sign in to the Azure portal.
2. In the top search field, type **Machine Learning**.

3. Select **Machine Learning**.

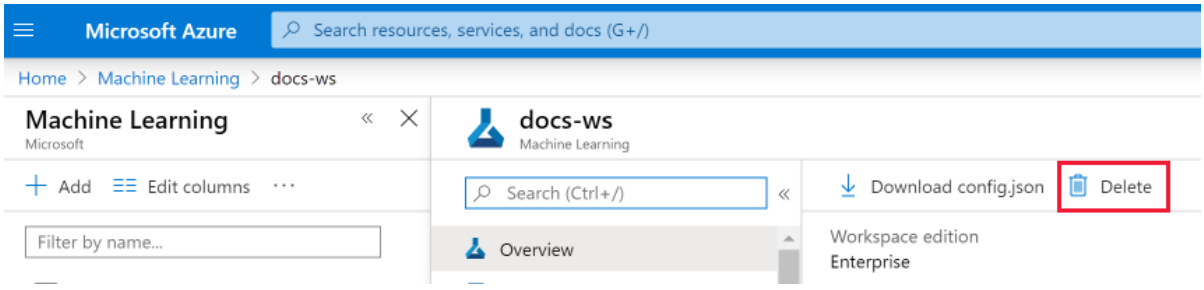


4. Look through the list of workspaces found. You can filter based on subscription, resource groups, and locations.

5. Select a workspace to display its properties.

Delete a workspace

In the Azure portal, select **Delete** at the top of the workspace you wish to delete.

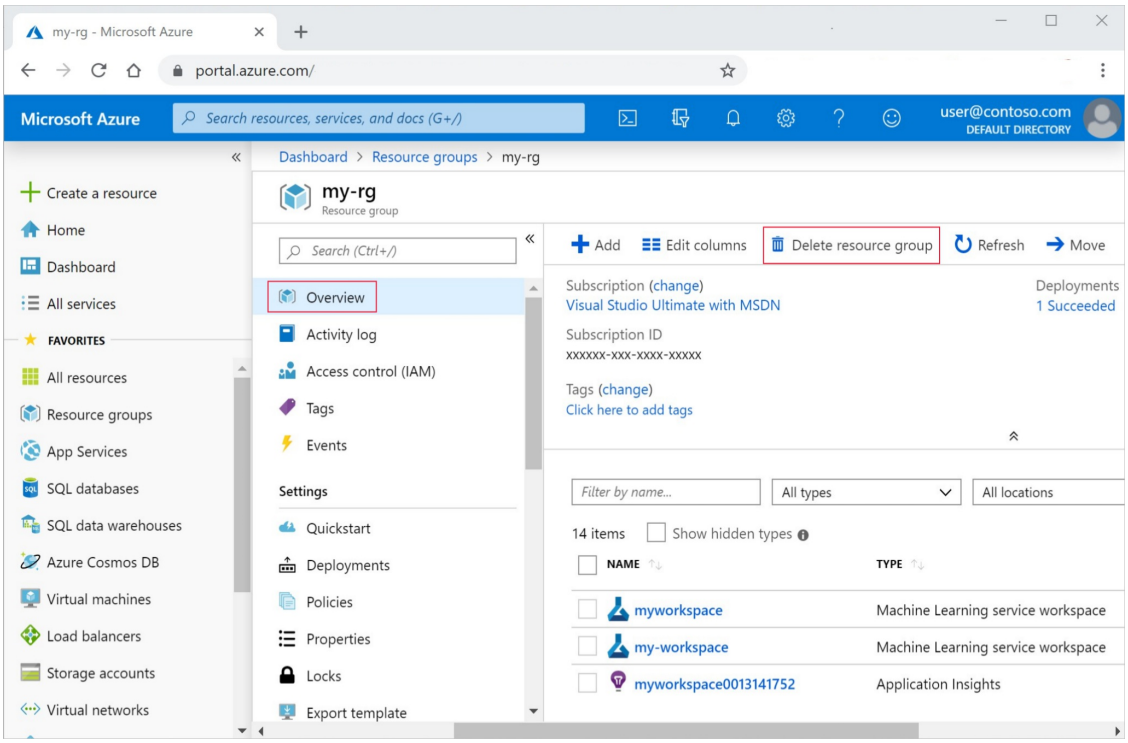


Clean up resources

> **IMPORTANT**
>
> The resources you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.



2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

Troubleshooting

Resource provider errors

When creating an Azure Machine Learning workspace, or a resource used by the workspace, you may receive an error similar to the following messages:

- `No registered resource provider found for location {location}`
- `The subscription is not registered to use namespace {resource-provider-namespace}`

Most resource providers are automatically registered, but not all. If you receive this message, you need to register the provider mentioned.

For information on registering resource providers, see Resolve errors for resource provider registration.

Moving the workspace

> **WARNING**
> Moving your Azure Machine Learning workspace to a different subscription, or moving the owning subscription to a new tenant, is not supported. Doing so may cause errors.

Deleting the Azure Container Registry

The Azure Machine Learning workspace uses Azure Container Registry (ACR) for some operations. It will automatically create an ACR instance when it first needs one.

> **WARNING**
> Once an Azure Container Registry has been created for a workspace, do not delete it. Doing so will break your Azure Machine Learning workspace.

Next steps

Follow the full-length tutorial to learn how to use a workspace to build, train, and deploy models with Azure Machine Learning.

Tutorial: Train models

Create a workspace for Azure Machine Learning with Azure CLI

4/17/2020 • 8 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to create an Azure Machine Learning workspace using the Azure CLI. The Azure CLI provides commands for managing Azure resources. The machine learning extension to the CLI provides commands for working with Azure Machine Learning resources.

Prerequisites

- An **Azure subscription**. If you do not have one, try the free or paid version of Azure Machine Learning.
- To use the CLI commands in this document from your **local environment**, you need the Azure CLI.

  If you use the Azure Cloud Shell, the CLI is accessed through the browser and lives in the cloud.

Connect the CLI to your Azure subscription

> **IMPORTANT**
>
> If you are using the Azure Cloud Shell, you can skip this section. The cloud shell automatically authenticates you using the account you log into your Azure subscription.

There are several ways that you can authenticate to your Azure subscription from the CLI. The most basic is to interactively authenticate using a browser. To authenticate interactively, open a command line or terminal and use the following command:

```
az login
```

If the CLI can open your default browser, it will do so and load a sign-in page. Otherwise, you need to open a browser and follow the instructions on the command line. The instructions involve browsing to https://aka.ms/devicelogin and entering an authorization code.

> **TIP**
>
> After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.
>
> To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

For other methods of authenticating, see Sign in with Azure CLI.

Install the machine learning extension

To install the machine learning extension, use the following command:

```
az extension add -n azure-cli-ml
```

Create a workspace

The Azure Machine Learning workspace relies on the following Azure services or entities:

> **IMPORTANT**
> If you do not specify an existing Azure service, one will be created automatically during workspace creation. You must always specify a resource group.

SERVICE	PARAMETER TO SPECIFY AN EXISTING INSTANCE
Azure resource group	`-g <resource-group-name>`
Azure Storage Account	`--storage-account <service-id>`
Azure Application Insights	`--application-insights <service-id>`
Azure Key Vault	`--keyvault <service-id>`
Azure Container Registry	`--container-registry <service-id>`

Create a resource group

The Azure Machine Learning workspace must be created inside a resource group. You can use an existing resource group or create a new one. To **create a new resource group**, use the following command. Replace `<resource-group-name>` with the name to use for this resource group. Replace `<location>` with the Azure region to use for this resource group:

> **TIP**
> You should select a region where Azure Machine Learning is available. For information, see Products available by region.

```
az group create --name <resource-group-name> --location <location>
```

The response from this command is similar to the following JSON:

```
{
  "id": "/subscriptions/<subscription-GUID>/resourceGroups/<resourcegroupname>",
  "location": "<location>",
  "managedBy": null,
  "name": "<resource-group-name>",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": null
}
```

For more information on working with resource groups, see az group.

Automatically create required resources

To create a new workspace where the **services are automatically created**, use the following command:

> **TIP**
> The commands in this section create a basic edition workspace. To create an enterprise workspace, use the `--sku enterprise` switch with the `az ml workspace create` command. For more information on Azure Machine Learning editions, see What is Azure Machine Learning.

```
az ml workspace create -w <workspace-name> -g <resource-group-name>
```

> **NOTE**
> The workspace name is case-insensitive.

The output of this command is similar to the following JSON:

```
{
  "applicationInsights": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.insights/components/<application-insight-name>",
  "containerRegistry": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.containerregistry/registries/<acr-name>",
  "creationTime": "2019-08-30T20:24:19.6984254+00:00",
  "description": "",
  "friendlyName": "<workspace-name>",
  "id": "/subscriptions/<service-GUID>/resourceGroups/<resource-group-
name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>",
  "identityPrincipalId": "<GUID>",
  "identityTenantId": "<GUID>",
  "identityType": "SystemAssigned",
  "keyVault": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.keyvault/vaults/<key-vault-name>",
  "location": "<location>",
  "name": "<workspace-name>",
  "resourceGroup": "<resource-group-name>",
  "storageAccount": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.storage/storageaccounts/<storage-account-name>",
  "type": "Microsoft.MachineLearningServices/workspaces",
  "workspaceid": "<GUID>"
}
```

Use existing resources

To create a workspace that uses existing resources, you must provide the ID for the resources. Use the following

commands to get the ID for the services:

> **IMPORTANT**
> You don't have to specify all existing resources. You can specify one or more. For example, you can specify an existing storage account and the workspace will create the other resources.

- **Azure Storage Account**: `az storage account show --name <storage-account-name> --query "id"`

  The response from this command is similar to the following text, and is the ID for your storage account:

  ```
  "/subscriptions/<service-GUID>/resourceGroups/<resource-group-name>/providers/Microsoft.Storage/storageAccounts/<storage-account-name>"
  ```

- **Azure Application Insights**:

  1. Install the application insights extension:

     ```
     az extension add -n application-insights
     ```

  2. Get the ID of your application insight service:

     ```
     az monitor app-insights component show --app <application-insight-name> -g <resource-group-name> --query "id"
     ```

     The response from this command is similar to the following text, and is the ID for your application insights service:

     ```
     "/subscriptions/<service-GUID>/resourceGroups/<resource-group-name>/providers/microsoft.insights/components/<application-insight-name>"
     ```

- **Azure Key Vault**: `az keyvault show --name <key-vault-name> --query "ID"`

  The response from this command is similar to the following text, and is the ID for your key vault:

  ```
  "/subscriptions/<service-GUID>/resourceGroups/<resource-group-name>/providers/Microsoft.KeyVault/vaults/<key-vault-name>"
  ```

- **Azure Container Registry**: `az acr show --name <acr-name> -g <resource-group-name> --query "id"`

  The response from this command is similar to the following text, and is the ID for the container registry:

  ```
  "/subscriptions/<service-GUID>/resourceGroups/<resource-group-name>/providers/Microsoft.ContainerRegistry/registries/<acr-name>"
  ```

  > **IMPORTANT**
  > The container registry must have the the admin account enabled before it can be used with an Azure Machine Learning workspace.

Once you have the IDs for the resource(s) that you want to use with the workspace, use the base `az workspace create -w <workspace-name> -g <resource-group-name>` command and add the parameter(s) and ID(s) for the existing resources. For example, the following command creates a workspace that uses an existing container registry:

```
az ml workspace create -w <workspace-name> -g <resource-group-name> --container-registry
"/subscriptions/<service-GUID>/resourceGroups/<resource-group-
name>/providers/Microsoft.ContainerRegistry/registries/<acr-name>"
```

The output of this command is similar to the following JSON:

```
{
  "applicationInsights": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.insights/components/<application-insight-name>",
  "containerRegistry": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.containerregistry/registries/<acr-name>",
  "creationTime": "2019-08-30T20:24:19.6984254+00:00",
  "description": "",
  "friendlyName": "<workspace-name>",
  "id": "/subscriptions/<service-GUID>/resourceGroups/<resource-group-
name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>",
  "identityPrincipalId": "<GUID>",
  "identityTenantId": "<GUID>",
  "identityType": "SystemAssigned",
  "keyVault": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.keyvault/vaults/<key-vault-name>",
  "location": "<location>",
  "name": "<workspace-name>",
  "resourceGroup": "<resource-group-name>",
  "storageAccount": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.storage/storageaccounts/<storage-account-name>",
  "type": "Microsoft.MachineLearningServices/workspaces",
  "workspaceid": "<GUID>"
}
```

List workspaces

To list all the workspaces for your Azure subscription, use the following command:

```
az ml workspace list
```

The output of this command is similar to the following JSON:

```
[
  {
    "resourceGroup": "myresourcegroup",
    "subscriptionId": "<subscription-id>",
    "workspaceName": "myml"
  },
  {
    "resourceGroup": "anotherresourcegroup",
    "subscriptionId": "<subscription-id>",
    "workspaceName": "anotherml"
  }
]
```

For more information, see the az ml workspace list documentation.

Get workspace information

To get information about a workspace, use the following command:

```
az ml workspace show -w <workspace-name> -g <resource-group-name>
```

The output of this command is similar to the following JSON:

```
{
  "applicationInsights": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.insights/components/application-insight-name>",
  "creationTime": "2019-08-30T18:55:03.1807976+00:00",
  "description": "",
  "friendlyName": "",
  "id": "/subscriptions/<service-GUID>/resourceGroups/<resource-group-
name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>",
  "identityPrincipalId": "<GUID>",
  "identityTenantId": "<GUID>",
  "identityType": "SystemAssigned",
  "keyVault": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.keyvault/vaults/<key-vault-name>",
  "location": "<location>",
  "name": "<workspace-name>",
  "resourceGroup": "<resource-group-name>",
  "storageAccount": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.storage/storageaccounts/<storage-account-name>",
  "tags": {},
  "type": "Microsoft.MachineLearningServices/workspaces",
  "workspaceid": "<GUID>"
}
```

For more information, see the az ml workspace show documentation.

Update a workspace

To update a workspace, use the following command:

```
az ml workspace update -w <workspace-name> -g <resource-group-name>
```

The output of this command is similar to the following JSON:

```
{
  "applicationInsights": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.insights/components/application-insight-name>",
  "creationTime": "2019-08-30T18:55:03.1807976+00:00",
  "description": "",
  "friendlyName": "",
  "id": "/subscriptions/<service-GUID>/resourceGroups/<resource-group-
name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>",
  "identityPrincipalId": "<GUID>",
  "identityTenantId": "<GUID>",
  "identityType": "SystemAssigned",
  "keyVault": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.keyvault/vaults/<key-vault-name>",
  "location": "<location>",
  "name": "<workspace-name>",
  "resourceGroup": "<resource-group-name>",
  "storageAccount": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-
name>/providers/microsoft.storage/storageaccounts/<storage-account-name>",
  "tags": {},
  "type": "Microsoft.MachineLearningServices/workspaces",
  "workspaceid": "<GUID>"
}
```

For more information, see the az ml workspace update documentation.

Share a workspace with another user

To share a workspace with another user on your subscription, use the following command:

```
az ml workspace share -w <workspace-name> -g <resource-group-name> --user <user> --role <role>
```

For more information on roles-based access control (RBAC) with Azure Machine Learning, see Manage users and roles.

For more information, see the az ml workspace share documentation.

Sync keys for dependent resources

If you change access keys for one of the resources used by your workspace, use the following command to sync the new keys with the workspace:

```
az ml workspace sync-keys -w <workspace-name> -g <resource-group-name>
```

For more information on changing keys, see Regenerate storage access keys.

For more information, see the az ml workspace sync-keys documentation.

Delete a workspace

To delete a workspace after it is no longer needed, use the following command:

```
az ml workspace delete -w <workspace-name> -g <resource-group-name>
```

> **IMPORTANT**
> Deleting a workspace does not delete the application insight, storage account, key vault, or container registry used by the workspace.

You can also delete the resource group, which deletes the workspace and all other Azure resources in the resource group. To delete the resource group, use the following command:

```
az group delete -g <resource-group-name>
```

For more information, see the az ml workspace delete documentation.

Troubleshooting

Resource provider errors

When creating an Azure Machine Learning workspace, or a resource used by the workspace, you may receive an error similar to the following messages:

- `No registered resource provider found for location {location}`
- `The subscription is not registered to use namespace {resource-provider-namespace}`

Most resource providers are automatically registered, but not all. If you receive this message, you need to register

the provider mentioned.

For information on registering resource providers, see Resolve errors for resource provider registration.

Moving the workspace

> **WARNING**
> Moving your Azure Machine Learning workspace to a different subscription, or moving the owning subscription to a new tenant, is not supported. Doing so may cause errors.

Deleting the Azure Container Registry

The Azure Machine Learning workspace uses Azure Container Registry (ACR) for some operations. It will automatically create an ACR instance when it first needs one.

> **WARNING**
> Once an Azure Container Registry has been created for a workspace, do not delete it. Doing so will break your Azure Machine Learning workspace.

Next steps

For more information on the Azure CLI extension for machine learning, see the az ml documentation.

Create, run, and delete Azure ML resources using REST

2/24/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

There are several ways to manage your Azure ML resources. You can use the portal, command-line interface, or Python SDK. Or, you can choose the REST API. The REST API uses HTTP verbs in a standard way to create, retrieve, update, and delete resources. The REST API works with any language or tool that can make HTTP requests. REST's straightforward structure often makes it a good choice in scripting environments and for MLOps automation.

In this article, you learn how to:

- Retrieve an authorization token
- Create a properly-formatted REST request using service principal authentication
- Use GET requests to retrieve information about Azure ML's hierarchical resources
- Use PUT and POST requests to create and modify resources
- Use DELETE requests to clean up resources
- Use key-based authorization to score deployed models

Prerequisites

- An **Azure subscription** for which you have administrative rights. If you don't have such a subscription, try the free or paid personal subscription
- An Azure Machine Learning Workspace
- Administrative REST requests use service principal authentication. Follow the steps in Set up authentication for Azure Machine Learning resources and workflows to create a service principal in your workspace
- The **curl** utility. The **curl** program is available in the Windows Subsystem for Linux or any UNIX distribution. In PowerShell, **curl** is an alias for **Invoke-WebRequest** and `curl -d "key=val" -X POST uri` becomes `Invoke-WebRequest -Body "key=val" -Method POST -Uri uri`.

Retrieve a service principal authentication token

Administrative REST requests are authenticated with an OAuth2 implicit flow. This authentication flow uses a token provided by your subscription's service principal. To retrieve this token, you'll need:

- Your tenant ID (identifying the organization to which your subscription belongs)
- Your client ID (which will be associated with the created token)
- Your client secret (which you should safeguard)

You should have these values from the response to the creation of your service principal as discussed in Set up authentication for Azure Machine Learning resources and workflows. If you're using your company subscription, you might not have permission to create a service principal. In that case, you should use either a free or paid personal subscription.

To retrieve a token:

1. Open a terminal window
2. Enter the following code at the command line

3. Substitute your own values for `{your-tenant-id}`, `{your-client-id}`, and `{your-client-secret}`. Throughout this article, strings surrounded by curly brackets are variables you'll have to replace with your own appropriate values.
4. Run the command

```
curl -X POST https://login.microsoftonline.com/{your-tenant-id}/oauth2/token \
-d "grant_type=client_credentials&resource=https%3A%2F%2Fmanagement.azure.com%2F&client_id={your-client-id}&client_secret={your-client-secret}" \
```

The response should provide an access token good for one hour:

```
{
    "token_type": "Bearer",
    "expires_in": "3599",
    "ext_expires_in": "3599",
    "expires_on": "1578523094",
    "not_before": "1578519194",
    "resource": "https://management.azure.com/",
    "access_token": "your-access-token"
}
```

Make note of the token, as you'll use it to authenticate all subsequent administrative requests. You'll do so by setting an Authorization header in all requests:

```
curl -h "Authentication: Bearer {your-access-token}" ...more args...
```

Note that the value starts with the string "Bearer " including a single space before you add the token.

Get a list of resource groups associated with your subscription

To retrieve the list of resource groups associated with your subscription, run:

```
curl https://management.azure.com/subscriptions/{your-subscription-id}/resourceGroups?api-version=2019-11-01 -H "Authorization:Bearer {your-access-token}"
```

Across Azure, many REST APIs are published. Each service provider updates their API on their own cadence, but does so without breaking existing programs. The service provider uses the `api-version` argument to ensure compatibility. The `api-version` argument varies from service to service. For the Machine Learning Service, for instance, the current API version is `2019-11-01`. For storage accounts, it's `2019-06-01`. For key vaults, it's `2019-09-01`. All REST calls should set the `api-version` argument to the expected value. You can rely on the syntax and semantics of the specified version even as the API continues to evolve. If you send a request to a provider without the `api-version` argument, the response will contain a human-readable list of supported values.

The above call will result in a compacted JSON response of the form:

```
{
    "value": [
        {
            "id": "/subscriptions/12345abc-abbc-1b2b-1234-57ab575a5a5a/resourceGroups/RG1",
            "name": "RG1",
            "type": "Microsoft.Resources/resourceGroups",
            "location": "westus2",
            "properties": {
                "provisioningState": "Succeeded"
            }
        },
        {
            "id": "/subscriptions/12345abc-abbc-1b2b-1234-57ab575a5a5a/resourceGroups/RG2",
            "name": "RG2",
            "type": "Microsoft.Resources/resourceGroups",
            "location": "eastus",
            "properties": {
                "provisioningState": "Succeeded"
            }
        }
    ]
}
```


Drill down into workspaces and their resources

To retrieve the set of workspaces in a resource group, run the following, substituting `{your-subscription-id}`, `{your-resource-group}`, and `{your-access-token}`:

```
curl https://management.azure.com/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/providers/Microsoft.MachineLearningServices/workspaces/?api-version=2019-11-01 \
-H "Authorization:Bearer {your-access-token}"
```

Again you'll receive a JSON list, this time containing a list, each item of which details a workspace:

```
{
    "id": "/subscriptions/12345abc-abbc-1b2b-1234-
57ab575a5a5a/resourceGroups/DeepLearningResourceGroup/providers/Microsoft.MachineLearningServices/workspaces/m
y-workspace",
    "name": "my-workspace",
    "type": "Microsoft.MachineLearningServices/workspaces",
    "location": "centralus",
    "tags": {},
    "etag": null,
    "properties": {
        "friendlyName": "",
        "description": "",
        "creationTime": "2020-01-03T19:56:09.7588299+00:00",
        "storageAccount": "/subscriptions/12345abc-abbc-1b2b-1234-
57ab575a5a5a/resourcegroups/DeepLearningResourceGroup/providers/microsoft.storage/storageaccounts/myworkspace0
275623111",
        "containerRegistry": null,
        "keyVault": "/subscriptions/12345abc-abbc-1b2b-1234-
57ab575a5a5a/resourcegroups/DeepLearningResourceGroup/providers/microsoft.keyvault/vaults/myworkspace252564932
4",
        "applicationInsights": "/subscriptions/12345abc-abbc-1b2b-1234-
57ab575a5a5a/resourcegroups/DeepLearningResourceGroup/providers/microsoft.insights/components/myworkspace20535
23719",
        "hbiWorkspace": false,
        "workspaceId": "cba12345-abab-abab-abab-abababab123456",
        "subscriptionState": null,
        "subscriptionStatusChangeTimeStampUtc": null,
        "discoveryUrl": "https://centralus.experiments.azureml.net/discovery"
    },
    "identity": {
        "type": "SystemAssigned",
        "principalId": "abcdef1-abab-1234-1234-abababab123456",
        "tenantId": "1fedcba-abab-1234-1234-abababab123456"
    },
    "sku": {
        "name": "Basic",
        "tier": "Basic"
    }
}
```


To work with resources within a workspace, you'll switch from the general **management.azure.com** server to a REST API server specific to the location of the workspace. Note the value of the `discoveryUrl` key in the above JSON response. If you GET that URL, you'll receive a response something like:


```
{
  "api": "https://centralus.api.azureml.ms",
  "catalog": "https://catalog.cortanaanalytics.com",
  "experimentation": "https://centralus.experiments.azureml.net",
  "gallery": "https://gallery.cortanaintelligence.com/project",
  "history": "https://centralus.experiments.azureml.net",
  "hyperdrive": "https://centralus.experiments.azureml.net",
  "labeling": "https://centralus.experiments.azureml.net",
  "modelmanagement": "https://centralus.modelmanagement.azureml.net",
  "pipelines": "https://centralus.aether.ms",
  "studiocoreservices": "https://centralus.studioservice.azureml.com"
}
```


The value of the `api` response is the URL of the server that you'll use for additional requests. To list experiments, for instance, send the following command. Replace `regional-api-server` with the value of the `api` response (for instance, `centralus.api.azureml.ms`). Also replace `your-subscription-id`, `your-resource-group`, `your-workspace-name`, and `your-access-token` as usual:

```
curl https://{regional-api-server}/history/v1.0/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/\
providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-name}/experiments?api-version=2019-11-01 \
-H "Authorization:Bearer {your-access-token}"
```

Similarly, to retrieve registered models in your workspace, send:

```
curl https://{regional-api-server}/modelmanagement/v1.0/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/\
providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-name}/models?api-version=2019-11-01 \
-H "Authorization:Bearer {your-access-token}"
```

Notice that to list experiments the path begins with `history/v1.0` while to list models, the path begins with `modelmanagement/v1.0`. The REST API is divided into several operational groups, each with a distinct path. The API Reference docs at the links below list the operations, parameters, and response codes for the various operations.

AREA	PATH	REFERENCE
Artifacts	artifact/v2.0/	REST API Reference
Data stores	datastore/v1.0/	REST API Reference
Hyperparameter tuning	hyperdrive/v1.0/	REST API Reference
Models	modelmanagement/v1.0/	REST API Reference
Run history	execution/v1.0/ and history/v1.0/	REST API Reference

You can explore the REST API using the general pattern of:

URL COMPONENT	EXAMPLE
https://	
regional-api-server/	centralus.api.azureml.ms/
operations-path/	history/v1.0/
subscriptions/{your-subscription-id}/	subscriptions/abcde123-abab-abab-1234-0123456789abc/
resourceGroups/{your-resource-group}/	resourceGroups/MyResourceGroup/
providers/operation-provider/	providers/Microsoft.MachineLearningServices/
provider-resource-path/	workspaces/MLWorkspace/MyWorkspace/FirstExperiment/runs/1/
operations-endpoint/	artifacts/metadata/

Create and modify resources using PUT and POST requests

In addition to resource retrieval with the GET verb, the REST API supports the creation of all the resources

necessary to train, deploy, and monitor ML solutions.

Training and running ML models require compute resources. You can list the compute resources of a workspace with:

```
curl https://management.azure.com/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/\
providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-name}/computes?api-version=2019-11-01 \
-H "Authorization:Bearer {your-access-token}"
```

To create or overwrite a named compute resource, you'll use a PUT request. In the following, in addition to the now-familiar substitutions of `your-subscription-id`, `your-resource-group`, `your-workspace-name`, and `your-access-token`, substitute `your-compute-name`, and values for `location`, `vmSize`, `vmPriority`, `scaleSettings`, `adminUserName`, and `adminUserPassword`. As specified in the reference at Machine Learning Compute - Create Or Update SDK Reference, the following command creates a dedicated, single-node Standard_D1 (a basic CPU compute resource) that will scale down after 30 minutes:

```
curl -X PUT \
  'https://management.azure.com/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-
group}/providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-name}/computes/{your-compute-
name}?api-version=2019-11-01' \
  -H 'Authorization:Bearer {your-access-token}' \
  -H 'Content-Type: application/json' \
  -d '{
    "location": "{your-azure-location}",
    "properties": {
        "computeType": "AmlCompute",
        "properties": {
            "vmSize": "Standard_D1",
            "vmPriority": "Dedicated",
            "scaleSettings": {
                "maxNodeCount": 1,
                "minNodeCount": 0,
                "nodeIdleTimeBeforeScaleDown": "PT30M"
            }
        }
    },
    "userAccountCredentials": {
        "adminUserName": "{adminUserName}",
        "adminUserPassword": "{adminUserPassword}"
    }
}'
```

NOTE

In Windows terminals you may have to escape the double-quote symbols when sending JSON data. That is, text such as `"location"` becomes `\"location\"`.

A successful request will get a `201 Created` response, but note that this response simply means that the provisioning process has begun. You'll need to poll (or use the portal) to confirm its successful completion.

Create an experimental run

To start a run within an experiment, you need a zip folder containing your training script and related files, and a run definition JSON file. The zip folder must have the Python entry file in its root directory. As an example, zip a trivial Python program such as the following into a folder called **train.zip**.

```
# hello.py
# Entry file for run
print("Hello, REST!")
```

Save this next snippet as **definition.json**. Confirm the "Script" value matches the name of the Python file you just zipped up. Confirm the "Target" value matches the name of an available compute resource.

```
{
    "Configuration":{
       "Script":"hello.py",
       "Arguments":[
          "234"
       ],
       "SourceDirectoryDataStore":null,
       "Framework":"Python",
       "Communicator":"None",
       "Target":"cpu-compute",
       "MaxRunDurationSeconds":1200,
       "NodeCount":1,
       "Environment":{
          "Python":{
             "InterpreterPath":"python",
             "UserManagedDependencies":false,
             "CondaDependencies":{
                "name":"project_environment",
                "dependencies":[
                   "python=3.6.2",
                   {
                      "pip":[
                         "azureml-defaults"
                      ]
                   }
                ]
             }
          },
          "Docker":{
             "BaseImage":"mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04"
          }
      },
       "History":{
          "OutputCollection":true
       }
    }
}
```

Post these files to the server using `multipart/form-data` content:

```
curl https://{regional-api-server}/execution/v1.0/subscriptions/{your-subscription-id}/resourceGroups/{your-
resource-group}/providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-
name}/experiments/{your-experiment-name}/startrun?api-version=2019-11-01 \
  -X POST \
  -H "Content-Type: multipart/form-data" \
  -H "Authorization:Bearer {your-access-token}" \
  -F projectZipFile=@train.zip \
  -F runDefinitionFile=@runDefinition.json
```

A successful POST request will generate a `200 OK` status, with a response body containing the identifier of the created run:

```
{
  "runId": "my-first-experiment_1579642222877"
}
```

You can monitor a run using the REST-ful pattern that should now be familiar:

```
curl 'https://{regional-api-server}/history/v1.0/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-name}/experiments/{your-experiment-names}/runs/{your-run-id}?api-version=2019-11-01' \
  -H 'Authorization:Bearer {your-access-token}'
```

Delete resources you no longer need

Some, but not all, resources support the DELETE verb. Check the API Reference before committing to the REST API for deletion use-cases. To delete a model, for instance, you can use:

```
curl
  -X DELETE \
'https://{regional-api-server}/modelmanagement/v1.0/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/providers/Microsoft.MachineLearningServices/workspaces/{your-workspace-name}/models/{your-model-id}?api-version=2019-11-01' \
  -H 'Authorization:Bearer {your-access-token}'
```

Use REST to score a deployed model

While it's possible to deploy a model so that it authenticates with a service principal, most client-facing deployments use key-based authentication. You can find the appropriate key in your deployment's page within the **Endpoints** tab of Studio. The same location will show your endpoint's scoring URI. Your model's inputs must be modeled as a JSON array named `data`:

```
curl 'https://{scoring-uri}' \
 -H 'Authorization:Bearer {your-key}' \
 -H 'Content-Type: application/json' \
  -d '{ "data" : [ {model-specific-data-structure} ] }
```

Create a workspace using REST

Every Azure ML workspace has a dependency on four other Azure resources: a container registry with administration enabled, a key vault, an Application Insights resource, and a storage account. You cannot create a workspace until these resources exist. Consult the REST API reference for the details of creating each such resource.

To create a workspace, PUT a call similar to the following to `management.azure.com`. While this call requires you to set a large number of variables, it's structurally identical to other calls that this article has discussed.

```
curl -X PUT \
  'https://management.azure.com/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}\
/providers/Microsoft.MachineLearningServices/workspaces/{your-new-workspace-name}?api-version=2019-11-01' \
  -H 'Authorization: Bearer {your-access-token}' \
  -H 'Content-Type: application/json' \
  -d '{
    "location": "{desired-region}",
    "properties": {
        "friendlyName" : "{your-workspace-friendly-name}",
        "description" : "{your-workspace-description}",
        "containerRegistry" : "/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/\
providers/Microsoft.ContainerRegistry/registries/{your-registry-name}",
        keyVault" : "/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}\
/providers/Microsoft.Keyvault/vaults/{your-keyvault-name}",
        "applicationInsights" : "subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/\
providers/Microsoft.insights/components/{your-application-insights-name}",
        "storageAccount" : "/subscriptions/{your-subscription-id}/resourceGroups/{your-resource-group}/\
providers/Microsoft.Storage/storageAccounts/{your-storage-account-name}"
    },
    "identity" : {
        "type" : "systemAssigned"
    }
}'
```

You should receive a `202 Accepted` response and, in the returned headers, a `Location` URI. You can GET this URI for information on the deployment, including helpful debugging information if there is a problem with one of your dependent resources (for instance, if you forgot to enable admin access on your container registry).

Troubleshooting

Resource provider errors

When creating an Azure Machine Learning workspace, or a resource used by the workspace, you may receive an error similar to the following messages:

- `No registered resource provider found for location {location}`
- `The subscription is not registered to use namespace {resource-provider-namespace}`

Most resource providers are automatically registered, but not all. If you receive this message, you need to register the provider mentioned.

For information on registering resource providers, see Resolve errors for resource provider registration.

Moving the workspace

> **WARNING**
> Moving your Azure Machine Learning workspace to a different subscription, or moving the owning subscription to a new tenant, is not supported. Doing so may cause errors.

Deleting the Azure Container Registry

The Azure Machine Learning workspace uses Azure Container Registry (ACR) for some operations. It will automatically create an ACR instance when it first needs one.

> **WARNING**
> Once an Azure Container Registry has been created for a workspace, do not delete it. Doing so will break your Azure Machine Learning workspace.

Next steps

- Explore the complete AzureML REST API reference.
- Learn how to use Studio & Designer to Predict automobile price with the designer (preview).
- Explore Azure Machine Learning with Jupyter notebooks.

4/20/2020 • 9 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Use an Azure Resource Manager template to create a workspace for Azure Machine Learning

In this article, you learn several ways to create an Azure Machine Learning workspace using Azure Resource Manager templates. A Resource Manager template makes it easy to create resources as a single, coordinated operation. A template is a JSON document that defines the resources that are needed for a deployment. It may also specify deployment parameters. Parameters are used to provide input values when using the template.

For more information, see Deploy an application with Azure Resource Manager template.

Prerequisites

- An **Azure subscription**. If you do not have one, try the free or paid version of Azure Machine Learning.
- To use a template from a CLI, you need either Azure PowerShell or the Azure CLI.

Resource Manager template

The following Resource Manager template can be used to create an Azure Machine Learning workspace and associated Azure resources:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "workspaceName": {
      "type": "string",
      "metadata": {
        "description": "Specifies the name of the Azure Machine Learning service workspace."
      }
    },
    "location": {
      "type": "string",
      "allowedValues": [
        "australiaeast",
        "brazilsouth",
        "canadacentral",
        "centralus",
        "eastasia",
        "eastus",
        "eastus2",
        "francecentral",
        "japaneast",
        "koreacentral",
        "northcentralus",
        "northeurope",
        "southeastasia",
        "southcentralus",
        "uksouth",
        "westcentralus",
```

```
        "westus",
        "westus2",
        "westeurope"
      ],
      "metadata": {
        "description": "Specifies the location for all resources."
      }
    },
    "sku":{
      "type": "string",
      "defaultValue": "basic",
        "allowedValues": [
          "basic",
          "enterprise"
        ],
        "metadata": {
          "description": "Specifies the sku, also referred as 'edition' of the Azure Machine Learning
workspace."
        }
    }
  },
  "variables": {
    "storageAccountName": "[concat('sa',uniqueString(resourceGroup().id))]",
    "storageAccountType": "Standard_LRS",
    "keyVaultName": "[concat('kv',uniqueString(resourceGroup().id))]",
    "tenantId": "[subscription().tenantId]",
    "applicationInsightsName": "[concat('ai',uniqueString(resourceGroup().id))]"
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-04-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[variables('storageAccountType')]"
      },
      "kind": "StorageV2",
      "properties": {
        "encryption": {
          "services": {
            "blob": {
              "enabled": true
            },
            "file": {
              "enabled": true
            }
          },
          "keySource": "Microsoft.Storage"
        },
        "supportsHttpsTrafficOnly": true
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2018-02-14",
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": [
        ]
      }
    },
    {
```

```
      "type": "Microsoft.Insights/components",
      "apiVersion": "2018-05-01-preview",
      "name": "[variables('applicationInsightsName')]",
      "location": "
[if(or(equals(parameters('location'),'eastus2'),equals(parameters('location'),'westcentralus')),'southcentralu
s',parameters('location'))]",
      "kind": "web",
      "properties": {
        "Application_Type": "web"
      }
    },
    {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2019-11-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
        "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]"
      ],
      "identity": {
        "type": "systemAssigned"
      },
      "sku": {
        "tier": "[parameters('sku')]",
        "name": "[parameters('sku')]"
      },
      "properties": {
        "friendlyName": "[parameters('workspaceName')]",
        "keyVault": "[resourceId('Microsoft.KeyVault/vaults',variables('keyVaultName'))]",
        "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
        "storageAccount": "[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]"
      }
    }
  ]
}
```

This template creates the following Azure services:

- Azure Resource Group
- Azure Storage Account
- Azure Key Vault
- Azure Application Insights
- Azure Container Registry
- Azure Machine Learning workspace

The resource group is the container that holds the services. The various services are required by the Azure Machine Learning workspace.

The example template has two parameters:

- The **location** where the resource group and services will be created.

  The template will use the location you select for most resources. The exception is the Application Insights service, which is not available in all of the locations that the other services are. If you select a location where it is not available, the service will be created in the South Central US location.

- The **workspace name**, which is the friendly name of the Azure Machine Learning workspace.

> **NOTE**
> The workspace name is case-insensitive.

The names of the other services are generated randomly.

> **TIP**
> While the template associated with this document creates a new Azure Container Registry, you can also create a new workspace without creating a container registry. One will be created when you perform an operation that requires a container registry. For example, training or deploying a model.
>
> You can also reference an existing container registry or storage account in the Azure Resource Manager template, instead of creating a new one.

> **WARNING**
> Once an Azure Container Registry has been created for a workspace, do not delete it. Doing so will break your Azure Machine Learning workspace.

For more information on templates, see the following articles:

- Author Azure Resource Manager templates
- Deploy an application with Azure Resource Manager templates
- Microsoft.MachineLearningServices resource types

Advanced template

The following example template demonstrates how to create a workspace with three settings:

- Enable high confidentiality settings for the workspace
- Enable encryption for the workspace
- Uses an existing Azure Key Vault to retrieve customer-managed keys

For more information, see Encryption at rest.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "workspaceName": {
      "type": "string",
      "metadata": {
        "description": "Specifies the name of the Azure Machine Learning workspace."
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "southcentralus",
      "allowedValues": [
        "eastus",
        "eastus2",
        "southcentralus",
        "southeastasia",
        "westcentralus",
        "westeurope",
        "westus2"
      ],
      "metadata": {
```

```
        "description": "Specifies the location for all resources."
      }
    },
    "sku":{
      "type": "string",
      "defaultValue": "basic",
      "allowedValues": [
        "basic",
        "enterprise"
      ],
      "metadata": {
        "description": "Specifies the sku, also referred to as 'edition' of the Azure Machine Learning
workspace."
      }
    },
    "high_confidentiality":{
      "type": "string",
      "defaultValue": "false",
      "allowedValues": [
        "false",
        "true"
      ],
      "metadata": {
        "description": "Specifies that the Azure Machine Learning workspace holds highly confidential data."
      }
    },
    "encryption_status":{
      "type": "string",
      "defaultValue": "Disabled",
      "allowedValues": [
        "Enabled",
        "Disabled"
      ],
      "metadata": {
        "description": "Specifies if the Azure Machine Learning workspace should be encrypted with the
customer managed key."
      }
    },
    "cmk_keyvault":{
      "type": "string",
      "metadata": {
        "description": "Specifies the customer managed keyvault Resource Manager ID."
      }
    },
    "resource_cmk_uri":{
      "type": "string",
      "metadata": {
        "description": "Specifies the customer managed keyvault key uri."
      }
    }
  },
  "variables": {
    "storageAccountName": "[concat('sa',uniqueString(resourceGroup().id))]",
    "storageAccountType": "Standard_LRS",
    "keyVaultName": "[concat('kv',uniqueString(resourceGroup().id))]",
    "tenantId": "[subscription().tenantId]",
    "applicationInsightsName": "[concat('ai',uniqueString(resourceGroup().id))]",
    "containerRegistryName": "[concat('cr',uniqueString(resourceGroup().id))]"
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2018-07-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[variables('storageAccountType')]"
      },
      "kind": "StorageV2",
```

```
      "properties": {
        "encryption": {
          "services": {
            "blob": {
              "enabled": true
            },
            "file": {
              "enabled": true
            }
          },
          "keySource": "Microsoft.Storage"
        },
        "supportsHttpsTrafficOnly": true
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2018-02-14",
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": []
      }
    },
    {
      "type": "Microsoft.Insights/components",
      "apiVersion": "2015-05-01",
      "name": "[variables('applicationInsightsName')]",
      "location": "
[if(or(equals(parameters('location'),'eastus2'),equals(parameters('location'),'westcentralus')),'southcentralu
s',parameters('location'))]",
      "kind": "web",
      "properties": {
        "Application_Type": "web"
      }
    },
    {
      "type": "Microsoft.ContainerRegistry/registries",
      "apiVersion": "2017-10-01",
      "name": "[variables('containerRegistryName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "Standard"
      },
      "properties": {
        "adminUserEnabled": true
      }
    },
    {
      "type": "Microsoft.MachineLearningServices/workspaces",
      "apiVersion": "2020-01-01",
      "name": "[parameters('workspaceName')]",
      "location": "[parameters('location')]",
      "dependsOn": [
        "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
        "[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",
        "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]",
        "[resourceId('Microsoft.ContainerRegistry/registries', variables('containerRegistryName'))]"
      ],
      "identity": {
        "type": "systemAssigned"
      },
      "sku": {
            "tier": "[parameters('sku')]",
```

```
            "name": "[parameters('sku')]"
        },
        "properties": {
          "friendlyName": "[parameters('workspaceName')]",
          "keyVault": "[resourceId('Microsoft.KeyVault/vaults',variables('keyVaultName'))]",
          "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
          "containerRegistry": "
[resourceId('Microsoft.ContainerRegistry/registries',variables('containerRegistryName'))]",
          "storageAccount": "
[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]",
           "encryption": {
                  "status": "[parameters('encryption_status')]",
                  "keyVaultProperties": {
                      "keyVaultArmId": "[parameters('cmk_keyvault')]",
                      "keyIdentifier": "[parameters('resource_cmk_uri')]"
                    }
              },
          "hbiWorkspace": "[parameters('high_confidentiality')]"
        }
      }
    ]
}
```

To get the ID of the Key Vault, and the key URI needed by this template, you can use the Azure CLI. The following command gets the Key Vault ID:

```
az keyvault show --name mykeyvault --resource-group myresourcegroup --query "id"
```

This command returns a value similar to `"/subscriptions/{subscription-guid}/resourceGroups/myresourcegroup/providers/Microsoft.KeyVault/vaults/mykeyvault"`.

To get the URI for the customer managed key, use the following command:

```
az keyvault key show --vault-name mykeyvault --name mykey --query "key.kid"
```

This command returns a value similar to `"https://mykeyvault.vault.azure.net/keys/mykey/{guid}"`.

> **IMPORTANT**
> Once a workspace has been created, you cannot change the settings for confidential data, encryption, key vault ID, or key identifiers. To change these values, you must create a new workspace using the new values.

Use the Azure portal

1. Follow the steps in Deploy resources from custom template. When you arrive at the **Edit template** screen, paste in the template from this document.

2. Select **Save** to use the template. Provide the following information and agree to the listed terms and conditions:

   - Subscription: Select the Azure subscription to use for these resources.
   - Resource group: Select or create a resource group to contain the services.
   - Workspace name: The name to use for the Azure Machine Learning workspace that will be created. The workspace name must be between 3 and 33 characters. It may only contain alphanumeric characters and

'-'.

- Location: Select the location where the resources will be created.

For more information, see Deploy resources from custom template.

Use Azure PowerShell

This example assumes that you have saved the template to a file named `azuredeploy.json` in the current directory:

```
New-AzResourceGroup -Name examplegroup -Location "East US"
new-azresourcegroupdeployment -name exampledeployment `
  -resourcegroupname examplegroup -location "East US" `
  -templatefile .\azuredeploy.json -workspaceName "exampleworkspace" -sku "basic"
```

For more information, see Deploy resources with Resource Manager templates and Azure PowerShell and Deploy private Resource Manager template with SAS token and Azure PowerShell.

Use the Azure CLI

This example assumes that you have saved the template to a file named `azuredeploy.json` in the current directory:

```
az group create --name examplegroup --location "East US"
az group deployment create \
  --name exampledeployment \
  --resource-group examplegroup \
  --template-file azuredeploy.json \
  --parameters workspaceName=exampleworkspace location=eastus sku=basic
```

For more information, see Deploy resources with Resource Manager templates and Azure CLI and Deploy private Resource Manager template with SAS token and Azure CLI.

Troubleshooting

Resource provider errors

When creating an Azure Machine Learning workspace, or a resource used by the workspace, you may receive an error similar to the following messages:

- `No registered resource provider found for location {location}`
- `The subscription is not registered to use namespace {resource-provider-namespace}`

Most resource providers are automatically registered, but not all. If you receive this message, you need to register the provider mentioned.

For information on registering resource providers, see Resolve errors for resource provider registration.

Azure Key Vault access policy and Azure Resource Manager templates

When you use an Azure Resource Manager template to create the workspace and associated resources (including Azure Key Vault), multiple times. For example, using the template multiple times with the same parameters as part of a continuous integration and deployment pipeline.

Most resource creation operations through templates are idempotent, but Key Vault clears the access policies each time the template is used. Clearing the access policies breaks access to the Key Vault for any existing workspace that is using it. For example, Stop/Create functionalities of Azure Notebooks VM may fail.

To avoid this problem, we recommend one of the following approaches:

- Do not deploy the template more than once for the same parameters. Or delete the existing resources before using the template to recreate them.
- Examine the Key Vault access policies and then use these policies to set the `accessPolicies` property of the template. To view the access policies, use the following Azure CLI command:

  ```
  az keyvault show --name mykeyvault --resource-group myresourcegroup --query properties.accessPolicies
  ```

  For more information on using the `accessPolicies` section of the template, see the AccessPolicyEntry object reference.
- Check if the Key Vault resource already exists. If it does, do not recreate it through the template. For example, to use the existing Key Vault instead of creating a new one, make the following changes to the template:
  - **Add** a parameter that accepts the ID of an existing Key Vault resource:

    ```
    "keyVaultId":{
      "type": "string",
      "metadata": {
        "description": "Specify the existing Key Vault ID."
      }
    }
    ```
  - **Remove** the section that creates a Key Vault resource:

    ```
    {
      "type": "Microsoft.KeyVault/vaults",
      "apiVersion": "2018-02-14",
      "name": "[variables('keyVaultName')]",
      "location": "[parameters('location')]",
      "properties": {
        "tenantId": "[variables('tenantId')]",
        "sku": {
          "name": "standard",
          "family": "A"
        },
        "accessPolicies": [
        ]
      }
    },
    ```
  - **Remove** the `"[resourceId('Microsoft.KeyVault/vaults', variables('keyVaultName'))]",` line from the `dependsOn` section of the workspace. Also **Change** the `keyVault` entry in the `properties` section of the workspace to reference the `keyVaultId` parameter:

```
{
  "type": "Microsoft.MachineLearningServices/workspaces",
  "apiVersion": "2019-11-01",
  "name": "[parameters('workspaceName')]",
  "location": "[parameters('location')]",
  "dependsOn": [
    "[resourceId('Microsoft.Storage/storageAccounts', variables('storageAccountName'))]",
    "[resourceId('Microsoft.Insights/components', variables('applicationInsightsName'))]"
  ],
  "identity": {
    "type": "systemAssigned"
  },
  "sku": {
    "tier": "[parameters('sku')]",
    "name": "[parameters('sku')]"
  },
  "properties": {
    "friendlyName": "[parameters('workspaceName')]",
    "keyVault": "[parameters('keyVaultId')]",
    "applicationInsights": "
[resourceId('Microsoft.Insights/components',variables('applicationInsightsName'))]",
    "storageAccount": "
[resourceId('Microsoft.Storage/storageAccounts/',variables('storageAccountName'))]"
  }
}
```

After these changes, you can specify the ID of the existing Key Vault resource when running the template. The template will then reuse the Key Vault by setting the `keyVault` property of the workspace to its ID.

To get the ID of the Key Vault, you can reference the output of the original template run or use the Azure CLI. The following command is an example of using the Azure CLI to get the Key Vault resource ID:

```
az keyvault show --name mykeyvault --resource-group myresourcegroup --query id
```

This command returns a value similar to the following text:

```
/subscriptions/{subscription-
guid}/resourceGroups/myresourcegroup/providers/Microsoft.KeyVault/vaults/mykeyvault
```

Next steps

- Deploy resources with Resource Manager templates and Resource Manager REST API.
- Creating and deploying Azure resource groups through Visual Studio.

Configure a development environment for Azure Machine Learning

4/16/2020 • 12 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to configure a development environment to work with Azure Machine Learning. Azure Machine Learning is platform agnostic. The only hard requirement for your development environment is Python 3. An isolated environment like Anaconda or Virtualenv is also recommended.

The following table shows each development environment covered in this article, along with pros and cons.

ENVIRONMENT	PROS	CONS
Cloud-based Azure Machine Learning compute instance (preview)	Easiest way to get started. The entire SDK is already installed in your workspace VM, and notebook tutorials are pre-cloned and ready to run.	Lack of control over your development environment and dependencies. Additional cost incurred for Linux VM (VM can be stopped when not in use to avoid charges). See pricing details.
Local environment	Full control of your development environment and dependencies. Run with any build tool, environment, or IDE of your choice.	Takes longer to get started. Necessary SDK packages must be installed, and an environment must also be installed if you don't already have one.
Azure Databricks	Ideal for running large-scale intensive machine learning workflows on the scalable Apache Spark platform.	Overkill for experimental machine learning, or smaller-scale experiments and workflows. Additional cost incurred for Azure Databricks. See pricing details.
The Data Science Virtual Machine (DSVM)	Similar to the cloud-based compute instance (Python and the SDK are pre-installed), but with additional popular data science and machine learning tools pre-installed. Easy to scale and combine with other custom tools and workflows.	A slower getting started experience compared to the cloud-based compute instance.

This article also provides additional usage tips for the following tools:

- Jupyter Notebooks: If you're already using the Jupyter Notebook, the SDK has some extras that you should install.
- Visual Studio Code: If you use Visual Studio Code, the Azure Machine Learning extension includes extensive language support for Python as well as features to make working with the Azure Machine Learning much more convenient and productive.

Prerequisites

An Azure Machine Learning workspace. To create the workspace, see Create an Azure Machine Learning workspace. A workspace is all you need to get started with your own cloud-based notebook server, a DSVM, or Azure Databricks.

To install the SDK environment for your local computer, Jupyter Notebook server or Visual Studio Code you also need:

- Either the Anaconda or Miniconda package manager.
- On Linux or macOS, you need the bash shell.

  > **TIP**
  > If you're on Linux or macOS and use a shell other than bash (for example, zsh) you might receive errors when you run some commands. To work around this problem, use the `bash` command to start a new bash shell and run the commands there.

- On Windows, you need the command prompt or Anaconda prompt (installed by Anaconda and Miniconda).

Your own cloud-based compute instance

The Azure Machine Learning compute instance (preview) is a secure, cloud-based Azure workstation that provides data scientists with a Jupyter notebook server, JupyterLab, and a fully prepared ML environment.

There is nothing to install or configure for a compute instance. Create one anytime from within your Azure Machine Learning workspace. Provide just a name and specify an Azure VM type. Try it now with this Tutorial: Setup environment and workspace.

Learn more about compute instances.

To stop incurring compute charges, stop the compute instance.

Data Science Virtual Machine

The DSVM is a customized virtual machine (VM) image. It's designed for data science work that's pre-configured with:

- Packages such as TensorFlow, PyTorch, Scikit-learn, XGBoost, and the Azure Machine Learning SDK
- Popular data science tools such as Spark Standalone and Drill
- Azure tools such as the Azure CLI, AzCopy, and Storage Explorer
- Integrated development environments (IDEs) such as Visual Studio Code and PyCharm
- Jupyter Notebook Server

The Azure Machine Learning SDK works on either the Ubuntu or Windows version of the DSVM. But if you plan to use the DSVM as a compute target as well, only Ubuntu is supported.

To use the DSVM as a development environment:

1. Create a DSVM in either of the following environments:
   - The Azure portal:
     - Create an Ubuntu Data Science Virtual Machine
     - Create a Windows Data Science Virtual Machine
   - The Azure CLI:

> **IMPORTANT**
>
> - When you use the Azure CLI, you must first sign in to your Azure subscription by using the `az login` command.
> - When you use the commands in this step, you must provide a resource group name, a name for the VM, a username, and a password.

- To create an Ubuntu Data Science Virtual Machine, use the following command:

```
# create a Ubuntu DSVM in your resource group
# note you need to be at least a contributor to the resource group in order to execute
this command successfully
# If you need to create a new resource group use: "az group create --name YOUR-RESOURCE-
GROUP-NAME --location YOUR-REGION (For example: westus2)"
az vm create --resource-group YOUR-RESOURCE-GROUP-NAME --name YOUR-VM-NAME --image
microsoft-dsvm:linux-data-science-vm-ubuntu:linuxdsvmubuntu:latest --admin-username YOUR-
USERNAME --admin-password YOUR-PASSWORD --generate-ssh-keys --authentication-type
password
```

- To create a Windows Data Science Virtual Machine, use the following command:

```
# create a Windows Server 2016 DSVM in your resource group
# note you need to be at least a contributor to the resource group in order to execute
this command successfully
az vm create --resource-group YOUR-RESOURCE-GROUP-NAME --name YOUR-VM-NAME --image
microsoft-dsvm:dsvm-windows:server-2016:latest --admin-username YOUR-USERNAME --admin-
password YOUR-PASSWORD --authentication-type password
```

2. The Azure Machine Learning SDK is already installed on the DSVM. To use the Conda environment that contains the SDK, use one of the following commands:

   - For Ubuntu DSVM:

   ```
   conda activate py36
   ```

   - For Windows DSVM:

   ```
   conda activate AzureML
   ```

3. To verify that you can access the SDK and check the version, use the following Python code:

```
import azureml.core
print(azureml.core.VERSION)
```

4. To configure the DSVM to use your Azure Machine Learning workspace, see the Create a workspace configuration file section.

For more information, see Data Science Virtual Machines.

Local computer

When you're using a local computer (which might also be a remote virtual machine), create an Anaconda environment and install the SDK. Here's an example:

1. Download and install Anaconda (Python 3.7 version) if you don't already have it.

2. Open an Anaconda prompt and create an environment with the following commands:

   Run the following command to create the environment.

   ```
   conda create -n myenv python=3.6.5
   ```

   Then activate the environment.

   ```
   conda activate myenv
   ```

   This example creates an environment using python 3.6.5, but any specific subversions can be chosen. SDK compatibility may not be guaranteed with certain major versions (3.5+ is recommended), and it's recommended to try a different version/subversion in your Anaconda environment if you run into errors. It will take several minutes to create the environment while components and packages are downloaded.

3. Run the following commands in your new environment to enable environment-specific IPython kernels. This will ensure expected kernel and package import behavior when working with Jupyter Notebooks within Anaconda environments:

   ```
   conda install notebook ipykernel
   ```

   Then run the following command to create the kernel:

   ```
   ipython kernel install --user --name myenv --display-name "Python (myenv)"
   ```

4. Use the following commands to install packages:

   This command installs the base Azure Machine Learning SDK with notebook and `automl` extras. The `automl` extra is a large install, and can be removed from the brackets if you don't intend to run automated machine learning experiments. The `automl` extra also includes the Azure Machine Learning Data Prep SDK by default as a dependency.

   ```
   pip install azureml-sdk[notebooks,automl]
   ```

   NOTE

   - If you get a message that PyYAML can't be uninstalled, use the following command instead:

     `pip install --upgrade azureml-sdk[notebooks,automl] --ignore-installed PyYAML`

   - Starting with macOS Catalina, zsh (Z shell) is the default login shell and interactive shell. In zsh, use the following command which escapes brackets with "\" (backslash):

     `pip install --upgrade azureml-sdk\[notebooks,automl\]`

   It will take several minutes to install the SDK. For more information on installation options, see the install guide.

5. Install other packages for your machine learning experimentation.

   Use either of the following commands and replace *<new package>* with the package you want to install.

Installing packages via `conda install` requires that the package is part of the current channels (new channels can be added in Anaconda Cloud).

```
conda install <new package>
```

Alternatively, you can install packages via `pip`.

```
pip install <new package>
```

Jupyter Notebooks

Jupyter Notebooks are part of the Jupyter Project. They provide an interactive coding experience where you create documents that mix live code with narrative text and graphics. Jupyter Notebooks are also a great way to share your results with others, because you can save the output of your code sections in the document. You can install Jupyter Notebooks on a variety of platforms.

The procedure in the Local computer section installs necessary components for running Jupyter Notebooks in an Anaconda environment.

To enable these components in your Jupyter Notebook environment:

1. Open an Anaconda prompt and activate your environment.

   ```
   conda activate myenv
   ```

2. Clone the GitHub repository for a set of sample notebooks.

   ```
   git clone https://github.com/Azure/MachineLearningNotebooks.git
   ```

3. Launch the Jupyter Notebook server with the following command:

   ```
   jupyter notebook
   ```

4. To verify that Jupyter Notebook can use the SDK, create a **New** notebook, select **Python 3** as your kernel, and then run the following command in a notebook cell:

   ```
   import azureml.core
   azureml.core.VERSION
   ```

5. If you encounter issues importing modules and receive a `ModuleNotFoundError`, ensure your Jupyter kernel is connected to the correct path for your environment by running the following code in a Notebook cell.

   ```
   import sys
   sys.path
   ```

6. To configure the Jupyter Notebook to use your Azure Machine Learning workspace, go to the Create a workspace configuration file section.

Visual Studio Code

Visual Studio Code is a very popular cross platform code editor that supports an extensive set of programming languages and tools through extensions available in the Visual Studio marketplace. The Azure Machine Learning extension installs the Python extension for coding in all types of Python environments (virtual, Anaconda, etc.). In

addition, it provides convenience features for working with Azure Machine Learning resources and running Azure Machine Learning experiments all without leaving Visual Studio Code.

To use Visual Studio Code for development:

1. Install the Azure Machine Learning extension for Visual Studio Code, see Azure Machine Learning.

   For more information, see Use Azure Machine Learning for Visual Studio Code.

2. Learn how to use Visual Studio Code for any type of Python development, see Get started with Python in VSCode.

   - To select the SDK Python environment containing the SDK, open VS Code, and then select Ctrl+Shift+P (Linux and Windows) or Command+Shift+P (Mac).
     - The **Command Palette** opens.
   - Enter **Python: Select Interpreter**, and then select the appropriate environment

3. To validate that you can use the SDK, create a new Python file (.py) that contains the following code:

   ```
   #%%
   import azureml.core
   azureml.core.VERSION
   ```

   Run this code by clicking the "Run cell" CodeLens or simply press shift-enter.

Azure Databricks

Azure Databricks is an Apache Spark-based environment in the Azure cloud. It provides a collaborative Notebook-based environment with CPU or GPU-based compute cluster.

How Azure Databricks works with Azure Machine Learning:

- You can train a model using Spark MLlib and deploy the model to ACI/AKS from within Azure Databricks.
- You can also use automated machine learning capabilities in a special Azure ML SDK with Azure Databricks.
- You can use Azure Databricks as a compute target from an Azure Machine Learning pipeline.

Set up your Databricks cluster

Create a Databricks cluster. Some settings apply only if you install the SDK for automated machine learning on Databricks. **It will take few minutes to create the cluster.**

Use these settings:

SETTING	APPLIES TO	VALUE
Cluster name	always	yourclustername
Databricks Runtime	always	Non-ML Runtime 6.5 (scala 2.11, spark 2.4.3)
Python version	always	3
Workers	always	2 or higher
Worker node VM types (determines max # of concurrent iterations)	Automated ML only	Memory optimized VM preferred

SETTING	APPLIES TO	VALUE
Enable Autoscaling	Automated ML only	Uncheck

Wait until the cluster is running before proceeding further.

Install the correct SDK into a Databricks library

Once the cluster is running, create a library to attach the appropriate Azure Machine Learning SDK package to your cluster.

1. Right-click the current Workspace folder where you want to store the library. Select **Create** > **Library**.
2. Choose **only one** option (no other SDK installation are supported)

SDK PACKAGE EXTRAS	SOURCE	PYPI NAME
For Databricks	Upload Python Egg or PyPI	azureml-sdk[databricks]
For Databricks -with- automated ML capabilities	Upload Python Egg or PyPI	azureml-sdk[automl]

> **WARNING**
> No other SDK extras can be installed. Choose only one of the preceding options [databricks] or [automl].

   - Do not select **Attach automatically to all clusters**.
   - Select **Attach** next to your cluster name.
3. Monitor for errors until status changes to **Attached**, which may take several minutes. If this step fails:

   Try restarting your cluster by:

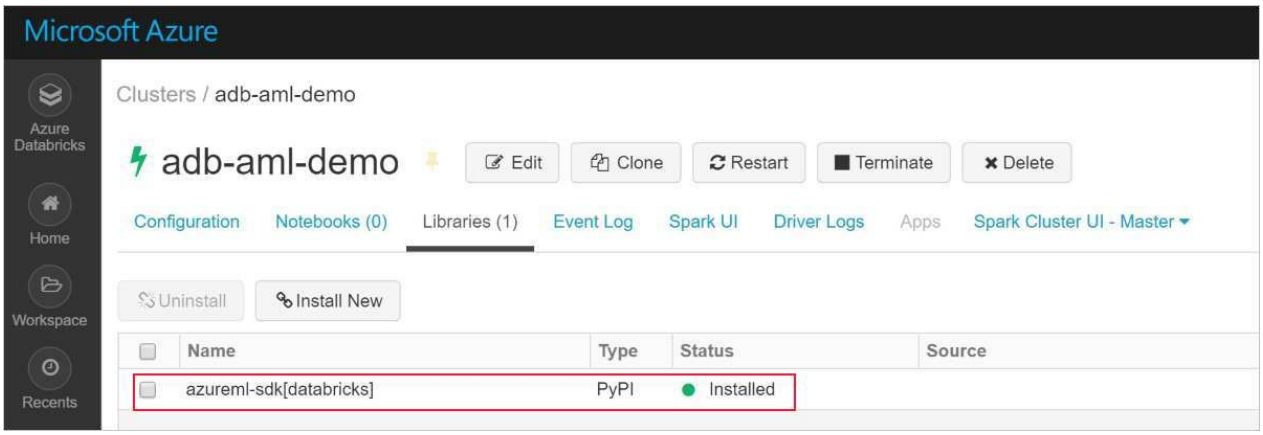
   a. In the left pane, select **Clusters**.
   b. In the table, select your cluster name.
   c. On the **Libraries** tab, select **Restart**.

   Also consider:

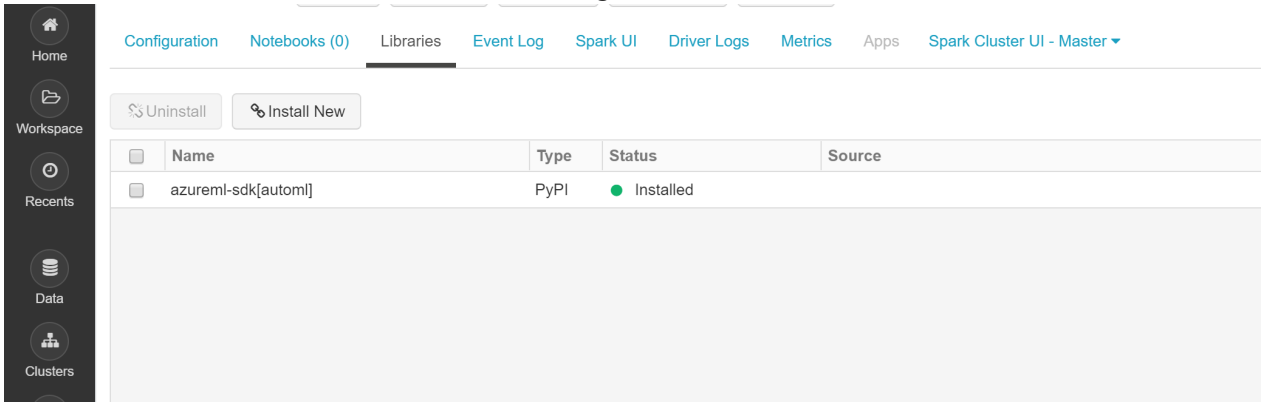
   - In AutoML config, when using Azure Databricks add the following parameters:
     a. `max_concurrent_iterations` is based on number of worker nodes in your cluster.
     b. `spark_context=sc` is based on the default spark context.
   - Or, if you have an old SDK version, deselect it from cluster's installed libs and move to trash. Install the new SDK version and restart the cluster. If there is an issue after the restart, detach and reattach your cluster.

If install was successful, the imported library should look like one of these:

SDK for Databricks ***without*** automated machine learning



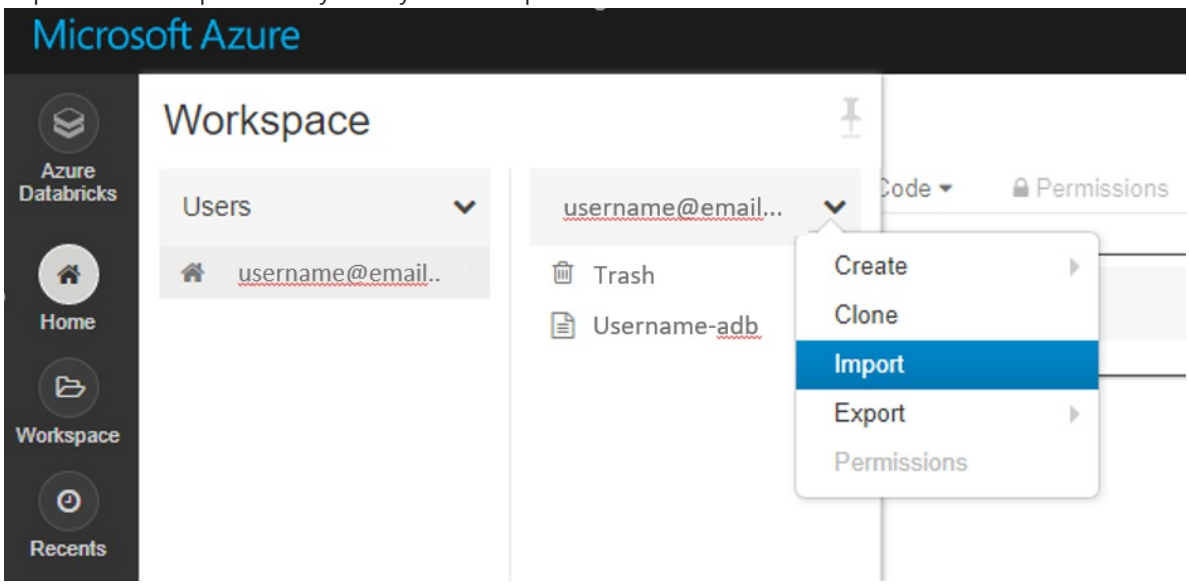
SDK for Databricks **WITH** automated machine learning

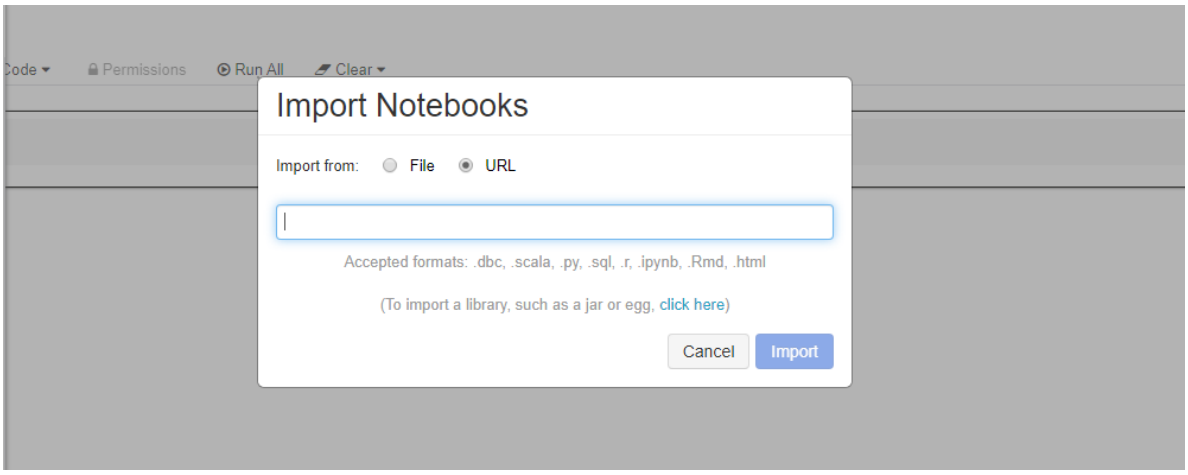


Start exploring

Try it out:

- While many sample notebooks are available, **only these sample notebooks work with Azure Databricks.**
- Import these samples directly from your workspace. See below:





- Learn how to create a pipeline with Databricks as the training compute.

Create a workspace configuration file

The workspace configuration file is a JSON file that tells the SDK how to communicate with your Azure Machine Learning workspace. The file is named *config.json*, and it has the following format:

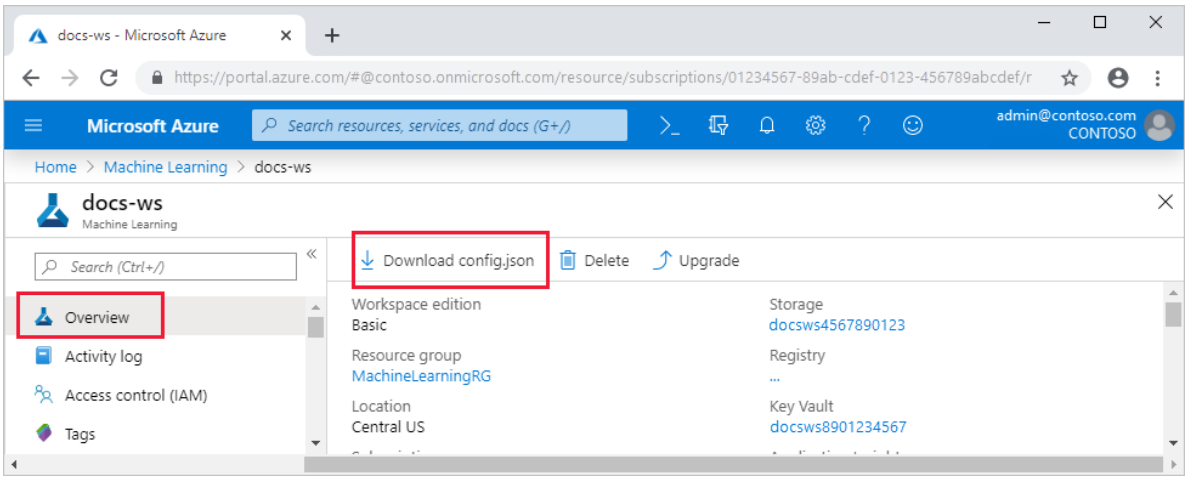
```
{
    "subscription_id": "<subscription-id>",
    "resource_group": "<resource-group>",
    "workspace_name": "<workspace-name>"
}
```

This JSON file must be in the directory structure that contains your Python scripts or Jupyter Notebooks. It can be in the same directory, a subdirectory named *.azureml*, or in a parent directory.

To use this file from your code, use `ws=Workspace.from_config()`. This code loads the information from the file and connects to your workspace.

You can create the configuration file in three ways:

- **Use ws.write_config**: to write a *config.json* file. The file contains the configuration information for your workspace. You can download or copy the *config.json* to other development environments.
- **Download the file**: In the Azure portal, select **Download config.json** from the **Overview** section of your workspace.



- **Create the file programmatically**: In the following code snippet, you connect to a workspace by providing the subscription ID, resource group, and workspace name. It then saves the workspace configuration to the file:

```
from azureml.core import Workspace

subscription_id = '<subscription-id>'
resource_group  = '<resource-group>'
workspace_name  = '<workspace-name>'

try:
    ws = Workspace(subscription_id = subscription_id, resource_group = resource_group, workspace_name
= workspace_name)
    ws.write_config()
    print('Library configuration succeeded')
except:
    print('Workspace not found')
```

This code writes the configuration file to the *.azureml/config.json* file.

Next steps

- Train a model on Azure Machine Learning with the MNIST dataset
- View the Azure Machine Learning SDK for Python reference

Reuse environments for training and deployment by using Azure Machine Learning

4/17/2020 • 12 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, learn how to create and manage Azure Machine Learning environments. Use the environments to track and reproduce your projects' software dependencies as they evolve.

Software dependency management is a common task for developers. You want to ensure that builds are reproducible without extensive manual software configuration. The Azure Machine Learning `Environment` class accounts for local development solutions such as pip and Conda, and it provides a solution for both local and distributed cloud development.

The examples in this article show how to:

- Create an environment and specify package dependencies.
- Retrieve and update environments.
- Use an environment for training.
- Use an environment for web service deployment.

For a high-level overview of how environments work in Azure Machine Learning, see What are ML environments?.

Prerequisites

- The Azure Machine Learning SDK for Python
- An Azure Machine Learning workspace

Create an environment

The following sections explore the multiple ways that you can create an environment for your experiments.

Use a curated environment

You can select one of the curated environments to start with:

- The *AzureML-Minimal* environment contains a minimal set of packages to enable run tracking and asset uploading. You can use it as a starting point for your own environment.
- The *AzureML-Tutorial* environment contains common data science packages. These packages include Scikit-Learn, Pandas, Matplotlib, and a larger set of azureml-sdk packages.

Curated environments are backed by cached Docker images. This backing reduces the run preparation cost.

Use the `Environment.get` method to select one of the curated environments:

```
from azureml.core import Workspace, Environment

ws = Workspace.from_config()
env = Environment.get(workspace=ws, name="AzureML-Minimal")
```

You can list the curated environments and their packages by using the following code:

```
envs = Environment.list(workspace=ws)

for env in envs:
    if env.startswith("AzureML"):
        print("Name",env)
        print("packages", envs[env].python.conda_dependencies.serialize_to_string())
```

> **WARNING**
> Don't start your own environment name with the *AzureML* prefix. This prefix is reserved for curated environments.

Instantiate an environment object

To manually create an environment, import the `Environment` class from the SDK. Then use the following code to instantiate an environment object.

```
from azureml.core.environment import Environment
Environment(name="myenv")
```

Use Conda and pip specification files

You can also create an environment from a Conda specification or a pip requirements file. Use the `from_conda_specification()` method or the `from_pip_requirements()` method. In the method argument, include your environment name and the file path of the file that you want.

```
# From a Conda specification file
myenv = Environment.from_conda_specification(name = "myenv",
                                             file_path = "path-to-conda-specification-file")

# From a pip requirements file
myenv = Environment.from_pip_requirements(name = "myenv"
                                          file_path = "path-to-pip-requirements-file")
```

Use existing Conda environments

If you have an existing Conda environment on your local computer, then you can use the service to create an environment object. By using this strategy, you can reuse your local interactive environment on remote runs.

The following code creates an environment object from the existing Conda environment `mycondaenv`. It uses the `from_existing_conda_environment()` method.

```
myenv = Environment.from_existing_conda_environment(name = "myenv",
                                                    conda_environment_name = "mycondaenv")
```

Create environments automatically

Automatically create an environment by submitting a training run. Submit the run by using the `submit()` method. When you submit a training run, the building of the new environment can take several minutes. The build duration depends on the size of the required dependencies.

If you don't specify an environment in your run configuration before you submit the run, then a default environment is created for you.

```
from azureml.core import ScriptRunConfig, Experiment, Environment
# Create experiment
myexp = Experiment(workspace=ws, name = "environment-example")

# Attach training script and compute target to run config
runconfig = ScriptRunConfig(source_directory=".", script="example.py")
runconfig.run_config.target = "local"

# Submit the run
run = myexp.submit(config=runconfig)

# Show each step of run
run.wait_for_completion(show_output=True)
```

Similarly, if you use an `Estimator` object for training, you can directly submit the estimator instance as a run without specifying an environment. The `Estimator` object already encapsulates the environment and the compute target.

Add packages to an environment

Add packages to an environment by using Conda, pip, or private wheel files. Specify each package dependency by using the `CondaDependency` class. Add it to the environment's `PythonSection`.

Conda and pip packages

If a package is available in a Conda package repository, then we recommend that you use the Conda installation rather than the pip installation. Conda packages typically come with prebuilt binaries that make installation more reliable.

The following example adds to the environment. It adds version 1.17.0 of `numpy`. It also adds the `pillow` package, `myenv`. The example uses the `add_conda_package()` method and the `add_pip_package()` method, respectively.

```
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies

myenv = Environment(name="myenv")
conda_dep = CondaDependencies()

# Installs numpy version 1.17.0 conda package
conda_dep.add_conda_package("numpy==1.17.0")

# Installs pillow package
conda_dep.add_pip_package("pillow")

# Adds dependencies to PythonSection of myenv
myenv.python.conda_dependencies=conda_dep
```

IMPORTANT

If you use the same environment definition for another run, the Azure Machine Learning service reuses the cached image of your environment. If you create an environment with an unpinned package dependency, for example `numpy`, that environment will keep using the package version installed *at the time of environment creation*. Also, any future environment with matching definition will keep using the old version. For more information, see Environment building, caching, and reuse.

Private wheel files

You can use private pip wheel files by first uploading them to your workspace storage. You upload them by using

a static `add_private_pip_wheel()` method. Then you capture the storage URL and pass the URL to the `add_pip_package()` method.

```
# During environment creation the service replaces the URL by secure SAS URL, so your wheel file is kept
private and secure
whl_url = Environment.add_private_pip_wheel(workspace=ws,file_path = "my-custom.whl")
myenv = Environment(name="myenv")
conda_dep = CondaDependencies()
conda_dep.add_pip_package(whl_url)
myenv.python.conda_dependencies=conda_dep
```

Manage environments

Manage environments so that you can update, track, and reuse them across compute targets and with other users of the workspace.

Register environments

The environment is automatically registered with your workspace when you submit a run or deploy a web service. You can also manually register the environment by using the `register()` method. This operation makes the environment into an entity that's tracked and versioned in the cloud. The entity can be shared between workspace users.

The following code registers the `myenv` environment to the `ws` workspace.

```
myenv.register(workspace=ws)
```

When you use the environment for the first time in training or deployment, it's registered with the workspace. Then it's built and deployed on the compute target. The service caches the environments. Reusing a cached environment takes much less time than using a new service or one that has been updated.

Get existing environments

The `Environment` class offers methods that allow you to retrieve existing environments in your workspace. You can retrieve environments by name, as a list, or by a specific training run. This information is helpful for troubleshooting, auditing, and reproducibility.

View a list of environments

View the environments in your workspace by using the `Environment.list(workspace="workspace_name")` class. Then select an environment to reuse.

Get an environment by name

You can also get a specific environment by name and version. The following code uses the `get()` method to retrieve version `1` of the `myenv` environment on the `ws` workspace.

```
restored_environment = Environment.get(workspace=ws,name="myenv",version="1")
```

Train a run-specific environment

To get the environment that was used for a specific run after the training finishes, use the `get_environment()` method in the `Run` class.

```
from azureml.core import Run
Run.get_environment()
```

Update an existing environment

Say you change an existing environment, for example, by adding a Python package. A new version of the environment is then created when you submit a run, deploy a model, or manually register the environment. The versioning allows you to view the environment's changes over time.

To update a Python package version in an existing environment, specify the version number for that package. If you don't use the exact version number, then Azure Machine Learning will reuse the existing environment with its original package versions.

Debug the image build

The following example uses the `build()` method to manually create an environment as a Docker image. It monitors the output logs from the image build by using `wait_for_completion()`. The built image then appears in the workspace's Azure Container Registry instance. This information is helpful for debugging.

```
from azureml.core import Image
build = env.build(workspace=ws)
build.wait_for_completion(show_output=True)
```

Enable Docker

The `DockerSection` of the Azure Machine Learning `Environment` class allows you to finely customize and control the guest operating system on which you run your training.

When you enable Docker, the service builds a Docker image. It also creates a Python environment that uses your specifications within that Docker container. This functionality provides additional isolation and reproducibility for your training runs.

```
# Creates the environment inside a Docker container.
myenv.docker.enabled = True
```

By default, the newly built Docker image appears in the container registry that's associated with the workspace. The repository name has the form *azureml/azureml_<uuid>*. The unique identifier (*uuid*) part of the name corresponds to a hash that's computed from the environment configuration. This correspondence allows the service to determine whether an image for the given environment already exists for reuse.

Additionally, the service automatically uses one of the Ubuntu Linux-based base images. It installs the specified Python packages. The base image has CPU versions and GPU versions. Azure Machine Learning automatically detects which version to use.

```
# Specify custom Docker base image and registry, if you don't want to use the defaults
myenv.docker.base_image="your_base-image"
myenv.docker.base_image_registry="your_registry_location"
```

You can also specify a custom Dockerfile. It's simplest to start from one of Azure Machine Learning base images using Docker `FROM` command, and then add your own custom steps. Use this approach if you need to install non-Python packages as dependencies.

```
# Specify docker steps as a string. Alternatively, load the string from a file.
dockerfile = r"""
FROM mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04
RUN echo "Hello from custom container!"
"""

# Set base image to None, because the image is defined by dockerfile.
myenv.docker.base_image = None
myenv.docker.base_dockerfile = dockerfile
```

Use user-managed dependencies

In some situations, your custom base image may already contain a Python environment with packages that you want to use.

By default, Azure Machine Learning service will build a Conda environment with dependencies you specified, and will execute the run in that environment instead of using any Python libraries that you installed on the base image.

To use your own installed packages, set the parameter `Environment.python.user_managed_dependencies = True`. Ensure that the base image contains a Python interpreter, and has the packages your training script needs.

For example, to run in a base Miniconda environment that has NumPy package installed, first specify a Dockerfile with a step to install the package. Then set the user-managed dependencies to `True`.

You can also specify a path to a specific Python interpreter within the image, by setting the `Environment.python.interpreter_path` variable.

```
dockerfile = """
FROM mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04
RUN conda install numpy
"""

myenv.docker.base_image = None
myenv.docker.base_dockerfile = dockerfile
myenv.python.user_managed_dependencies=True
myenv.python.interpreter_path = "/opt/miniconda/bin/python"
```

Use environments for training

To submit a training run, you need to combine your environment, compute target, and your training Python script into a run configuration. This configuration is a wrapper object that's used for submitting runs.

When you submit a training run, the building of a new environment can take several minutes. The duration depends on the size of the required dependencies. The environments are cached by the service. So as long as the environment definition remains unchanged, you incur the full setup time only once.

The following local script run example shows where you would use `ScriptRunConfig` as your wrapper object.

```
from azureml.core import ScriptRunConfig, Experiment
from azureml.core.environment import Environment

exp = Experiment(name="myexp", workspace = ws)
# Instantiate environment
myenv = Environment(name="myenv")

# Add training script to run config
runconfig = ScriptRunConfig(source_directory=".", script="train.py")

# Attach compute target to run config
runconfig.run_config.target = "local"

# Attach environment to run config
runconfig.run_config.environment = myenv

# Submit run
run = exp.submit(runconfig)
```

> **NOTE**
> To disable the run history or run snapshots, use the setting under `ScriptRunConfig.run_config.history`.

If you don't specify the environment in your run configuration, then the service creates a default environment when you submit your run.

Use an estimator for training

If you use an estimator for training, then you can submit the estimator instance directly. It already encapsulates the environment and the compute target.

The following code uses an estimator for a single-node training run. It runs on a remote compute for a `scikit-learn` model. It assumes that you previously created a compute target object, `compute_target`, and a datastore object, `ds`.

```
from azureml.train.estimator import Estimator

script_params = {
    '--data-folder': ds.as_mount(),
    '--regularization': 0.8
}

sk_est = Estimator(source_directory='./my-sklearn-proj',
                   script_params=script_params,
                   compute_target=compute_target,
                   entry_script='train.py',
                   conda_packages=['scikit-learn'])

# Submit the run
run = experiment.submit(sk_est)
```

Use environments for web service deployment

You can use environments when you deploy your model as a web service. This capability enables a reproducible, connected workflow. In this workflow, you can train, test, and deploy your model by using the same libraries in both your training compute and your inference compute.

To deploy a web service, combine the environment, inference compute, scoring script, and registered model in your deployment object, `deploy()`. For more information, see How and where to deploy models.

In this example, assume that you've completed a training run. Now you want to deploy that model to Azure Container Instances. When you build the web service, the model and scoring files are mounted on the image, and the Azure Machine Learning inference stack is added to the image.

```
from azureml.core.model import InferenceConfig, Model
from azureml.core.webservice import AciWebservice, Webservice

# Register the model to deploy
model = run.register_model(model_name = "mymodel", model_path = "outputs/model.pkl")

# Combine scoring script & environment in Inference configuration
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)

# Set deployment configuration
deployment_config = AciWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)

# Define the model, inference, & deployment configuration and web service name and location to deploy
service = Model.deploy(
    workspace = ws,
    name = "my_web_service",
    models = [model],
    inference_config = inference_config,
    deployment_config = deployment_config)
```

Example notebooks

This example notebook expands upon concepts and methods demonstrated in this article.

Deploy a model using a custom Docker base image demonstrates how to deploy a model using a custom Docker base image.

This example notebook demonstrates how to deploy a Spark model as a web service.

Create and manage environments with the CLI

The Azure Machine Learning CLI mirrors most of the functionality of the Python SDK. You can use it to create and manage environments. The commands that we discuss in this section demonstrate basic functionality.

The following command scaffolds the files for a default environment definition in the specified directory. These files are JSON files. They work like the corresponding class in the SDK. You can use the files to create new environments that have custom settings.

```
az ml environment scaffold -n myenv -d myenvdir
```

Run the following command to register an environment from a specified directory.

```
az ml environment register -d myenvdir
```

Run the following command to list all registered environments.

```
az ml environment list
```

Download a registered environment by using the following command.

```
az ml environment download -n myenv -d downloaddir
```

Next steps

- To use a managed compute target to train a model, see Tutorial: Train a model.
- After you have a trained model, learn how and where to deploy models.
- View the `Environment` class SDK reference.
- For more information about the concepts and methods described in this article, see the example notebook.

Enable logging in Azure Machine Learning

3/5/2020 • 2 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

The Azure Machine Learning Python SDK allows you to enable logging using both the default Python logging package, as well as using SDK-specific functionality both for local logging and logging to your workspace in the portal. Logs provide developers with real-time information about the application state, and can help with diagnosing errors or warnings. In this article, you learn different ways of enabling logging in the following areas:

- Training models and compute targets
- Image creation
- Deployed models
- Python `logging` settings

Create an Azure Machine Learning workspace. Use the guide for more information the SDK.

Training models and compute target logging

There are multiple ways to enable logging during the model training process, and the examples shown will illustrate common design patterns. You can easily log run-related data to your workspace in the cloud by using the `start_logging` function on the `Experiment` class.

```
from azureml.core import Experiment

exp = Experiment(workspace=ws, name='test_experiment')
run = exp.start_logging()
run.log("test-val", 10)
```

See the reference documentation for the Run class for additional logging functions.

To enable local logging of application state during training progress, use the `show_output` parameter. Enabling verbose logging allows you to see details from the training process as well as information about any remote resources or compute targets. Use the following code to enable logging upon experiment submission.

```
from azureml.core import Experiment

experiment = Experiment(ws, experiment_name)
run = experiment.submit(config=run_config_object, show_output=True)
```

You can also use the same parameter in the `wait_for_completion` function on the resulting run.

```
run.wait_for_completion(show_output=True)
```

The SDK also supports using the default python logging package in certain scenarios for training. The following example enables a logging level of `INFO` in an `AutoMLConfig` object.

```
from azureml.train.automl import AutoMLConfig
import logging

automated_ml_config = AutoMLConfig(task='regression',
                                   verbosity=logging.INFO,
                                   X=your_training_features,
                                   y=your_training_labels,
                                   iterations=30,
                                   iteration_timeout_minutes=5,
                                   primary_metric="spearman_correlation")
```

You can also use the `show_output` parameter when creating a persistent compute target. Specify the parameter in the `wait_for_completion` function to enable logging during compute target creation.

```
from azureml.core.compute import ComputeTarget

compute_target = ComputeTarget.attach(
    workspace=ws, name="example", attach_configuration=config)
compute.wait_for_completion(show_output=True)
```

Logging for deployed models

To retrieve logs from a previously deployed web service, load the service and use the `get_logs()` function. The logs may contain detailed information about any errors that occurred during deployment.

```
from azureml.core.webservice import Webservice

# load existing web service
service = Webservice(name="service-name", workspace=ws)
logs = service.get_logs()
```

You can also log custom stack traces for your web service by enabling Application Insights, which allows you to monitor request/response times, failure rates, and exceptions. Call the `update()` function on an existing web service to enable Application Insights.

```
service.update(enable_app_insights=True)
```

For more information, see Monitor and collect data from ML web service endpoints.

Python native logging settings

Certain logs in the SDK may contain an error that instructs you to set the logging level to DEBUG. To set the logging level, add the following code to your script.

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Next steps

- Monitor and collect data from ML web service endpoints

Where to save and write files for Azure Machine Learning experiments

3/10/2020 • 3 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn where to save input files, and where to write output files from your experiments to prevent storage limit errors and experiment latency.

When launching training runs on a compute target, they are isolated from outside environments. The purpose of this design is to ensure reproducibility and portability of the experiment. If you run the same script twice, on the same or another compute target, you receive the same results. With this design, you can treat compute targets as stateless computation resources, each having no affinity to the jobs that are running after they are finished.

Where to save input files

Before you can initiate an experiment on a compute target or your local machine, you must ensure that the necessary files are available to that compute target, such as dependency files and data files your code needs to run.

Azure Machine Learning runs training scripts by copying the entire script folder to the target compute context, and then takes a snapshot. The storage limit for experiment snapshots is 300 MB and/or 2000 files.

For this reason, we recommend:

- **Storing your files in an Azure Machine Learning datastore.** This prevents experiment latency issues, and has the advantages of accessing data from a remote compute target, which means authentication and mounting are managed by Azure Machine Learning. Learn more about specifying a datastore as your source directory, and uploading files to your datastore in the Access data from your datastores article.
- **If you only need a couple data files and dependency scripts and can't use a datastore,** place the files in the same folder directory as your training script. Specify this folder as your `source_directory` directly in your training script, or in the code that calls your training script.

Storage limits of experiment snapshots

For experiments, Azure Machine Learning automatically makes an experiment snapshot of your code based on the directory you suggest when you configure the run. This has a total limit of 300 MB and/or 2000 files. If you exceed this limit, you'll see the following error:

```
While attempting to take snapshot of .
Your total snapshot size exceeds the limit of 300.0 MB
```

To resolve this error, store your experiment files on a datastore. If you can't use a datastore, the below table offers possible alternate solutions.

EXPERIMENT DESCRIPTION	STORAGE LIMIT SOLUTION
Less than 2000 files & can't use a datastore	Override snapshot size limit with `azureml._restclient.snapshots_client.SNAPSHOT_MAX_SIZE_BYTES = 'insert_desired_size'` This may take several minutes depending on the number and size of files.

EXPERIMENT DESCRIPTION	STORAGE LIMIT SOLUTION
Must use specific script directory	Make a `.amlignore` file to exclude files from your experiment snapshot that are not part of the source code. Add the filenames to the `.amlignore` file and place it in the same directory as your training script. The `.amlignore` file uses the same syntax and patterns as a `.gitignore` file.
Pipeline	Use a different subdirectory for each step
Jupyter notebooks	Create a `.amlignore` file or move your notebook into a new, empty, subdirectory and run your code again.

Where to write files

Due to the isolation of training experiments, the changes to files that happen during runs are not necessarily persisted outside of your environment. If your script modifies the files local to compute, the changes are not persisted for your next experiment run, and they're not propagated back to the client machine automatically. Therefore, the changes made during the first experiment run don't and shouldn't affect those in the second.

When writing changes, we recommend writing files to an Azure Machine Learning datastore. See Access data from your datastores.

If you don't require a datastore, write files to the `./outputs` and/or `./logs` folder.

> **IMPORTANT**
> Two folders, *outputs* and *logs*, receive special treatment by Azure Machine Learning. During training, when you write files to `./outputs` and `./logs` folders, the files will automatically upload to your run history, so that you have access to them once your run is finished.

- **For output such as status messages or scoring results,** write files to the `./outputs` folder, so they are persisted as artifacts in run history. Be mindful of the number and size of files written to this folder, as latency may occur when the contents are uploaded to run history. If latency is a concern, writing files to a datastore is recommended.
- **To save written file as logs in run history,** write files to `./logs` folder. The logs are uploaded in real time, so this method is suitable for streaming live updates from a remote run.

Next steps

- Learn more about accessing data from your datastores.
- Learn more about How to Set Up Training Targets.

Debug interactively on an Azure Machine Learning Compute Instance with VS Code Remote

2/12/2020 • 2 minutes to read • Edit Online

In this article, you'll learn how to set up Visual Studio Code Remote on an Azure Machine Learning Compute Instance so you can **interactively debug your code** from VS Code.

- An Azure Machine Learning Compute Instance is a fully managed cloud-based workstation for data scientists and provides management and enterprise readiness capabilities for IT administrators.
- Visual Studio Code Remote Development allows you to use a container, remote machine, or the Windows Subsystem for Linux (WSL) as a full-featured development environment.

Prerequisite

On Windows platforms, you must install an OpenSSH compatible SSH client if one is not already present.

> **NOTE**
> PuTTY is not supported on Windows since the ssh command must be in the path.

Get IP and SSH port

1. Go to the Azure Machine Learning studio at https://ml.azure.com/.
2. Select your workspace.
3. Click the **Compute Instances** tab.
4. In the **Application URI** column, click the **SSH** link of the compute instance you want to use as a remote compute.
5. In the dialog, take note of the IP Address and SSH port.
6. Save your private key to the ~/.ssh/ directory on your local computer; for instance, open an editor for a new file and paste the key in:

   Linux:

   ```
   vi ~/.ssh/id_azmlcitest_rsa
   ```

   Windows:

   ```
   notepad C:\Users\<username>\.ssh\id_azmlcitest_rsa
   ```

   The private key will look somewhat like this:

```
-----BEGIN RSA PRIVATE KEY-----

MIIEpAIBAAKCAQEAr99EPm0P4CaTPT2KtBt+kpN3rmsNNE5dS0vmGWxIXq4vAWXD
.....
ewMtLnDgXWYJo0IyQ91ynOdxbFoVOuuGNdDoBykUZPQfeHDONy2Raw==

-----END RSA PRIVATE KEY-----
```

7. Change permissions on file to make sure only you can read the file.

```
chmod 600 ~/.ssh/id_azmlcitest_rsa
```

Add instance as a host

Open the file `~/.ssh/config` (Linux) or `C:\Users<username>.ssh\config` (Windows) in an editor and add a new entry similar to this:

```
Host azmlci1

    HostName 13.69.56.51

    Port 50000

    User azureuser

    IdentityFile ~/.ssh/id_azmlcitest_rsa
```

Here some details on the fields:

FIELD	DESCRIPTION
Host	Use whatever shorthand you like for the compute instance
HostName	This is the IP address of the compute instance
Port	This is the port shown on the SSH dialog above
User	This needs to be `azureuser`
IdentityFile	Should point to the file where you saved the private key

Now, you should be able to ssh to your compute instance using the shorthand you used above, `ssh azmlci1`.

Connect VS Code to the instance

1. Install Visual Studio Code.
2. Install the Remote SSH Extension.
3. Click the Remote-SSH icon on the left to show your SSH configurations.
4. Right-click the SSH host configuration you just created.
5. Select **Connect to Host in Current Window**.

From here on, you are entirely working on the compute instance and you can now edit, debug, use git, use

extensions, etc. -- just like you can with your local Visual Studio Code.

Next steps

Now that you've set up Visual Studio Code Remote, you can use a compute instance as remote compute from Visual Studio Code to interactively debug your code.

Tutorial: Train your first ML model shows how to use a compute instance with an integrated notebook.

Git integration for Azure Machine Learning

3/5/2020 • 3 minutes to read • Edit Online

Git is a popular version control system that allows you to share and collaborate on your projects.

Azure Machine Learning fully supports Git repositories for tracking work - you can clone repositories directly onto your shared workspace file system, use Git on your local workstation, or use Git from a CI/CD pipeline.

When submitting a job to Azure Machine Learning, if source files are stored in a local git repository then information about the repo is tracked as part of the training process.

Since Azure Machine Learning tracks information from a local git repo, it isn't tied to any specific central repository. Your repository can be cloned from GitHub, GitLab, Bitbucket, Azure DevOps, or any other git-compatible service.

Clone Git repositories into your workspace file system

Azure Machine Learning provides a shared file system for all users in the workspace. To clone a Git repository into this file share, we recommend that you create a Compute Instance & open a terminal. Once the terminal is opened, you have access to a full Git client and can clone and work with Git via the Git CLI experience.

We recommend that you clone the repository into your users directory so that others will not make collisions directly on your working branch.

You can clone any Git repository you can authenticate to (GitHub, Azure Repos, BitBucket, etc.)

For a guide on how to use the Git CLI, read here here.

Track code that comes from Git repositories

When you submit a training run from the Python SDK or Machine Learning CLI, the files needed to train the model are uploaded to your workspace. If the `git` command is available on your development environment, the upload process uses it to check if the files are stored in a git repository. If so, then information from your git repository is also uploaded as part of the training run. This information is stored in the following properties for the training run:

PROPERTY	GIT COMMAND USED TO GET THE VALUE	DESCRIPTION
`azureml.git.repository_uri`	`git ls-remote --get-url`	The URI that your repository was cloned from.
`mlflow.source.git.repoURL`	`git ls-remote --get-url`	The URI that your repository was cloned from.
`azureml.git.branch`	`git symbolic-ref --short HEAD`	The active branch when the run was submitted.
`mlflow.source.git.branch`	`git symbolic-ref --short HEAD`	The active branch when the run was submitted.
`azureml.git.commit`	`git rev-parse HEAD`	The commit hash of the code that was submitted for the run.

PROPERTY	GIT COMMAND USED TO GET THE VALUE	DESCRIPTION
`mlflow.source.git.commit`	`git rev-parse HEAD`	The commit hash of the code that was submitted for the run.
`azureml.git.dirty`	`git status --porcelain .`	`True`, if the branch/commit is dirty; otherwise, `false`.

This information is sent for runs that use an estimator, machine learning pipeline, or script run.

If your training files are not located in a git repository on your development environment, or the `git` command is not available, then no git-related information is tracked.

TIP

To check if the git command is available on your development environment, open a shell session, command prompt, PowerShell or other command line interface and type the following command:

```
git --version
```

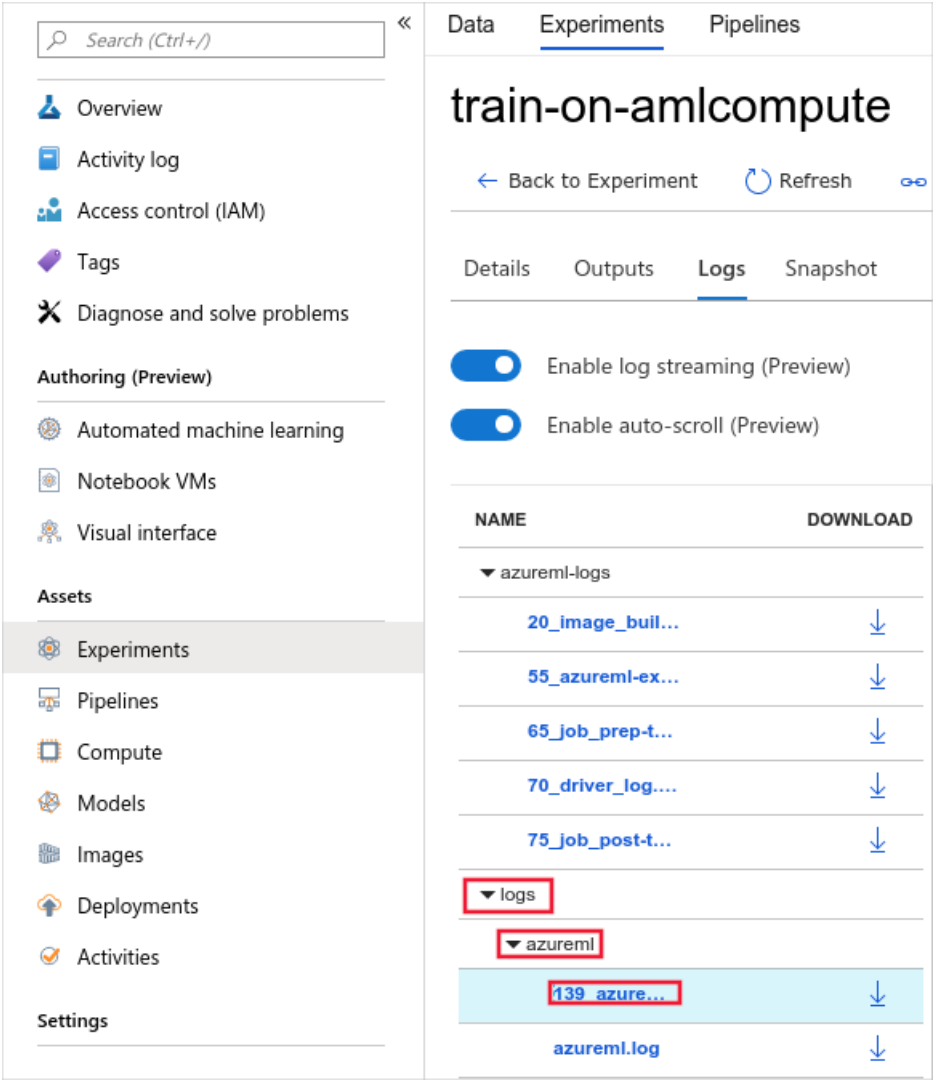
If installed, and in the path, you receive a response similar to `git version 2.4.1`. For more information on installing git on your development environment, see the Git website.

View the logged information

The git information is stored in the properties for a training run. You can view this information using the Azure portal, Python SDK, and CLI.

Azure portal

1. From the Azure portal, select your workspace.
2. Select **Experiments**, and then select one of your experiments.
3. Select one of the runs from the **RUN NUMBER** column.
4. Select **Logs**, and then expand the **logs** and **azureml** entries. Select the link that begins with **###_azure**.



The logged information contains text similar to the following JSON:

```
"properties": {
    "_azureml.ComputeTargetType": "batchai",
    "ContentSnapshotId": "5ca66406-cbac-4d7d-bc95-f5a51dd3e57e",
    "azureml.git.repository_uri": "git@github.com:azure/machinelearningnotebooks",
    "mlflow.source.git.repoURL": "git@github.com:azure/machinelearningnotebooks",
    "azureml.git.branch": "master",
    "mlflow.source.git.branch": "master",
    "azureml.git.commit": "4d2b93784676893f8e346d5f0b9fb894a9cf0742",
    "mlflow.source.git.commit": "4d2b93784676893f8e346d5f0b9fb894a9cf0742",
    "azureml.git.dirty": "True",
    "AzureML.DerivedImageName": "azureml/azureml_9d3568242c6bfef9631879915768deaf",
    "ProcessInfoFile": "azureml-logs/process_info.json",
    "ProcessStatusFile": "azureml-logs/process_status.json"
}
```

Python SDK

After submitting a training run, a Run object is returned. The `properties` attribute of this object contains the logged git information. For example, the following code retrieves the commit hash:

```
run.properties['azureml.git.commit']
```

CLI

The `az ml run` CLI command can be used to retrieve the properties from a run. For example, the following command returns the properties for the last run in the experiment named `train-on-amlcompute`:

```
az ml run list -e train-on-amlcompute --last 1 -w myworkspace -g myresourcegroup --query '[].properties'
```

For more information, see the az ml run reference documentation.

Next steps

- Set up and use compute targets for model training

Create a data labeling project and export labels

4/7/2020 • 11 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Labeling voluminous data in machine learning projects is often a headache. Projects that have a computer-vision component, such as image classification or object detection, generally require labels for thousands of images.

Azure Machine Learning gives you a central place to create, manage, and monitor labeling projects (public preview). Use it to coordinate data, labels, and team members to efficiently manage labeling tasks. Machine Learning supports image classification, either multi-label or multi-class, and object identification with bounded boxes.

Machine Learning tracks progress and maintains the queue of incomplete labeling tasks. Labelers don't need an Azure account to participate. After they are authenticated with your Microsoft account or Azure Active Directory, they can do as much labeling as their time allows.

You start and stop the project, add and remove labelers and teams, and monitor the labeling progress. You can export labeled data in COCO format or as an Azure Machine Learning dataset.

> **IMPORTANT**
> Only image classification and object identification labeling projects are currently supported. Additionally, the data images must be available in an Azure blob datastore. (If you do not have an existing datastore, you may upload images during project creation.)

In this article, you'll learn how to:

- Create a project
- Specify the project's data and structure
- Manage the teams and people who work on the project
- Run and monitor the project
- Export the labels

Prerequisites

- The data that you want to label, either in local files or in Azure blob storage.
- The set of labels that you want to apply.
- The instructions for labeling.
- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin.
- A Machine Learning workspace. See Create an Azure Machine Learning workspace.

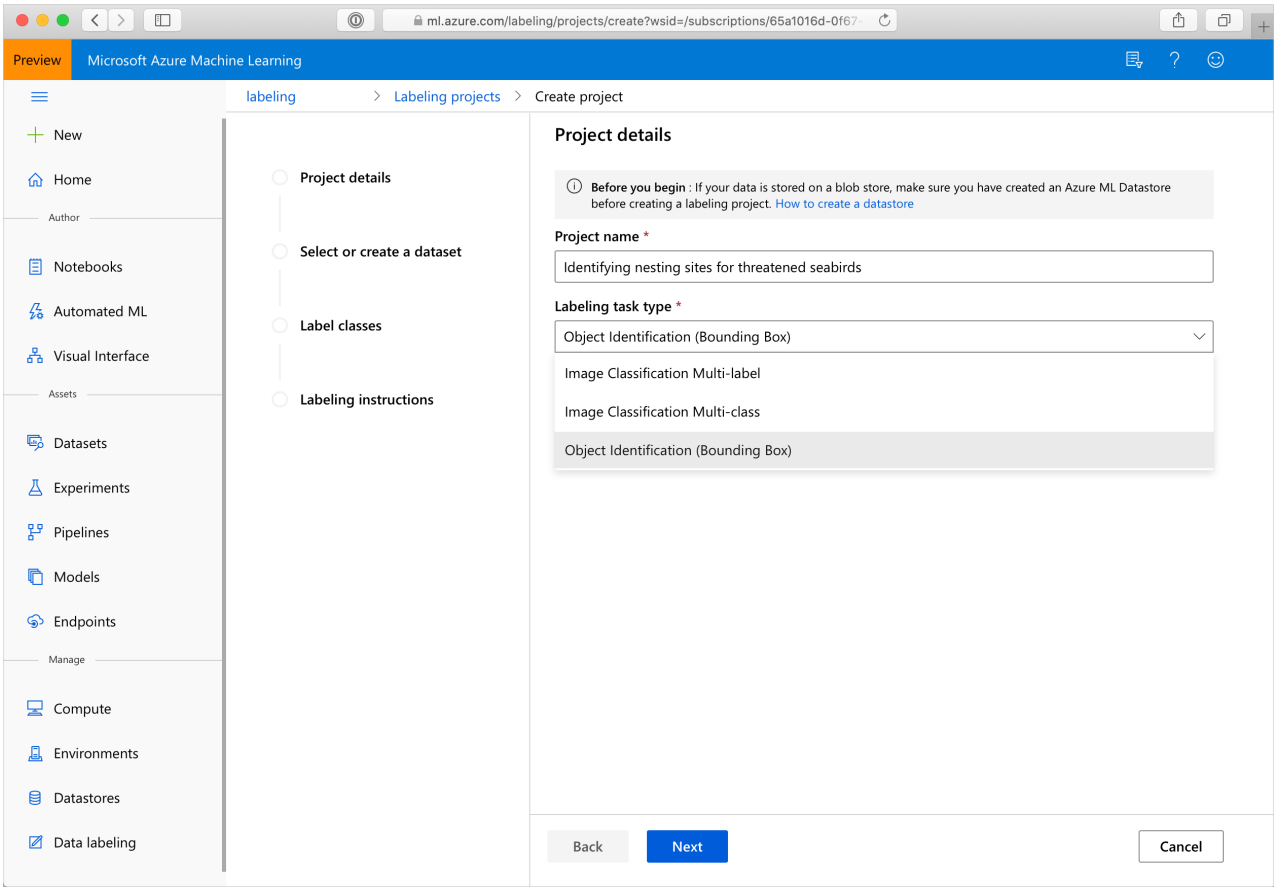
Create a labeling project

Labeling projects are administered from Azure Machine Learning. You use the **Labeling projects** page to manage your projects and people. A project has one or more teams assigned to it, and a team has one or more people assigned to it.

If your data is already in Azure Blob storage, you should make it available as a datastore before you create the labeling project. For an example of using a datastore, see Tutorial: Create your first image classification labeling

project.

To create a project, select **Add project**. Give the project an appropriate name and select **Labeling task type**.



- Choose **Image Classification Multi-class** for projects when you want to apply only a *single class* from a set of classes to an image.
- Choose **Image Classification Multi-label** for projects when you want to apply *one or more* labels from a set of classes to an image. For instance, a photo of a dog might be labeled with both *dog* and *daytime*.
- Choose **Object Identification (Bounding Box)** for projects when you want to assign a class and a bounding box to each object within an image.

Select **Next** when you're ready to continue.

Specify the data to label

If you already created a dataset that contains your data, select it from the **Select an existing dataset** drop-down list. Or, select **Create a dataset** to use an existing Azure datastore or to upload local files.

Create a dataset from an Azure datastore

In many cases, it's fine to just upload local files. But Azure Storage Explorer provides a faster and more robust way to transfer a large amount of data. We recommend Storage Explorer as the default way to move files.

To create a dataset from data that you've already stored in Azure Blob storage:

1. Select **Create a dataset** > **From datastore**.
2. Assign a **Name** to your dataset.
3. Choose **File** as the **Dataset type**.
4. Select the datastore.
5. If your data is in a subfolder within your blob storage, choose **Browse** to select the path.
   - Append "/**" to the path to include all the files in subfolders of the selected path.
   - Append "**/." to include all the data in the current container and its subfolders.

6. Provide a description for your dataset.
7. Select **Next**.
8. Confirm the details. Select **Back** to modify the settings or **Create** to create the dataset.

> **NOTE**
> The data you choose is loaded into your project. Adding more data to the datastore will not appear in this project once the project is created.

Create a dataset from uploaded data

To directly upload your data:

1. Select **Create a dataset** > **From local files**.
2. Assign a **Name** to your dataset.
3. Choose "File" as the **Dataset type**.
4. *Optional:* Select **Advanced settings** to customize the datastore, container, and path to your data.
5. Select **Browse** to select the local files to upload.
6. Provide a description of your dataset.
7. Select **Next**.
8. Confirm the details. Select **Back** to modify the settings or **Create** to create the dataset.

The data gets uploaded to the default blob store ("workspaceblobstore") of your Machine Learning workspace.

Specify label classes

On the **Label classes** page, specify the set of classes to categorize your data. Do this carefully, because your labelers' accuracy and speed will be affected by their ability to choose among the classes. For instance, instead of spelling out the full genus and species for plants or animals, use a field code or abbreviate the genus.

Enter one label per row. Use the **+** button to add a new row. If you have more than 3 or 4 labels but fewer than 10, you may want to prefix the names with numbers ("1: ", "2: ") so the labelers can use the number keys to speed their work.

Describe the labeling task

It's important to clearly explain the labeling task. On the **Labeling instructions** page, you can add a link to an external site for labeling instructions, or provide instructions in the edit box on the page. Keep the instructions task-oriented and appropriate to the audience. Consider these questions:

- What are the labels they'll see, and how will they choose among them? Is there a reference text to refer to?
- What should they do if no label seems appropriate?
- What should they do if multiple labels seem appropriate?
- What confidence threshold should they apply to a label? Do you want their "best guess" if they aren't certain?
- What should they do with partially occluded or overlapping objects of interest?
- What should they do if an object of interest is clipped by the edge of the image?
- What should they do after they submit a label if they think they made a mistake?

For bounding boxes, important questions include:

- How is the bounding box defined for this task? Should it be entirely on the interior of the object, or should it be on the exterior? Should it be cropped as closely as possible, or is some clearance acceptable?
- What level of care and consistency do you expect the labelers to apply in defining bounding boxes?

- How to label the object that is partially shown in the image?
- How to label the object that partially covered by other object?

> **NOTE**
> Be sure to note that the labelers will be able to select the first 9 labels by using number keys 1-9.

Use ML assisted labeling

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

The **ML assisted labeling** page lets you trigger automatic machine learning models to accelerate the labeling task. At the beginning of your labeling project, the images are shuffled into a random order to reduce potential bias. However, any biases that are present in the dataset will be reflected in the trained model. For example, if 80% of your images are of a single class, then approximately 80% of the data used to train the model will be of that class. This training does not include active learning.

This feature is available for image classification (multi-class or multi-label) tasks.

Select *Enable ML assisted labeling* and specify a GPU to enable assisted labeling, which consists of two phases:

- Clustering
- Prelabeling

The exact number of labeled images necessary to start assisted labeling is not a fixed number. This can vary significantly from one labeling project to another. For some projects, is sometimes possible to see prelabel or cluster tasks after 300 images have been manually labeled. ML Assisted Labeling uses a technique called *Transfer Learning*, which uses a pre-trained model to jump-start the training process. If your dataset's classes are similar to those in the pre-trained model, pre-labels may be available after only a few hundred manually labeled images. If your dataset is significantly different from the data used to pre-train the model, it may take much longer.

Since the final labels still rely on input from the labeler, this technology is sometimes called *human in the loop* labeling.

Clustering

After a certain number of labels are submitted, the machine learning model starts to group together similar images. These similar images are presented to the labelers on the same screen to speed up manual tagging. Clustering is especially useful when the labeler is viewing a grid of 4, 6, or 9 images.

Once a machine learning model has been trained on your manually labeled data, the model is truncated to its last fully-connected layer. Unlabeled images are then passed through the truncated model in a process commonly known as "embedding" or "featurization." This embeds each image in a high-dimensional space defined by this model layer. Images which are nearest neighbors in the space are used for clustering tasks.

Prelabeling

After more image labels are submitted, a classification model is used to predict image tags. The labeler now sees pages that contain predicted labels already present on each image. The task is then to review these labels and correct any mis-labeled images before submitting the page.

Once a machine learning model has been trained on your manually labeled data, the model is evaluated on a test set of manually labeled images to determine its accuracy at a variety of different confidence thresholds. This evaluation process is used to determine a confidence threshold above which the model is accurate enough to show pre-labels. The model is then evaluated against unlabeled data. Images with predictions more confident than this threshold are used for pre-labeling.

> **NOTE**
> ML assisted labeling is available **only** in Enterprise edition workspaces.

Initialize the labeling project

After the labeling project is initialized, some aspects of the project are immutable. You can't change the task type or dataset. You *can* modify labels and the URL for the task description. Carefully review the settings before you create the project. After you submit the project, you're returned to the **Data Labeling** homepage, which will show the project as **Initializing**. This page doesn't automatically refresh. So, after a pause, manually refresh the page to see the project's status as **Created**.

Manage teams and people

By default, each labeling project that you create gets a new team with you as a member. But teams can also be shared between projects. And projects can have more than one team. To create a team, select **Add team** on the **Teams** page.

You manage people on the **Labelers** page. Add and remove people by email address. Each labeler has to authenticate through your Microsoft account or Azure Active Directory, if you use it.

After you add a person, you can assign that person to one or more teams: Go to the **Teams** page, select the team, and then select **Assign people** or **Remove people**.

To send an email to the team, select the team to view the **Team details** page. On this page, select **Email team** to open an email draft with the addresses of everyone on the team.

Run and monitor the project

After you initialize the project, Azure will begin running it. Select the project on the main **Data Labeling** page to go to **Project details**. The **Dashboard** tab shows the progress of the labeling task.

On the **Data** tab, you can see your dataset and review labeled data. If you see incorrectly labeled data, select it and choose **Reject**, which will remove the labels and put the data back into the unlabeled queue.

Use the **Team** tab to assign or unassign teams to the project.

To pause or restart the project, select the **Pause/Start** button. You can only label data when the project is running.

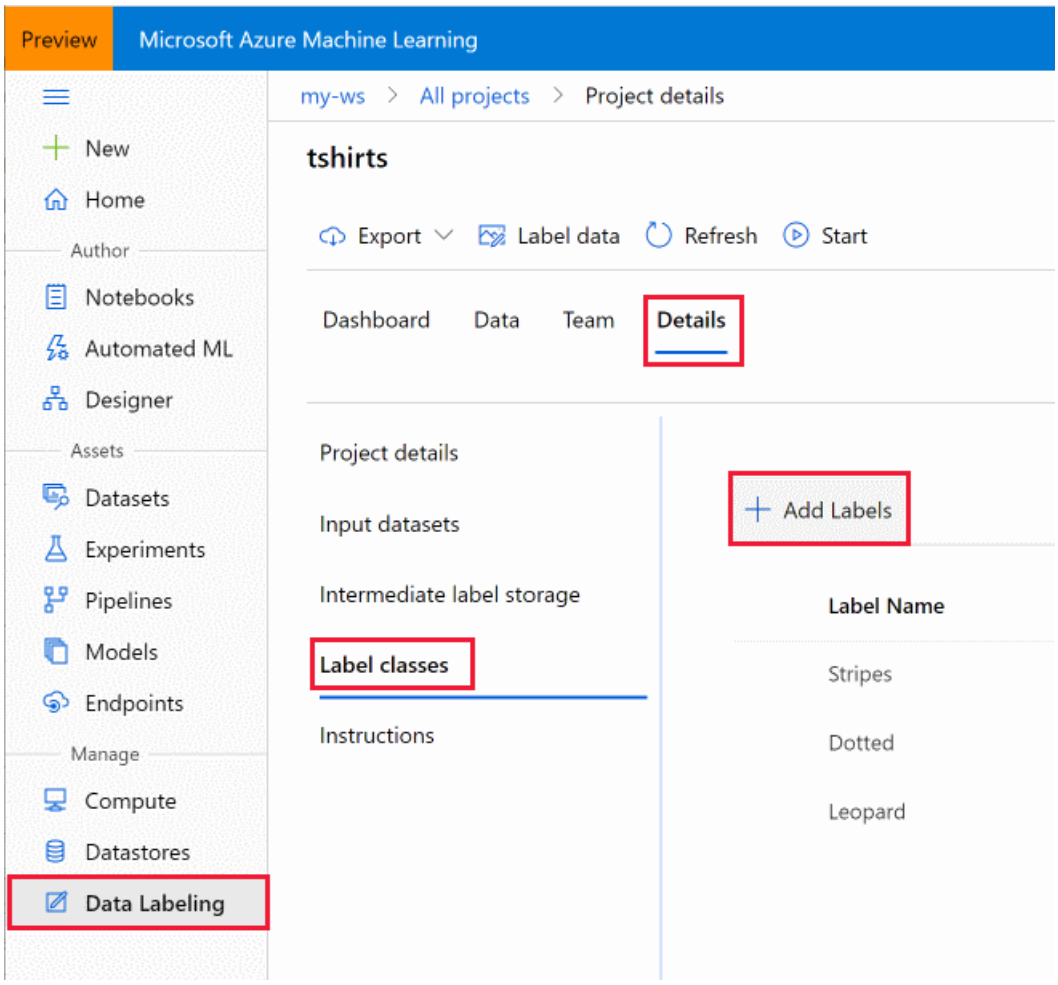
You can label data directly from the **Project details** page by selecting **Label data**.

Add new label class to a project

During the labeling process, you may find that additional labels are needed to classify your images. For example, you may want to add an "Unknown" or "Other" label to indicate confusing images.

Use these steps to add one or more labels to a project:

1. Select the project on the main **Data Labeling** page.
2. At the top of the page, select **Pause** to stop labelers from their activity.
3. Select the **Details** tab.
4. In the list on the left, select **Label classes**.
5. At the top of the list, select **+ Add Labels**

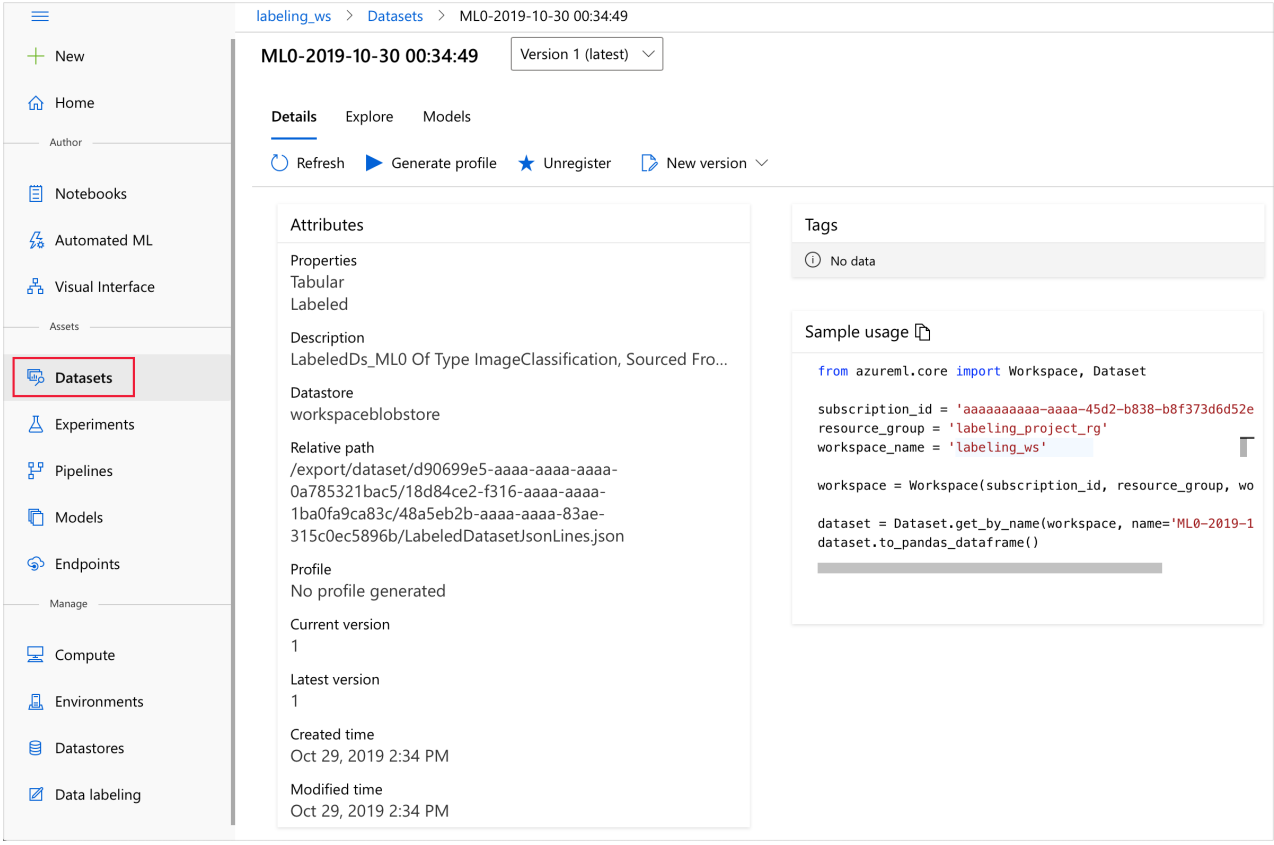


6. In the form, add your new label and choose how to proceed. Since you've changed the available labels for an image, you choose how to treat the already labeled data:
   - Start over, removing all existing labels. Choose this option if you want to start labeling from the beginning with the new full set of labels.
   - Start over, keeping all existing labels. Choose this option to mark all data as unlabeled, but keep the existing labels as a default tag for images that were previously labeled.
   - Continue, keeping all existing labels. Choose this option to keep all data already labeled as is, and start using the new label for data not yet labeled.
7. Modify your instructions page as necessary for the new label(s).
8. Once you have added all new labels, at the top of the page select **Start** to restart the project.

Export the labels

You can export the label data for Machine Learning experimentation at any time. Image labels can be exported in COCO format or as an Azure Machine Learning dataset. Use the **Export** button on the **Project details** page of your labeling project.

The COCO file is created in the default blob store of the Azure Machine Learning workspace in a folder within *export/coco*. You can access the exported Azure Machine Learning dataset in the **Datasets** section of Machine Learning. The dataset details page also provides sample code to access your labels from Python.



Next steps

- Tutorial: Create your first image classification labeling project.
- Label images for image classification or object detection
- Learn more about Azure Machine Learning and Machine Learning Studio (classic)

Tag images in a labeling project

4/7/2020 • 6 minutes to read • Edit Online

After your project administrator creates a labeling project in Azure Machine Learning, you can use the labeling tool (public preview) to rapidly prepare data for a Machine Learning project. This article describes:

- How to access your labeling projects
- The labeling tools
- How to use the tools for specific labeling tasks

Prerequisites

- The labeling portal URL for a running data labeling project
- A Microsoft account or an Azure Active Directory account for the organization and project

> **NOTE**
> The project administrator can find the labeling portal URL on the **Details** tab of the **Project details** page.

Sign in to the project's labeling portal

Go to the labeling portal URL that's provided by the project administrator. Sign in by using the email account that the administrator used to add you to the team. For most users, it will be your Microsoft account. If the labeling project uses Azure Active Directory, that's how you'll sign in.

Understand the labeling task

After you sign in, you'll see the project's overview page.

Go to **View detailed instructions**. These instructions are specific to your project. They explain the type of data that you're facing, how you should make your decisions, and other relevant information. After you read this information, return to the project page and select **Start labeling**.

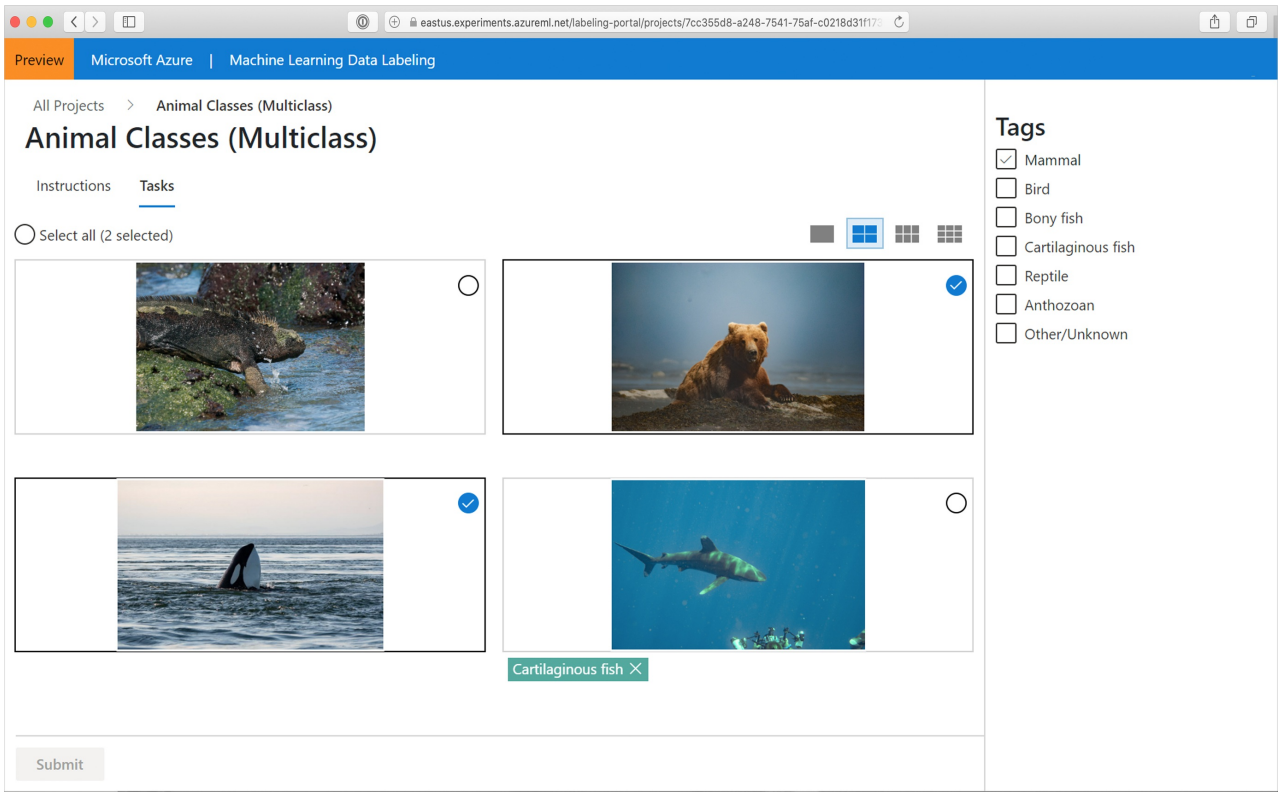
Common features of the labeling task

In all image-labeling tasks, you choose an appropriate tag or tags from a set that's specified by the project administrator. You can select the first nine tags by using the number keys on your keyboard.

In image-classification tasks, you can choose to view multiple images simultaneously. Use the icons above the image area to select the layout.

To select all the displayed images simultaneously, use **Select all**. To select individual images, use the circular selection button in the upper-right corner of the image. You must select at least one image to apply a tag. If you select multiple images, any tag that you select will be applied to all the selected images.

Here we've chosen a two-by-two layout and are about to apply the tag "Mammal" to the images of the bear and orca. The image of the shark was already tagged as "Cartilaginous fish," and the iguana hasn't been tagged yet.



IMPORTANT

Only switch layouts when you have a fresh page of unlabeled data. Switching layouts clears the page's in-progress tagging work.

Azure enables the **Submit** button when you've tagged all the images on the page. Select **Submit** to save your work.

After you submit tags for the data at hand, Azure refreshes the page with a new set of images from the work queue.

Assisted machine learning

Machine learning algorithms may be triggered during a multi-class or multi-label classification task. If these algorithms are enabled in your project, you may see the following:

- After some amount of images have been labeled, you may see **Tasks clustered** at the top of your screen next to the project name. This means that images are grouped together to present similar images on the same page. If so, switch to one of the multiple image views to take advantage of the grouping.
- At a later point, you may see **Tasks prelabeled** next to the project name. Images will then appear with a suggested label that comes from a machine learning classification model. No machine learning model has 100% accuracy. While we only use images for which the model is confident, these images might still be incorrectly prelabeled. When you see these labels, correct any wrong labels before submitting the page.

Especially early in a labeling project, the machine learning model may only be accurate enough to prelabel a small subset of images. Once these images are labeled, the labeling project will return to manual labeling to gather more data for the next round of model training. Over time, the model will become more confident about a higher proportion of images, resulting in more prelabel tasks later in the project.

Tag images for multi-class classification

If your project is of type "Image Classification Multi-Class," you'll assign a single tag to the entire image. To review the directions at any time, go to the **Instructions** page and select **View detailed instructions**.

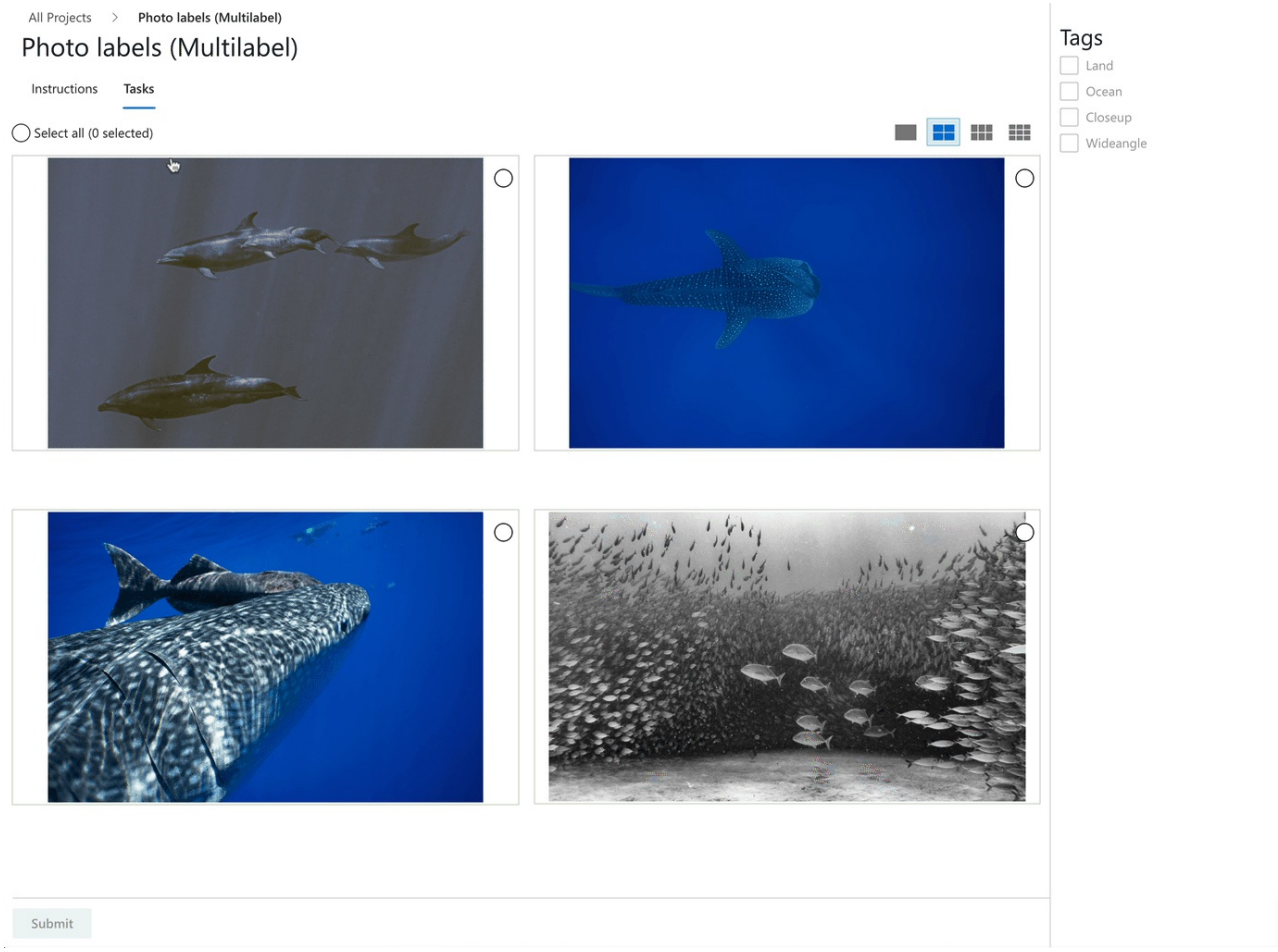
If you realize that you made a mistake after you assign a tag to an image, you can fix it. Select the "**X**" on the label that's displayed below the image to clear the tag. Or, select the image and choose another class. The newly selected value will replace the previously applied tag.

Tag images for multi-label classification

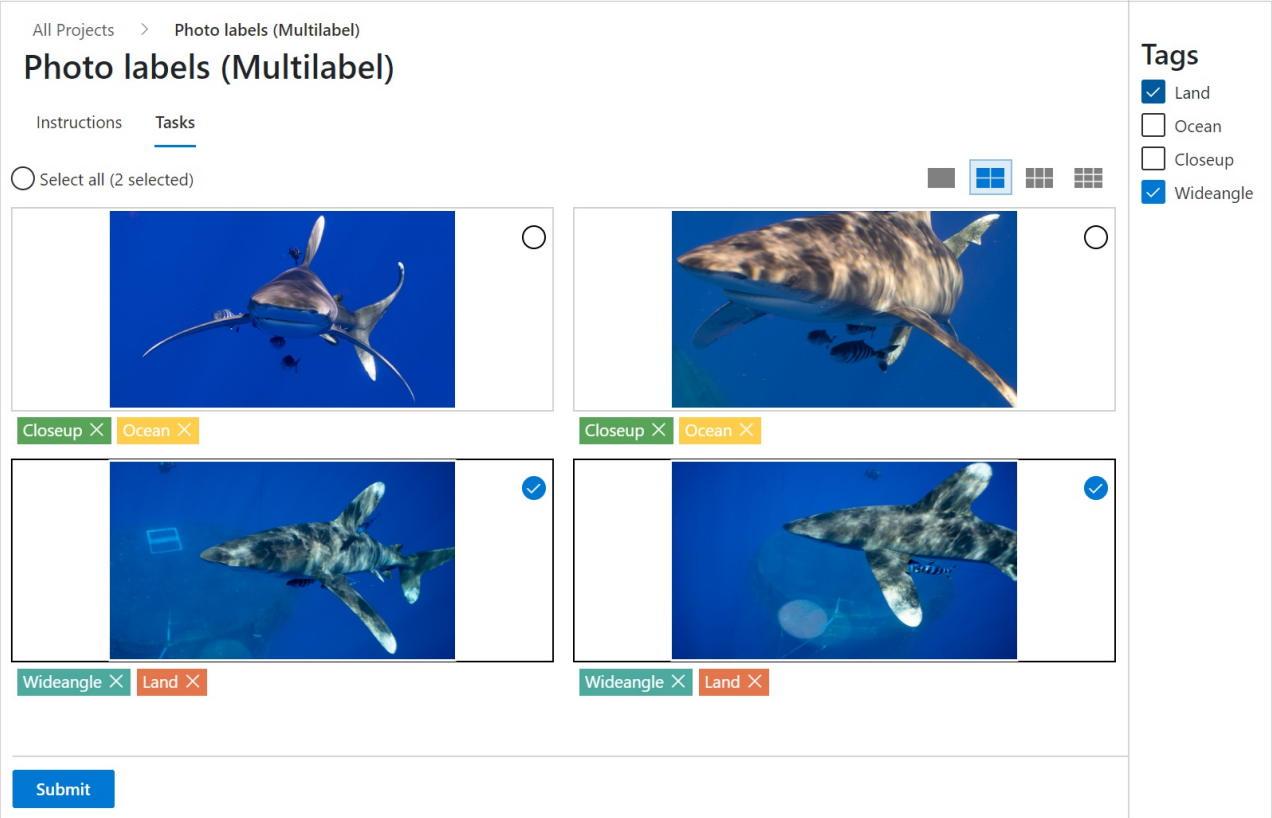
If you're working on a project of type "Image Classification Multi-Label," you'll apply one *or more* tags to an image. To see the project-specific directions, select **Instructions** and go to **View detailed instructions**.

Select the image that you want to label and then select the tag. The tag is applied to all the selected images, and then the images are deselected. To apply more tags, you must reselect the images. The following animation shows multi-label tagging:

1. **Select all** is used to apply the "Ocean" tag.
2. A single image is selected and tagged "Closeup."
3. Three images are selected and tagged "Wide angle."



To correct a mistake, click the "**X**" to clear an individual tag or select the images and then select the tag, which clears the tag from all the selected images. This scenario is shown here. Clicking on "Land" will clear that tag from the two selected images.

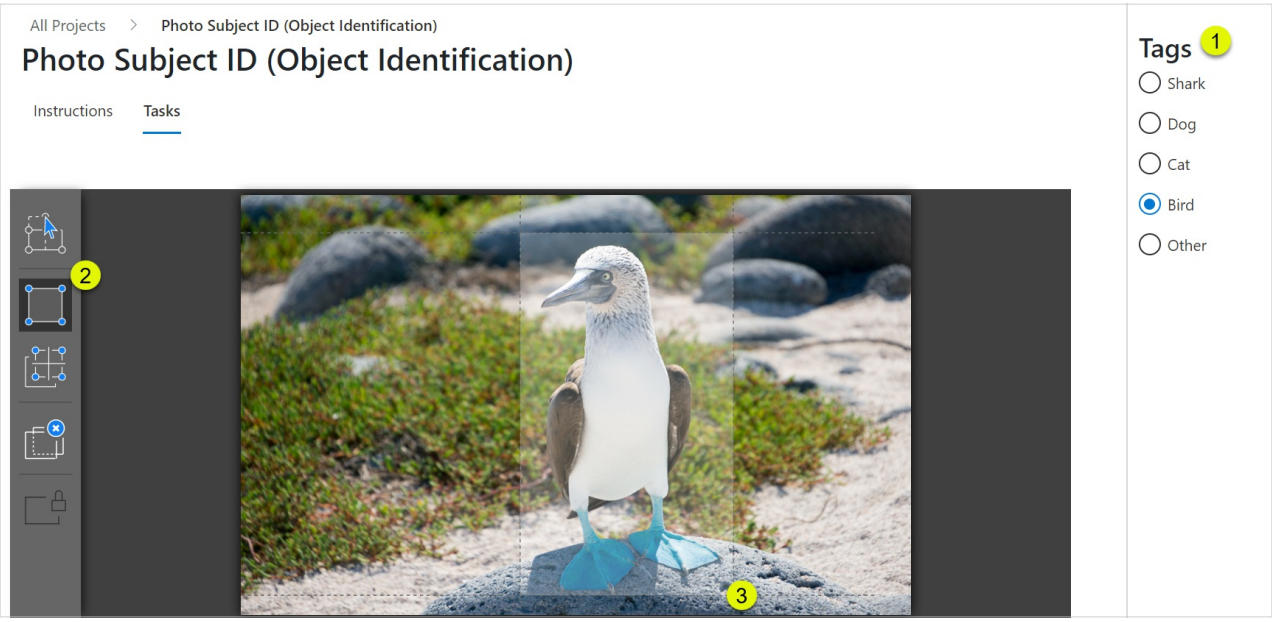


Azure will only enable the **Submit** button after you've applied at least one tag to each image. Select **Submit** to save your work.

Tag images and specify bounding boxes for object detection

If your project is of type "Object Identification (Bounding Boxes)," you'll specify one or more bounding boxes in the image and apply a tag to each box. Images can have multiple bounding boxes, each with a single tag. Use **View detailed instructions** to determine if multiple bounding boxes are used in your project.

1. Select a tag for the bounding box that you plan to create.
2. Select the **Rectangular box** tool  or select "R."
3. Click and drag diagonally across your target to create a rough bounding box. To adjust the bounding box, drag the edges or corners.



To delete a bounding box, click the X-shaped target that appears next to the bounding box after creation.

You can't change the tag of an existing bounding box. If you make a tag-assignment mistake, you have to delete the bounding box and create a new one with the correct tag.

By default, you can edit existing bounding boxes. The **Lock/unlock regions** tool  or "L" toggles that behavior. If regions are locked, you can only change the shape or location of a new bounding box.

Use the **Regions manipulation** tool  or "M" to adjust an existing bounding box. Drag the edges or corners to adjust the shape. Click in the interior to be able to drag the whole bounding box. If you can't edit a region, you've probably toggled the **Lock/unlock regions** tool.

Use the **Template-based box** tool  or "T" to create multiple bounding boxes of the same size. If the image has no bounding boxes and you activate template-based boxes, the tool will produce 50-by-50-pixel boxes. If you create a bounding box and then activate template-based boxes, any new bounding boxes will be the size of the last box that you created. Template-based boxes can be resized after placement. Resizing a template-based box only resizes that particular box.

To delete *all* bounding boxes in the current image, select the **Delete all regions** tool  .

After you create the bounding boxes for an image, select **Submit** to save your work, or your work in progress won't be saved.

Finish up

When you submit a page of tagged data, Azure assigns new unlabeled data to you from a work queue. If there's no more unlabeled data available, you'll get a message noting this along with a link to the portal home page.

When you're done labeling, select your name in the upper-right corner of the labeling portal and then select **sign-out**. If you don't sign out, eventually Azure will "time you out" and assign your data to another labeler.

Next steps

- Learn to train image classification models in Azure

Create and explore Azure Machine Learning dataset with labels

1/23/2020 • 2 minutes to read • Edit Online

In this article, you'll learn how to export the data labels from an Azure Machine Learning data labeling project and load them into popular formats such as, a pandas dataframe for data exploration or a Torchvision dataset for image transformation.

What are datasets with labels

Azure Machine Learning datasets with labels are TabularDatasets with a label property, we will refer to them as labeled datasets. These specific types of TabularDatasets are only created as an output of Azure Machine Learning data labeling projects. Create a data labeling project with these steps. Machine Learning supports data labeling projects for image classification, either multi-label or multi-class, and object identification together with bounded boxes.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin.
- The Azure Machine Learning SDK for Python, or access to Azure Machine Learning studio.
  - Install the azure-contrib-dataset package
- A Machine Learning workspace. See Create an Azure Machine Learning workspace.
- Access to an Azure Machine Learning data labeling project. If you don't have a labeling project, create one with these steps.

Export data labels

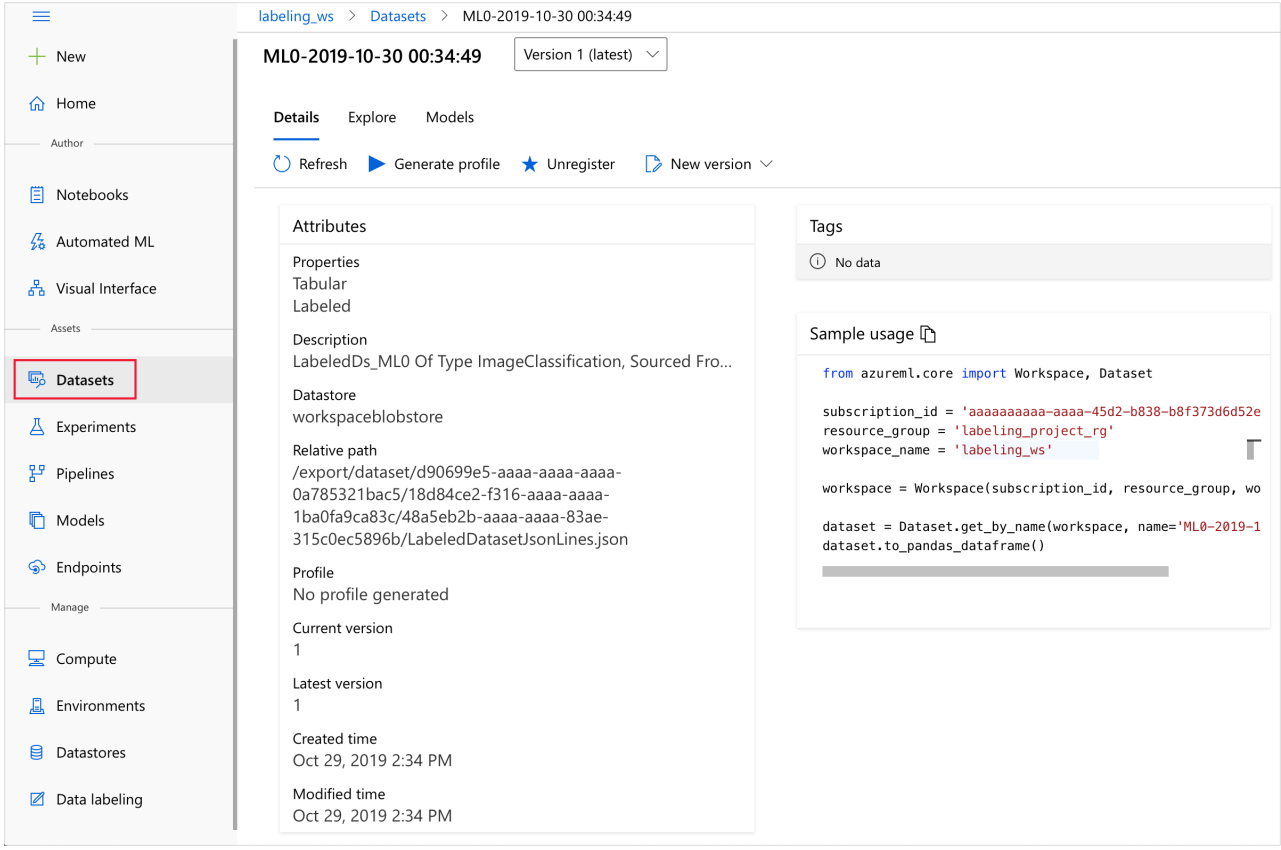
When you complete a data labeling project, you can export the label data from a labeling project. Doing so, allows you to capture both the reference to the data and its labels, and export them in COCO format or as an Azure Machine Learning dataset. Use the **Export** button on the **Project details** page of your labeling project.

COCO

The COCO file is created in the default blob store of the Azure Machine Learning workspace in a folder within *export/coco*.

Azure Machine Learning dataset

You can access the exported Azure Machine Learning dataset in the **Datasets** section of your Azure Machine Learning studio. The dataset **Details** page also provides sample code to access your labels from Python.



Explore labeled datasets

Load your labeled datasets into a pandas dataframe or Torchvision dataset to leverage popular open-source libraries for data exploration, as well as PyTorch provided libraries for image transformation and training.

Pandas dataframe

You can load labeled datasets into a pandas dataframe with the `to_pandas_dataframe()` method from the `azureml-contrib-dataset` class. Install the class with the following shell command:

```
pip install azureml-contrib-dataset
```

> **NOTE**
> The azureml.contrib namespace changes frequently, as we work to improve the service. As such, anything in this namespace should be considered as a preview, and not fully supported by Microsoft.

We offer the following file handling options for file streams when converting to a pandas dataframe.

- Download: Download your data files to a local path.
- Mount: Mount your data files to a mount point. Mount only works for Linux-based compute, including Azure Machine Learning notebook VM and Azure Machine Learning Compute.

```
import azureml.contrib.dataset
from azureml.contrib.dataset import FileHandlingOption
animal_pd = animal_labels.to_pandas_dataframe(file_handling_option=FileHandlingOption.DOWNLOAD,
target_path='./download/', overwrite_download=True)

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

#read images from downloaded path
img = mpimg.imread(animal_pd.loc[0,'image_url'])
imgplot = plt.imshow(img)
```

Torchvision datasets

You can load labeled datasets into Torchvision dataset with the to_torchvision() method also from the `azureml-contrib-dataset` class. To use this method, you need to have PyTorch installed.

```
from torchvision.transforms import functional as F

# load animal_labels dataset into torchvision dataset
pytorch_dataset = animal_labels.to_torchvision()
img = pytorch_dataset[0][0]
print(type(img))

# use methods from torchvision to transform the img into grayscale
pil_image = F.to_pil_image(img)
gray_image = F.to_grayscale(pil_image, num_output_channels=3)

imgplot = plt.imshow(gray_image)
```

Next steps

- See the dataset with labels notebook for complete training sample.

Data ingestion with Azure Data Factory

3/3/2020 • 4 minutes to read • Edit Online

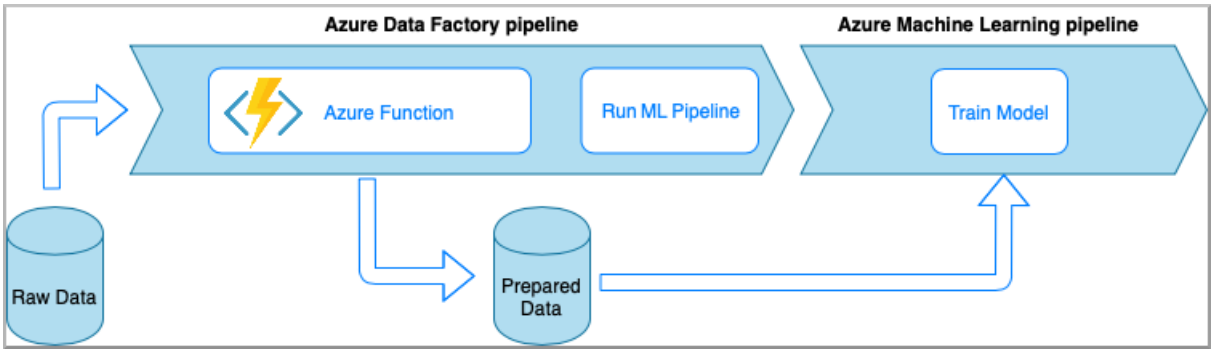
In this article, you learn how to build a data ingestion pipeline with Azure Data Factory (ADF). This pipeline is used to ingest data for use with Azure Machine Learning. Azure Data Factory allows you to easily extract, transform, and load (ETL) data. Once the data has been transformed and loaded into storage, it can be used to train your machine learning models.

Simple data transformation can be handled with native ADF activities and instruments such as data flow. When it comes to more complicated scenarios, the data can be processed with some custom code. For example, Python or R code.

There are several common techniques of using Azure Data Factory to transform data during ingestion. Each technique has pros and cons that determine if it is a good fit for a specific use case:

TECHNIQUE	PROS	CONS
ADF + Azure Functions	Low latency, serverless compute Stateful functions Reusable functions	Only good for short running processing
ADF + custom component	Large-scale parallel computing Suited for heavy algorithms	Wrapping code into an executable Complexity of handling dependencies and IO
ADF + Azure Databricks notebook	Apache Spark Native Python environment	Can be expensive Creating clusters initially takes time and adds latency

ADF with Azure functions



Azure Functions allows you to run small pieces of code (functions) without worrying about application infrastructure. In this option, the data is processed with custom Python code wrapped into an Azure Function.

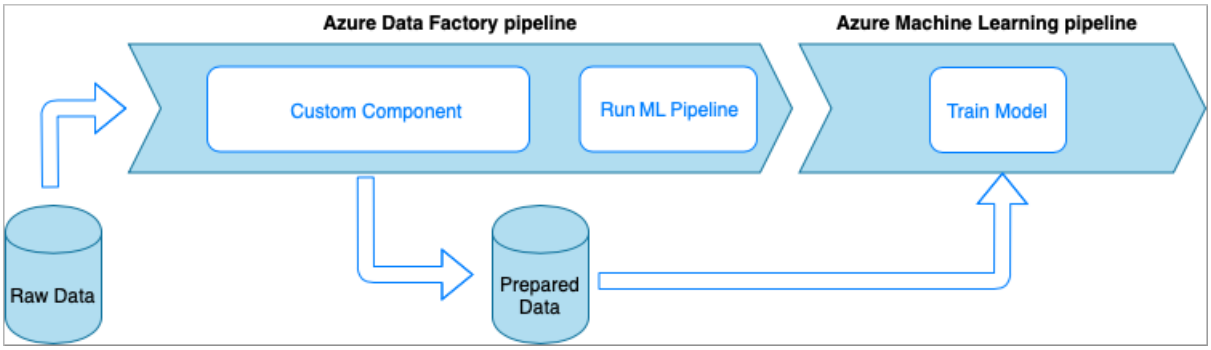
The function is invoked with the ADF Azure Function activity. This approach is a good option for lightweight data transformations.

- Pros:
  - The data is processed on a serverless compute with a relatively low latency
  - ADF pipeline can invoke a Durable Azure Function that may implement a sophisticated data transformation flow
  - The details of the data transformation are abstracted away by the Azure Function that can be reused and

invoked from other places

- Cons:
  - The Azure Functions must be created before use with ADF
  - Azure Functions is good only for short running data processing

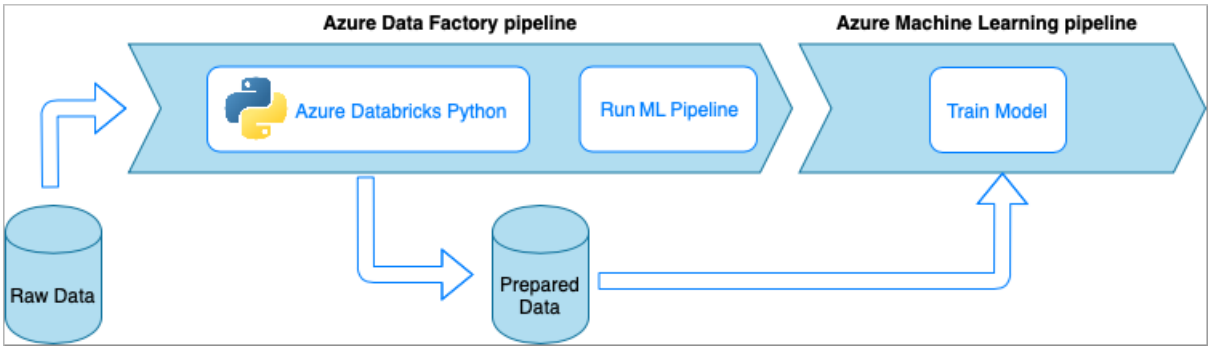
ADF with Custom Component Activity



In this option, the data is processed with custom Python code wrapped into an executable. It is invoked with an ADF Custom Component activity. This approach is a better fit for large data than the previous technique.

- Pros:
  - The data is processed on Azure Batch pool, which provides large-scale parallel and high-performance computing
  - Can be used to run heavy algorithms and process significant amounts of data
- Cons:
  - Azure Batch pool must be created before use with ADF
  - Over engineering related to wrapping Python code into an executable. Complexity of handling dependencies and input/output parameters

ADF with Azure Databricks Python notebook



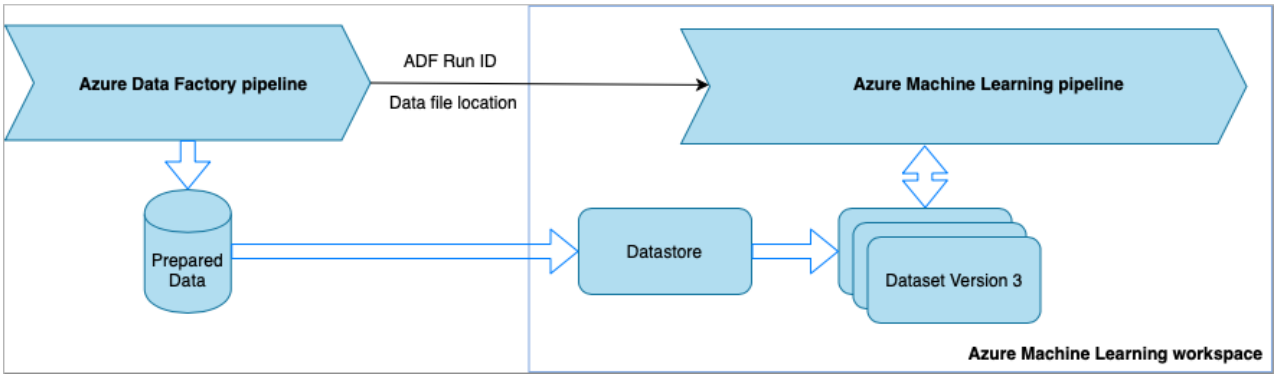
Azure Databricks is an Apache Spark-based analytics platform in the Microsoft cloud.

In this technique, the data transformation is performed by a Python notebook, running on an Azure Databricks cluster. This is probably, the most common approach that leverages the full power of an Azure Databricks service. It is designed for distributed data processing at scale.

- Pros:
  - The data is transformed on the most powerful data processing Azure service, which is backed up by Apache Spark environment
  - Native support of Python along with data science frameworks and libraries including TensorFlow, PyTorch, and scikit-learn
  - There is no need to wrap the Python code into functions or executable modules. The code works as is.
- Cons:

- Azure Databricks infrastructure must be created before use with ADF
- Can be expensive depending on Azure Databricks configuration
- Spinning up compute clusters from "cold" mode takes some time that brings high latency to the solution

Consuming data in Azure Machine Learning pipelines



The transformed data from the ADF pipeline is saved to data storage (such as Azure Blob). Azure Machine Learning can access this data using datastores and datasets.

Each time the ADF pipeline runs, the data is saved to a different location in storage. To pass the location to Azure Machine Learning, the ADF pipeline calls an Azure Machine Learning pipeline. When calling the ML pipeline, the data location and run ID are sent as parameters. The ML pipeline can then create a datastore/dataset using the data location.

> **TIP**
> Datasets support versioning, so the ML pipeline can register a new version of the dataset that points to the most recent data from the ADF pipeline.

Once the data is accessible through a datastore or dataset, you can use it to train an ML model. The training process might be part of the same ML pipeline that is called from ADF. Or it might be a separate process such as experimentation in a Jupyter notebook.

Since datasets support versioning, and each run from the pipeline creates a new version, it's easy to understand which version of the data was used to train a model.

Next steps

- Run a Databricks notebook in Azure Data Factory
- Access data in Azure storage services
- Train models with datasets in Azure Machine Learning
- DevOps for a data ingestion pipeline

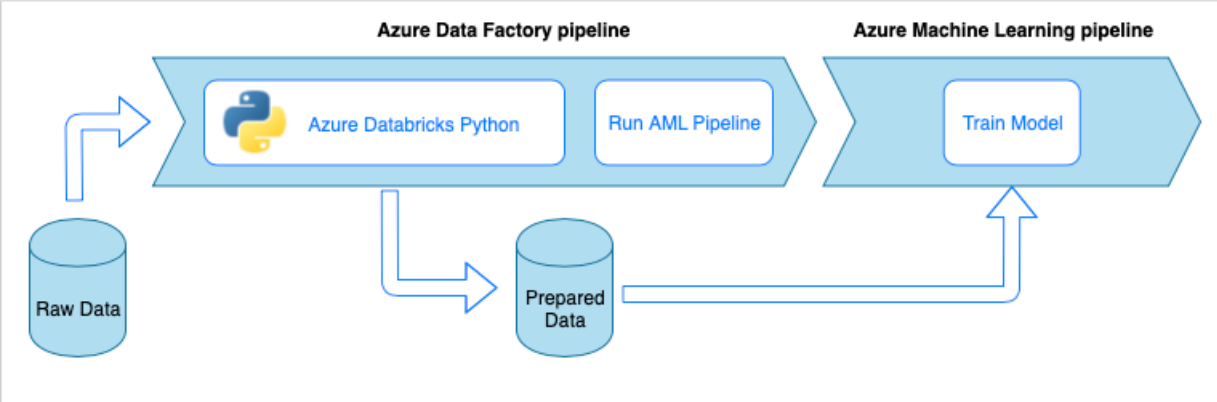
DevOps for a data ingestion pipeline

3/25/2020 • 11 minutes to read • Edit Online

In most scenarios, a data ingestion solution is a composition of scripts, service invocations, and a pipeline orchestrating all the activities. In this article, you learn how to apply DevOps practices to the development lifecycle of a common data ingestion pipeline. The pipeline prepares the data for the Machine Learning model training.

The solution

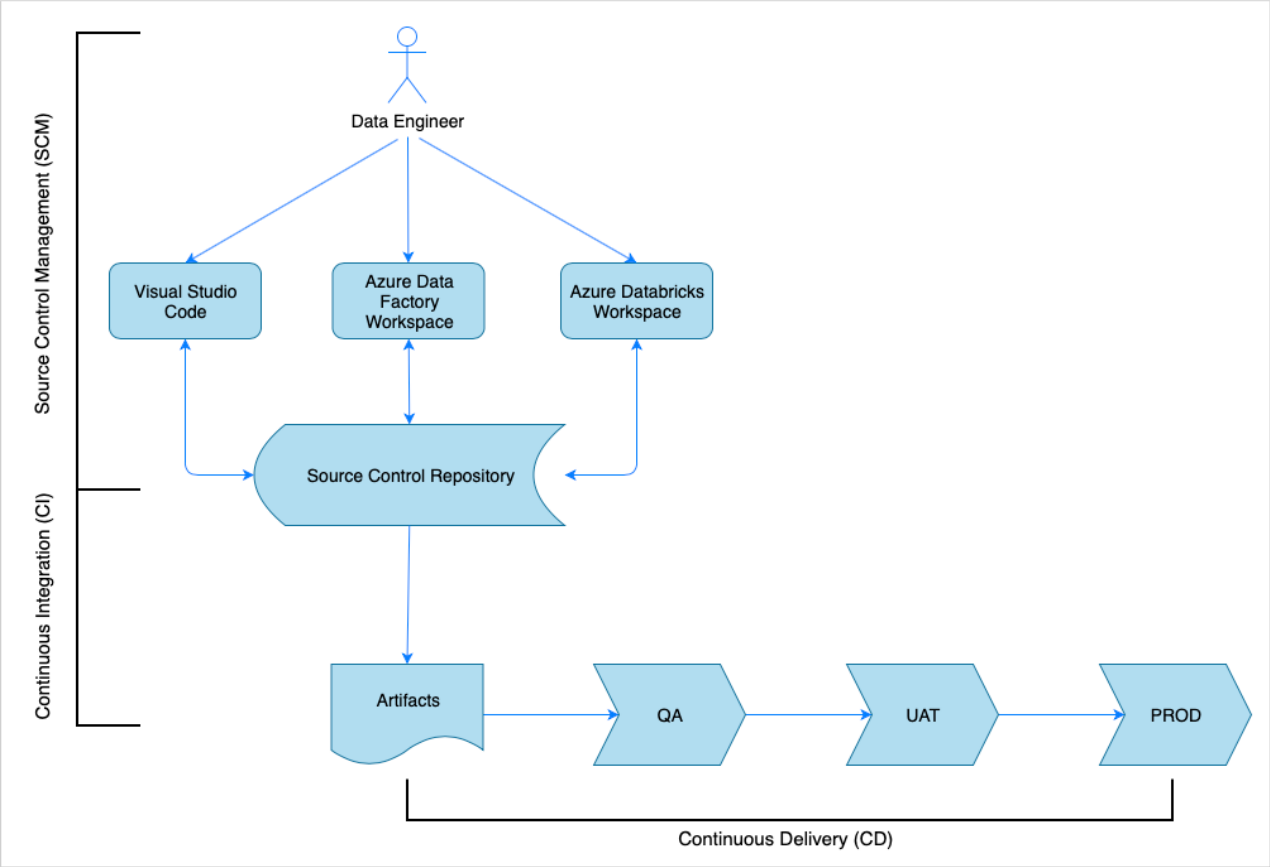
Consider the following data ingestion workflow:



In this approach, the training data is stored in an Azure blob storage. An Azure Data Factory pipeline fetches the data from an input blob container, transforms it and saves the data to the output blob container. This container serves as a data storage for the Azure Machine Learning service. Having the data prepared, the Data Factory pipeline invokes a training Machine Learning pipeline to train a model. In this specific example the data transformation is performed by a Python notebook, running on an Azure Databricks cluster.

What we are building

As with any software solution, there is a team (for example, Data Engineers) working on it.



They collaborate and share the same Azure resources such as Azure Data Factory, Azure Databricks, Azure Storage account and such. The collection of these resources is a Development environment. The data engineers contribute to the same source code base. The Continuous Integration process assembles the code, checks it with the code quality tests, unit tests and produces artifacts such as tested code and Azure Resource Manager templates. The Continuous Delivery process deploys the artifacts to the downstream environments. This article demonstrates how to automate the CI and CD processes with Azure Pipelines.

Source control management

The team members work in slightly different ways to collaborate on the Python notebook source code and the Azure Data Factory source code. However, in both cases the code is stored in a source control repository (for example, Azure DevOps, GitHub, GitLab) and the collaboration is normally based on some branching model (for example, GitFlow).

Python Notebook Source Code

The data engineers work with the Python notebook source code either locally in an IDE (for example, Visual Studio Code) or directly in the Databricks workspace. The latter gives the ability to debug the code on the development environment. In any case, the code is going to be merged to the repository following a branching policy. It's highly recommended to store the code in `.py` files rather than in `.ipynb` Jupyter notebook format. It improves the code readability and enables automatic code quality checks in the CI process.

Azure Data Factory Source Code

The source code of Azure Data Factory pipelines is a collection of json files generated by a workspace. Normally the data engineers work with a visual designer in the Azure Data Factory workspace rather than with the source code files directly. Configure the workspace with a source control repository as it is described in the Azure Data Factory documentation. With this configuration in place, the data engineers are able to collaborate on the source code following a preferred branching workflow.

Continuous integration (CI)

The ultimate goal of the Continuous Integration process is to gather the joint team work from the source code and prepare it for the deployment to the downstream environments. As with the source code management this process is different for the Python notebooks and Azure Data Factory pipelines.

Python Notebook CI

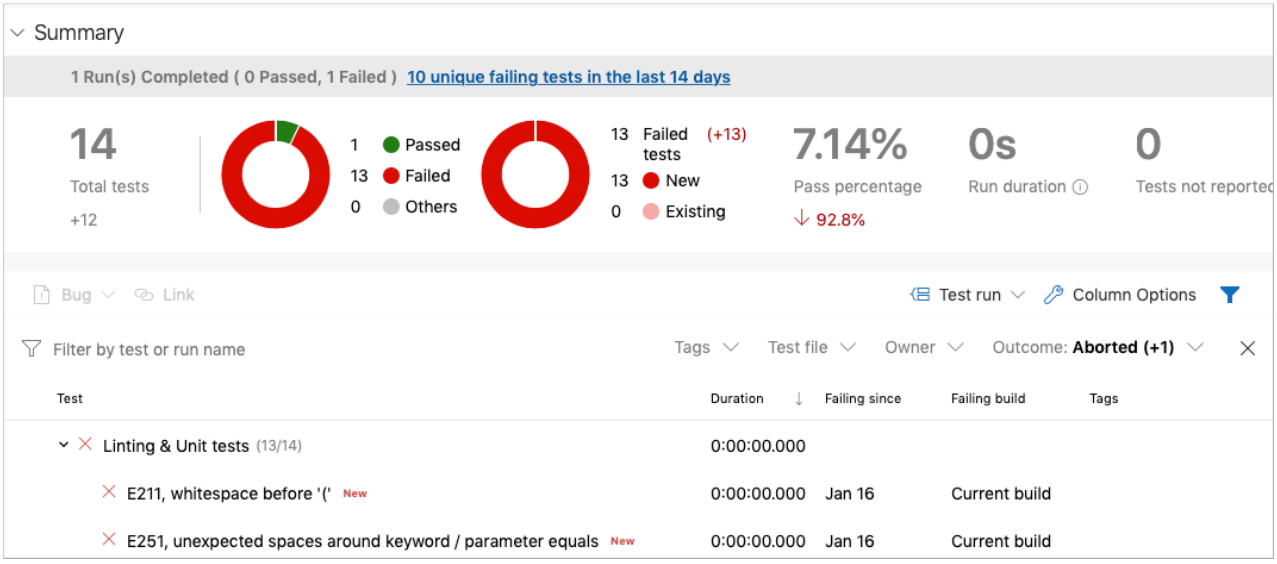
The CI process for the Python Notebooks gets the code from the collaboration branch (for example, ***master*** or ***develop***) and performs the following activities:

- Code linting
- Unit testing
- Saving the code as an artifact

The following code snippet demonstrates the implementation of these steps in an Azure DevOps ***yaml*** pipeline:

```
steps:
- script: |
   flake8 --output-file=$(Build.BinariesDirectory)/lint-testresults.xml --format junit-xml
  workingDirectory: '$(Build.SourcesDirectory)'
  displayName: 'Run flake8 (code style analysis)'

- script: |
   python -m pytest --junitxml=$(Build.BinariesDirectory)/unit-testresults.xml $(Build.SourcesDirectory)
  displayName: 'Run unit tests'

- task: PublishTestResults@2
  condition: succeededOrFailed()
  inputs:
    testResultsFiles: '$(Build.BinariesDirectory)/*-testresults.xml'
    testRunTitle: 'Linting & Unit tests'
    failTaskOnFailedTests: true
  displayName: 'Publish linting and unit test results'

- publish: $(Build.SourcesDirectory)
    artifact: di-notebooks
```

The pipeline uses ***flake8*** to do the Python code linting. It runs the unit tests defined in the source code and publishes the linting and test results so they're available in the Azure Pipeline execution screen:



If the linting and unit testing is successful, the pipeline will copy the source code to the artifact repository to be used by the subsequent deployment steps.

Azure Data Factory CI

CI process for an Azure Data Factory pipeline is a bottleneck in the whole CI/CD story for a data ingestion pipeline. There's no ***Continuous*** Integration. A deployable artifact for Azure Data Factory is a collection of Azure Resource Manager templates. The only way to produce those templates is to click the ***publish*** button in the Azure Data Factory workspace. There's no automation here. The data engineers merge the source code from their feature branches into the collaboration branch, for example, ***master*** or ***develop***. Then, someone with the granted permissions clicks the ***publish*** button to generate Azure Resource Manager templates from the source code in the collaboration branch. When the button is clicked, the workspace validates the pipelines (think of it as of linting and unit testing), generates Azure Resource Manager templates (think of it as of building) and saves the generated templates to a technical branch ***adf_publish*** in the same code repository (think of it as of publishing artifacts). This branch is created automatically by the Azure Data Factory workspace. This process is described in details in the Azure Data Factory documentation.

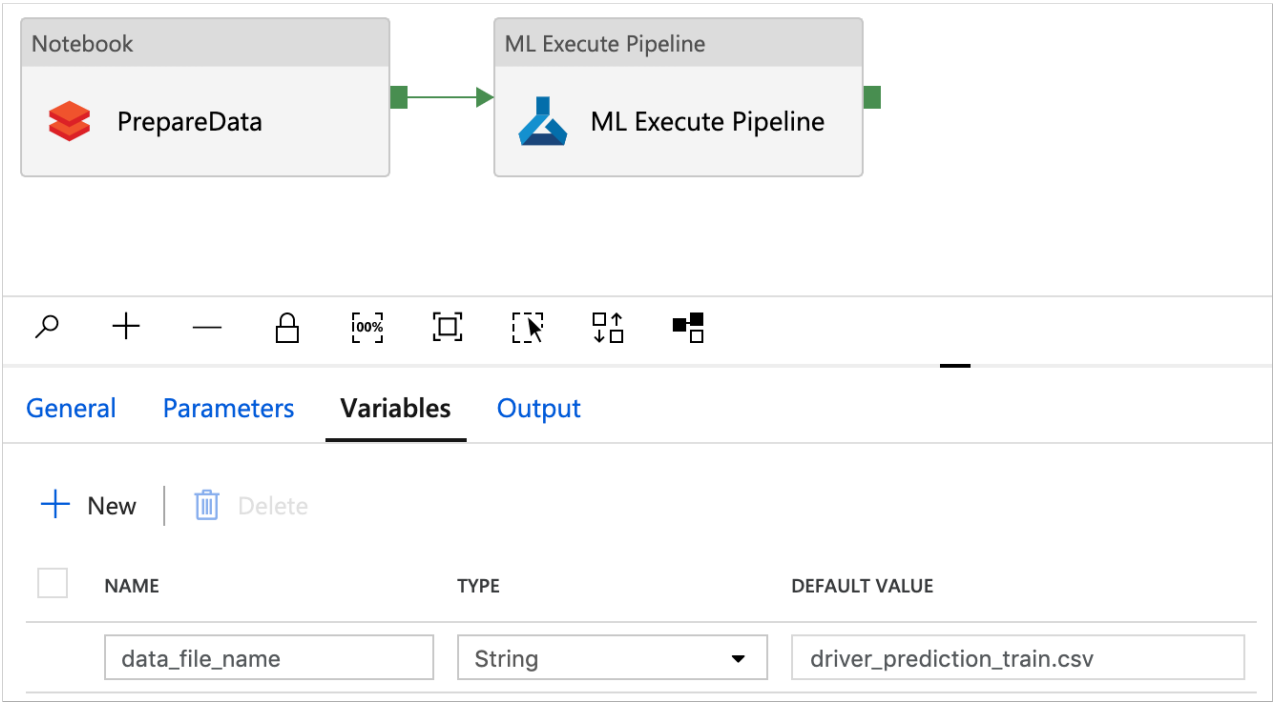
It's important to make sure that the generated Azure Resource Manager templates are environment agnostic. This means that all values that may differ from between environments are parametrized. The Azure Data Factory is smart enough to expose the majority of such values as parameters. For example, in the following template the connection properties to an Azure Machine Learning workspace are exposed as parameters:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "factoryName": {
            "value": "devops-ds-adf"
        },
        "AzureMLService_servicePrincipalKey": {
            "value": ""
        },
        "AzureMLService_properties_typeProperties_subscriptionId": {
            "value": "0fe1c235-5cfa-4152-17d7-5dff45a8d4ba"
        },
        "AzureMLService_properties_typeProperties_resourceGroupName": {
            "value": "devops-ds-rg"
        },
        "AzureMLService_properties_typeProperties_servicePrincipalId": {
            "value": "6e35e589-3b22-4edb-89d0-2ab7fc08d488"
        },
        "AzureMLService_properties_typeProperties_tenant": {
            "value": "72f988bf-86f1-41af-912b-2d7cd611db47"
        }
    }
}
```

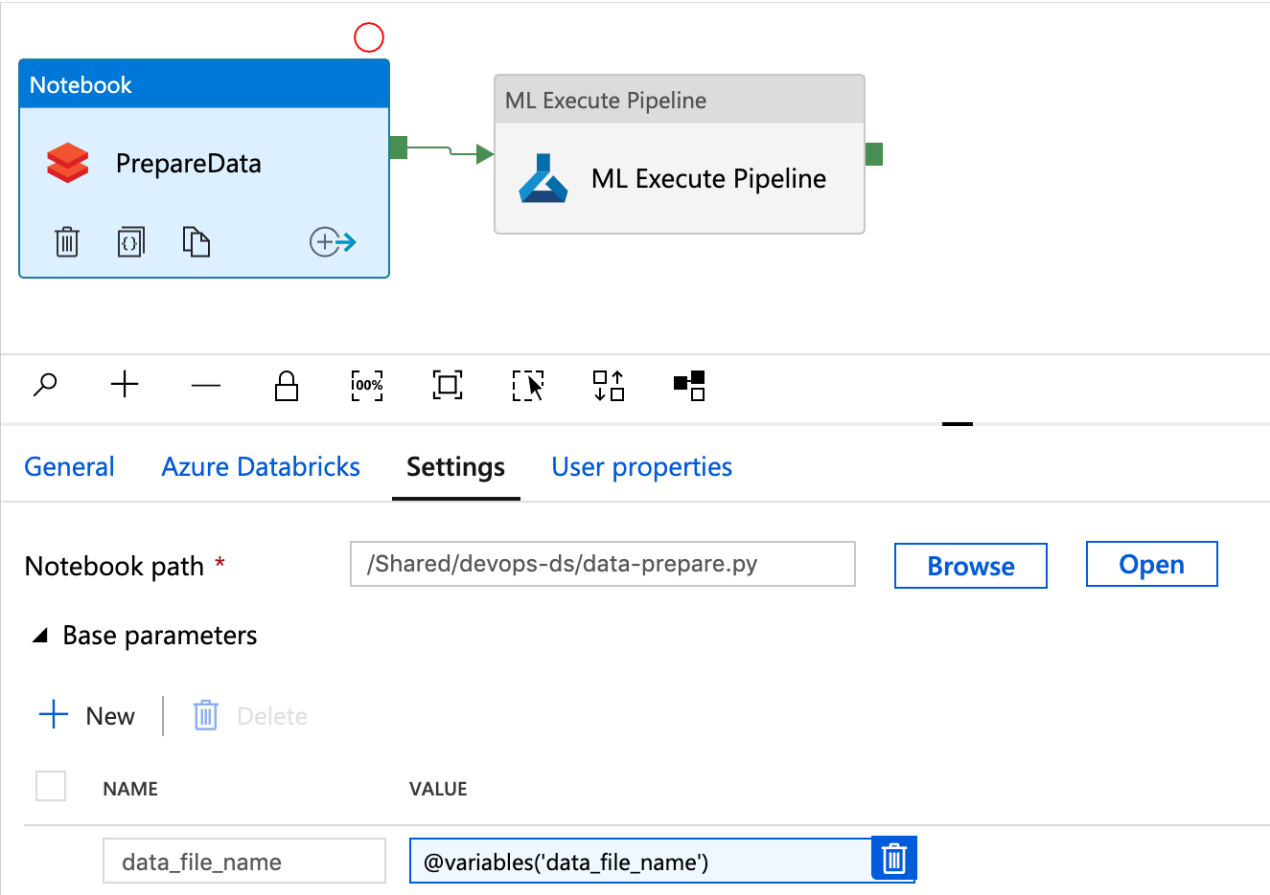
However, you may want to expose your custom properties that are not handled by the Azure Data Factory workspace by default. In the scenario of this article an Azure Data Factory pipeline invokes a Python notebook processing the data. The notebook accepts a parameter with the name of an input data file.

```
import pandas as pd
import numpy as np

data_file_name = getArgument("data_file_name")
data = pd.read_csv(data_file_name)

labels = np.array(data['target'])
...
```

This name is different for ***Dev***, ***QA***, ***UAT***, and ***PROD*** environments. In a complex pipeline with multiple activities, there can be several custom properties. It's good practice to collect all those values in one place and define them as pipeline ***variables***:



The pipeline activities may refer to the pipeline variables while actually using them:



The Azure Data Factory workspace ***doesn't*** expose pipeline variables as Azure Resource Manager templates parameters by default. The workspace uses the Default Parameterization Template dictating what pipeline properties should be exposed as Azure Resource Manager template parameters. In order to add pipeline variables to the list, update the "Microsoft.DataFactory/factories/pipelines" section of the Default Parameterization Template with the following snippet and place the result json file in the root of the source folder:

```
"Microsoft.DataFactory/factories/pipelines": {
        "properties": {
            "variables": {
                "*": {
                    "defaultValue": "="
                }
            }
        }
    }
```

Doing so will force the Azure Data Factory workspace to add the variables to the parameters list when the ***publish*** button is clicked:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "factoryName": {
            "value": "devops-ds-adf"
        },
        ...
        "data-ingestion-pipeline_properties_variables_data_file_name_defaultValue": {
            "value": "driver_prediction_train.csv"
        }
    }
}
```

The values in the json file are default values configured in the pipeline definition. They're expected to be overridden with the target environment values when the Azure Resource Manager template is deployed.

Continuous delivery (CD)

The Continuous Delivery process takes the artifacts and deploys them to the first target environment. It makes sure that the solution works by running tests. If successful, it continues to the next environment. The CD Azure Pipeline consists of multiple stages representing the environments. Each stage contains deployments and jobs that perform the following steps:

- Deploy a Python Notebook to Azure Databricks workspace
- Deploy an Azure Data Factory pipeline
- Run the pipeline
- Check the data ingestion result

The pipeline stages can be configured with approvals and gates that provide additional control on how the deployment process evolves through the chain of environments.

Deploy a Python Notebook

The following code snippet defines an Azure Pipeline deployment that copies a Python notebook to a Databricks cluster:

```
- stage: 'Deploy_to_QA'
  displayName: 'Deploy to QA'
  variables:
  - group: devops-ds-qa-vg
  jobs:
  - deployment: "Deploy_to_Databricks"
    displayName: 'Deploy to Databricks'
    timeoutInMinutes: 0
    environment: qa
    strategy:
      runOnce:
        deploy:
          steps:
            - task: UsePythonVersion@0
              inputs:
                versionSpec: '3.x'
                addToPath: true
                architecture: 'x64'
              displayName: 'Use Python3'

            - task: configuredatabricks@0
              inputs:
                url: '$(DATABRICKS_URL)'
                token: '$(DATABRICKS_TOKEN)'
              displayName: 'Configure Databricks CLI'

            - task: deploynotebooks@0
              inputs:
                notebooksFolderPath: '$(Pipeline.Workspace)/di-notebooks'
                workspaceFolder: '/Shared/devops-ds'
              displayName: 'Deploy (copy) data processing notebook to the Databricks cluster'
```

The artifacts produced by the CI are automatically copied to the deployment agent and are available in the *$(Pipeline.Workspace)* folder. In this case, the deployment task refers to the *di-notebooks* artifact containing the Python notebook. This deployment uses the Databricks Azure DevOps extension to copy the notebook files to the Databricks workspace. The *Deploy_to_QA* stage contains a reference to *devops-ds-qa-vg* variable group defined in the Azure DevOps project. The steps in this stage refer to the variables from this variable group (for example, $(DATABRICKS_URL), $(DATABRICKS_TOKEN)). The idea is that the next stage (for example, *Deploy_to_UAT*) will operate with the same variable names defined in its own UAT-scoped variable group.

Deploy an Azure Data Factory pipeline

A deployable artifact for Azure Data Factory is an Azure Resource Manager template. Therefore, it's going to be deployed with the *Azure Resource Group Deployment* task as it is demonstrated in the following snippet:

```
  - deployment: "Deploy_to_ADF"
    displayName: 'Deploy to ADF'
    timeoutInMinutes: 0
    environment: qa
    strategy:
      runOnce:
        deploy:
          steps:
            - task: AzureResourceGroupDeployment@2
              displayName: 'Deploy ADF resources'
              inputs:
                azureSubscription: $(AZURE_RM_CONNECTION)
                resourceGroupName: $(RESOURCE_GROUP)
                location: $(LOCATION)
                csmFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateForFactory.json'
                csmParametersFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateParametersForFactory.json'
                overrideParameters: -data-ingestion-pipeline_properties_variables_data_file_name_defaultValue
"$(DATA_FILE_NAME)"
```

The value of the data filename parameter comes from the $(DATA_FILE_NAME) variable defined in a QA stage variable group. Similarly, all parameters defined in ***ARMTemplateForFactory.json*** can be overridden. If they are not, then the default values are used.

Run the pipeline and check the data ingestion result

The next step is to make sure that the deployed solution is working. The following job definition runs an Azure Data Factory pipeline with a PowerShell script and executes a Python notebook on an Azure Databricks cluster. The notebook checks if the data has been ingested correctly and validates the result data file with $(bin_FILE_NAME) name.

```
  - job: "Integration_test_job"
    displayName: "Integration test job"
    dependsOn: [Deploy_to_Databricks, Deploy_to_ADF]
    pool:
      vmImage: 'ubuntu-latest'
    timeoutInMinutes: 0
    steps:
    - task: AzurePowerShell@4
      displayName: 'Execute ADF Pipeline'
      inputs:
        azureSubscription: $(AZURE_RM_CONNECTION)
        ScriptPath: '$(Build.SourcesDirectory)/adf/utils/Invoke-ADFPipeline.ps1'
        ScriptArguments: '-ResourceGroupName $(RESOURCE_GROUP) -DataFactoryName $(DATA_FACTORY_NAME) -
PipelineName $(PIPELINE_NAME)'
        azurePowerShellVersion: LatestVersion
    - task: UsePythonVersion@0
      inputs:
        versionSpec: '3.x'
        addToPath: true
        architecture: 'x64'
      displayName: 'Use Python3'

    - task: configuredatabricks@0
      inputs:
        url: '$(DATABRICKS_URL)'
        token: '$(DATABRICKS_TOKEN)'
      displayName: 'Configure Databricks CLI'

    - task: executenotebook@0
      inputs:
        notebookPath: '/Shared/devops-ds/test-data-ingestion'
        existingClusterId: '$(DATABRICKS_CLUSTER_ID)'
        executionParams: '{"bin_file_name":"$(bin_FILE_NAME)"}'
      displayName: 'Test data ingestion'

    - task: waitexecution@0
      displayName: 'Wait until the testing is done'
```

The final task in the job checks the result of the notebook execution. If it returns an error, it sets the status of pipeline execution to failed.

Putting pieces together

The outcome of this article is a CI/CD Azure Pipeline that consists of the following stages:

- CI
- Deploy To QA
  - Deploy to Databricks + Deploy to ADF
  - Integration Test

It contains a number of ***Deploy*** stages equal to the number of target environments you have. Each ***Deploy*** stage

contains two deployments that run in parallel and a job that runs after deployments to test the solution on the environment.

A sample implementation of the pipeline is assembled in the following *yaml* snippet:

```
variables:
- group: devops-ds-vg

stages:
- stage: 'CI'
  displayName: 'CI'
  jobs:
  - job: "CI_Job"
    displayName: "CI Job"
    pool:
      vmImage: 'ubuntu-latest'
    timeoutInMinutes: 0
    steps:
    - task: UsePythonVersion@0
      inputs:
        versionSpec: '3.x'
        addToPath: true
        architecture: 'x64'
      displayName: 'Use Python3'
    - script: pip install --upgrade flake8 flake8_formatter_junit_xml
      displayName: 'Install flake8'
    - checkout: self
    - script: |
       flake8 --output-file=$(Build.BinariesDirectory)/lint-testresults.xml --format junit-xml
    workingDirectory: '$(Build.SourcesDirectory)'
    displayName: 'Run flake8 (code style analysis)'
    - script: |
       python -m pytest --junitxml=$(Build.BinariesDirectory)/unit-testresults.xml $(Build.SourcesDirectory)
    displayName: 'Run unit tests'
    - task: PublishTestResults@2
    condition: succeededOrFailed()
    inputs:
        testResultsFiles: '$(Build.BinariesDirectory)/*-testresults.xml'
        testRunTitle: 'Linting & Unit tests'
        failTaskOnFailedTests: true
    displayName: 'Publish linting and unit test results'

    # The CI stage produces two artifacts (notebooks and ADF pipelines).
    # The pipelines Azure Resource Manager templates are stored in a technical branch "adf_publish"
    - publish: $(Build.SourcesDirectory)/$(Build.Repository.Name)/code/dataingestion
      artifact: di-notebooks
    - checkout: git://${{variables['System.TeamProject']}}@adf_publish
    - publish: $(Build.SourcesDirectory)/$(Build.Repository.Name)/devops-ds-adf
      artifact: adf-pipelines

- stage: 'Deploy_to_QA'
  displayName: 'Deploy to QA'
  variables:
  - group: devops-ds-qa-vg
  jobs:
  - deployment: "Deploy_to_Databricks"
    displayName: 'Deploy to Databricks'
    timeoutInMinutes: 0
    environment: qa
    strategy:
      runOnce:
        deploy:
          steps:
            - task: UsePythonVersion@0
              inputs:
                versionSpec: '3.x'
                addToPath: true
                architecture: 'x64'
```

```
              architecture: 'x64'
            displayName: 'Use Python3'

          - task: configuredatabricks@0
            inputs:
              url: '$(DATABRICKS_URL)'
              token: '$(DATABRICKS_TOKEN)'
            displayName: 'Configure Databricks CLI'

          - task: deploynotebooks@0
            inputs:
              notebooksFolderPath: '$(Pipeline.Workspace)/di-notebooks'
              workspaceFolder: '/Shared/devops-ds'
            displayName: 'Deploy (copy) data processing notebook to the Databricks cluster'
  - deployment: "Deploy_to_ADF"
    displayName: 'Deploy to ADF'
    timeoutInMinutes: 0
    environment: qa
    strategy:
      runOnce:
        deploy:
          steps:
            - task: AzureResourceGroupDeployment@2
              displayName: 'Deploy ADF resources'
              inputs:
                azureSubscription: $(AZURE_RM_CONNECTION)
                resourceGroupName: $(RESOURCE_GROUP)
                location: $(LOCATION)
                csmFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateForFactory.json'
                csmParametersFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateParametersForFactory.json'
                overrideParameters: -data-ingestion-pipeline_properties_variables_data_file_name_defaultValue
"$(DATA_FILE_NAME)"
  - job: "Integration_test_job"
    displayName: "Integration test job"
    dependsOn: [Deploy_to_Databricks, Deploy_to_ADF]
    pool:
      vmImage: 'ubuntu-latest'
    timeoutInMinutes: 0
    steps:
    - task: AzurePowerShell@4
      displayName: 'Execute ADF Pipeline'
      inputs:
        azureSubscription: $(AZURE_RM_CONNECTION)
        ScriptPath: '$(Build.SourcesDirectory)/adf/utils/Invoke-ADFPipeline.ps1'
        ScriptArguments: '-ResourceGroupName $(RESOURCE_GROUP) -DataFactoryName $(DATA_FACTORY_NAME) -
PipelineName $(PIPELINE_NAME)'
        azurePowerShellVersion: LatestVersion
    - task: UsePythonVersion@0
      inputs:
        versionSpec: '3.x'
        addToPath: true
        architecture: 'x64'
      displayName: 'Use Python3'

    - task: configuredatabricks@0
      inputs:
        url: '$(DATABRICKS_URL)'
        token: '$(DATABRICKS_TOKEN)'
      displayName: 'Configure Databricks CLI'

    - task: executenotebook@0
      inputs:
        notebookPath: '/Shared/devops-ds/test-data-ingestion'
        existingClusterId: '$(DATABRICKS_CLUSTER_ID)'
        executionParams: '{"bin_file_name":"$(bin_FILE_NAME)"}'
      displayName: 'Test data ingestion'

    - task: waitexecution@0
      displayName: 'Wait until the testing is done'
```

Next steps

- Source Control in Azure Data Factory
- Continuous integration and delivery in Azure Data Factory
- DevOps for Azure Databricks

Import your data into Azure Machine Learning designer (preview)

4/1/2020 • 2 minutes to read • Edit Online

In this article, you learn how to import your own data in the designer to create custom solutions. There are two ways you can import data into the designer:

- **Azure Machine Learning datasets** - Register datasets in Azure Machine Learning to enable advanced features that help you manage your data.
- **Import Data module** - Use the Import Data module to directly access data from online datasources.

Use Azure Machine Learning datasets

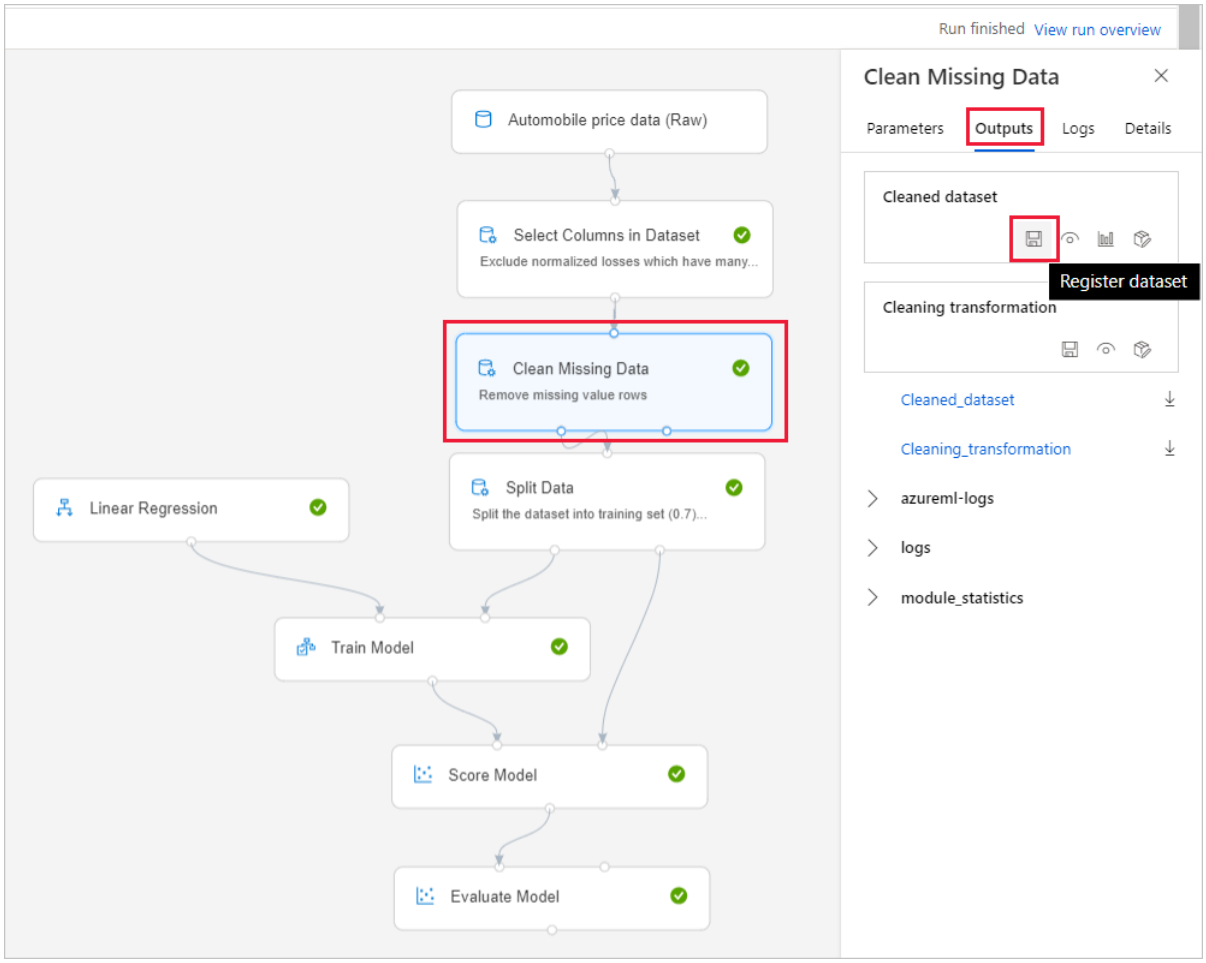
We recommend that you use datasets to import data into the designer. When you register a dataset, you can take full advantage of advanced data features like versioning and tracking and data monitoring.

Register a dataset

You can register existing datasets programatically with the SDK or visually in Azure Machine Learning studio.

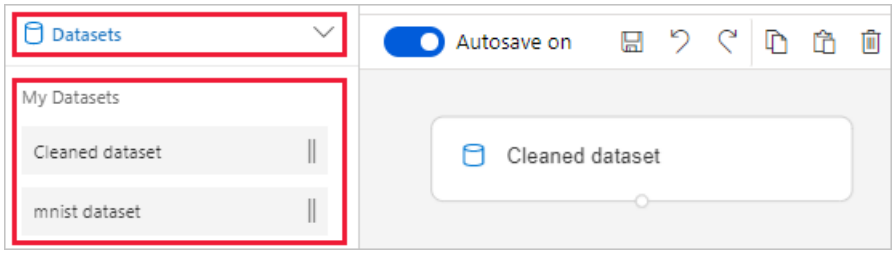
You can also register the output for any designer module as a dataset.

1. Select the module that outputs the data you want to register.
2. In the properties pane, select **Outputs** > **Register dataset**.



Use a dataset

Your registered datasets can be found in the module palette, under **Datasets** > **My Datasets**. To use a dataset, drag and drop it onto the pipeline canvas. Then, connect the output port of the dataset to other modules in the palette.



> **NOTE**
> The designer currently only supports processing tabular datasets. If you want to use file datasets, use the Azure Machine Learning SDK available for Python and R.

Import data using the Import Data module

While we recommend that you use datasets to import data, you can also use the Import Data module. The Import Data module skips registering your dataset in Azure Machine Learning and imports data directly from a datastore or HTTP URL.

For detailed information on how to use the Import Data module, see the Import Data reference page.

> **NOTE**
> If your dataset has too many columns, you may encounter the following error: "Validation failed due to size limitation". To avoid this, register the dataset in the Datasets interface.

Supported sources

This section lists the data sources supported by the designer. Data comes into the designer from either a datastore or from tabular dataset.

Datastore sources

For a list of supported datastore sources, see Access data in Azure storage services.

Tabular dataset sources

The designer supports tabular datasets created from the following sources:

- Delimited files
- JSON files
- Parquet files
- SQL queries

Data types

The designer internally recognizes the following data types:

- String
- Integer
- Decimal

- Boolean
- Date

The designer uses an internal data type to pass data between modules. You can explicitly convert your data into data table format using the Convert to Dataset module. Any module that accepts formats other than the internal format will convert the data silently before passing it to the next module.

Data constraints

Modules in the designer are limited by the size of the compute target. For larger datasets, you should use a larger Azure Machine Learning compute resource. For more information on Azure Machine Learning compute, see What are compute targets in Azure Machine Learning?

Next steps

Learn the basics of the designer with Tutorial: Predict automobile price with the designer.

Connect to Azure storage services

4/21/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, learn how to connect to Azure storage services via Azure Machine Learning datastores. Datastores store connection information, like your subscription ID and token authorization in your Key Vault associated with the workspace, so you can securely access your storage without having to hard code them in your scripts.

You can create datastores from these Azure storage solutions. For unsupported storage solutions, and to save data egress cost during machine learning experiments, we recommend that you move your data to supported Azure storage solutions.

Prerequisites

You'll need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning.
- An Azure storage account with an Azure blob container or Azure file share.
- The Azure Machine Learning SDK for Python, or access to Azure Machine Learning studio.
- An Azure Machine Learning workspace.

  Either create an Azure Machine Learning workspace or use an existing one via the Python SDK. Import the `Workspace` and `Datastore` class, and load your subscription information from the file `config.json` using the function `from_config()`. This looks for the JSON file in the current directory by default, but you can also specify a path parameter to point to the file using `from_config(path="your/file/path")`.

```
import azureml.core
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()
```

Supported data storage service types

Datastores currently support storing connection information to the storage services listed in the following matrix.

STORAGE TYPE	AUTHENTICATION TYPE	AZURE MACHINE LEARNING STUDIO	AZURE MACHINE LEARNING PYTHON SDK	AZURE MACHINE LEARNING CLI	AZURE MACHINE LEARNING REST API
Azure Blob Storage	Account key SAS token	✓	✓	✓	✓
Azure File Share	Account key SAS token	✓	✓	✓	✓
Azure Data Lake Storage Gen 1	Service principal	✓	✓	✓	✓

STORAGE TYPE	AUTHENTICATION TYPE	AZURE MACHINE LEARNING STUDIO	AZURE MACHINE LEARNING PYTHON SDK	AZURE MACHINE LEARNING CLI	AZURE MACHINE LEARNING REST API
Azure Data Lake Storage Gen 2	Service principal	✓	✓	✓	✓
Azure SQL Database	SQL authentication Service principal	✓	✓	✓	✓
Azure PostgreSQL	SQL authentication	✓	✓	✓	✓
Azure Database for MySQL	SQL authentication		✓*	✓*	✓*
Databricks File System	No authentication		✓**	✓ **	✓**

*MySQL is only supported for pipeline DataTransferStep.
**Databricks is only supported for pipeline DatabricksStep

Storage guidance

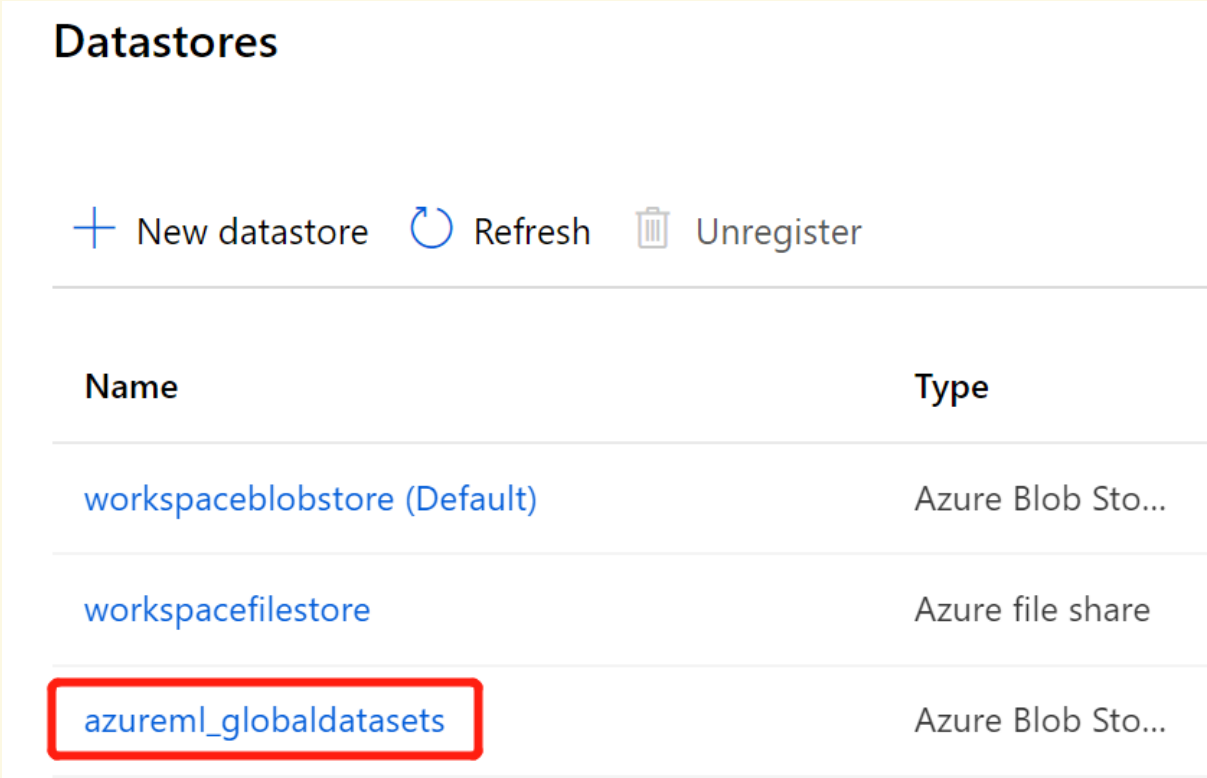
We recommend creating a datastore for an Azure Blob container. Both standard and premium storage are available for blobs. Although premium storage is more expensive, its faster throughput speeds might improve the speed of your training runs, particularly if you train against a large dataset. For information about the cost of storage accounts, see the Azure pricing calculator.

Azure Data Lake Storage Gen2 is built on top of Azure Blob storage and designed for enterprise big data analytics. A fundamental part of Data Lake Storage Gen2 is the addition of a hierarchical namespace to Blob storage. The hierarchical namespace organizes objects/files into a hierarchy of directories for efficient data access.

When you create a workspace, an Azure blob container and an Azure file share are automatically registered to the workspace. They're named `workspaceblobstore` and `workspacefilestore`, respectively. `workspaceblobstore` is used to store workspace artifacts and your machine learning experiment logs. `workspacefilestore` is used to store notebooks and R scripts authorized via compute instance. The `workspaceblobstore` container is set as the default datastore.

IMPORTANT

Azure Machine Learning designer (preview) will create a datastore named **azureml_globaldatasets** automatically when you open a sample in the designer homepage. This datastore only contains sample datasets. Please **do not** use this datastore for any confidential data access.



Create and register datastores

When you register an Azure storage solution as a datastore, you automatically create and register that datastore to a specific workspace. You can create and register datastores to a workspace by using the Python SDK or Azure Machine Learning studio.

IMPORTANT

As part of the initial datastore create and register process, Azure Machine Learning validates that the underlying storage service exists and that the user provided principal (username, service principal or SAS token) has access to that storage. For Azure Data Lake Storage Gen 1 and 2 datastores, however, this validation happens later, when data access methods like `from_files()` or `from_delimited_files()` are called.

After datastore creation, this validation is only performed for methods that require access to the underlying storage container, **not** each time datastore objects are retrieved. For example, validation happens if you want to download files from your datastore; but if you just want to change your default datastore, then validation does not happen.

Python SDK

All the register methods are on the `Datastore` class and have the form `register_azure_*`.

You can find the information that you need to populate the `register()` method on the Azure portal. Select **Storage Accounts** on the left pane, and choose the storage account that you want to register. The **Overview** page provides information such as the account name, container, and file share name.

- For authentication items, like account key or SAS token, go to **Access keys** on the **Settings** pane.
- For service principal items like, tenant ID and client ID, go to your **App registrations** and select which

app you want to use. Its corresponding **Overview** page will contain these items.

> **IMPORTANT**
>
> If your storage account is in a virtual network, only creation of datastores **via the SDK** is supported.

The following examples show how to register an Azure blob container, an Azure file share, and Azure Data Lake Storage Generation 2 as a datastore. For other storage services, please see the reference documentation for the applicable `register_azure_*` methods.

Blob container

To register an Azure blob container as a datastore, use `register_azure_blob-container()`.

The following code creates and registers the `blob_datastore_name` datastore to the `ws` workspace. This datastore accesses the `my-container-name` blob container on the `my-account-name` storage account, by using the provided account access key.

```
blob_datastore_name='azblobsdk' # Name of the datastore to workspace
container_name=os.getenv("BLOB_CONTAINER", "<my-container-name>") # Name of Azure blob container
account_name=os.getenv("BLOB_ACCOUNTNAME", "<my-account-name>") # Storage account name
account_key=os.getenv("BLOB_ACCOUNT_KEY", "<my-account-key>") # Storage account access key

blob_datastore = Datastore.register_azure_blob_container(workspace=ws,
                                                         datastore_name=blob_datastore_name,
                                                         container_name=container_name,
                                                         account_name=account_name,
                                                         account_key=account_key)
```

If your blob container is in virtual network, set `skip_validation=True` using `register_azure_blob-container()`.

File share

To register an Azure file share as a datastore, use `register_azure_file_share()`.

The following code creates and registers the `file_datastore_name` datastore to the `ws` workspace. This datastore accesses the `my-fileshare-name` file share on the `my-account-name` storage account, by using the provided account access key.

```
file_datastore_name='azfilesharesdk' # Name of the datastore to workspace
file_share_name=os.getenv("FILE_SHARE_CONTAINER", "<my-fileshare-name>") # Name of Azure file share container
account_name=os.getenv("FILE_SHARE_ACCOUNTNAME", "<my-account-name>") # Storage account name
account_key=os.getenv("FILE_SHARE_ACCOUNT_KEY", "<my-account-key>") # Storage account access key

file_datastore = Datastore.register_azure_file_share(workspace=ws,
                                                     datastore_name=file_datastore_name,
                                                     file_share_name=file_share_name,
                                                     account_name=account_name,
                                                     account_key=account_key)
```

If your file share is in virtual network, set `skip_validation=True` using `register_azure_file_share()`.

Azure Data Lake Storage Generation 2

For an Azure Data Lake Storage Generation 2 (ADLS Gen 2) datastore, use register_azure_data_lake_gen2() to register a credential datastore connected to an Azure DataLake Gen 2 storage with service principal permissions. In order to utilize your service principal you need to register your application and grant the service principal with the right data access. Learn more about access control set up for ADLS Gen 2.

The following code creates and registers the `adlsgen2_datastore_name` datastore to the `ws` workspace. This datastore accesses the file system `test` on the `account_name` storage account, by using the provided service

principal credentials.

```
adlsgen2_datastore_name = 'adlsgen2datastore'

subscription_id=os.getenv("ADL_SUBSCRIPTION", "<my_subscription_id>") # subscription id of ADLS account
resource_group=os.getenv("ADL_RESOURCE_GROUP", "<my_resource_group>") # resource group of ADLS account

account_name=os.getenv("ADLSGEN2_ACCOUNTNAME", "<my_account_name>") # ADLS Gen2 account name
tenant_id=os.getenv("ADLSGEN2_TENANT", "<my_tenant_id>") # tenant id of service principal
client_id=os.getenv("ADLSGEN2_CLIENTID", "<my_client_id>") # client id of service principal
client_secret=os.getenv("ADLSGEN2_CLIENT_SECRET", "<my_client_secret>") # the secret of service principal

adlsgen2_datastore = Datastore.register_azure_data_lake_gen2(workspace=ws,
                                                             datastore_name=adlsgen2_datastore_name,
                                                             account_name=account_name, # ADLS Gen2 account
name
                                                             filesystem='test', # ADLS Gen2 filesystem
                                                             tenant_id=tenant_id, # tenant id of service
principal
                                                             client_id=client_id, # client id of service
principal
                                                             client_secret=client_secret) # the secret of
service principal
```

Azure Machine Learning studio

Create a new datastore in a few steps in Azure Machine Learning studio:

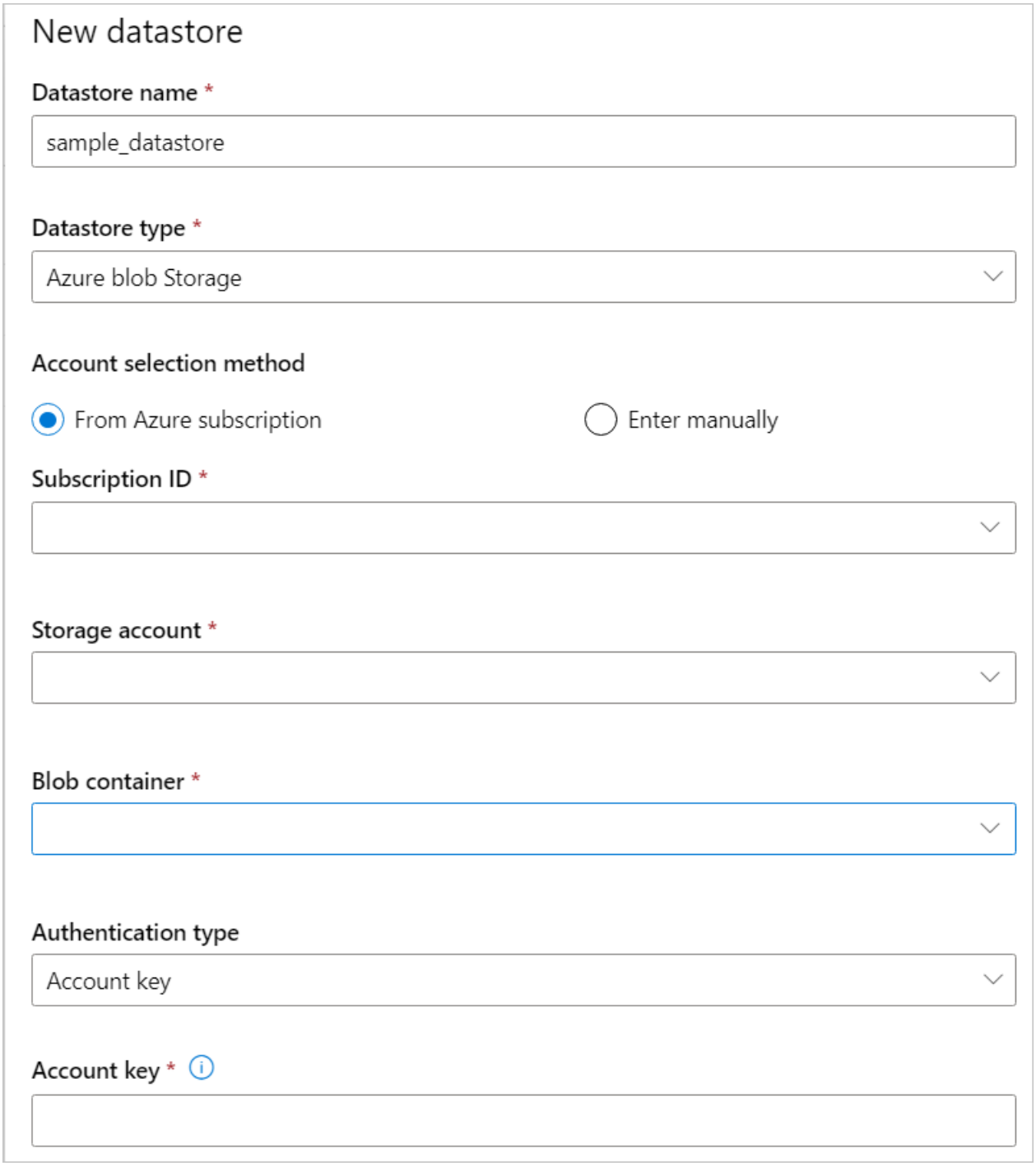
> **IMPORTANT**
>
> If your storage account is in a virtual network, only creation of datastores via the SDK is supported.

1. Sign in to Azure Machine Learning studio.
2. Select **Datastores** on the left pane under **Manage**.
3. Select **+ New datastore**.
4. Complete the form for a new datastore. The form intelligently updates itself based on your selections for Azure storage type and authentication type.

You can find the information that you need to populate the form on the Azure portal. Select **Storage Accounts** on the left pane, and choose the storage account that you want to register. The **Overview** page provides information such as the account name, container, and file share name.

- For authentication items, like account key or SAS token, go to **Access keys** on the **Settings** pane.
- For service principal items like, tenant ID and client ID, go to your **App registrations** and select which app you want to use. Its corresponding **Overview** page will contain these items.

The following example demonstrates what the form looks like when you create an Azure blob datastore:



Get datastores from your workspace

To get a specific datastore registered in the current workspace, use the `get()` static method on the `Datastore` class:

```
# Get a named datastore from the current workspace
datastore = Datastore.get(ws, datastore_name='your datastore name')
```

To get the list of datastores registered with a given workspace, you can use the `datastores` property on a workspace object:

```
# List all datastores registered in the current workspace
datastores = ws.datastores
for name, datastore in datastores.items():
    print(name, datastore.datastore_type)
```

To get the workspace's default datastore, use this line:

```
datastore = ws.get_default_datastore()
```

Upload and download data

The `upload()` and `download()` methods described in the following examples are specific to, and operate identically for, the AzureBlobDatastore and AzureFileDatastore classes.

Upload

Upload either a directory or individual files to the datastore by using the Python SDK:

```
datastore.upload(src_dir='your source directory',
                 target_path='your target path',
                 overwrite=True,
                 show_progress=True)
```

The `target_path` parameter specifies the location in the file share (or blob container) to upload. It defaults to `None`, so the data is uploaded to root. If `overwrite=True`, any existing data at `target_path` is overwritten.

You can also upload a list of individual files to the datastore via the `upload_files()` method.

Download

Download data from a datastore to your local file system:

```
datastore.download(target_path='your target path',
                   prefix='your prefix',
                   show_progress=True)
```

The `target_path` parameter is the location of the local directory to download the data to. To specify a path to the folder in the file share (or blob container) to download, provide that path to `prefix`. If `prefix` is `None`, all the contents of your file share (or blob container) will be downloaded.

Access your data during training

To interact with data in your datastores or to package your data into a consumable object for machine learning tasks, like training, create an Azure Machine Learning dataset. Datasets provide functions that load tabular data into a pandas or Spark DataFrame. Datasets also provide the ability to download or mount files of any format from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL. Learn more about how to train with datasets.

Accessing source code during training

Azure Blob storage has higher throughput speeds than an Azure file share and will scale to large numbers of jobs started in parallel. For this reason, we recommend configuring your runs to use Blob storage for transferring source code files.

The following code example specifies in the run configuration which blob datastore to use for source code transfers.

```
# workspaceblobstore is the default blob storage
run_config.source_directory_data_store = "workspaceblobstore"
```

Access data during scoring

Azure Machine Learning provides several ways to use your models for scoring. Some of these methods don't provide access to datastores. Use the following table to understand which methods allow you to access datastores during scoring:

METHOD	DATASTORE ACCESS	DESCRIPTION
Batch prediction	✔	Make predictions on large quantities of data asynchronously.
Web service		Deploy models as a web service.
Azure IoT Edge module		Deploy models to IoT Edge devices.

For situations where the SDK doesn't provide access to datastores, you might be able to create custom code by using the relevant Azure SDK to access the data. For example, the Azure Storage SDK for Python is a client library that you can use to access data stored in blobs or files.

Move data to supported Azure storage solutions

Azure Machine Learning supports accessing data from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL. If you're using unsupported storage, we recommend that you move your data to supported Azure storage solutions by using Azure Data Factory and these steps. Moving data to supported storage can help you save data egress costs during machine learning experiments.

Azure Data Factory provides efficient and resilient data transfer with more than 80 prebuilt connectors at no additional cost. These connectors include Azure data services, on-premises data sources, Amazon S3 and Redshift, and Google BigQuery.

Next steps

- Create an Azure machine learning dataset
- Train a model
- Deploy a model

Create Azure Machine Learning datasets

4/19/2020 • 11 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to create Azure Machine Learning datasets to access data for your local or remote experiments.

With Azure Machine Learning datasets, you can:

- Keep a single copy of data in your storage, referenced by datasets.
- Seamlessly access data during model training without worrying about connection strings or data paths.
- Share data and collaborate with other users.

Prerequisites

' To create and work with datasets, you need:

- An Azure subscription. If you don't have one, create a free account before you begin. Try the free or paid version of Azure Machine Learning.
- An Azure Machine Learning workspace.
- The Azure Machine Learning SDK for Python installed, which includes the azureml-datasets package.

> **NOTE**
> Some dataset classes have dependencies on the azureml-dataprep package. For Linux users, these classes are supported only on the following distributions: Red Hat Enterprise Linux, Ubuntu, Fedora, and CentOS.

Compute size guidance

When creating a dataset review your compute processing power and the size of your data in memory. The size of your data in storage is not the same as the size of data in a dataframe. For example, data in CSV files can expand up to 10x in a dataframe, so a 1 GB CSV file can become 10 GB in a dataframe.

The main factor is how large the dataset is in-memory, i.e. as a dataframe. We recommend your compute size and processing power contain 2x the size of RAM. So if your dataframe is 10GB, you want a compute target with 20+ GB of RAM to ensure that the dataframe can comfortable fit in memory and be processed. If your data is compressed, it can expand further; 20 GB of relatively sparse data stored in compressed parquet format can expand to ~800 GB in memory. Since Parquet files store data in a columnar format, if you only need half of the columns, then you only need to load ~400 GB in memory.

If you're using Pandas, there's no reason to have more than 1 vCPU since that's all it will use. You can easily parallelize to many vCPUs on a single Azure Machine Learning compute instance/node via Modin and Dask/Ray, and scale out to a large cluster if needed, by simply changing `import pandas as pd` to `import modin.pandas as pd`.

If you can't get a big enough virtual for the data, you have two options: use a framework like Spark or Dask to perform the processing on the data 'out of memory', i.e. the dataframe is loaded into RAM partition by partition and processed, with the final result being gathered at the end. If this is too slow, Spark or Dask allow you to

scale out to a cluster which can still be used interactively.

Dataset types

There are two dataset types, based on how users consume them in training:

- TabularDataset represents data in a tabular format by parsing the provided file or list of files. This provides you with the ability to materialize the data into a Pandas or Spark DataFrame. You can create a `TabularDataset` object from .csv, .tsv, .parquet, .jsonl files, and from SQL query results. For a complete list, see TabularDatasetFactory class.
- The FileDataset class references single or multiple files in your datastores or public URLs. By this method, you can download or mount the files to your compute as a FileDataset object. The files can be in any format, which enables a wider range of machine learning scenarios, including deep learning.

To learn more about upcoming API changes, see Dataset API change notice.

Create datasets

By creating a dataset, you create a reference to the data source location, along with a copy of its metadata. Because the data remains in its existing location, you incur no extra storage cost. You can create both `TabularDataset` and `FileDataset` data sets by using the Python SDK or at https://ml.azure.com.

For the data to be accessible by Azure Machine Learning, datasets must be created from paths in Azure datastores or public web URLs.

Use the SDK

To create datasets from an Azure datastore by using the Python SDK:

1. Verify that you have `contributor` or `owner` access to the registered Azure datastore.
2. Create the dataset by referencing paths in the datastore.

> **NOTE**
> You can create a dataset from multiple paths in multiple datastores. There is no hard limit on the number of files or data size that you can create a dataset from. However, for each data path, a few requests will be sent to the storage service to check whether it points to a file or a folder. This overhead may lead to degraded performance or failure. A dataset referencing one folder with 1000 files inside is considered referencing one data path. We'd recommend creating dataset referencing less than 100 paths in datastores for optimal performance.

Create a TabularDataset

Use the `from_delimited_files()` method on the `TabularDatasetFactory` class to read files in .csv or .tsv format, and to create an unregistered TabularDataset. If you're reading from multiple files, results will be aggregated into one tabular representation.

```
from azureml.core import Workspace, Datastore, Dataset

datastore_name = 'your datastore name'

# get existing workspace
workspace = Workspace.from_config()

# retrieve an existing datastore in the workspace by name
datastore = Datastore.get(workspace, datastore_name)

# create a TabularDataset from 3 paths in datastore
datastore_paths = [(datastore, 'weather/2018/11.csv'),
                   (datastore, 'weather/2018/12.csv'),
                   (datastore, 'weather/2019/*.csv')]
weather_ds = Dataset.Tabular.from_delimited_files(path=datastore_paths)
```

By default, when you create a TabularDataset, column data types are inferred automatically. If the inferred types don't match your expectations, you can specify column types by using the following code. The parameter `infer_column_type` is only applicable for datasets created from delimited files.You can also learn more about supported data types.

IMPORTANT

If your storage is behind a virtual network or firewall, only creation of a dataset via the SDK is supported. To create your dataset, be sure to include the parameters `validate=False` and `infer_column_types=False` in your `from_delimited_files()` method. This bypasses the initial validation check and ensures that you can create your dataset from these secure files.

```
from azureml.core import Dataset
from azureml.data.dataset_factory import DataType

# create a TabularDataset from a delimited file behind a public web url and convert column "Survived" to
boolean
web_path ='https://dprepdata.blob.core.windows.net/demo/Titanic.csv'
titanic_ds = Dataset.Tabular.from_delimited_files(path=web_path, set_column_types={'Survived':
DataType.to_bool()})

# preview the first 3 rows of titanic_ds
titanic_ds.take(3).to_pandas_dataframe()
```

	PASSENGERID	SURVIVED	PCLASS	NAME	SEX	AGE	SIBSP	PARCH	TICKET	FARE	CABIN	EMBARKED
0	1	False	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500		S

	PASSENGERID	SURVIVED	PCLASS	NAME	SEX	AGE	SIBSP	PARCH	TICKET	FARE	CABIN	EMBARKED
1	2	True	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	True	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250		S

To create a dataset from an in memory pandas dataframe, write the data to a local file, like a csv, and create your dataset from that file. The following code demonstrates this workflow.

```
local_path = 'data/prepared.csv'
dataframe.to_csv(local_path)
upload the local file to a datastore on the cloud
# azureml-core of version 1.0.72 or higher is required
# azureml-dataprep[pandas] of version 1.1.34 or higher is required
from azureml.core import Workspace, Dataset

subscription_id = 'xxxxxxxxxxxxxxxxxxxx'
resource_group = 'xxxxxx'
workspace_name = 'xxxxxxxxxxxxxxx'

workspace = Workspace(subscription_id, resource_group, workspace_name)

# get the datastore to upload prepared data
datastore = workspace.get_default_datastore()

# upload the local file from src_dir to the target_path in datastore
datastore.upload(src_dir='data', target_path='data')
# create a dataset referencing the cloud location
dataset = Dataset.Tabular.from_delimited_files(datastore.path('data/prepared.csv'))
```

Use the `from_sql_query()` method on the `TabularDatasetFactory` class to read from Azure SQL Database:

```
from azureml.core import Dataset, Datastore

# create tabular dataset from a SQL database in datastore
sql_datastore = Datastore.get(workspace, 'mssql')
sql_ds = Dataset.Tabular.from_sql_query((sql_datastore, 'SELECT * FROM my_table'))
```

In TabularDatasets, you can specify a time stamp from a column in the data or from wherever the path pattern data is stored to enable a time series trait. This specification allows for easy and efficient filtering by time.

Use the `with_timestamp_columns()` method on the `TabularDataset` class to specify your time stamp column and to enable filtering by time. For more information, see Tabular time series-related API demo with NOAA weather data.

```
# create a TabularDataset with time series trait
datastore_paths = [(datastore, 'weather/*/*/*/data.parquet')]

# get a coarse timestamp column from the path pattern
dataset = Dataset.Tabular.from_parquet_files(path=datastore_path,
partition_format='weather/{coarse_time:yyy/MM/dd}/data.parquet')

# set coarse timestamp to the virtual column created, and fine grain timestamp from a column in the data
dataset = dataset.with_timestamp_columns(fine_grain_timestamp='datetime',
coarse_grain_timestamp='coarse_time')

# filter with time-series-trait-specific methods
data_slice = dataset.time_before(datetime(2019, 1, 1))
data_slice = dataset.time_after(datetime(2019, 1, 1))
data_slice = dataset.time_between(datetime(2019, 1, 1), datetime(2019, 2, 1))
data_slice = dataset.time_recent(timedelta(weeks=1, days=1))
```

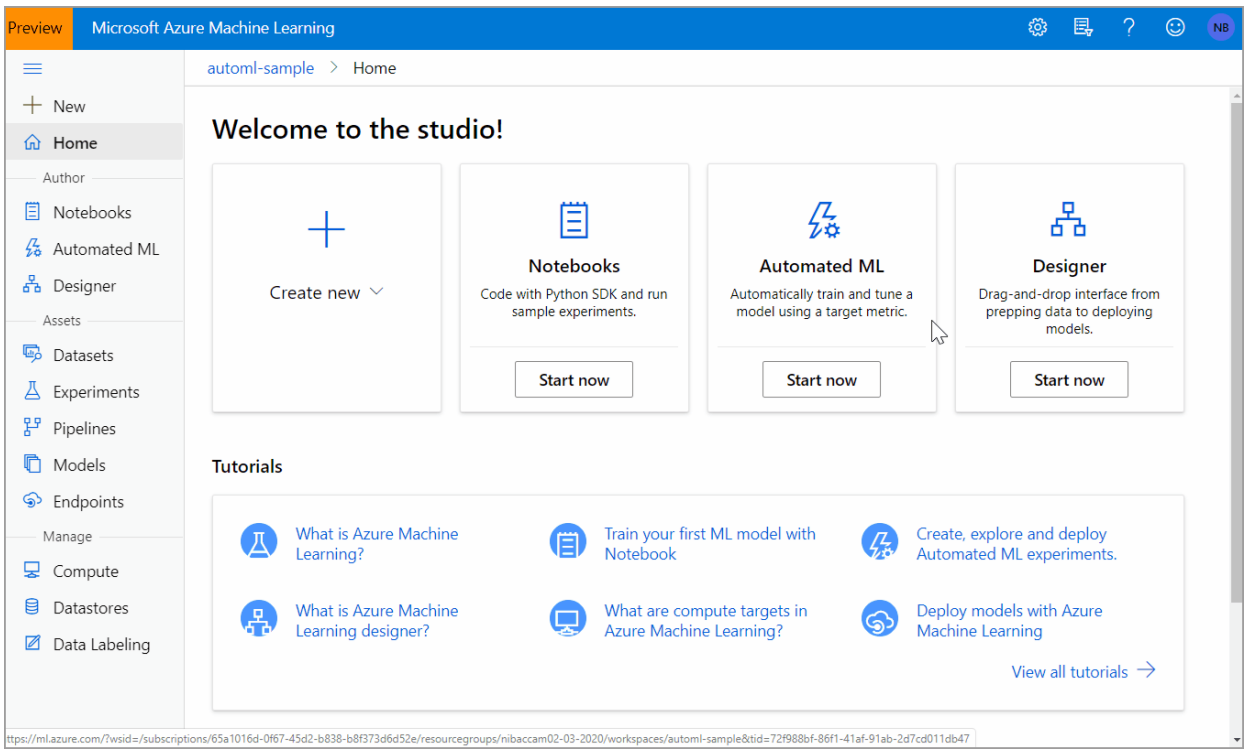
Create a FileDataset

Use the `from_files()` method on the `FileDatasetFactory` class to load files in any format and to create an unregistered FileDataset. If your storage is behind a a virtual network or firewall, set the parameter `validate =False` in your `from_files()` method. This bypasses the initial validation step, and ensures that you can create your dataset from these secure files.

```
# create a FileDataset pointing to files in 'animals' folder and its subfolders recursively
datastore_paths = [(datastore, 'animals')]
animal_ds = Dataset.File.from_files(path=datastore_paths)

# create a FileDataset from image and label files behind public web urls
web_paths = ['https://azureopendatastorage.blob.core.windows.net/mnist/train-images-idx3-ubyte.gz',
             'https://azureopendatastorage.blob.core.windows.net/mnist/train-labels-idx1-ubyte.gz']
mnist_ds = Dataset.File.from_files(path=web_paths)
```

On the web

The following steps and animation show how to create a dataset in Azure Machine Learning studio, https://ml.azure.com.



To create a dataset in the studio:

1. Sign in at https://ml.azure.com.
2. Select **Datasets** in the **Assets** section of the left pane.
3. Select **Create Dataset** to choose the source of your dataset. This source can be local files, a datastore, or public URLs.
4. Select **Tabular** or **File** for Dataset type.
5. Select **Next** to open the **Datastore and file selection** form. On this form you select where to keep your dataset after creation, as well as select what data files to use for your dataset.
6. Select **Next** to populate the **Settings and preview** and **Schema** forms; they are intelligently populated based on file type and you can further configure your dataset prior to creation on these forms.
7. Select **Next** to review the **Confirm details** form. Check your selections and create an optional data profile for your dataset. Learn more about data profiling.
8. Select **Create** to complete your dataset creation.

Register datasets

To complete the creation process, register your datasets with a workspace. Use the `register()` method to register datasets with your workspace in order to share them with others and reuse them across various experiments:

```
titanic_ds = titanic_ds.register(workspace=workspace,
                                 name='titanic_ds',
                                 description='titanic training data')
```

> **NOTE**
> Datasets created through Azure Machine Learning studio are automatically registered to the workspace.

Create datasets with Azure Open Datasets

Azure Open Datasets are curated public datasets that you can use to add scenario-specific features to machine learning solutions for more accurate models. Datasets include public-domain data for weather, census, holidays, public safety, and location that help you train machine learning models and enrich predictive solutions. Open Datasets are in the cloud on Microsoft Azure and are included in both the SDK and the workspace UI.

Use the SDK

To create datasets with Azure Open Datasets from the SDK, make sure you've installed the package with `pip install azureml-opendatasets`. Each discrete data set is represented by its own class in the SDK, and certain classes are available as either a `TabularDataset`, `FileDataset`, or both. See the reference documentation for a full list of classes.

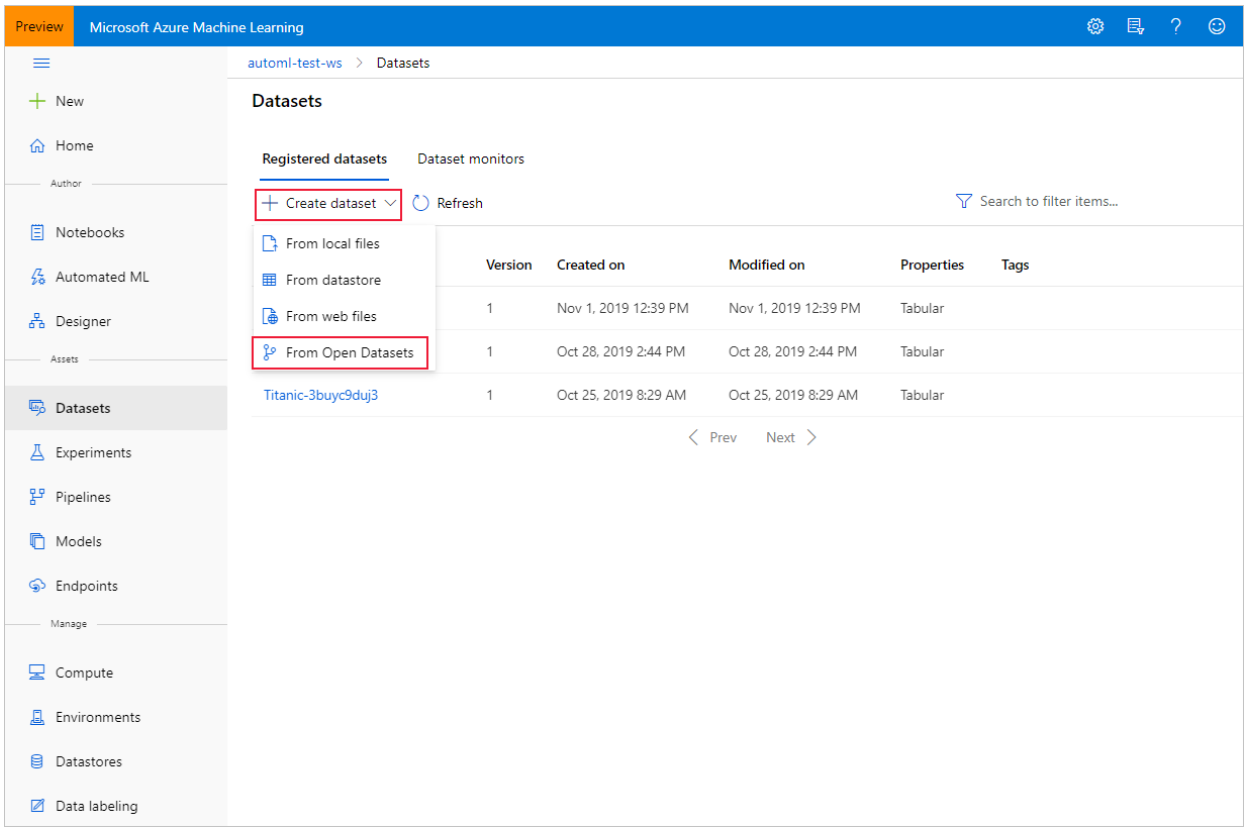
You can retrieve certain classes as either a `TabularDataset` or `FileDataset`, which allows you to manipulate and/or download the files directly. Other classes can get a dataset **only** by using one of `get_tabular_dataset()` or `get_file_dataset()` functions. The following code sample shows a few examples of these types of classes.

```
from azureml.opendatasets import MNIST

# MNIST class can return either TabularDataset or FileDataset
tabular_dataset = MNIST.get_tabular_dataset()
file_dataset = MNIST.get_file_dataset()

from azureml.opendatasets import Diabetes

# Diabetes class can return ONLY TabularDataset and must be called from the static function
diabetes_tabular = Diabetes.get_tabular_dataset()
```

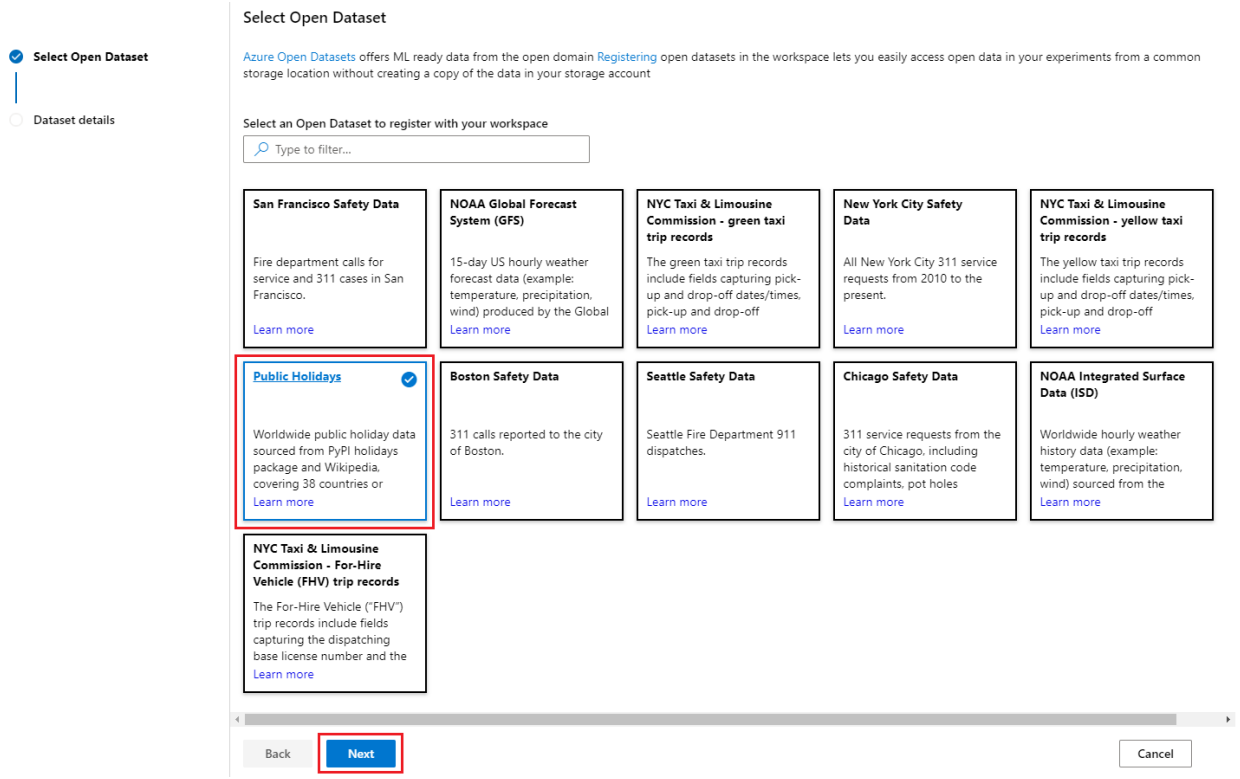
When you register a dataset created from Open Datasets, no data is immediately downloaded, but the data will be accessed later when requested (during training, for example) from a central storage location.

Use the UI

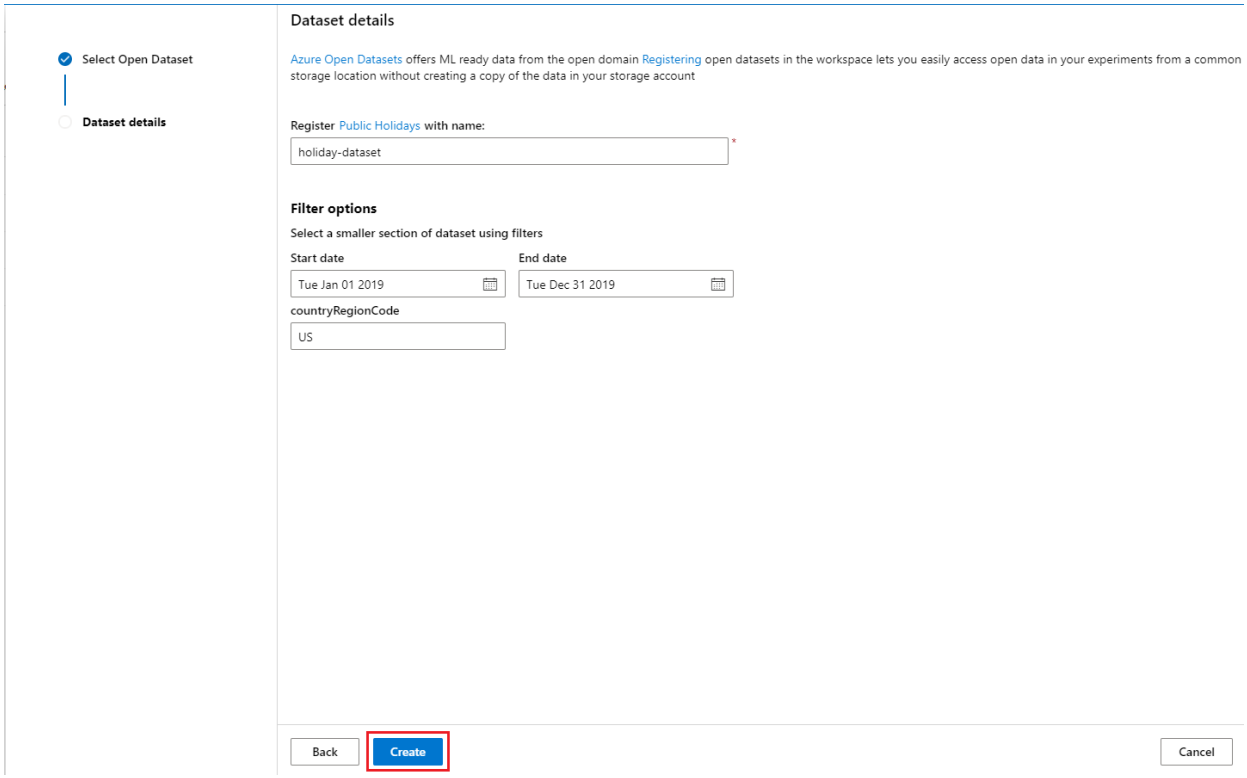
You can also create datasets from Open Datasets classes through the UI. In your workspace, select the **Datasets** tab under **Assets**. On the **Create dataset** drop-down menu, select **From Open Datasets**.



Select a dataset by selecting its tile. (You have the option to filter by using the search bar.) Select **Next**.



Choose a name under which to register the dataset, and optionally filter the data by using the available filters. In this case, for the public holidays dataset, you filter the time period to one year and the country code to only the US. Select **Create**.



The dataset is now available in your workspace under **Datasets**. You can use it in the same way as other datasets you've created.

Version datasets

You can register a new dataset under the same name by creating a new version. A dataset version is a way to bookmark the state of your data so that you can apply a specific version of the dataset for experimentation or future reproduction. Learn more about dataset versions.

```
# create a TabularDataset from Titanic training data
web_paths = ['https://dprepdata.blob.core.windows.net/demo/Titanic.csv',
             'https://dprepdata.blob.core.windows.net/demo/Titanic2.csv']
titanic_ds = Dataset.Tabular.from_delimited_files(path=web_paths)

# create a new version of titanic_ds
titanic_ds = titanic_ds.register(workspace = workspace,
                                 name = 'titanic_ds',
                                 description = 'new titanic training data',
                                 create_new_version = True)
```

Access datasets in your script

Registered datasets are accessible both locally and remotely on compute clusters like the Azure Machine Learning compute. To access your registered dataset across experiments, use the following code to access your workspace and registered dataset by name. By default, the `get_by_name()` method on the `Dataset` class returns the latest version of the dataset that's registered with the workspace.

```
%%writefile $script_folder/train.py

from azureml.core import Dataset, Run

run = Run.get_context()
workspace = run.experiment.workspace

dataset_name = 'titanic_ds'

# Get a dataset by name
titanic_ds = Dataset.get_by_name(workspace=workspace, name=dataset_name)

# Load a TabularDataset into pandas DataFrame
df = titanic_ds.to_pandas_dataframe()
```

Next steps

- Learn how to train with datasets.
- Use automated machine learning to train with TabularDatasets.
- For more dataset training examples, see the sample notebooks.

Train with datasets in Azure Machine Learning

3/9/2020 • 5 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn the two ways to consume Azure Machine Learning datasets in a remote experiment training runs without worrying about connection strings or data paths.

- Option 1: If you have structured data, create a TabularDataset and use it directly in your training script.
- Option 2: If you have unstructured data, create a FileDataset and mount or download files to a remote compute for training.

Azure Machine Learning datasets provide a seamless integration with Azure Machine Learning training products like ScriptRun, Estimator, HyperDrive and Azure Machine Learning pipelines.

Prerequisites

To create and train with datasets, you need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- An Azure Machine Learning workspace.
- The Azure Machine Learning SDK for Python installed, which includes the azureml-datasets package.

> **NOTE**
> Some Dataset classes have dependencies on the azureml-dataprep package. For Linux users, these classes are supported only on the following distributions: Red Hat Enterprise Linux, Ubuntu, Fedora, and CentOS.

Option 1: Use datasets directly in training scripts

In this example, you create a TabularDataset and use it as a direct input to your `estimator` object for training.

Create a TabularDataset

The following code creates an unregistered TabularDataset from a web url. You can also create datasets from local files or paths in datastores. Learn more about how to create datasets.

```
from azureml.core.dataset import Dataset

web_path ='https://dprepdata.blob.core.windows.net/demo/Titanic.csv'
titanic_ds = Dataset.Tabular.from_delimited_files(path=web_path)
```

Access the input dataset in your training script

TabularDataset objects provide the ability to load the data into a pandas or spark DataFrame so that you can work with familiar data preparation and training libraries. To leverage this capability, you can pass a TabularDataset as the input in your training configuration, and then retrieve it in your script.

To do so, access the input dataset through the `Run` object in your training script and use the `to_pandas_dataframe()` method.

```
%%writefile $script_folder/train_titanic.py

from azureml.core import Dataset, Run

run = Run.get_context()
# get the input dataset by name
dataset = run.input_datasets['titanic']
# load the TabularDataset to pandas DataFrame
df = dataset.to_pandas_dataframe()
```

Configure the estimator

An estimator object is used to submit the experiment run. Azure Machine Learning has pre-configured estimators for common machine learning frameworks, as well as a generic estimator.

This code creates a generic estimator object, `est`, that specifies

- A script directory for your scripts. All the files in this directory are uploaded into the cluster nodes for execution.
- The training script, *train_titanic.py*.
- The input dataset for training, `titanic`. `as_named_input()` is required so that the input dataset can be referenced by the assigned name in your training script.
- The compute target for the experiment.
- The environment definition for the experiment.

```
est = Estimator(source_directory=script_folder,
                entry_script='train_titanic.py',
                # pass dataset object as an input with name 'titanic'
                inputs=[titanic_ds.as_named_input('titanic')],
                compute_target=compute_target,
                environment_definition= conda_env)

# Submit the estimator as part of your experiment run
experiment_run = experiment.submit(est)
experiment_run.wait_for_completion(show_output=True)
```

Option 2: Mount files to a remote compute target

If you want to make your data files available on the compute target for training, use FileDataset to mount or download files referred by it.

Mount vs. Download

Mounting or downloading files of any format are supported for datasets created from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL.

When you mount a dataset, you attach the files referenced by the dataset to a directory (mount point) and make it available on the compute target. Mounting is supported for Linux-based computes, including Azure Machine Learning Compute, virtual machines, and HDInsight. When you download a dataset, all the files referenced by the dataset will be downloaded to the compute target. Downloading is supported for all compute types.

If your script processes all files referenced by the dataset, and your compute disk can fit your full dataset, downloading is recommended to avoid the overhead of streaming data from storage services. If your data size exceeds the compute disk size, downloading is not possible. For this scenario, we recommend mounting since only the data files used by your script are loaded at the time of processing.

The following code mounts `dataset` to the temp directory at `mounted_path`

```
import tempfile
mounted_path = tempfile.mkdtemp()

# mount dataset onto the mounted_path of a Linux-based compute
mount_context = dataset.mount(mounted_path)

mount_context.start()

import os
print(os.listdir(mounted_path))
print (mounted_path)
```

Create a FileDataset

The following example creates an unregistered FileDataset from web urls. Learn more about how to create datasets from other sources.

```
from azureml.core.dataset import Dataset

web_paths = [
            'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
            ]
mnist_ds = Dataset.File.from_files(path = web_paths)
```

Configure the estimator

Besides passing the dataset through the `inputs` parameter in the estimator, you can also pass the dataset through `script_params` and get the data path (mounting point) in your training script via arguments. This way, you can keep your training script independent of azureml-sdk. In other words, you will be able use the same training script for local debugging and remote training on any cloud platform.

An SKLearn estimator object is used to submit the run for scikit-learn experiments. Learn more about training with the SKlearn estimator.

```
from azureml.train.sklearn import SKLearn

script_params = {
    # mount the dataset on the remote compute and pass the mounted path as an argument to the training script
    '--data-folder': mnist_ds.as_named_input('mnist').as_mount(),
    '--regularization': 0.5
}

est = SKLearn(source_directory=script_folder,
              script_params=script_params,
              compute_target=compute_target,
              environment_definition=env,
              entry_script='train_mnist.py')

# Run the experiment
run = experiment.submit(est)
run.wait_for_completion(show_output=True)
```

Retrieve the data in your training script

After you submit the run, data files referred by the `mnist` dataset will be mounted to the compute target. The following code shows how to retrieve the data in your script.

```
%%writefile $script_folder/train_mnist.py

import argparse
import os
import numpy as np
import glob

from utils import load_data

# retrieve the 2 arguments configured through script_params in estimator
parser = argparse.ArgumentParser()
parser.add_argument('--data-folder', type=str, dest='data_folder', help='data folder mounting point')
parser.add_argument('--regularization', type=float, dest='reg', default=0.01, help='regularization rate')
args = parser.parse_args()

data_folder = args.data_folder
print('Data folder:', data_folder)

# get the file paths on the compute
X_train_path = glob.glob(os.path.join(data_folder, '**/train-images-idx3-ubyte.gz'), recursive=True)[0]
X_test_path = glob.glob(os.path.join(data_folder, '**/t10k-images-idx3-ubyte.gz'), recursive=True)[0]
y_train_path = glob.glob(os.path.join(data_folder, '**/train-labels-idx1-ubyte.gz'), recursive=True)[0]
y_test = glob.glob(os.path.join(data_folder, '**/t10k-labels-idx1-ubyte.gz'), recursive=True)[0]

# load train and test set into numpy arrays
X_train = load_data(X_train_path, False) / 255.0
X_test = load_data(X_test_path, False) / 255.0
y_train = load_data(y_train_path, True).reshape(-1)
y_test = load_data(y_test, True).reshape(-1)
```

Notebook examples

The dataset notebooks demonstrate and expand upon concepts in this article.

Next steps

- Auto train machine learning models with TabularDatasets
- Train image classification models with FileDatasets
- Train with datasets using pipelines

Detect data drift (preview) on datasets

4/19/2020 • 13 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to create Azure Machine Learning dataset monitors (preview), monitor for data drift and statistical changes in datasets, and set up alerts.

With Azure Machine Learning dataset monitors, you can:

- **Analyze drift in your data** to understand how it changes over time.
- **Monitor model data** for differences between training and serving datasets.
- **Monitor new data** for differences between any baseline and target dataset.
- **Profile features in data** to track how statistical properties change over time.
- **Set up alerts on data drift** for early warnings to potential issues.

Metrics and insights are available through the Azure Application Insights resource associated with the Azure Machine Learning workspace.

> **IMPORTANT**
> Please note that monitoring data drift with the SDK is available in all editions, while monitoring data drift through the studio on the web is Enterprise edition only.

Prerequisites

To create and work with dataset monitors, you need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- An Azure Machine Learning workspace.
- The Azure Machine Learning SDK for Python installed, which includes the azureml-datasets package.
- Structured (tabular) data with a timestamp specified in the file path, file name, or column in the data.

What is data drift?

In the context of machine learning, data drift is the change in model input data that leads to model performance degradation. It is one of the top reasons model accuracy degrades over time, thus monitoring data drift helps detect model performance issues.

Causes of data drift include:

- Upstream process changes, such as a sensor being replaced that changes the units of measurement from inches to centimeters.
- Data quality issues, such as a broken sensor always reading 0.
- Natural drift in the data, such as mean temperature changing with the seasons.
- Change in relation between features, or covariate shift.

With Azure Machine Learning dataset monitors you can set up alerts that assist in data drift detection in datasets over time.

Dataset monitors

You can create a dataset monitor to detect and alert to data drift on new data in a dataset, analyze historical data for drift, and profile new data over time. The data drift algorithm provides an overall measure of change in data and indication of which features are responsible for further investigation. Dataset monitors produce a number of other metrics by profiling new data in the `timeseries` dataset. Custom alerting can be set up on all metrics generated by the monitor through Azure Application Insights. Dataset monitors can be used to quickly catch data issues and reduce the time to debug the issue by identifying likely causes.

Conceptually, there are three primary scenarios for setting up dataset monitors in Azure Machine Learning.

SCENARIO	DESCRIPTION
Monitoring a model's serving data for drift from the model's training data	Results from this scenario can be interpreted as monitoring a proxy for the model's accuracy, given that model accuracy degrades if the serving data drifts from the training data.
Monitoring a time series dataset for drift from a previous time period.	This scenario is more general, and can be used to monitor datasets involved upstream or downstream of model building. The target dataset must have a timestamp column, while the baseline dataset can be any tabular dataset that has features in common with the target dataset.
Performing analysis on past data.	This scenario can be used to understand historical data and inform decisions in settings for dataset monitors.

How dataset can monitor data

Using Azure Machine Learning, data drift is monitored through datasets. To monitor for data drift, a baseline dataset - usually the training dataset for a model - is specified. A target dataset - usually model input data - is compared over time to your baseline dataset. This comparison means that your target dataset must have a timestamp column specified.

Set the `timeseries` trait in the target dataset

The target dataset needs to have the `timeseries` trait set on it by specifying the timestamp column either from a column in the data or a virtual column derived from the path pattern of the files. This can be done through the Python SDK or Azure Machine Learning studio. A column representing a "fine grain" timestamp must be specified to add `timeseries` trait to the dataset. If your data is partitioned into folder structure with time info, such as '{yyyy/MM/dd}', you can create a virtual column through the path pattern setting and set it as the "coarse grain" timestamp to improve the importance of time series functionality.

Python SDK

The `Dataset` class' `with_timestamp_columns()` method defines the time stamp column for the dataset.

```
from azureml.core import Workspace, Dataset, Datastore

# get workspace object
ws = Workspace.from_config()

# get datastore object
dstore = Datastore.get(ws, 'your datastore name')

# specify datastore paths
dstore_paths = [(dstore, 'weather/*/*/*/*/data.parquet')]

# specify partition format
partition_format = 'weather/{state}/{date:yyyy/MM/dd}/data.parquet'

# create the Tabular dataset with 'state' and 'date' as virtual columns
dset = Dataset.Tabular.from_parquet_files(path=dstore_paths, partition_format=partition_format)

# assign the timestamp attribute to a real or virtual column in the dataset
dset = dset.with_timestamp_columns('date')

# register the dataset as the target dataset
dset = dset.register(ws, 'target')
```

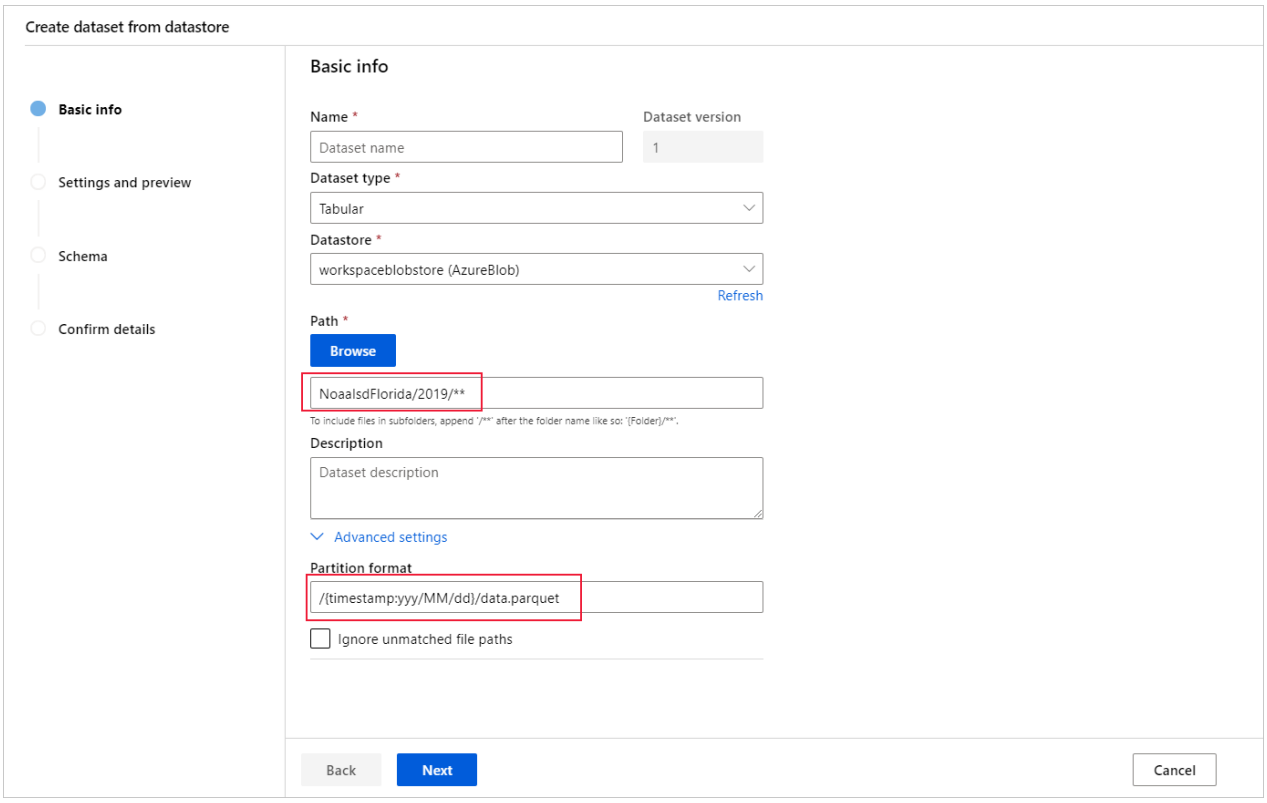
For a full example of using the `timeseries` trait of datasets, see the example notebook or the datasets SDK documentation.

Azure Machine Learning studio

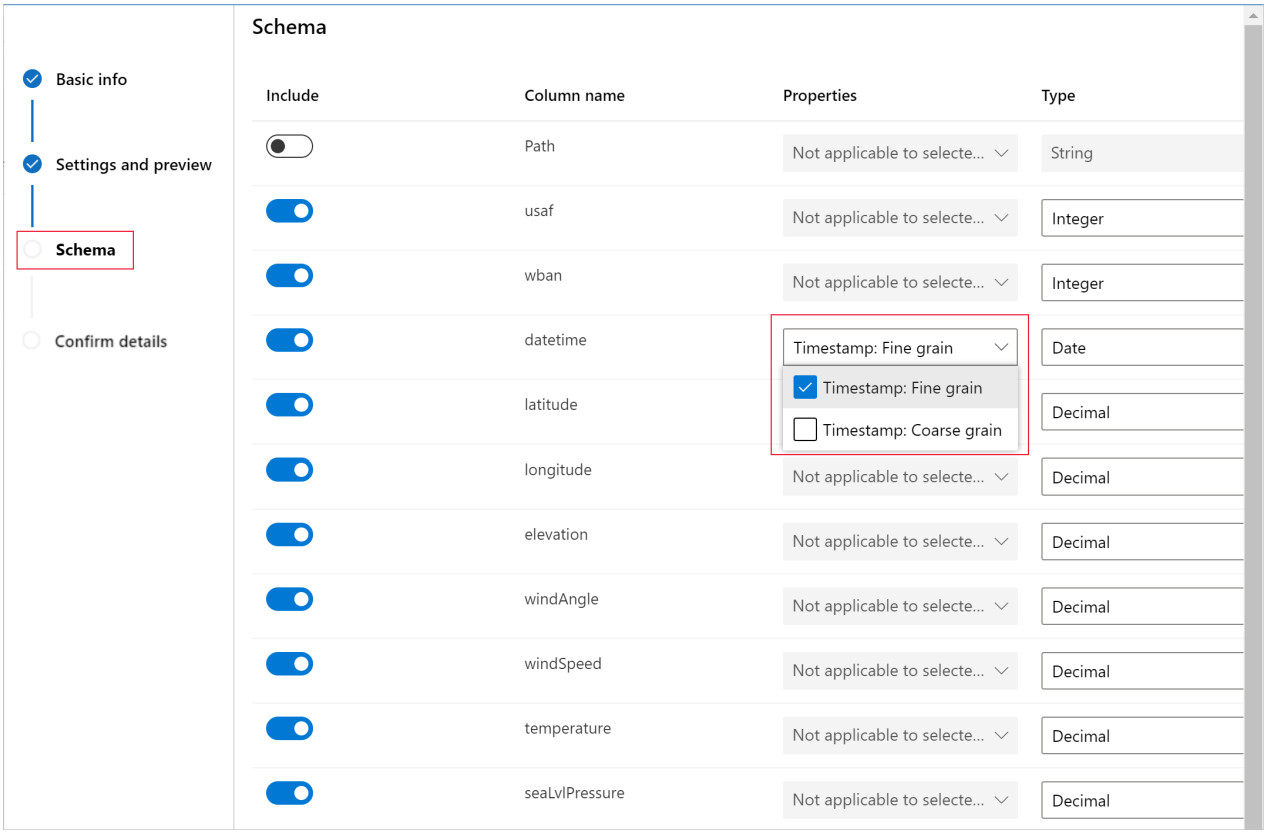
> **IMPORTANT**
>
> The functionality in this studio, https://ml.azure.com, is **accessible from Enterprise workspaces only**. Learn more about editions and upgrading.

If you create your dataset using Azure Machine Learning studio, ensure the path to your data contains timestamp information, include all subfolders with data, and set the partition format.

In the following example, all data under the subfolder *NoaaIsdFlorida/2019* is taken, and the partition format specifies the timestamp's year, month, and day.



In the **Schema** settings, specify the timestamp column from a virtual or real column in the specified dataset:



Dataset monitor settings

Once you create your dataset with the specified timestamp settings, you're ready to configure your dataset monitor.

The various dataset monitor settings are broken into three groups: **Basic info, Monitor settings** and **Backfill settings**.

Basic info

This table contains basic settings used for the dataset monitor.

SETTING	DESCRIPTION	TIPS	MUTABLE
Name	Name of the dataset monitor.		No
Baseline dataset	Tabular dataset that will be used as the baseline for comparison of the target dataset over time.	The baseline dataset must have features in common with the target dataset. Generally, the baseline should be set to a model's training dataset or a slice of the target dataset.	No
Target dataset	Tabular dataset with timestamp column specified which will be analyzed for data drift.	The target dataset must have features in common with the baseline dataset, and should be a `timeseries` dataset, which new data is appended to. Historical data in the target dataset can be analyzed, or new data can be monitored.	No
Frequency	The frequency that will be used to schedule the pipeline job and analyze historical data if running a backfill. Options include daily, weekly, or monthly.	Adjust this setting to include a comparable size of data to the baseline.	No
Features	List of features that will be analyzed for data drift over time.	Set to a model's output feature(s) to measure concept drift. Do not include features that naturally drift over time (month, year, index, etc.). You can backfill and existing data drift monitor after adjusting the list of features.	Yes
Compute target	Azure Machine Learning compute target to run the dataset monitor jobs.		Yes

Monitor settings

These settings are for the scheduled dataset monitor pipeline, which will be created.

SETTING	DESCRIPTION	TIPS	MUTABLE
Enable	Enable or disable the schedule on the dataset monitor pipeline	Disable the schedule to analyze historical data with the backfill setting. It can be enabled after the dataset monitor is created.	Yes

SETTING	DESCRIPTION	TIPS	MUTABLE
Latency	Time, in hours, it takes for data to arrive in the dataset. For instance, if it takes three days for data to arrive in the SQL DB the dataset encapsulates, set the latency to 72.	Cannot be changed after the dataset monitor is created	No
Email addresses	Email addresses for alerting based on breach of the data drift percentage threshold.	Emails are sent through Azure Monitor.	Yes
Threshold	Data drift percentage threshold for email alerting.	Further alerts and events can be set on many other metrics in the workspace's associated Application Insights resource.	Yes

Backfill settings

These settings are for running a backfill on past data for data drift metrics.

SETTING	DESCRIPTION	TIPS
Start date	Start date of the backfill job.	
End date	End date of the backfill job.	The end date cannot be more than 31*frequency units of time from the start date. On an existing dataset monitor, metrics can be backfilled to analyze historical data or replace metrics with updated settings.

Create dataset monitors

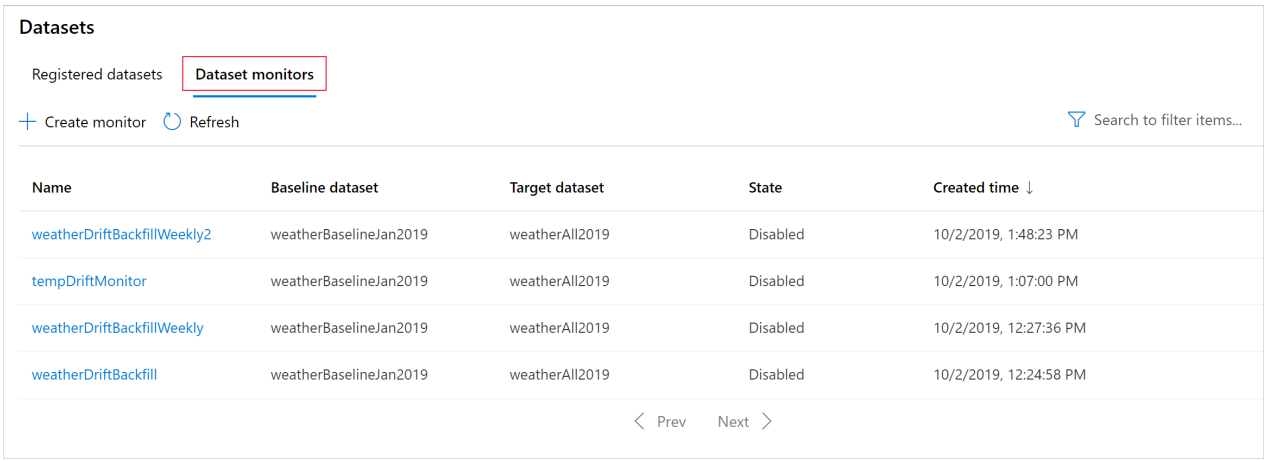
Create dataset monitors to detect and alert to data drift on a new dataset with Azure Machine Learning studio or the Python SDK.

Azure Machine Learning studio

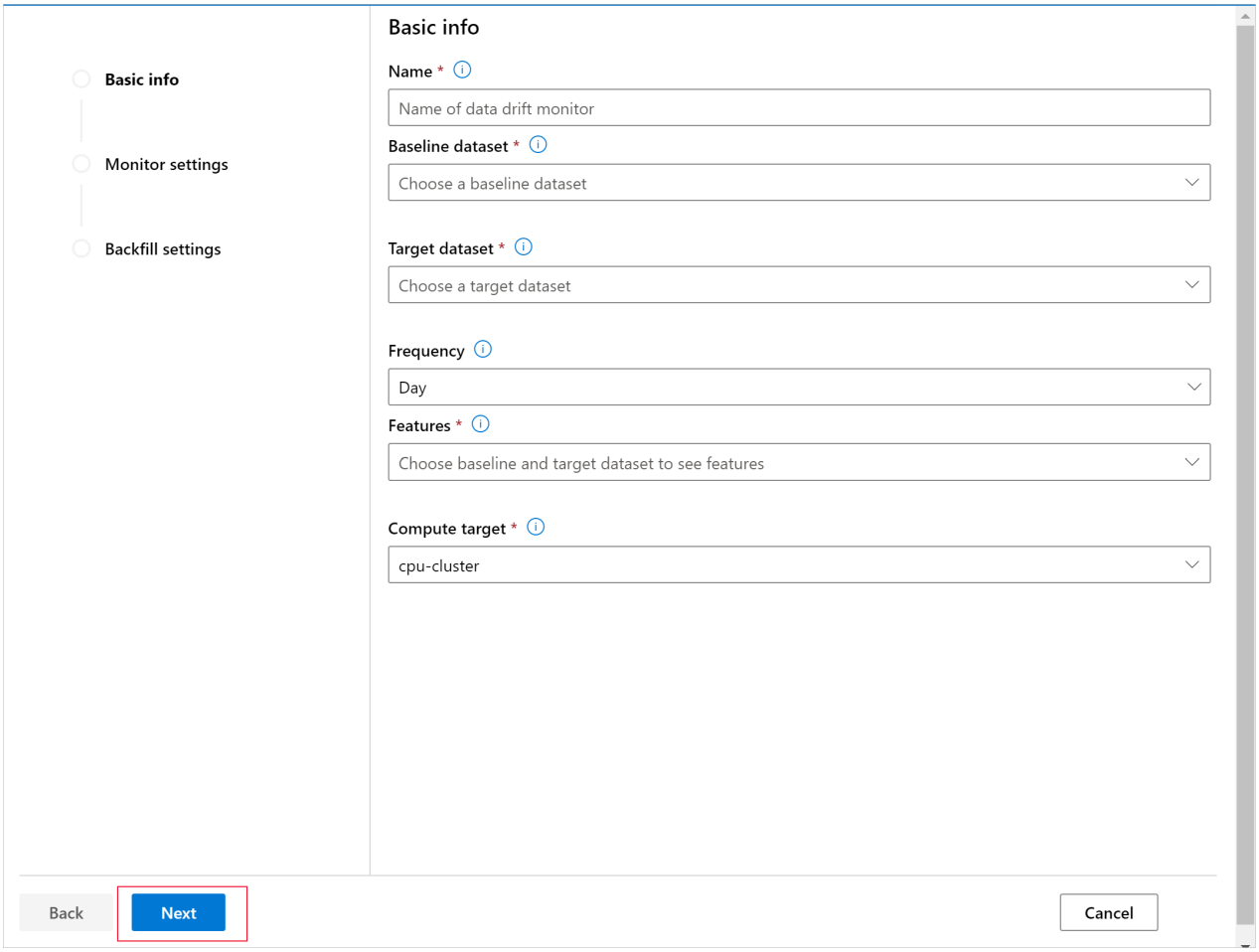
> **IMPORTANT**
> The functionality in this studio, https://ml.azure.com, is **accessible from Enterprise workspaces only**. Learn more about editions and upgrading.

To set up alerts on your dataset monitor, the workspace that contains the dataset you want to create a monitor for must have Enterprise edition capabilities.

After the workspace functionality is confirmed, navigate to the studio's homepage and select the Datasets tab on the left. Select Dataset monitors.



Click on the **+Create monitor** button and continue through the wizard by clicking **Next**.



The resulting dataset monitor will appear in the list. Select it to go to that monitor's details page.

From Python SDK

See the Python SDK reference documentation on data drift for full details.

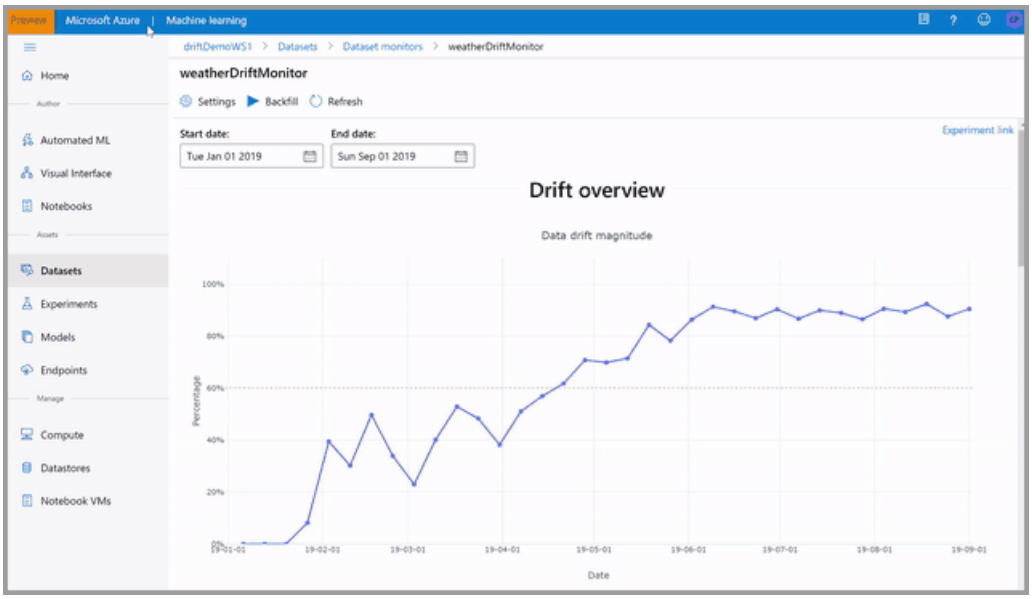
The following example shows how to create a dataset monitor using the Python SDK

```
from azureml.core import Workspace, Dataset
from azureml.datadrift import DataDriftDetector
from datetime import datetime

# get the workspace object
ws = Workspace.from_config()

# get the target dataset
dset = Dataset.get_by_name(ws, 'target')

# set the baseline dataset
baseline = target.time_before(datetime(2019, 2, 1))

# set up feature list
features = ['latitude', 'longitude', 'elevation', 'windAngle', 'windSpeed', 'temperature', 'snowDepth',
'stationName', 'countryOrRegion']

# set up data drift detector
monitor = DataDriftDetector.create_from_datasets(ws, 'drift-monitor', baseline, target,
                                                 compute_target='cpu-cluster',
                                                 frequency='Week',
                                                 feature_list=None,
                                                 drift_threshold=.6,
                                                 latency=24)

# get data drift detector by name
monitor = DataDriftDetector.get_by_name(ws, 'drift-monitor')

# update data drift detector
monitor = monitor.update(feature_list=features)

# run a backfill for January through May
backfill1 = monitor.backfill(datetime(2019, 1, 1), datetime(2019, 5, 1))

# run a backfill for May through today
backfill1 = monitor.backfill(datetime(2019, 5, 1), datetime.today())

# disable the pipeline schedule for the data drift detector
monitor = monitor.disable_schedule()

# enable the pipeline schedule for the data drift detector
monitor = monitor.enable_schedule()
```

For a full example of setting up a `timeseries` dataset and data drift detector, see our example notebook.

Understanding data drift results

The data monitor produces two groups of results: Drift overview and Feature details. The following animation shows the available drift monitor charts based on the selected feature and metric.



Drift overview

The **Drift overview** section contains top-level insights into the magnitude of data drift and which features should be further investigated.

METRIC	DESCRIPTION	TIPS
Data drift magnitude	Given as a percentage between the baseline and target dataset over time. Ranging from 0 to 100 where 0 indicates identical datasets and 100 indicates the Azure Machine Learning data drift capability can completely tell the two datasets apart.	Noise in the precise percentage measured is expected due to machine learning techniques being used to generate this magnitude.
Drift contribution by feature	The contribution of each feature in the target dataset to the measured drift magnitude.	Due to covariate shift, the underlying distribution of a feature does not necessarily need to change to have relatively high feature importance.

The following image is an example of charts seen in the **Drift overview** results in Azure Machine Learning studio, resulting from a backfill of NOAA Integrated Surface Data. Data was sampled to `stationName contains 'FLORIDA'`, with January 2019 being used as the baseline dataset and all 2019 data used as the target.

Feature details

The **Feature details** section contains feature-level insights into the change in the selected feature's distribution, as well as other statistics, over time.

The target dataset is also profiled over time. The statistical distance between the baseline distribution of each feature is compared with the target dataset's over time, which is conceptually similar to the data drift magnitude with the exception that this statistical distance is for an individual feature. Min, max, and mean are also available.

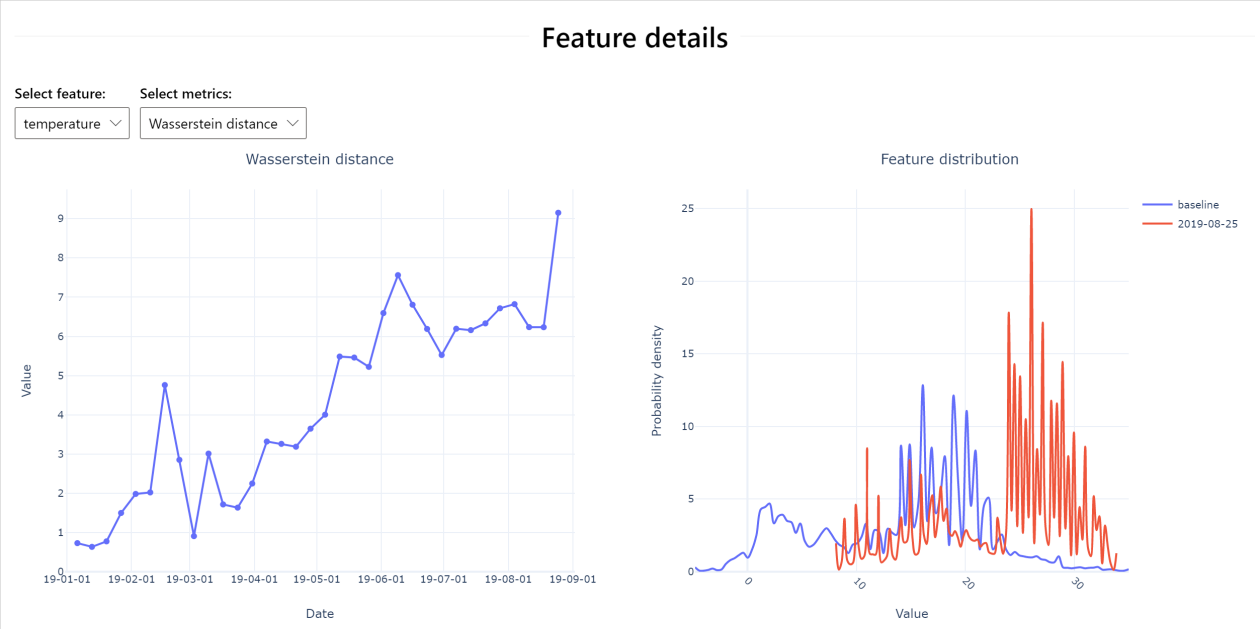
In the Azure Machine Learning studio, if you click on a data point in the graph the distribution of the feature being shown will adjust accordingly. By default, it shows the baseline dataset's distribution and the most recent run's distribution of the same feature.

These metrics can also be retrieved in the Python SDK through the `get_metrics()` method on a `DataDriftDetector` object.

Numeric features

Numeric features are profiled in each dataset monitor run. The following are exposed in the Azure Machine Learning studio. Probability density is shown for the distribution.

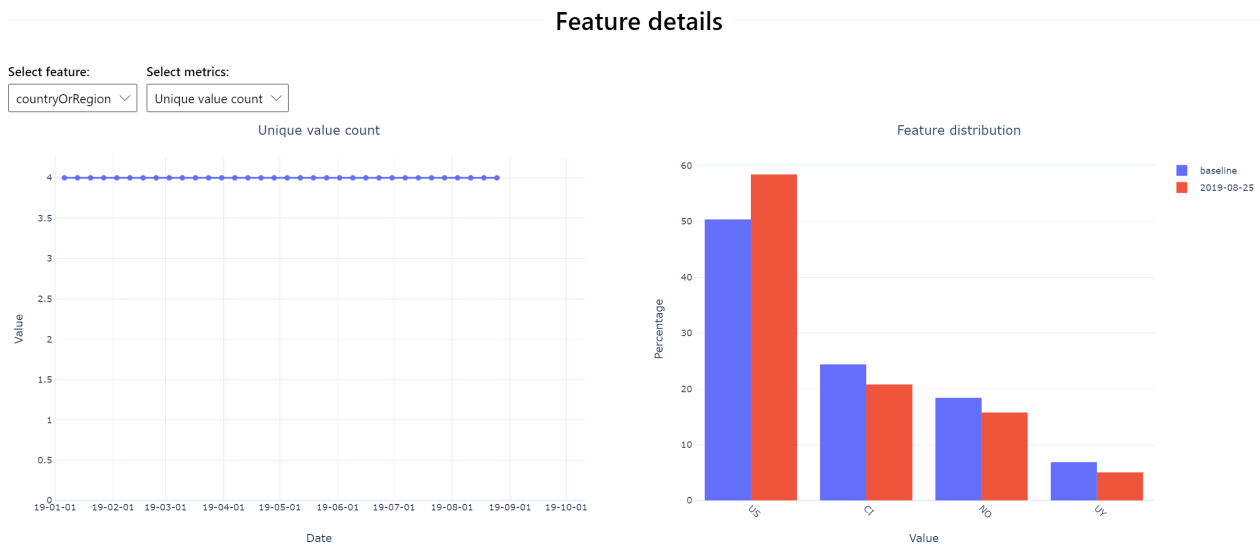
METRIC	DESCRIPTION
Wasserstein distance	Minimum amount of work to transform baseline distribution into the target distribution.
Mean value	Average value of the feature.
Min value	Minimum value of the feature.
Max value	Maximum value of the feature.



Categorical features

Numeric features are profiled in each dataset monitor run. The following are exposed in the Azure Machine Learning studio. A histogram is shown for the distribution.

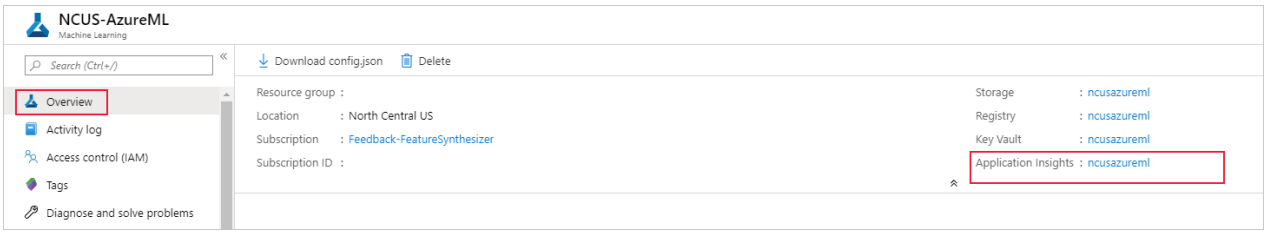
METRIC	DESCRIPTION
Euclidian distance	Geometric distance between baseline and target distributions.
Unique values	Number of unique values (cardinality) of the feature.



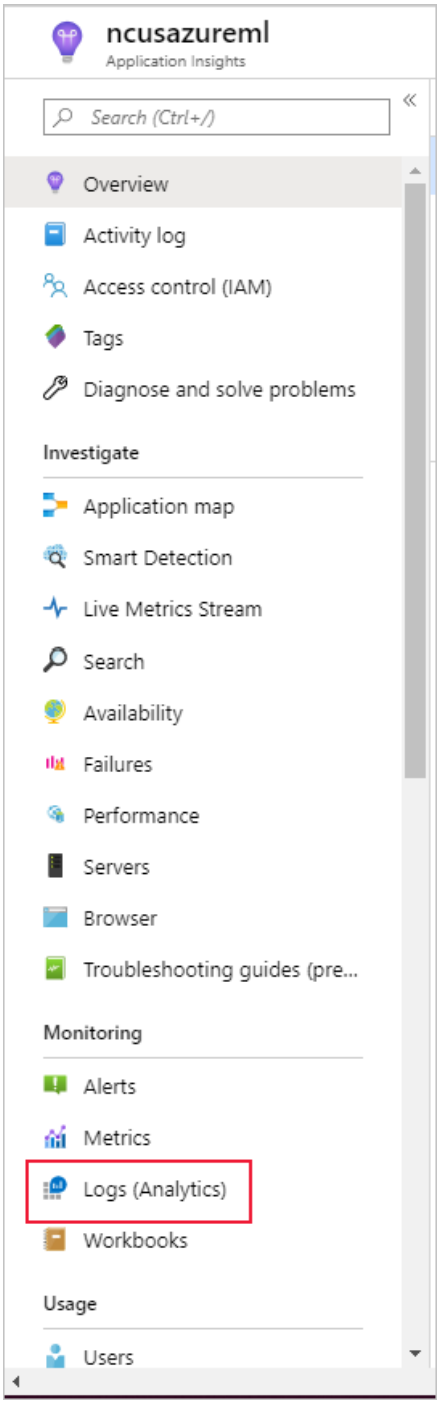
Metrics, alerts, and events

Metrics can be queried in the Azure Application Insights resource associated with your machine learning workspace. Which gives access to all features of Application Insights including set up for custom alert rules and action groups to trigger an action such as, an Email/SMS/Push/Voice or Azure Function. Please refer to the complete Application Insights documentation for details.

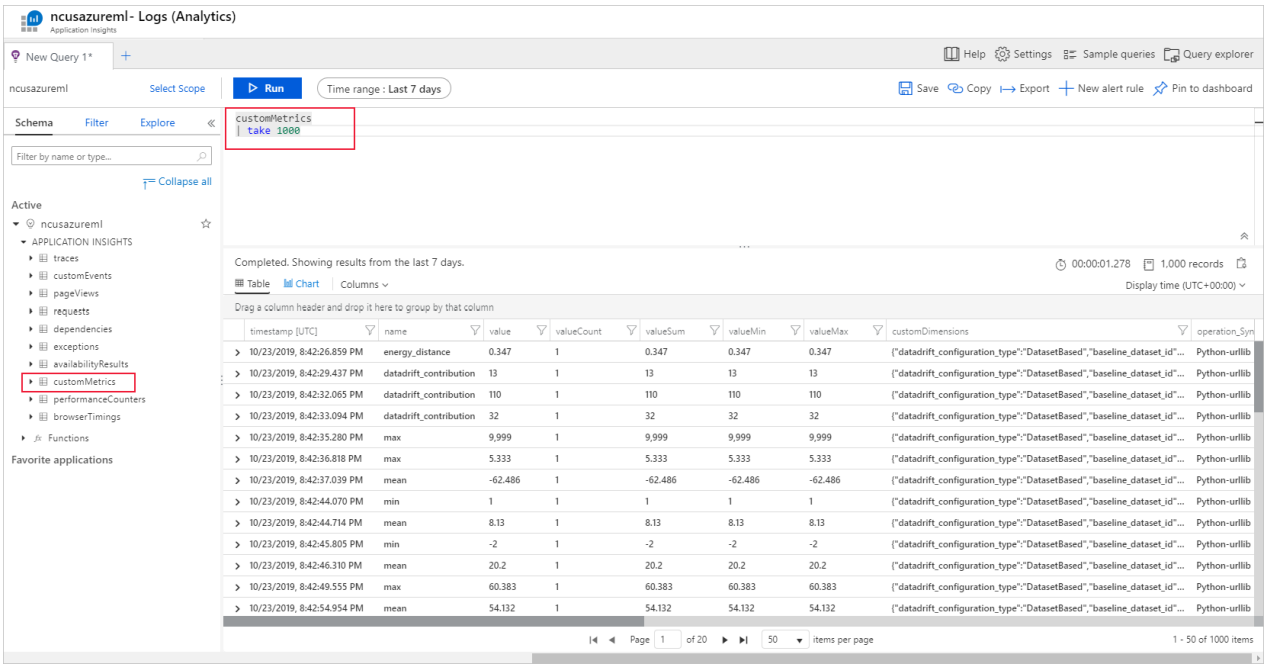
To get started, navigate to the Azure portal and select your workspace's **Overview** page. The associated Application Insights resource is on the far right:



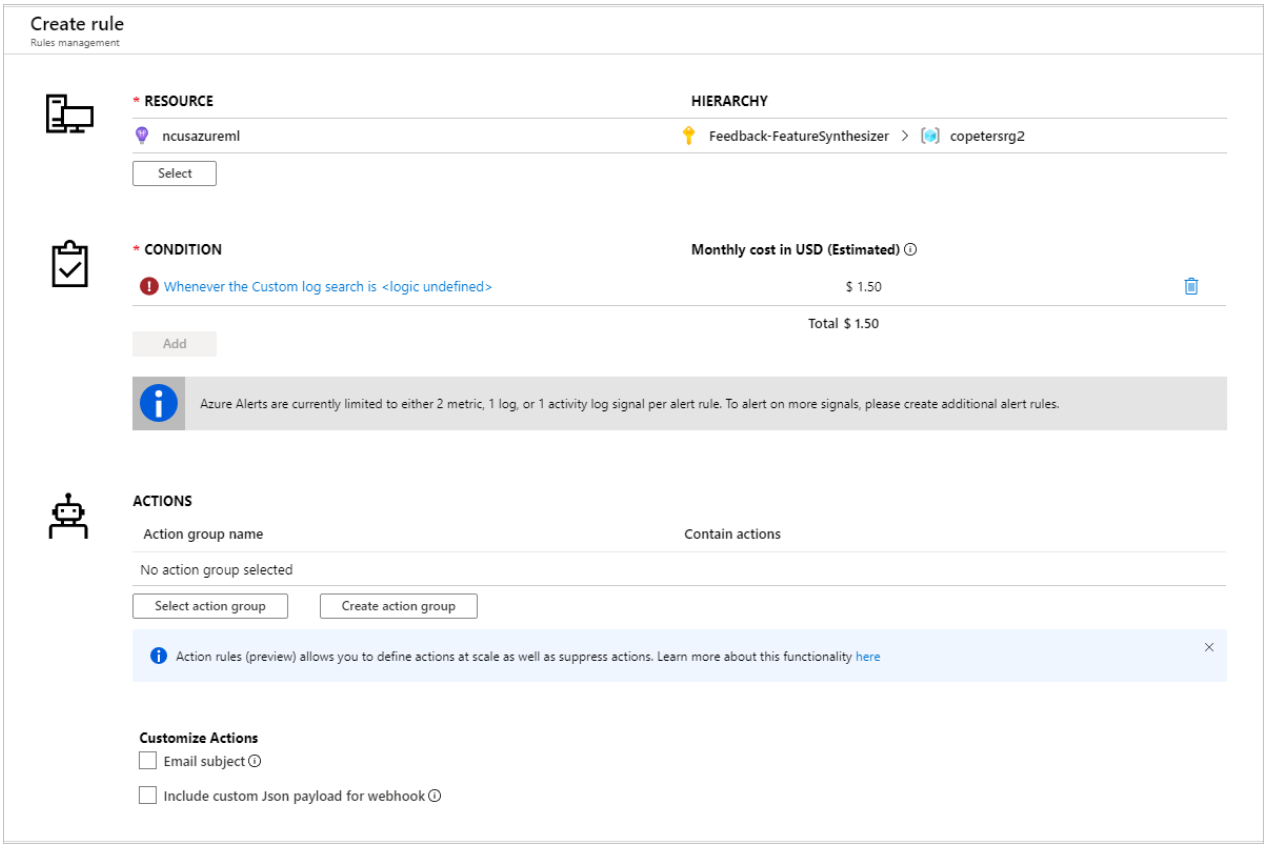
Select Logs (Analytics) under Monitoring on the left pane:



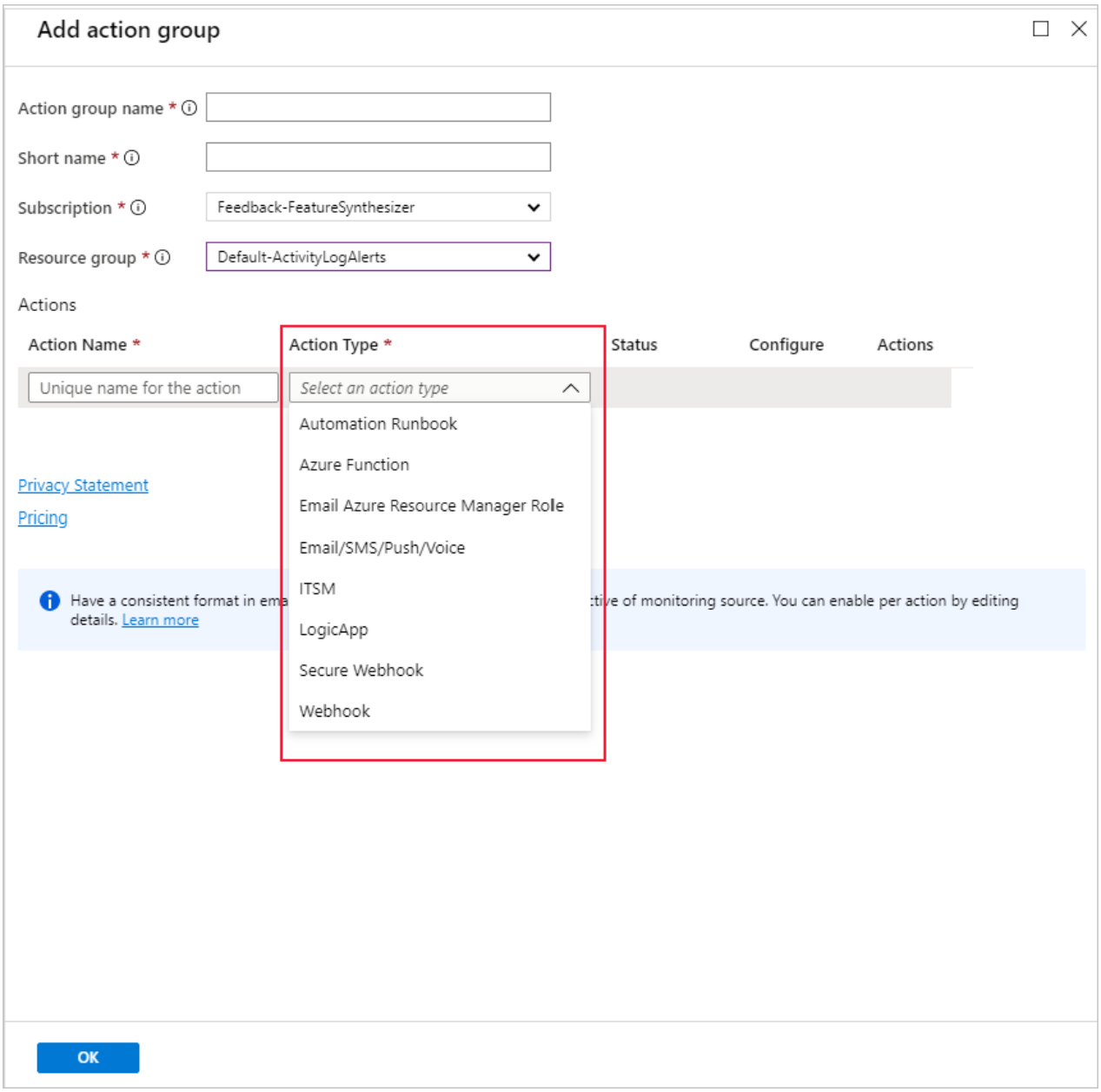
The dataset monitor metrics are stored as `customMetrics`. You can write and run a query after setting up a dataset monitor to view them:



After identifying metrics to set up alert rules, create a new alert rule:



You can use an existing action group, or create a new one to define the action to be taken when the set conditions are met:



Troubleshooting

Limitations and known issues:

- Time range of backfill jobs are limited to 31 intervals of the monitor's frequency setting.
- Limitation of 200 features, unless a feature list is not specified (all features used).
- Compute size must be large enough to handle the data.
- Ensure your dataset has data within the start and end date for a given monitor run.
- Dataset monitors will only work on datasets that contain 50 rows or more.

Columns, or features, in the dataset are classified as categorical or numeric based on the conditions in the following table. If the feature does not meet these conditions - for instance, a column of type string with >100 unique values - the feature is dropped from our data drift algorithm, but is still profiled.

FEATURE TYPE	DATA TYPE	CONDITION	LIMITATIONS
Categorical	string, bool, int, float	The number of unique values in the feature is less than 100 and less than 5% of the number of rows.	Null is treated as its own category.

FEATURE TYPE	DATA TYPE	CONDITION	LIMITATIONS
Numerical	int, float	The values in the feature are of a numerical data type and do not meet the condition for a categorical feature.	Feature dropped if >15% of values are null.

Next steps

- Head to the Azure Machine Learning studio or the Python notebook to set up a dataset monitor.
- See how to set up data drift on models deployed to Azure Kubernetes Service.
- Set up dataset drift monitors with event grid.

Version and track datasets in experiments

3/31/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you'll learn how to version and track Azure Machine Learning datasets for reproducibility. Dataset versioning is a way to bookmark the state of your data so that you can apply a specific version of the dataset for future experiments.

Typical versioning scenarios:

- When new data is available for retraining
- When you're applying different data preparation or feature engineering approaches

Prerequisites

For this tutorial, you need:

- Azure Machine Learning SDK for Python installed. This SDK includes the azureml-datasets package.
- An Azure Machine Learning workspace. Retrieve an existing one by running the following code, or create a new workspace.

```
import azureml.core
from azureml.core import Workspace

ws = Workspace.from_config()
```

- An Azure Machine Learning dataset.

Register and retrieve dataset versions

By registering a dataset, you can version, reuse, and share it across experiments and with colleagues. You can register multiple datasets under the same name and retrieve a specific version by name and version number.

Register a dataset version

The following code registers a new version of the `titanic_ds` dataset by setting the `create_new_version` parameter to `True`. If there's no existing `titanic_ds` dataset registered with the workspace, the code creates a new dataset with the name `titanic_ds` and sets its version to 1.

```
titanic_ds = titanic_ds.register(workspace = workspace,
                                 name = 'titanic_ds',
                                 description = 'titanic training data',
                                 create_new_version = True)
```

Retrieve a dataset by name

By default, the get_by_name() method on the `Dataset` class returns the latest version of the dataset registered with the workspace.

The following code gets version 1 of the `titanic_ds` dataset.

```
from azureml.core import Dataset
# Get a dataset by name and version number
titanic_ds = Dataset.get_by_name(workspace = workspace,
                                 name = 'titanic_ds',
                                 version = 1)
```

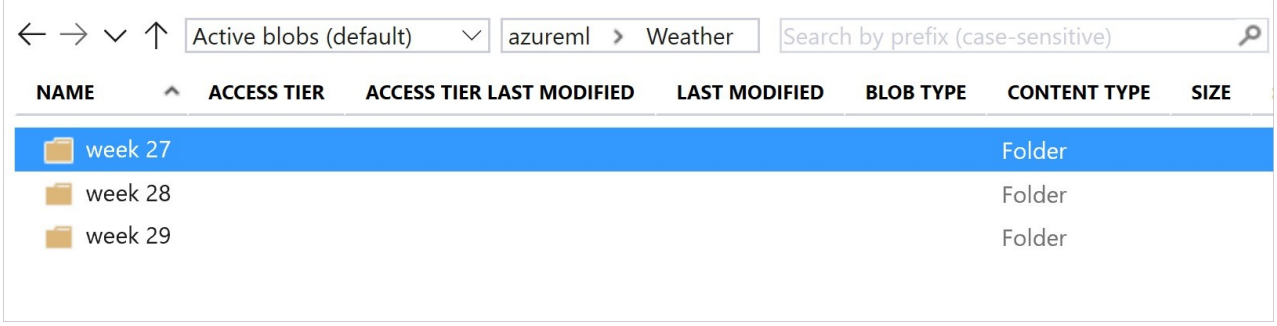
Versioning best practice

When you create a dataset version, you're *not* creating an extra copy of data with the workspace. Because datasets are references to the data in your storage service, you have a single source of truth, managed by your storage service.

> **IMPORTANT**
> If the data referenced by your dataset is overwritten or deleted, calling a specific version of the dataset does *not* revert the change.

When you load data from a dataset, the current data content referenced by the dataset is always loaded. If you want to make sure that each dataset version is reproducible, we recommend that you not modify data content referenced by the dataset version. When new data comes in, save new data files into a separate data folder and then create a new dataset version to include data from that new folder.

The following image and sample code show the recommended way to structure your data folders and to create dataset versions that reference those folders:




```
from azureml.core import Dataset

# get the default datastore of the workspace
datastore = workspace.get_default_datastore()

# create & register weather_ds version 1 pointing to all files in the folder of week 27
datastore_path1 = [(datastore, 'Weather/week 27')]
dataset1 = Dataset.File.from_files(path=datastore_path1)
dataset1.register(workspace = workspace,
                  name = 'weather_ds',
                  description = 'weather data in week 27',
                  create_new_version = True)

# create & register weather_ds version 2 pointing to all files in the folder of week 27 and 28
datastore_path2 = [(datastore, 'Weather/week 27'), (datastore, 'Weather/week 28')]
dataset2 = Dataset.File.from_files(path = datastore_path2)
dataset2.register(workspace = workspace,
                  name = 'weather_ds',
                  description = 'weather data in week 27, 28',
                  create_new_version = True)
```

Version a pipeline output dataset

You can use a dataset as the input and output of each Machine Learning pipeline step. When you rerun pipelines, the output of each pipeline step is registered as a new dataset version.

Because Machine Learning pipelines populate the output of each step into a new folder every time the pipeline reruns, the versioned output datasets are reproducible. Learn more about datasets in pipelines.

```
from azureml.core import Dataset
from azureml.pipeline.steps import PythonScriptStep
from azureml.pipeline.core import Pipeline, PipelineData
from azureml.core. runconfig import CondaDependencies, RunConfiguration

# get input dataset
input_ds = Dataset.get_by_name(workspace, 'weather_ds')

# register pipeline output as dataset
output_ds = PipelineData('prepared_weather_ds', datastore=datastore).as_dataset()
output_ds = output_ds.register(name='prepared_weather_ds', create_new_version=True)

conda = CondaDependencies.create(
    pip_packages=['azureml-defaults', 'azureml-dataprep[fuse,pandas]'],
    pin_sdk_version=False)

run_config = RunConfiguration()
run_config.environment.docker.enabled = True
run_config.environment.python.conda_dependencies = conda

# configure pipeline step to use dataset as the input and output
prep_step = PythonScriptStep(script_name="prepare.py",
                             inputs=[input_ds.as_named_input('weather_ds')],
                             outputs=[output_ds],
                             runconfig=run_config,
                             compute_target=compute_target,
                             source_directory=project_folder)
```

Track datasets in experiments

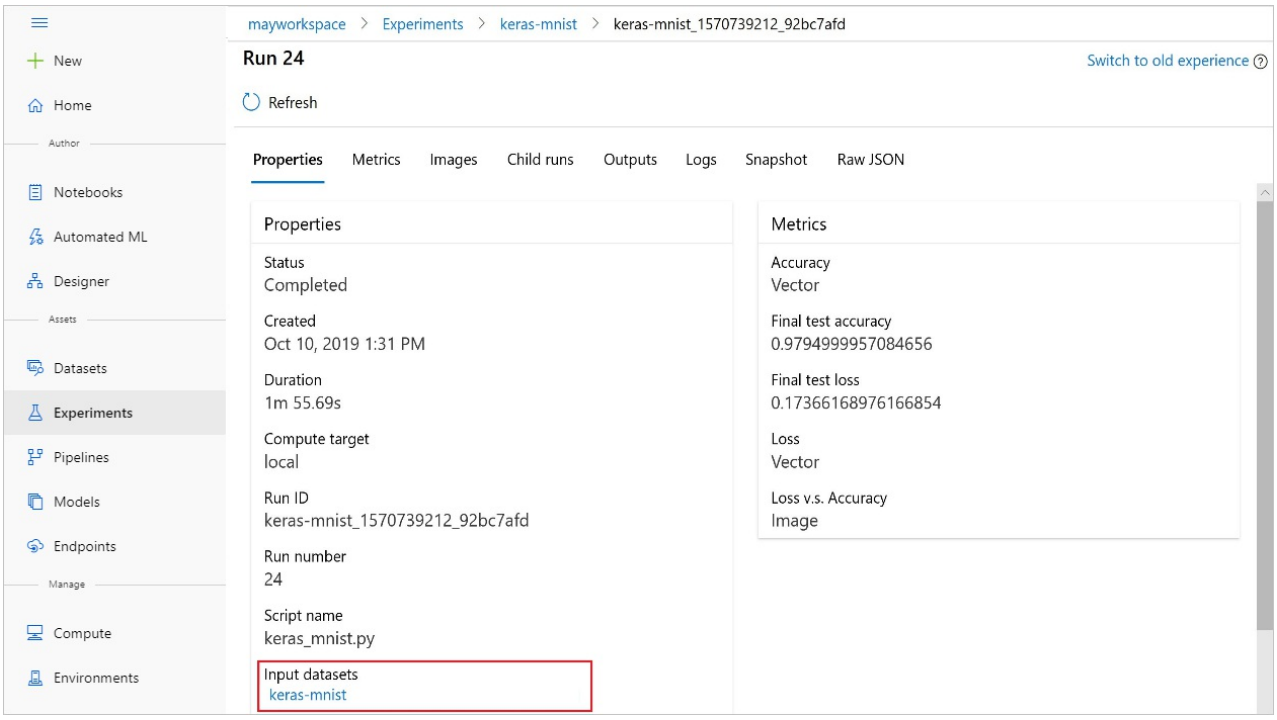
For each Machine Learning experiment, you can easily trace the datasets used as the input through the experiment `Run` object.

The following code uses the `get_details()` method to track which input datasets were used with the experiment run:

```
# get input datasets
inputs = run.get_details()['inputDatasets']
input_dataset = inputs[0]['dataset']

# list the files referenced by input_dataset
input_dataset.to_path()
```

You can also find the `input_datasets` from experiments by using https://ml.azure.com/.

The following image shows where to find the input dataset of an experiment on Azure Machine Learning studio. For this example, go to your **Experiments** pane and open the **Properties** tab for a specific run of your experiment, `keras-mnist`.

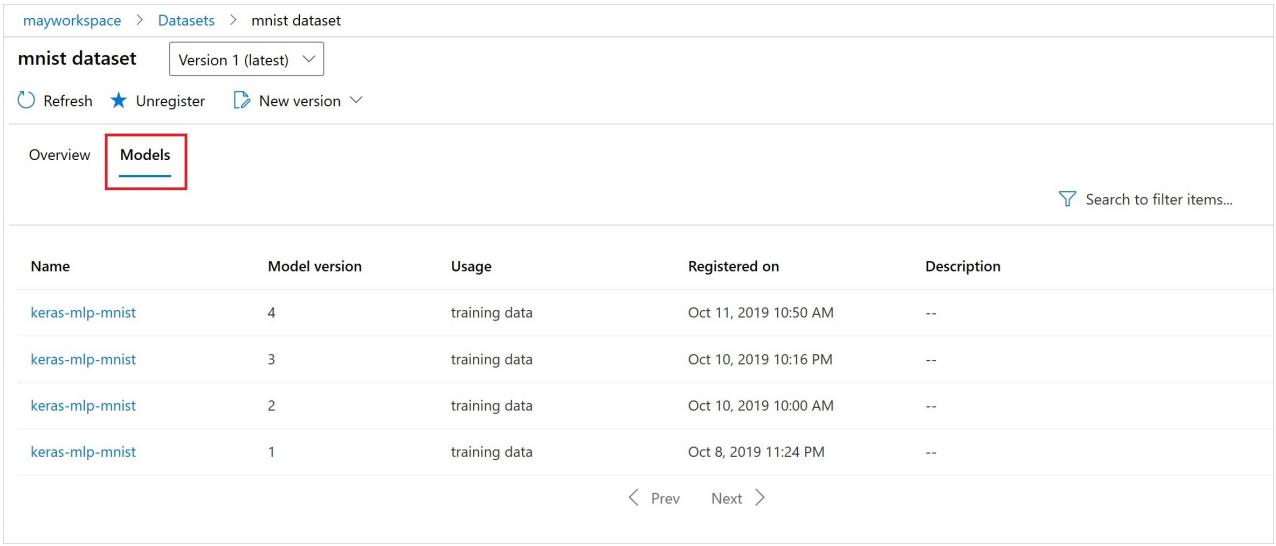


Use the following code to register models with datasets:

```
model = run.register_model(model_name='keras-mlp-mnist',
                           model_path=model_path,
                           datasets =[('training data',train_dataset)])
```

After registration, you can see the list of models registered with the dataset by using Python or go to https://ml.azure.com/.

The following view is from the **Datasets** pane under **Assets**. Select the dataset and then select the **Models** tab for a list of the models that are registered with the dataset.



Next steps

- Train with datasets
- More sample dataset notebooks

Train models with Azure Machine Learning using estimator

4/19/2020 • 5 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

With Azure Machine Learning, you can easily submit your training script to various compute targets, using a RunConfiguration object and a ScriptRunConfig object. That pattern gives you a lot of flexibility and maximum control.

To facilitate deep learning model training, the Azure Machine Learning Python SDK provides an alternative higher-level abstraction, the estimator class, which allows users to easily construct run configurations. You can create and use a generic Estimator to submit training script using any learning framework you choose (such as scikit-learn) on any compute target you choose, whether it's your local machine, a single VM in Azure, or a GPU cluster in Azure. For PyTorch, TensorFlow and Chainer tasks, Azure Machine Learning also provides respective PyTorch, TensorFlow, and Chainer estimators to simplify using these frameworks.

Train with an estimator

Once you've created your workspace and set up your development environment, training a model in Azure Machine Learning involves the following steps:

1. Create a remote compute target (note you can also use local computer as compute target)
2. Upload your training data to datastore (Optional)
3. Create your training script
4. Create an `Estimator` object
5. Submit the estimator to an experiment object under the workspace

This article focuses on steps 4-5. For steps 1-3, refer to the train a model tutorial for an example.

Single-node training

Use an `Estimator` for a single-node training run on remote compute in Azure for a scikit-learn model. You should have already created your compute target object `compute_target` and your FileDataset object `ds`.

```
from azureml.train.estimator import Estimator

script_params = {
    # to mount files referenced by mnist dataset
    '--data-folder': ds.as_named_input('mnist').as_mount(),
    '--regularization': 0.8
}

sk_est = Estimator(source_directory='./my-sklearn-proj',
                   script_params=script_params,
                   compute_target=compute_target,
                   entry_script='train.py',
                   conda_packages=['scikit-learn'])
```

This code snippet specifies the following parameters to the `Estimator` constructor.

PARAMETER	DESCRIPTION
`source_directory`	Local directory that contains all of your code needed for the training job. This folder gets copied from your local machine to the remote compute.
`script_params`	Dictionary specifying the command-line arguments to pass to your training script `entry_script`, in the form of `<command-line argument, value>` pairs. To specify a verbose flag in `script_params`, use `<command-line argument, "">`.
`compute_target`	Remote compute target that your training script will run on, in this case an Azure Machine Learning Compute (AmlCompute) cluster. (Note even though AmlCompute cluster is the commonly used target, it is also possible to choose other compute target types such as Azure VMs or even local computer.)
`entry_script`	Filepath (relative to the `source_directory`) of the training script to be run on the remote compute. This file, and any additional files it depends on, should be located in this folder.
`conda_packages`	List of Python packages to be installed via conda needed by your training script.

The constructor has another parameter called `pip_packages` that you use for any pip packages needed.

Now that you've created your `Estimator` object, submit the training job to be run on the remote compute with a call to the `submit` function on your Experiment object `experiment`.

```
run = experiment.submit(sk_est)
print(run.get_portal_url())
```

IMPORTANT

Special Folders Two folders, *outputs* and *logs*, receive special treatment by Azure Machine Learning. During training, when you write files to folders named *outputs* and *logs* that are relative to the root directory (`./outputs` and `./logs`, respectively), the files will automatically upload to your run history so that you have access to them once your run is finished.

To create artifacts during training (such as model files, checkpoints, data files, or plotted images) write these to the `./outputs` folder.

Similarly, you can write any logs from your training run to the `./logs` folder. To utilize Azure Machine Learning's TensorBoard integration make sure you write your TensorBoard logs to this folder. While your run is in progress, you will be able to launch TensorBoard and stream these logs. Later, you will also be able to restore the logs from any of your previous runs.

For example, to download a file written to the *outputs* folder to your local machine after your remote training run: `run.download_file(name='outputs/my_output_file', output_file_path='my_destination_path')`

Distributed training and custom Docker images

There are two additional training scenarios you can carry out with the `Estimator`:

- Using a custom Docker image

- Distributed training on a multi-node cluster

The following code shows how to carry out distributed training for a Keras model. In addition, instead of using the default Azure Machine Learning images, it specifies a custom docker image from Docker Hub `continuumio/miniconda` for training.

You should have already created your compute target object `compute_target`. You create the estimator as follows:

```
from azureml.train.estimator import Estimator
from azureml.core.runconfig import MpiConfiguration

estimator = Estimator(source_directory='./my-keras-proj',
                      compute_target=compute_target,
                      entry_script='train.py',
                      node_count=2,
                      process_count_per_node=1,
                      distributed_training=MpiConfiguration(),
                      conda_packages=['tensorflow', 'keras'],
                      custom_docker_image='continuumio/miniconda')
```

The above code exposes the following new parameters to the `Estimator` constructor:

PARAMETER	DESCRIPTION	DEFAULT
`custom_docker_image`	Name of the image you want to use. Only provide images available in public docker repositories (in this case Docker Hub). To use an image from a private docker repository, use the constructor's `environment_definition` parameter instead. See example.	`None`
`node_count`	Number of nodes to use for your training job.	`1`
`process_count_per_node`	Number of processes (or "workers") to run on each node. In this case, you use the `2` GPUs available on each node.	`1`
`distributed_training`	MPIConfiguration object for launching distributed training using MPI backend.	`None`

Finally, submit the training job:

```
run = experiment.submit(estimator)
print(run.get_portal_url())
```

Registering a model

Once you've trained the model, you can save and register it to your workspace. Model registration lets you store and version your models in your workspace to simplify model management and deployment.

Running the following code will register the model to your workspace, and will make it available to reference by name in remote compute contexts or deployment scripts. See `register_model` in the reference docs for more information and additional parameters.

```
model = run.register_model(model_name='sklearn-sample', model_path=None)
```

GitHub tracking and integration

When you start a training run where the source directory is a local Git repository, information about the repository is stored in the run history. For more information, see Git integration for Azure Machine Learning.

Examples

For a notebook that shows the basics of an estimator pattern, see:

- how-to-use-azureml/training-with-deep-learning/how-to-use-estimator

For a notebook that trains a scikit-learn model by using estimator, see:

- tutorials/img-classification-part1-training.ipynb

For notebooks on training models by using deep-learning-framework specific estimators, see:

- how-to-use-azureml/ml-frameworks

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Next steps

- Track run metrics during training
- Train PyTorch models
- Train TensorFlow models
- Tune hyperparameters
- Deploy a trained model
- Create and manage environments for training and deployment

Set up and use compute targets for model training

4/24/2020 • 21 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

With Azure Machine Learning, you can train your model on a variety of resources or environments, collectively referred to as **compute targets**. A compute target can be a local machine or a cloud resource, such as an Azure Machine Learning Compute, Azure HDInsight or a remote virtual machine. You can also create compute targets for model deployment as described in "Where and how to deploy your models".

You can create and manage a compute target using the Azure Machine Learning SDK, Azure Machine Learning studio, Azure CLI or Azure Machine Learning VS Code extension. If you have compute targets that were created through another service (for example, an HDInsight cluster), you can use them by attaching them to your Azure Machine Learning workspace.

In this article, you learn how to use various compute targets for model training. The steps for all compute targets follow the same workflow:

1. **Create** a compute target if you don't already have one.
2. **Attach** the compute target to your workspace.
3. **Configure** the compute target so that it contains the Python environment and package dependencies needed by your script.

> **NOTE**
> Code in this article was tested with Azure Machine Learning SDK version 1.0.74.

Compute targets for training

Azure Machine Learning has varying support across different compute targets. A typical model development lifecycle starts with dev/experimentation on a small amount of data. At this stage, we recommend using a local environment. For example, your local computer or a cloud-based VM. As you scale up your training on larger data sets, or do distributed training, we recommend using Azure Machine Learning Compute to create a single- or multi-node cluster that autoscales each time you submit a run. You can also attach your own compute resource, although support for various scenarios may vary as detailed below:

Compute targets can be reused from one training job to the next. For example, once you attach a remote VM to your workspace, you can reuse it for multiple jobs. For machine learning pipelines, use the appropriate pipeline step for each compute target.

TRAINING TARGETS	AUTOMATED ML	ML PIPELINES	AZURE MACHINE LEARNING DESIGNER
Local computer	yes		
Azure Machine Learning compute cluster	yes & hyperparameter tuning	yes	yes
Remote VM	yes & hyperparameter tuning	yes	
Azure Databricks	yes (SDK local mode only)	yes	

TRAINING TARGETS	AUTOMATED ML	ML PIPELINES	AZURE MACHINE LEARNING DESIGNER
Azure Data Lake Analytics		yes	
Azure HDInsight		yes	
Azure Batch		yes	

> **NOTE**
> Azure Machine Learning Compute can be created as a persistent resource or created dynamically when you request a run. Run-based creation removes the compute target after the training run is complete, so you cannot reuse compute targets created this way.

What's a run configuration?

When training, it is common to start on your local computer, and later run that training script on a different compute target. With Azure Machine Learning, you can run your script on various compute targets without having to change your script.

All you need to do is define the environment for each compute target within a **run configuration**. Then, when you want to run your training experiment on a different compute target, specify the run configuration for that compute. For details of specifying an environment and binding it to run configuration, see Create and manage environments for training and deployment.

Learn more about submitting experiments at the end of this article.

What's an estimator?

To facilitate model training using popular frameworks, the Azure Machine Learning Python SDK provides an alternative higher-level abstraction, the estimator class. This class allows you to easily construct run configurations. You can create and use a generic Estimator to submit training scripts that use any learning framework you choose (such as scikit-learn). We recommend using an estimator for training as it automatically constructs embedded objects like an environment or RunConfiguration objects for you. If you wish to have more control over how these objects are created and specify what packages to install for your experiment run, follow these steps to submit your training experiments using a RunConfiguration object on an Azure Machine Learning Compute.

For PyTorch, TensorFlow, and Chainer tasks, Azure Machine Learning also provides respective PyTorch, TensorFlow, and Chainer estimators to simplify using these frameworks.

For more information, see Train ML Models with estimators.

What's an ML Pipeline?

With ML pipelines, you can optimize your workflow with simplicity, speed, portability, and reuse. When building pipelines with Azure Machine Learning, you can focus on your expertise, machine learning, rather than on infrastructure and automation.

ML pipelines are constructed from multiple **steps**, which are distinct computational units in the pipeline. Each step can run independently and use isolated compute resources. This allows multiple data scientists to work on the same pipeline at the same time without over-taxing compute resources, and also makes it easy to use different compute types/sizes for each step.

> **TIP**
> ML Pipelines can use run configuration or estimators when training models.

While ML pipelines can train models, they can also prepare data before training and deploy models after training. One of the primary use cases for pipelines is batch scoring. For more information, see Pipelines: Optimize machine learning workflows.

Set up in Python

Use the sections below to configure these compute targets:

- Local computer
- Azure Machine Learning Compute
- Remote virtual machines
- Azure HDInsight

Local computer

1. **Create and attach**: There's no need to create or attach a compute target to use your local computer as the training environment.
2. **Configure**: When you use your local computer as a compute target, the training code is run in your development environment. If that environment already has the Python packages you need, use the user-managed environment.

```
from azureml.core.runconfig import RunConfiguration

# Edit a run configuration property on the fly.
run_local = RunConfiguration()

run_local.environment.python.user_managed_dependencies = True
```

Now that you've attached the compute and configured your run, the next step is to submit the training run.

Azure Machine Learning Compute

Azure Machine Learning Compute is a managed-compute infrastructure that allows the user to easily create a single or multi-node compute. The compute is created within your workspace region as a resource that can be shared with other users in your workspace. The compute scales up automatically when a job is submitted, and can be put in an Azure Virtual Network. The compute executes in a containerized environment and packages your model dependencies in a Docker container.

You can use Azure Machine Learning Compute to distribute the training process across a cluster of CPU or GPU compute nodes in the cloud. For more information on the VM sizes that include GPUs, see GPU-optimized virtual machine sizes.

Azure Machine Learning Compute has default limits, such as the number of cores that can be allocated. For more information, see Manage and request quotas for Azure resources.

> **TIP**
> Clusters can generally scale upto 100 nodes as long as you have enough quota for the number of cores required. By default clusters are setup with inter-node communication enabled between the nodes of the cluster to support MPI jobs for example. However you can scale your clusters to 1000s of nodes by simply raising a support ticket, and requesting to whitelist your subscription, or workspace, or a specific cluster for disabling inter-node communication.

Azure Machine Learning Compute can be reused across runs. The compute can be shared with other users in the

workspace and is retained between runs, automatically scaling nodes up or down based on the number of runs submitted, and the max_nodes set on your cluster.

1. **Create and attach**: To create a persistent Azure Machine Learning Compute resource in Python, specify the **vm_size** and **max_nodes** properties. Azure Machine Learning then uses smart defaults for the other properties. The compute autoscales down to zero nodes when it isn't used. Dedicated VMs are created to run your jobs as needed.

   - **vm_size**: The VM family of the nodes created by Azure Machine Learning Compute.
   - **max_nodes**: The max number of nodes to autoscale up to when you run a job on Azure Machine Learning Compute.

   ```
   from azureml.core.compute import ComputeTarget, AmlCompute
   from azureml.core.compute_target import ComputeTargetException

   # Choose a name for your CPU cluster
   cpu_cluster_name = "cpucluster"

   # Verify that cluster does not exist already
   try:
       cpu_cluster = ComputeTarget(workspace=ws, name=cpu_cluster_name)
       print('Found existing cluster, use it.')
   except ComputeTargetException:
       compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                              max_nodes=4)
       cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)

   cpu_cluster.wait_for_completion(show_output=True)
   ```

   You can also configure several advanced properties when you create Azure Machine Learning Compute. The properties allow you to create a persistent cluster of fixed size, or within an existing Azure Virtual Network in your subscription. See the AmlCompute class for details.

   Or you can create and attach a persistent Azure Machine Learning Compute resource in Azure Machine Learning studio.

2. **Configure**: Create a run configuration for the persistent compute target.

   ```
   from azureml.core.runconfig import RunConfiguration
   from azureml.core.conda_dependencies import CondaDependencies
   from azureml.core.runconfig import DEFAULT_CPU_IMAGE

   # Create a new runconfig object
   run_amlcompute = RunConfiguration()

   # Use the cpu_cluster you created above.
   run_amlcompute.target = cpu_cluster

   # Enable Docker
   run_amlcompute.environment.docker.enabled = True

   # Set Docker base image to the default CPU-based image
   run_amlcompute.environment.docker.base_image = DEFAULT_CPU_IMAGE

   # Use conda_dependencies.yml to create a conda environment in the Docker image for execution
   run_amlcompute.environment.python.user_managed_dependencies = False

   # Specify CondaDependencies obj, add necessary packages
   run_amlcompute.environment.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-
   learn'])
   ```

Now that you've attached the compute and configured your run, the next step is to submit the training run.

Remote virtual machines

Azure Machine Learning also supports bringing your own compute resource and attaching it to your workspace. One such resource type is an arbitrary remote VM, as long as it's accessible from Azure Machine Learning. The resource can be an Azure VM, a remote server in your organization, or on-premises. Specifically, given the IP address and credentials (user name and password, or SSH key), you can use any accessible VM for remote runs.

You can use a system-built conda environment, an already existing Python environment, or a Docker container. To execute on a Docker container, you must have a Docker Engine running on the VM. This functionality is especially useful when you want a more flexible, cloud-based dev/experimentation environment than your local machine.

Use the Azure Data Science Virtual Machine (DSVM) as the Azure VM of choice for this scenario. This VM is a pre-configured data science and AI development environment in Azure. The VM offers a curated choice of tools and frameworks for full-lifecycle machine learning development. For more information on how to use the DSVM with Azure Machine Learning, see Configure a development environment.

1. **Create**: Create a DSVM before using it to train your model. To create this resource, see Provision the Data Science Virtual Machine for Linux (Ubuntu).

   > **WARNING**
   > Azure Machine Learning only supports virtual machines that run Ubuntu. When you create a VM or choose an existing VM, you must select a VM that uses Ubuntu.

2. **Attach**: To attach an existing virtual machine as a compute target, you must provide the resource ID, user name, and password for the virtual machine. The resource ID of the VM can be constructed using the subscription ID, resource group name, and VM name using the following string format:

   `/subscriptions/<subscription_id>/resourceGroups/<resource_group>/providers/Microsoft.Compute/virtualMachines/<vm_name>`

   ```
   from azureml.core.compute import RemoteCompute, ComputeTarget

   # Create the compute config
   compute_target_name = "attach-dsvm"

   attach_config = RemoteCompute.attach_configuration(resource_id='<resource_id>',
                                                      ssh_port=22,
                                                      username='<username>',
                                                      password="<password>")

   # If you authenticate with SSH keys instead, use this code:
   #                                                    ssh_port=22,
   #                                                    username='<username>',
   #                                                    password=None,
   #                                                    private_key_file="<path-to-file>",
   #                                                    private_key_passphrase="<passphrase>")

   # Attach the compute
   compute = ComputeTarget.attach(ws, compute_target_name, attach_config)

   compute.wait_for_completion(show_output=True)
   ```

   Or you can attach the DSVM to your workspace using Azure Machine Learning studio.

3. **Configure**: Create a run configuration for the DSVM compute target. Docker and conda are used to create and configure the training environment on the DSVM.

```
import azureml.core
from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies

run_dsvm = RunConfiguration(framework = "python")

# Set the compute target to the Linux DSVM
run_dsvm.target = compute_target_name

# Use Docker in the remote VM
run_dsvm.environment.docker.enabled = True

# Use the CPU base image
# To use GPU in DSVM, you must also use the GPU base Docker image
"azureml.core.runconfig.DEFAULT_GPU_IMAGE"
run_dsvm.environment.docker.base_image = azureml.core.runconfig.DEFAULT_CPU_IMAGE
print('Base Docker image is:', run_dsvm.environment.docker.base_image)

# Specify the CondaDependencies object
run_dsvm.environment.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-
learn'])
```

Now that you've attached the compute and configured your run, the next step is to submit the training run.

Azure HDInsight

Azure HDInsight is a popular platform for big-data analytics. The platform provides Apache Spark, which can be used to train your model.

1. **Create**: Create the HDInsight cluster before you use it to train your model. To create a Spark on HDInsight cluster, see Create a Spark Cluster in HDInsight.

   When you create the cluster, you must specify an SSH user name and password. Take note of these values, as you need them to use HDInsight as a compute target.

   After the cluster is created, connect to it with the hostname <clustername>-ssh.azurehdinsight.net, where <clustername> is the name that you provided for the cluster.

2. **Attach**: To attach an HDInsight cluster as a compute target, you must provide the resource ID, user name, and password for the HDInsight cluster. The resource ID of the HDInsight cluster can be constructed using the subscription ID, resource group name, and HDInsight cluster name using the following string format:
   `/subscriptions/<subscription_id>/resourceGroups/<resource_group>/providers/Microsoft.HDInsight/clusters/<cluster_name>`

```
from azureml.core.compute import ComputeTarget, HDInsightCompute
from azureml.exceptions import ComputeTargetException

try:
 # if you want to connect using SSH key instead of username/password you can provide parameters
private_key_file and private_key_passphrase

 attach_config = HDInsightCompute.attach_configuration(resource_id='<resource_id>',
                                                        ssh_port=22,
                                                        username='<ssh-username>',
                                                        password='<ssh-pwd>')
 hdi_compute = ComputeTarget.attach(workspace=ws,
                                    name='myhdi',
                                    attach_configuration=attach_config)

except ComputeTargetException as e:
 print("Caught = {}".format(e.message))

hdi_compute.wait_for_completion(show_output=True)
```

Or you can attach the HDInsight cluster to your workspace using Azure Machine Learning studio.

3. **Configure**: Create a run configuration for the HDI compute target.

```
from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies


# use pyspark framework
run_hdi = RunConfiguration(framework="pyspark")

# Set compute target to the HDI cluster
run_hdi.target = hdi_compute.name

# specify CondaDependencies object to ask system installing numpy
cd = CondaDependencies()
cd.add_conda_package('numpy')
run_hdi.environment.python.conda_dependencies = cd
```

Now that you've attached the compute and configured your run, the next step is to submit the training run.

Azure Batch

Azure Batch is used to run large-scale parallel and high-performance computing (HPC) applications efficiently in the cloud. AzureBatchStep can be used in an Azure Machine Learning Pipeline to submit jobs to an Azure Batch pool of machines.

To attach Azure Batch as a compute target, you must use the Azure Machine Learning SDK and provide the following information:

- **Azure Batch compute name**: A friendly name to be used for the compute within the workspace
- **Azure Batch account name**: The name of the Azure Batch account
- **Resource Group**: The resource group that contains the Azure Batch account.

The following code demonstrates how to attach Azure Batch as a compute target:

```
from azureml.core.compute import ComputeTarget, BatchCompute
from azureml.exceptions import ComputeTargetException

# Name to associate with new compute in workspace
batch_compute_name = 'mybatchcompute'

# Batch account details needed to attach as compute to workspace
batch_account_name = "<batch_account_name>"  # Name of the Batch account
# Name of the resource group which contains this account
batch_resource_group = "<batch_resource_group>"

try:
    # check if the compute is already attached
    batch_compute = BatchCompute(ws, batch_compute_name)
except ComputeTargetException:
    print('Attaching Batch compute...')
    provisioning_config = BatchCompute.attach_configuration(
        resource_group=batch_resource_group, account_name=batch_account_name)
    batch_compute = ComputeTarget.attach(
        ws, batch_compute_name, provisioning_config)
    batch_compute.wait_for_completion()
    print("Provisioning state:{}".format(batch_compute.provisioning_state))
    print("Provisioning errors:{}".format(batch_compute.provisioning_errors))

print("Using Batch compute:{}".format(batch_compute.cluster_resource_id))
```

Set up in Azure Machine Learning studio

You can access the compute targets that are associated with your workspace in the Azure Machine Learning studio. You can use the studio to:

- View compute targets attached to your workspace
- Create a compute target in your workspace
- Attach a compute target that was created outside the workspace

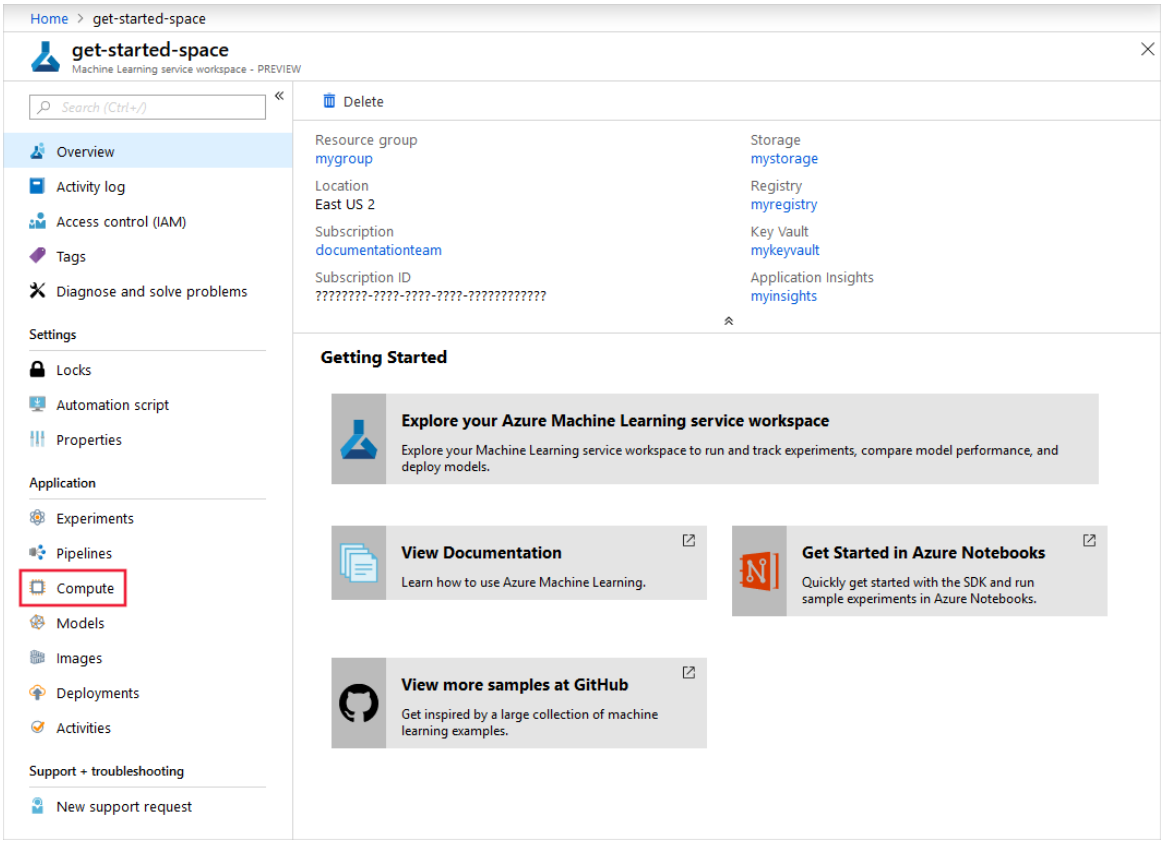
After a target is created and attached to your workspace, you will use it in your run configuration with a `ComputeTarget` object:

```
from azureml.core.compute import ComputeTarget
myvm = ComputeTarget(workspace=ws, name='my-vm-name')
```

View compute targets

To see the compute targets for your workspace, use the following steps:

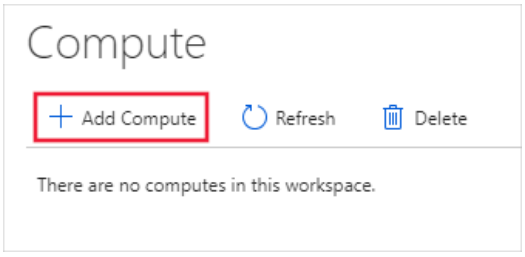
1. Navigate to Azure Machine Learning studio.
2. Under **Applications**, select **Compute**.



Create a compute target

Follow the previous steps to view the list of compute targets. Then use these steps to create a compute target:

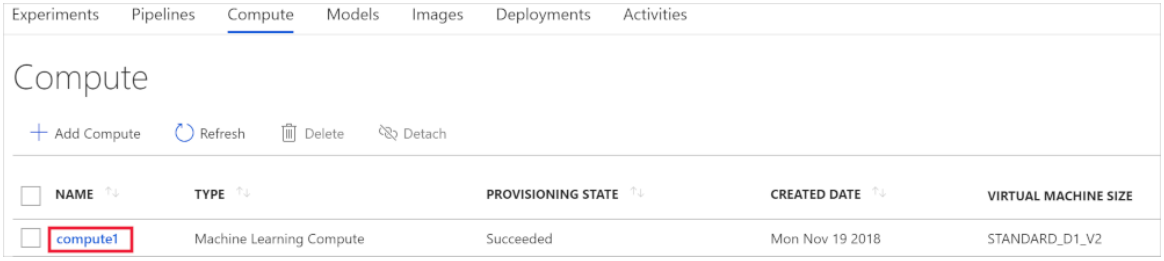
1. Select the plus sign (+) to add a compute target.



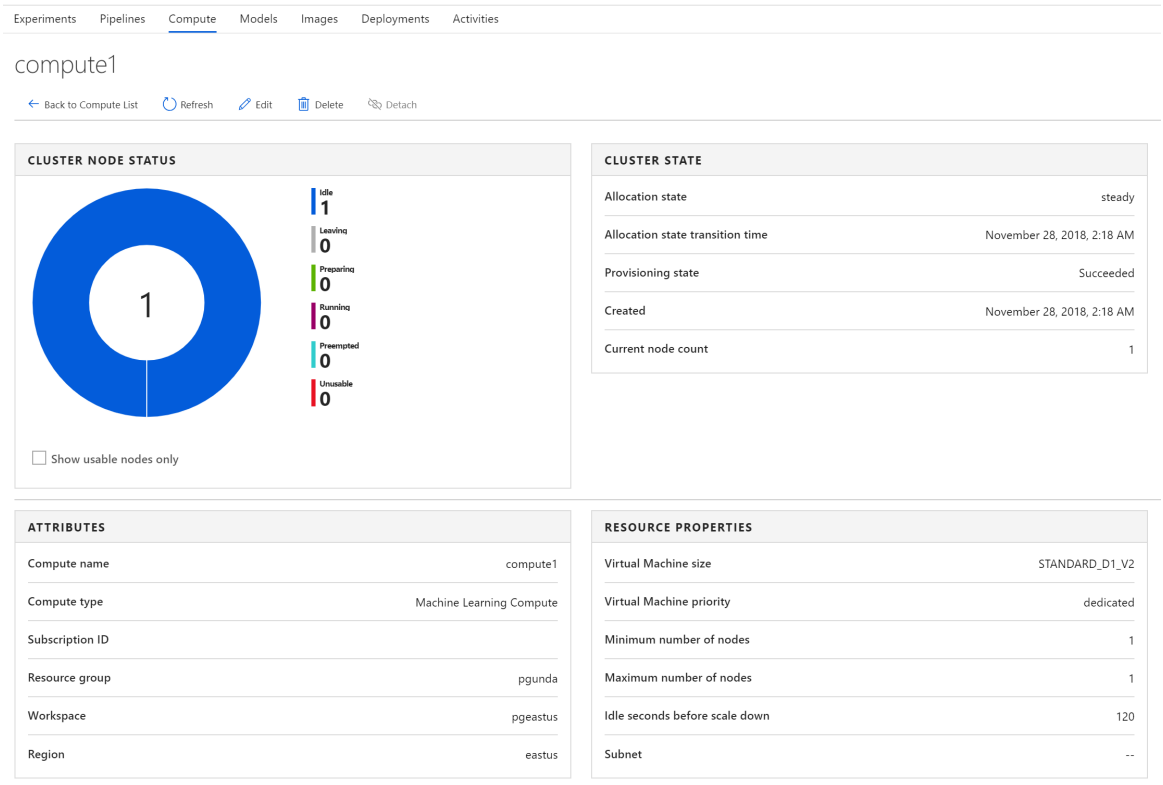
2. Enter a name for the compute target.
3. Select **Machine Learning Compute** as the type of compute to use for **Training**.

> **NOTE**
> Azure Machine Learning Compute is the only managed-compute resource you can create in Azure Machine Learning studio. All other compute resources can be attached after they are created.

4. Fill out the form. Provide values for the required properties, especially **VM Family**, and the **maximum nodes** to use to spin up the compute.
5. Select **Create**.
6. View the status of the create operation by selecting the compute target from the list:



7. You then see the details for the compute target:



Attach compute targets

To use compute targets created outside the Azure Machine Learning workspace, you must attach them. Attaching a compute target makes it available to your workspace.

Follow the steps described earlier to view the list of compute targets. Then use the following steps to attach a compute target:

1. Select the plus sign (+) to add a compute target.
2. Enter a name for the compute target.
3. Select the type of compute to attach for **Training**:

> **IMPORTANT**
>
> Not all compute types can be attached from Azure Machine Learning studio. The compute types that can currently be attached for training include:
>
> - A remote VM
> - Azure Databricks (for use in machine learning pipelines)
> - Azure Data Lake Analytics (for use in machine learning pipelines)
> - Azure HDInsight

4. Fill out the form and provide values for the required properties.

> **NOTE**
>
> Microsoft recommends that you use SSH keys, which are more secure than passwords. Passwords are vulnerable to brute force attacks. SSH keys rely on cryptographic signatures. For information on how to create SSH keys for use with Azure Virtual Machines, see the following documents:
>
> - Create and use SSH keys on Linux or macOS
> - Create and use SSH keys on Windows

5. Select **Attach**.

6. View the status of the attach operation by selecting the compute target from the list.

Set up with CLI

You can access the compute targets that are associated with your workspace using the CLI extension for Azure Machine Learning. You can use the CLI to:

- Create a managed compute target
- Update a managed compute target
- Attach an unmanaged compute target

For more information, see Resource management.

Set up with VS Code

You can access, create, and manage the compute targets that are associated with your workspace using the VS Code extension for Azure Machine Learning.

Submit training run using Azure Machine Learning SDK

After you create a run configuration, you use it to run your experiment. The code pattern to submit a training run is the same for all types of compute targets:

1. Create an experiment to run
2. Submit the run.
3. Wait for the run to complete.

IMPORTANT

When you submit the training run, a snapshot of the directory that contains your training scripts is created and sent to the compute target. It is also stored as part of the experiment in your workspace. If you change files and submit the run again, only the changed files will be uploaded.

To prevent files from being included in the snapshot, create a .gitignore or `.amlignore` file in the directory and add the files to it. The `.amlignore` file uses the same syntax and patterns as the .gitignore file. If both files exist, the `.amlignore` file takes precedence.

For more information, see Snapshots.

Create an experiment

First, create an experiment in your workspace.

```
from azureml.core import Experiment
experiment_name = 'my_experiment'

exp = Experiment(workspace=ws, name=experiment_name)
```

Submit the experiment

Submit the experiment with a `ScriptRunConfig` object. This object includes the:

- **source_directory**: The source directory that contains your training script
- **script**: Identify the training script
- **run_config**: The run configuration, which in turn defines where the training will occur.

For example, to use the local target configuration:

```
from azureml.core import ScriptRunConfig
import os

script_folder = os.getcwd()
src = ScriptRunConfig(source_directory = script_folder, script = 'train.py', run_config = run_local)
run = exp.submit(src)
run.wait_for_completion(show_output = True)
```

Switch the same experiment to run in a different compute target by using a different run configuration, such as the amlcompute target:

```
from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory = script_folder, script = 'train.py', run_config = run_amlcompute)
run = exp.submit(src)
run.wait_for_completion(show_output = True)
```

TIP

This example defaults to only using one node of the compute target for training. To use more than one node, set the `node_count` of the run configuration to the desired number of nodes. For example, the following code sets the number of nodes used for training to four:

```
src.run_config.node_count = 4
```

Or you can:

- Submit the experiment with an `Estimator` object as shown in Train ML models with estimators.
- Submit a HyperDrive run for hyperparameter tuning.
- Submit an experiment via the VS Code extension.

For more information, see the ScriptRunConfig and RunConfiguration documentation.

Create run configuration and submit run using Azure Machine Learning CLI

You can use Azure CLI and Machine Learning CLI extension to create run configurations and submit runs on different compute targets. The following examples assume that you have an existing Azure Machine Learning Workspace and you have logged in to Azure using `az login` CLI command.

> **TIP**
> After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.
>
> To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

Create run configuration

The simplest way to create run configuration is to navigate the folder that contains your machine learning Python scripts, and use CLI command

```
az ml folder attach
```

This command creates a subfolder `.azureml` that contains template run configuration files for different compute targets. You can copy and edit these files to customize your configuration, for example to add Python packages or change Docker settings.

Structure of run configuration file

The run configuration file is YAML formatted, with following sections

- The script to run and its arguments
- Compute target name, either "local" or name of a compute under the workspace.
- Parameters for executing the run: framework, communicator for distributed runs, maximum duration, and number of compute nodes.
- Environment section. See Create and manage environments for training and deployment for details of the fields in this section.
  - To specify Python packages to install for the run, create conda environment file, and set **condaDependenciesFile** field.
- Run history details to specify log file folder, and to enable or disable output collection and run history snapshots.
- Configuration details specific to the framework selected.
- Data reference and data store details.
- Configuration details specific for Machine Learning Compute for creating a new cluster.

See the example JSON file for a full runconfig schema.

Create an experiment

First, create an experiment for your runs

```
az ml experiment create -n <experiment>
```

Script run

To submit a script run, execute a command

```
az ml run submit-script -e <experiment> -c <runconfig> my_train.py
```

HyperDrive run

You can use HyperDrive with Azure CLI to perform parameter tuning runs. First, create a HyperDrive configuration file in the following format. See Tune hyperparameters for your model article for details on hyperparameter tuning parameters.

```
# hdconfig.yml
sampling:
    type: random # Supported options: Random, Grid, Bayesian
    parameter_space: # specify a name|expression|values tuple for each parameter.
    - name: --penalty # The name of a script parameter to generate values for.
      expression: choice # supported options: choice, randint, uniform, quniform, loguniform, qloguniform,
normal, qnormal, lognormal, qlognormal
      values: [0.5, 1, 1.5] # The list of values, the number of values is dependent on the expression
specified.
policy:
    type: BanditPolicy # Supported options: BanditPolicy, MedianStoppingPolicy, TruncationSelectionPolicy,
NoTerminationPolicy
    evaluation_interval: 1 # Policy properties are policy specific. See the above link for policy specific
parameter details.
    slack_factor: 0.2
primary_metric_name: Accuracy # The metric used when evaluating the policy
primary_metric_goal: Maximize # Maximize|Minimize
max_total_runs: 8 # The maximum number of runs to generate
max_concurrent_runs: 2 # The number of runs that can run concurrently.
max_duration_minutes: 100 # The maximum length of time to run the experiment before cancelling.
```

Add this file alongside the run configuration files. Then submit a HyperDrive run using:

```
az ml run submit-hyperdrive -e <experiment> -c <runconfig> --hyperdrive-configuration-name <hdconfig>
my_train.py
```

Note the *arguments* section in runconfig and *parameter space* in HyperDrive config. They contain the command-line arguments to be passed to training script. The value in runconfig stays the same for each iteration, while the range in HyperDrive config is iterated over. Do not specify the same argument in both files.

For more details on these `az ml` CLI commands and full set of arguments, see the reference documentation.

Git tracking and integration

When you start a training run where the source directory is a local Git repository, information about the repository is stored in the run history. For more information, see Git integration for Azure Machine Learning.

Notebook examples

See these notebooks for examples of training with various compute targets:

- how-to-use-azureml/training
- tutorials/img-classification-part1-training.ipynb

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Next steps

- Tutorial: Train a model uses a managed compute target to train a model.
- Learn how to efficiently tune hyperparameters to build better models.
- Once you have a trained model, learn how and where to deploy models.
- View the RunConfiguration class SDK reference.
- Use Azure Machine Learning with Azure Virtual Networks

Tune hyperparameters for your model with Azure Machine Learning

3/30/2020 • 15 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Efficiently tune hyperparameters for your model using Azure Machine Learning. Hyperparameter tuning includes the following steps:

- Define the parameter search space
- Specify a primary metric to optimize
- Specify early termination criteria for poorly performing runs
- Allocate resources for hyperparameter tuning
- Launch an experiment with the above configuration
- Visualize the training runs
- Select the best performing configuration for your model

What are hyperparameters?

Hyperparameters are adjustable parameters you choose to train a model that govern the training process itself. For example, to train a deep neural network, you decide the number of hidden layers in the network and the number of nodes in each layer prior to training the model. These values usually stay constant during the training process.

In deep learning / machine learning scenarios, model performance depends heavily on the hyperparameter values selected. The goal of hyperparameter exploration is to search across various hyperparameter configurations to find a configuration that results in the best performance. Typically, the hyperparameter exploration process is painstakingly manual, given that the search space is vast and evaluation of each configuration can be expensive.

Azure Machine Learning allows you to automate hyperparameter exploration in an efficient manner, saving you significant time and resources. You specify the range of hyperparameter values and a maximum number of training runs. The system then automatically launches multiple simultaneous runs with different parameter configurations and finds the configuration that results in the best performance, measured by the metric you choose. Poorly performing training runs are automatically early terminated, reducing wastage of compute resources. These resources are instead used to explore other hyperparameter configurations.

Define search space

Automatically tune hyperparameters by exploring the range of values defined for each hyperparameter.

Types of hyperparameters

Each hyperparameter can either be discrete or continuous and has a distribution of values described by a parameter expression.

Discrete hyperparameters

Discrete hyperparameters are specified as a `choice` among discrete values. `choice` can be:

- one or more comma-separated values
- a `range` object

- any arbitrary `list` object

```
{
    "batch_size": choice(16, 32, 64, 128)
    "number_of_hidden_layers": choice(range(1,5))
}
```

In this case, `batch_size` takes on one of the values [16, 32, 64, 128] and `number_of_hidden_layers` takes on one of the values [1, 2, 3, 4].

Advanced discrete hyperparameters can also be specified using a distribution. The following distributions are supported:

- `quniform(low, high, q)` - Returns a value like round(uniform(low, high) / q) * q
- `qloguniform(low, high, q)` - Returns a value like round(exp(uniform(low, high)) / q) * q
- `qnormal(mu, sigma, q)` - Returns a value like round(normal(mu, sigma) / q) * q
- `qlognormal(mu, sigma, q)` - Returns a value like round(exp(normal(mu, sigma)) / q) * q

Continuous hyperparameters

Continuous hyperparameters are specified as a distribution over a continuous range of values. Supported distributions include:

- `uniform(low, high)` - Returns a value uniformly distributed between low and high
- `loguniform(low, high)` - Returns a value drawn according to exp(uniform(low, high)) so that the logarithm of the return value is uniformly distributed
- `normal(mu, sigma)` - Returns a real value that's normally distributed with mean mu and standard deviation sigma
- `lognormal(mu, sigma)` - Returns a value drawn according to exp(normal(mu, sigma)) so that the logarithm of the return value is normally distributed

An example of a parameter space definition:

```
{
    "learning_rate": normal(10, 3),
    "keep_probability": uniform(0.05, 0.1)
}
```

This code defines a search space with two parameters - `learning_rate` and `keep_probability`. `learning_rate` has a normal distribution with mean value 10 and a standard deviation of 3. `keep_probability` has a uniform distribution with a minimum value of 0.05 and a maximum value of 0.1.

Sampling the hyperparameter space

You can also specify the parameter sampling method to use over the hyperparameter space definition. Azure Machine Learning supports random sampling, grid sampling, and Bayesian sampling.

Picking a sampling method

- Grid sampling can be used if your hyperparameter space can be defined as a choice among discrete values and if you have sufficient budget to exhaustively search over all values in the defined search space. Additionally, one can use automated early termination of poorly performing runs, which reduces wastage of resources.
- Random sampling allows the hyperparameter space to include both discrete and continuous hyperparameters. In practice it produces good results most of the times and also allows the use of automated early termination of poorly performing runs. Some users perform an initial search using random sampling and then iteratively refine the search space to improve results.

- Bayesian sampling leverages knowledge of previous samples when choosing hyperparameter values, effectively trying to improve the reported primary metric. Bayesian sampling is recommended when you have sufficient budget to explore the hyperparameter space - for best results with Bayesian Sampling we recommend using a maximum number of runs greater than or equal to 20 times the number of hyperparameters being tuned. Note that Bayesian sampling does not currently support any early termination policy.

Random sampling

In random sampling, hyperparameter values are randomly selected from the defined search space. Random sampling allows the search space to include both discrete and continuous hyperparameters.

```
from azureml.train.hyperdrive import RandomParameterSampling
param_sampling = RandomParameterSampling( {
        "learning_rate": normal(10, 3),
        "keep_probability": uniform(0.05, 0.1),
        "batch_size": choice(16, 32, 64, 128)
    }
)
```

Grid sampling

Grid sampling performs a simple grid search over all feasible values in the defined search space. It can only be used with hyperparameters specified using `choice`. For example, the following space has a total of six samples:

```
from azureml.train.hyperdrive import GridParameterSampling
param_sampling = GridParameterSampling( {
        "num_hidden_layers": choice(1, 2, 3),
        "batch_size": choice(16, 32)
    }
)
```

Bayesian sampling

Bayesian sampling is based on the Bayesian optimization algorithm and makes intelligent choices on the hyperparameter values to sample next. It picks the sample based on how the previous samples performed, such that the new sample improves the reported primary metric.

When you use Bayesian sampling, the number of concurrent runs has an impact on the effectiveness of the tuning process. Typically, a smaller number of concurrent runs can lead to better sampling convergence, since the smaller degree of parallelism increases the number of runs that benefit from previously completed runs.

Bayesian sampling only supports `choice`, `uniform`, and `quniform` distributions over the search space.

```
from azureml.train.hyperdrive import BayesianParameterSampling
param_sampling = BayesianParameterSampling( {
        "learning_rate": uniform(0.05, 0.1),
        "batch_size": choice(16, 32, 64, 128)
    }
)
```

> **NOTE**
>
> Bayesian sampling does not support any early termination policy (See Specify an early termination policy). When using Bayesian parameter sampling, set `early_termination_policy = None`, or leave off the `early_termination_policy` parameter.

Specify primary metric

Specify the primary metric you want the hyperparameter tuning experiment to optimize. Each training run is evaluated for the primary metric. Poorly performing runs (where the primary metric does not meet criteria set by the early termination policy) will be terminated. In addition to the primary metric name, you also specify the goal of the optimization - whether to maximize or minimize the primary metric.

- `primary_metric_name`: The name of the primary metric to optimize. The name of the primary metric needs to exactly match the name of the metric logged by the training script. See Log metrics for hyperparameter tuning.
- `primary_metric_goal`: It can be either `PrimaryMetricGoal.MAXIMIZE` or `PrimaryMetricGoal.MINIMIZE` and determines whether the primary metric will be maximized or minimized when evaluating the runs.

```
primary_metric_name="accuracy",
primary_metric_goal=PrimaryMetricGoal.MAXIMIZE
```

Optimize the runs to maximize "accuracy". Make sure to log this value in your training script.

Log metrics for hyperparameter tuning

The training script for your model must log the relevant metrics during model training. When you configure the hyperparameter tuning, you specify the primary metric to use for evaluating run performance. (See Specify a primary metric to optimize.) In your training script, you must log this metric so it is available to the hyperparameter tuning process.

Log this metric in your training script with the following sample snippet:

```
from azureml.core.run import Run
run_logger = Run.get_context()
run_logger.log("accuracy", float(val_accuracy))
```

The training script calculates the `val_accuracy` and logs it as "accuracy", which is used as the primary metric. Each time the metric is logged it is received by the hyperparameter tuning service. It is up to the model developer to determine how frequently to report this metric.

Specify early termination policy

Terminate poorly performing runs automatically with an early termination policy. Termination reduces wastage of resources and instead uses these resources for exploring other parameter configurations.

When using an early termination policy, you can configure the following parameters that control when a policy is applied:

- `evaluation_interval`: the frequency for applying the policy. Each time the training script logs the primary metric counts as one interval. Thus an `evaluation_interval` of 1 will apply the policy every time the training script reports the primary metric. An `evaluation_interval` of 2 will apply the policy every other time the training script reports the primary metric. If not specified, `evaluation_interval` is set to 1 by default.
- `delay_evaluation`: delays the first policy evaluation for a specified number of intervals. It is an optional parameter that allows all configurations to run for an initial minimum number of intervals, avoiding premature termination of training runs. If specified, the policy applies every multiple of evaluation_interval that is greater than or equal to delay_evaluation.

Azure Machine Learning supports the following Early Termination Policies.

Bandit policy

Bandit is a termination policy based on slack factor/slack amount and evaluation interval. The policy early terminates any runs where the primary metric is not within the specified slack factor / slack amount with respect

to the best performing training run. It takes the following configuration parameters:

- `slack_factor` or `slack_amount`: the slack allowed with respect to the best performing training run. `slack_factor` specifies the allowable slack as a ratio. `slack_amount` specifies the allowable slack as an absolute amount, instead of a ratio.

  For example, consider a Bandit policy being applied at interval 10. Assume that the best performing run at interval 10 reported a primary metric 0.8 with a goal to maximize the primary metric. If the policy was specified with a `slack_factor` of 0.2, any training runs, whose best metric at interval 10 is less than 0.66 (0.8/(1+`slack_factor`)) will be terminated. If instead, the policy was specified with a `slack_amount` of 0.2, any training runs, whose best metric at interval 10 is less than 0.6 (0.8 - `slack_amount`) will be terminated.

- `evaluation_interval`: the frequency for applying the policy (optional parameter).
- `delay_evaluation`: delays the first policy evaluation for a specified number of intervals (optional parameter).

```
from azureml.train.hyperdrive import BanditPolicy
early_termination_policy = BanditPolicy(slack_factor = 0.1, evaluation_interval=1, delay_evaluation=5)
```

In this example, the early termination policy is applied at every interval when metrics are reported, starting at evaluation interval 5. Any run whose best metric is less than (1/(1+0.1) or 91% of the best performing run will be terminated.

Median stopping policy

Median stopping is an early termination policy based on running averages of primary metrics reported by the runs. This policy computes running averages across all training runs and terminates runs whose performance is worse than the median of the running averages. This policy takes the following configuration parameters:

- `evaluation_interval`: the frequency for applying the policy (optional parameter).
- `delay_evaluation`: delays the first policy evaluation for a specified number of intervals (optional parameter).

```
from azureml.train.hyperdrive import MedianStoppingPolicy
early_termination_policy = MedianStoppingPolicy(evaluation_interval=1, delay_evaluation=5)
```

In this example, the early termination policy is applied at every interval starting at evaluation interval 5. A run will be terminated at interval 5 if its best primary metric is worse than the median of the running averages over intervals 1:5 across all training runs.

Truncation selection policy

Truncation selection cancels a given percentage of lowest performing runs at each evaluation interval. Runs are compared based on their performance on the primary metric and the lowest X% are terminated. It takes the following configuration parameters:

- `truncation_percentage`: the percentage of lowest performing runs to terminate at each evaluation interval. Specify an integer value between 1 and 99.
- `evaluation_interval`: the frequency for applying the policy (optional parameter).
- `delay_evaluation`: delays the first policy evaluation for a specified number of intervals (optional parameter).

```
from azureml.train.hyperdrive import TruncationSelectionPolicy
early_termination_policy = TruncationSelectionPolicy(evaluation_interval=1, truncation_percentage=20,
delay_evaluation=5)
```

In this example, the early termination policy is applied at every interval starting at evaluation interval 5. A run will

be terminated at interval 5 if its performance at interval 5 is in the lowest 20% of performance of all runs at interval 5.

No termination policy

If you want all training runs to run to completion, set policy to None. This will have the effect of not applying any early termination policy.

```
policy=None
```

Default policy

If no policy is specified, the hyperparameter tuning service will let all training runs execute to completion.

Picking an early termination policy

- If you are looking for a conservative policy that provides savings without terminating promising jobs, you can use a Median Stopping Policy with `evaluation_interval` 1 and `delay_evaluation` 5. These are conservative settings, that can provide approximately 25%-35% savings with no loss on primary metric (based on our evaluation data).
- If you are looking for more aggressive savings from early termination, you can either use Bandit Policy with a stricter (smaller) allowable slack or Truncation Selection Policy with a larger truncation percentage.

Allocate resources

Control your resource budget for your hyperparameter tuning experiment by specifying the maximum total number of training runs. Optionally specify the maximum duration for your hyperparameter tuning experiment.

- `max_total_runs`: Maximum total number of training runs that will be created. Upper bound - there may be fewer runs, for instance, if the hyperparameter space is finite and has fewer samples. Must be a number between 1 and 1000.
- `max_duration_minutes`: Maximum duration in minutes of the hyperparameter tuning experiment. Parameter is optional, and if present, any runs that would be running after this duration are automatically canceled.

> **NOTE**
> If both `max_total_runs` and `max_duration_minutes` are specified, the hyperparameter tuning experiment terminates when the first of these two thresholds is reached.

Additionally, specify the maximum number of training runs to run concurrently during your hyperparameter tuning search.

- `max_concurrent_runs`: Maximum number of runs to run concurrently at any given moment. If none specified, all `max_total_runs` will be launched in parallel. If specified, must be a number between 1 and 100.

> **NOTE**
> The number of concurrent runs is gated on the resources available in the specified compute target. Hence, you need to ensure that the compute target has the available resources for the desired concurrency.

Allocate resources for hyperparameter tuning:

```
max_total_runs=20,
max_concurrent_runs=4
```

This code configures the hyperparameter tuning experiment to use a maximum of 20 total runs, running four configurations at a time.

Configure experiment

Configure your hyperparameter tuning experiment using the defined hyperparameter search space, early termination policy, primary metric, and resource allocation from the sections above. Additionally, provide an `estimator` that will be called with the sampled hyperparameters. The `estimator` describes the training script you run, the resources per job (single or multi-gpu), and the compute target to use. Since concurrency for your hyperparameter tuning experiment is gated on the resources available, ensure that the compute target specified in the `estimator` has sufficient resources for your desired concurrency. (For more information on estimators, see how to train models.)

Configure your hyperparameter tuning experiment:

```
from azureml.train.hyperdrive import HyperDriveConfig
hyperdrive_run_config = HyperDriveConfig(estimator=estimator,
                          hyperparameter_sampling=param_sampling,
                          policy=early_termination_policy,
                          primary_metric_name="accuracy",
                          primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
                          max_total_runs=100,
                          max_concurrent_runs=4)
```

Submit experiment

Once you define your hyperparameter tuning configuration, submit an experiment:

```
from azureml.core.experiment import Experiment
experiment = Experiment(workspace, experiment_name)
hyperdrive_run = experiment.submit(hyperdrive_run_config)
```

`experiment_name` is the name you assign to your hyperparameter tuning experiment, and `workspace` is the workspace in which you want to create the experiment (For more information on experiments, see How does Azure Machine Learning work?)

Warm start your hyperparameter tuning experiment (optional)

Often, finding the best hyperparameter values for your model can be an iterative process, needing multiple tuning runs that learn from previous hyperparameter tuning runs. Reusing knowledge from these previous runs will accelerate the hyperparameter tuning process, thereby reducing the cost of tuning the model and will potentially improve the primary metric of the resulting model. When warm starting a hyperparameter tuning experiment with Bayesian sampling, trials from the previous run will be used as prior knowledge to intelligently pick new samples, to improve the primary metric. Additionally, when using Random or Grid sampling, any early termination decisions will leverage metrics from the previous runs to determine poorly performing training runs.

Azure Machine Learning allows you to warm start your hyperparameter tuning run by leveraging knowledge from up to 5 previously completed / cancelled hyperparameter tuning parent runs. You can specify the list of parent runs you want to warm start from using this snippet:

```
from azureml.train.hyperdrive import HyperDriveRun

warmstart_parent_1 = HyperDriveRun(experiment, "warmstart_parent_run_ID_1")
warmstart_parent_2 = HyperDriveRun(experiment, "warmstart_parent_run_ID_2")
warmstart_parents_to_resume_from = [warmstart_parent_1, warmstart_parent_2]
```

Additionally, there may be occasions when individual training runs of a hyperparameter tuning experiment are cancelled due to budget constraints or fail due to other reasons. It is now possible to resume such individual training runs from the last checkpoint (assuming your training script handles checkpoints). Resuming an individual training run will use the same hyperparameter configuration and mount the outputs folder used for that run. The training script should accept the `resume-from` argument, which contains the checkpoint or model files from which to resume the training run. You can resume individual training runs using the following snippet:

```
from azureml.core.run import Run

resume_child_run_1 = Run(experiment, "resume_child_run_ID_1")
resume_child_run_2 = Run(experiment, "resume_child_run_ID_2")
child_runs_to_resume = [resume_child_run_1, resume_child_run_2]
```

You can configure your hyperparameter tuning experiment to warm start from a previous experiment or resume individual training runs using the optional parameters `resume_from` and `resume_child_runs` in the config:

```
from azureml.train.hyperdrive import HyperDriveConfig

hyperdrive_run_config = HyperDriveConfig(estimator=estimator,
                          hyperparameter_sampling=param_sampling,
                          policy=early_termination_policy,
                          resume_from=warmstart_parents_to_resume_from,
                          resume_child_runs=child_runs_to_resume,
                          primary_metric_name="accuracy",
                          primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
                          max_total_runs=100,
                          max_concurrent_runs=4)
```

Visualize experiment

The Azure Machine Learning SDK provides a Notebook widget that visualizes the progress of your training runs. The following snippet visualizes all your hyperparameter tuning runs in one place in a Jupyter notebook:

```
from azureml.widgets import RunDetails
RunDetails(hyperdrive_run).show()
```

This code displays a table with details about the training runs for each of the hyperparameter configurations.

Run Properties

Status	Completed
Start Time	9/4/2018 2:55:54 PM
Duration	7 days, 0:01:41
Run Id	cifar_1536098154351
Max concurrent runs	4
Max total runs	100

Output Logs

```
previous status = 'RUNNING')]
[2018-09-11T21:57:36.389083][CONTROLLER][INFO]Experiment
was 'ExperimentStatus.RUNNING', is 'ExperimentStatus.FINISHED'.

Run is completed.
```

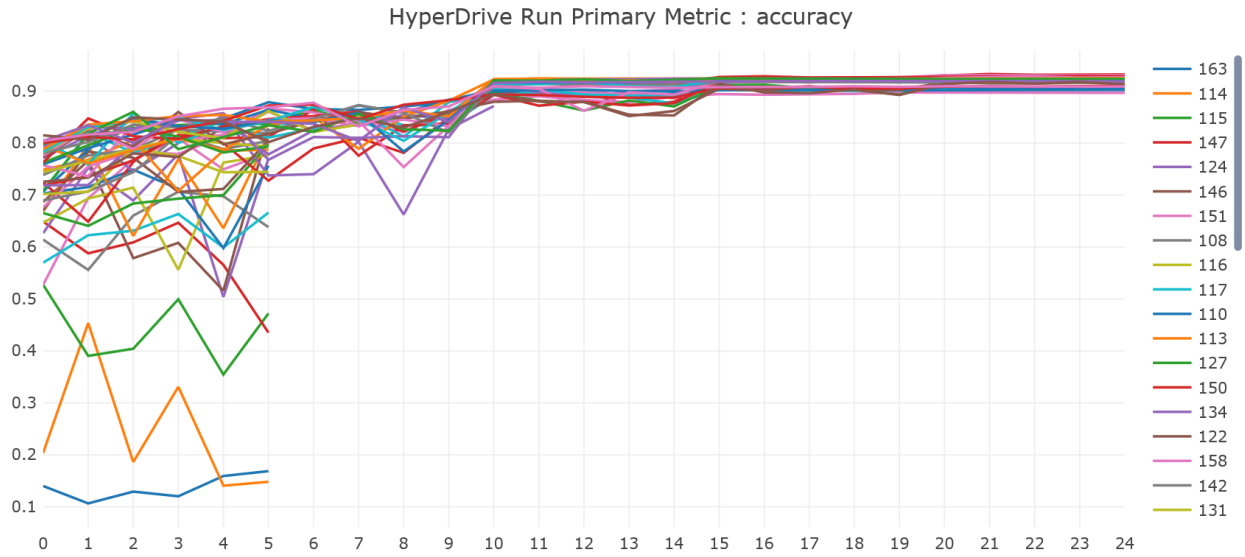
Failed (6) | Completed (36) | Canceled (58)

Run	Best Metric*	Status	Started	Duration	Run Id
103	0.909600019454956	Completed	Sep 4, 2018 2:56 PM	5:58:16	swatig-cifar_1536098154351_0
105	0.9283999800682068	Completed	Sep 4, 2018 2:56 PM	6:04:28	swatig-cifar_1536098154351_3
104	0.9247000217437744	Completed	Sep 4, 2018 2:56 PM	3:32:35	swatig-cifar_1536098154351_2
106	0.9251000285148621	Completed	Sep 4, 2018 2:56 PM	6:01:17	swatig-cifar_1536098154351_1
107	0.9108999967575073	Completed	Sep 4, 2018 6:29 PM	4:18:35	swatig-cifar_1536098154351_4

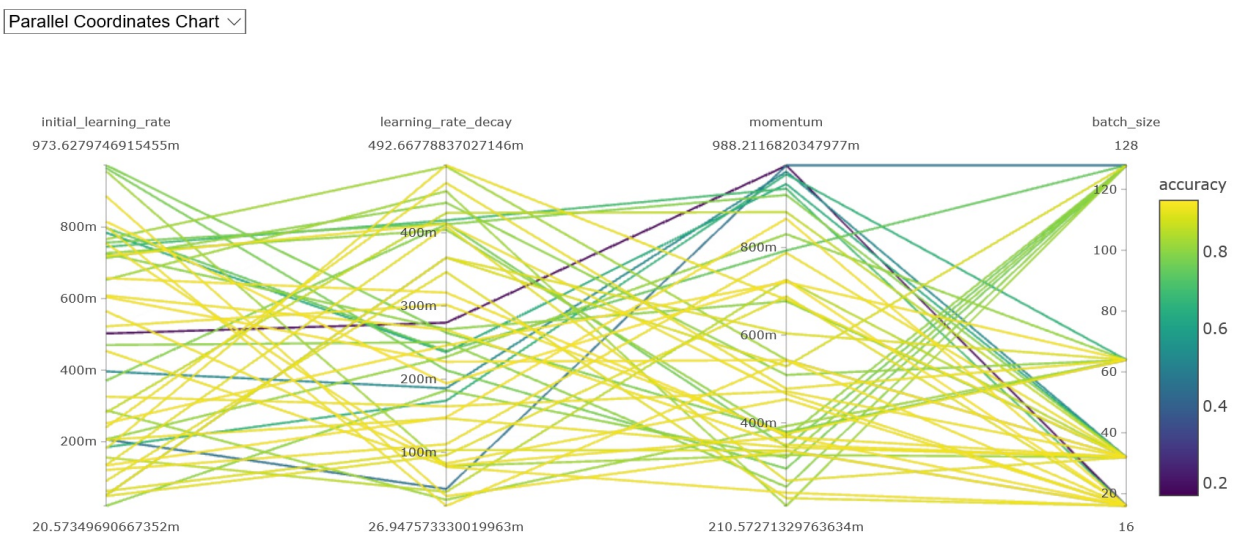
Pages: **1** 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... Next Last

* The best metric field is obtained from the min/max of primary metric achieved by a run

You can also visualize the performance of each of the runs as training progresses.



Additionally, you can visually identify the correlation between performance and values of individual hyperparameters using a Parallel Coordinates Plot.



You can visualize all your hyperparameter tuning runs in the Azure web portal as well. For more information on how to view an experiment in the web portal, see how to track experiments.

Find the best model

Once all of the hyperparameter tuning runs have completed, identify the best performing configuration and the corresponding hyperparameter values:

```
best_run = hyperdrive_run.get_best_run_by_primary_metric()
best_run_metrics = best_run.get_metrics()
parameter_values = best_run.get_details()['runDefinition']['Arguments']

print('Best Run Id: ', best_run.id)
print('\n Accuracy:', best_run_metrics['accuracy'])
print('\n learning rate:',parameter_values[3])
print('\n keep probability:',parameter_values[5])
print('\n batch size:',parameter_values[7])
```

Sample notebook

Refer to train-hyperparameter-* notebooks in this folder:

- how-to-use-azureml/training-with-deep-learning

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Next steps

- Track an experiment
- Deploy a trained model

Use secrets in training runs

3/9/2020 • 2 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to use secrets in training runs securely. Authentication information such as your user name and password are secrets. For example, if you connect to an external database in order to query training data, you would need to pass your username and password to the remote run context. Coding such values into training scripts in cleartext is insecure as it would expose the secret.

Instead, your Azure Machine Learning workspace has an associated resource called a Azure Key Vault. Use this Key Vault to pass secrets to remote runs securely through a set of APIs in the Azure Machine Learning Python SDK.

The basic flow for using secrets is:

1. On local computer, log in to Azure and connect to your workspace.
2. On local computer, set a secret in Workspace Key Vault.
3. Submit a remote run.
4. Within the remote run, get the secret from Key Vault and use it.

Set secrets

In the Azure Machine Learning, the Keyvault class contains methods for setting secrets. In your local Python session, first obtain a reference to your workspace Key Vault, and then use the `set_secret()` method to set a secret by name and value. The **set_secret** method updates the secret value if the name already exists.

```
from azureml.core import Workspace
from azureml.core import Keyvault
import os


ws = Workspace.from_config()
my_secret = os.environ.get("MY_SECRET")
keyvault = ws.get_default_keyvault()
keyvault.set_secret(name="mysecret", value = my_secret)
```

Do not put the secret value in your Python code as it is insecure to store it in file as cleartext. Instead, obtain the secret value from an environment variable, for example Azure DevOps build secret, or from interactive user input.

You can list secret names using the `list_secrets()` method and there is also a batch version,set_secrets() that allows you to set multiple secrets at a time.

Get secrets

In your local code, you can use the `get_secret()` method to get the secret value by name.

For runs submitted the `Experiment.submit` , use the `get_secret()` method with the `Run` class. Because a submitted run is aware of its workspace, this method shortcuts the Workspace instantiation and returns the secret value directly.

```
# Code in submitted run
from azureml.core import Experiment, Run

run = Run.get_context()
secret_value = run.get_secret(name="mysecret")
```

Be careful not to expose the secret value by writing or printing it out.

There is also a batch version, get_secrets() for accessing multiple secrets at once.

Next steps

- View example notebook
- Learn about enterprise security with Azure Machine Learning

Build scikit-learn models at scale with Azure Machine Learning

3/9/2020 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, learn how to run your scikit-learn training scripts at enterprise scale by using the Azure Machine Learning SKlearn estimator class.

The example scripts in this article are used to classify iris flower images to build a machine learning model based on scikit-learn's iris dataset.

Whether you're training a machine learning scikit-learn model from the ground-up or you're bringing an existing model into the cloud, you can use Azure Machine Learning to scale out open-source training jobs using elastic cloud compute resources. You can build, deploy, version and monitor production-grade models with Azure Machine Learning.

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
  - Complete the Tutorial: Setup environment and workspace to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
  - In the samples training folder on the notebook server, find a completed and expanded notebook by navigating to this directory: **how-to-use-azureml > ml-frameworks > scikit-learn > training > train-hyperparameter-tune-deploy-with-sklearn** folder.
- Your own Jupyter Notebook server
  - Install the Azure Machine Learning SDK.
  - Create a workspace configuration file.
  - Download the dataset and sample script file
    - iris dataset
    - train_iris.py
  - You can also find a completed Jupyter Notebook version of this guide on the GitHub samples page. The notebook includes an expanded section covering intelligent hyperparameter tuning and retrieving the best model by primary metrics.

Set up the experiment

This section sets up the training experiment by loading the required python packages, initializing a workspace, creating an experiment, and uploading the training data and training scripts.

Import packages

First, import the necessary Python libraries.

```
import os
import urllib
import shutil
import azureml

from azureml.core import Experiment
from azureml.core import Workspace, Run

from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
```

Initialize a workspace

The Azure Machine Learning workspace is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the prerequisites section.

```
ws = Workspace.from_config()
```

Create a machine learning experiment

Create an experiment and a folder to hold your training scripts. In this example, create an experiment called "sklearn-iris".

```
project_folder = './sklearn-iris'
os.makedirs(project_folder, exist_ok=True)

exp = Experiment(workspace=ws, name='sklearn-iris')
```

Prepare training script

In this tutorial, the training script **train_iris.py** is already provided for you. In practice, you should be able to take any custom training script as is and run it with Azure ML without having to modify your code.

To use the Azure ML tracking and metrics capabilities, add a small amount of Azure ML code inside your training script. The training script **train_iris.py** shows how to log some metrics to your Azure ML run using the `Run` object within the script.

The provided training script uses example data from the `iris = datasets.load_iris()` function. For your own data, you may need to use steps such as Upload dataset and scripts to make data available during training.

Copy the training script **train_iris.py** into your project directory.

```
import shutil
shutil.copy('./train_iris.py', project_folder)
```

Create or get a compute target

Create a compute target for your scikit-learn job to run on. Scikit-learn only supports single node, CPU computing.

The following code, creates an Azure Machine Learning managed compute (AmlCompute) for your remote training compute resource. Creation of AmlCompute takes approximately 5 minutes. If the AmlCompute with that name is already in your workspace, this code will skip the creation process.

```
cluster_name = "cpu-cluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                           max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

For more information on compute targets, see the what is a compute target article.

Create a scikit-learn estimator

The scikit-learn estimator provides a simple way of launching a scikit-learn training job on a compute target. It is implemented through the `SKLearn` class, which can be used to support single-node CPU training.

If your training script needs additional pip or conda packages to run, you can have the packages installed on the resulting docker image by passing their names through the `pip_packages` and `conda_packages` arguments.

```
from azureml.train.sklearn import SKLearn

script_params = {
    '--kernel': 'linear',
    '--penalty': 1.0,
}

estimator = SKLearn(source_directory=project_folder,
                    script_params=script_params,
                    compute_target=compute_target,
                    entry_script='train_iris.py'
                    pip_packages=['joblib']
                   )
```

For more information on customizing your Python environment, see Create and manage environments for training and deployment.

Submit a run

The Run object provides the interface to the run history while the job is running and after it has completed.

```
run = experiment.submit(estimator)
run.wait_for_completion(show_output=True)
```

As the run is executed, it goes through the following stages:

- **Preparing**: A docker image is created according to the TensorFlow estimator. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress.
- **Scaling**: The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running**: All scripts in the script folder are uploaded to the compute target, data stores are mounted or

copied, and the entry_script is executed. Outputs from stdout and the ./logs folder are streamed to the run history and can be used to monitor the run.

- **Post-Processing**: The ./outputs folder of the run is copied over to the run history.

Save and register the model

Once you've trained the model, you can save and register it to your workspace. Model registration lets you store and version your models in your workspace to simplify model management and deployment.

Add the following code to your training script, train_iris.py, to save the model.

```
import joblib

joblib.dump(svm_model_linear, 'model.joblib')
```

Register the model to your workspace with the following code. By specifying the parameters `model_framework`, `model_framework_version`, and `resource_configuration`, no-code model deployment becomes available. This allows you to directly deploy your model as a web service from the registered model, and the `ResourceConfiguration` object defines the compute resource for the web service.

```
from azureml.core import Model
from azureml.core.resource_configuration import ResourceConfiguration

model = run.register_model(model_name='sklearn-iris',
                           model_path='outputs/model.joblib',
                           model_framework=Model.Framework.SCIKITLEARN,
                           model_framework_version='0.19.1',
                           resource_configuration=ResourceConfiguration(cpu=1, memory_in_gb=0.5))
```

Deployment

The model you just registered can be deployed the exact same way as any other registered model in Azure Machine Learning, regardless of which estimator you used for training. The deployment how-to contains a section on registering models, but you can skip directly to creating a compute target for deployment, since you already have a registered model.

(Preview) No-code model deployment

Instead of the traditional deployment route, you can also use the no-code deployment feature (preview)for scikit-learn. No-code model deployment is supported for all built-in scikit-learn model types. By registering your model as shown above with the `model_framework`, `model_framework_version`, and `resource_configuration` parameters, you can simply use the `deploy()` static function to deploy your model.

```
web_service = Model.deploy(ws, "scikit-learn-service", [model])
```

NOTE: These dependencies are included in the pre-built scikit-learn inference container.

```
    - azureml-defaults
    - inference-schema[numpy-support]
    - scikit-learn
    - numpy
```

The full how-to covers deployment in Azure Machine Learning in greater depth.

Next steps

In this article, you trained and registered a scikit-learn model, and learned about deployment options. See these other articles to learn more about Azure Machine Learning.

- Track run metrics during training
- Tune hyperparameters
- Reference architecture for distributed deep learning training in Azure

Build a TensorFlow deep learning model at scale with Azure Machine Learning

3/3/2020 • 8 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article shows you how to run your TensorFlow training scripts at scale using Azure Machine Learning's TensorFlow estimator class. This example trains and registers a TensorFlow model to classify handwritten digits using a deep neural network (DNN).

Whether you're developing a TensorFlow model from the ground-up or you're bringing an existing model into the cloud, you can use Azure Machine Learning to scale out open-source training jobs to build, deploy, version, and monitor production-grade models.

Learn more about deep learning vs machine learning.

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
  - Complete the Tutorial: Setup environment and workspace to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
  - In the samples deep learning folder on the notebook server, find a completed and expanded notebook by navigating to this directory: **how-to-use-azureml > ml-frameworks > tensorflow > deployment > train-hyperparameter-tune-deploy-with-tensorflow** folder.
- Your own Jupyter Notebook server
  - Install the Azure Machine Learning SDK.
  - Create a workspace configuration file.
  - Download the sample script files `mnist-tf.py` and `utils.py`

  You can also find a completed Jupyter Notebook version of this guide on the GitHub samples page. The notebook includes expanded sections covering intelligent hyperparameter tuning, model deployment, and notebook widgets.

Set up the experiment

This section sets up the training experiment by loading the required Python packages, initializing a workspace, creating an experiment, and uploading the training data and training scripts.

Import packages

First, import the necessary Python libraries.

```
import os
import urllib
import shutil
import azureml

from azureml.core import Experiment
from azureml.core import Workspace, Run

from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
from azureml.train.dnn import TensorFlow
```

Initialize a workspace

The Azure Machine Learning workspace is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the prerequisites section.

```
ws = Workspace.from_config()
```

Create a deep learning experiment

Create an experiment and a folder to hold your training scripts. In this example, create an experiment called "tf-mnist".

```
script_folder = './tf-mnist'
os.makedirs(script_folder, exist_ok=True)

exp = Experiment(workspace=ws, name='tf-mnist')
```

Create a file dataset

A `FileDataset` object references one or multiple files in your workspace datastore or public urls. The files can be of any format, and the class provides you with the ability to download or mount the files to your compute. By creating a `FileDataset`, you create a reference to the data source location. If you applied any transformations to the data set, they will be stored in the data set as well. The data remains in its existing location, so no extra storage cost is incurred. See the how-to guide on the `Dataset` package for more information.

```
from azureml.core.dataset import Dataset

web_paths = [
            'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
            ]
dataset = Dataset.File.from_files(path=web_paths)
```

Use the `register()` method to register the data set to your workspace so they can be shared with others, reused across various experiments, and referred to by name in your training script.

```
dataset = dataset.register(workspace=ws,
                           name='mnist dataset',
                           description='training and test dataset',
                           create_new_version=True)

# list the files referenced by dataset
dataset.to_path()
```

Create a compute target

Create a compute target for your TensorFlow job to run on. In this example, create a GPU-enabled Azure Machine Learning compute cluster.

```
cluster_name = "gpucluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                           max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

For more information on compute targets, see the what is a compute target article.

Create a TensorFlow estimator

The TensorFlow estimator provides a simple way of launching a TensorFlow training job on a compute target.

The TensorFlow estimator is implemented through the generic `estimator` class, which can be used to support any framework. For more information about training models using the generic estimator, see train models with Azure Machine Learning using estimator

If your training script needs additional pip or conda packages to run, you can have the packages installed on the resulting Docker image by passing their names through the `pip_packages` and `conda_packages` arguments.

```
script_params = {
    '--data-folder': dataset.as_named_input('mnist').as_mount(),
    '--batch-size': 50,
    '--first-layer-neurons': 300,
    '--second-layer-neurons': 100,
    '--learning-rate': 0.01
}

est = TensorFlow(source_directory=script_folder,
                 entry_script='tf_mnist.py',
                 script_params=script_params,
                 compute_target=compute_target,
                 use_gpu=True,
                 pip_packages=['azureml-dataprep[pandas,fuse]'])
```

> **TIP**
> Support for **Tensorflow 2.0** has been added to the Tensorflow estimator class. See the blog post for more information.

For more information on customizing your Python environment, see Create and manage environments for training and deployment.

Submit a run

The Run object provides the interface to the run history while the job is running and after it has completed.

```
run = exp.submit(est)
run.wait_for_completion(show_output=True)
```

As the Run is executed, it goes through the following stages:

- **Preparing**: A Docker image is created according to the TensorFlow estimator. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress.
- **Scaling**: The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running**: All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the entry_script is executed. Outputs from stdout and the ./logs folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing**: The ./outputs folder of the run is copied over to the run history.

Register or download a model

Once you've trained the model, you can register it to your workspace. Model registration lets you store and version your models in your workspace to simplify model management and deployment. By specifying the parameters `model_framework`, `model_framework_version`, and `resource_configuration`, no-code model deployment becomes available. This allows you to directly deploy your model as a web service from the registered model, and the `ResourceConfiguration` object defines the compute resource for the web service.

```
from azureml.core import Model
from azureml.core.resource_configuration import ResourceConfiguration

model = run.register_model(model_name='tf-dnn-mnist',
                           model_path='outputs/model',
                           model_framework=Model.Framework.TENSORFLOW,
                           model_framework_version='1.13.0',
                           resource_configuration=ResourceConfiguration(cpu=1, memory_in_gb=0.5))
```

You can also download a local copy of the model by using the Run object. In the training script `mnist-tf.py`, a TensorFlow saver object persists the model to a local folder (local to the compute target). You can use the Run object to download a copy.

```
# Create a model folder in the current directory
os.makedirs('./model', exist_ok=True)

for f in run.get_file_names():
    if f.startswith('outputs/model'):
        output_file_path = os.path.join('./model', f.split('/')[-1])
        print('Downloading from {} to {} ...'.format(f, output_file_path))
        run.download_file(name=f, output_file_path=output_file_path)
```

Distributed training

The `TensorFlow` estimator also supports distributed training across CPU and GPU clusters. You can easily run distributed TensorFlow jobs and Azure Machine Learning will manage the orchestration for you.

Azure Machine Learning supports two methods of distributed training in TensorFlow:

- MPI-based distributed training using the Horovod framework
- Native distributed TensorFlow using the parameter server method

Horovod

Horovod is an open-source framework for distributed training developed by Uber. It offers an easy path to distributed GPU TensorFlow jobs.

To use Horovod, specify an `MpiConfiguration` object for the `distributed_training` parameter in the TensorFlow constructor. This parameter ensures that Horovod library is installed for you to use in your training script.

```
from azureml.core.runconfig import MpiConfiguration
from azureml.train.dnn import TensorFlow

# Tensorflow constructor
estimator= TensorFlow(source_directory=project_folder,
                      compute_target=compute_target,
                      script_params=script_params,
                      entry_script='script.py',
                      node_count=2,
                      process_count_per_node=1,
                      distributed_training=MpiConfiguration(),
                      framework_version='1.13',
                      use_gpu=True,
                      pip_packages=['azureml-dataprep[pandas,fuse]'])
```

Parameter server

You can also run native distributed TensorFlow, which uses the parameter server model. In this method, you train across a cluster of parameter servers and workers. The workers calculate the gradients during training, while the parameter servers aggregate the gradients.

To use the parameter server method, specify a `TensorflowConfiguration` object for the `distributed_training` parameter in the TensorFlow constructor.

```
from azureml.train.dnn import TensorFlow

distributed_training = TensorflowConfiguration()
distributed_training.worker_count = 2

# Tensorflow constructor
tf_est= TensorFlow(source_directory=project_folder,
                      compute_target=compute_target,
                      script_params=script_params,
                      entry_script='script.py',
                      node_count=2,
                      process_count_per_node=1,
                      distributed_training=distributed_training,
                      use_gpu=True,
                      pip_packages=['azureml-dataprep[pandas,fuse]'])

# submit the TensorFlow job
run = exp.submit(tf_est)
```

Define cluster specifications in 'TF_CONFIG`

You also need the network addresses and ports of the cluster for the `tf.train.ClusterSpec`, so Azure Machine Learning sets the `TF_CONFIG` environment variable for you.

The `TF_CONFIG` environment variable is a JSON string. Here is an example of the variable for a parameter server:

```
TF_CONFIG='{
    "cluster": {
        "ps": ["host0:2222", "host1:2222"],
        "worker": ["host2:2222", "host3:2222", "host4:2222"],
    },
    "task": {"type": "ps", "index": 0},
    "environment": "cloud"
}'
```

For TensorFlow's high level `tf.estimator` API, TensorFlow parses the `TF_CONFIG` variable and builds the cluster spec for you.

For TensorFlow's lower-level core APIs for training, parse the `TF_CONFIG` variable and build the `tf.train.ClusterSpec` in your training code.

```
import os, json
import tensorflow as tf

tf_config = os.environ.get('TF_CONFIG')
if not tf_config or tf_config == "":
    raise ValueError("TF_CONFIG not found.")
tf_config_json = json.loads(tf_config)
cluster_spec = tf.train.ClusterSpec(cluster)
```

Deploy a TensorFlow model

The model you just registered can be deployed the exact same way as any other registered model in Azure Machine Learning, regardless of which estimator you used for training. The deployment how-to contains a section on registering models, but you can skip directly to creating a compute target for deployment, since you already have a registered model.

(Preview) No-code model deployment

Instead of the traditional deployment route, you can also use the no-code deployment feature (preview) for Tensorflow. By registering your model as shown above with the `model_framework`, `model_framework_version`, and `resource_configuration` parameters, you can simply use the `deploy()` static function to deploy your model.

```
service = Model.deploy(ws, "tensorflow-web-service", [model])
```

The full how-to covers deployment in Azure Machine Learning in greater depth.

Next steps

In this article, you trained and registered a TensorFlow model, and learned about options for deployment. See these other articles to learn more about Azure Machine Learning.

- Track run metrics during training
- Tune hyperparameters
- Reference architecture for distributed deep learning training in Azure

Train and register a Keras classification model with Azure Machine Learning

1/8/2020 • 5 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article shows you how to train and register a Keras classification model built on TensorFlow using Azure Machine Learning. It uses the popular MNIST dataset to classify handwritten digits using a deep neural network (DNN) built using the Keras Python library running on top of TensorFlow.

Keras is a high-level neural network API capable of running top of other popular DNN frameworks to simplify development. With Azure Machine Learning, you can rapidly scale out training jobs using elastic cloud compute resources. You can also track your training runs, version models, deploy models, and much more.

Whether you're developing a Keras model from the ground-up or you're bringing an existing model into the cloud, Azure Machine Learning can help you build production-ready models.

See the conceptual article for information on the differences between machine learning and deep learning.

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
  - Complete the Tutorial: Setup environment and workspace to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
  - In the samples folder on the notebook server, find a completed and expanded notebook by navigating to this directory: **how-to-use-azureml > training-with-deep-learning > train-hyperparameter-tune-deploy-with-keras** folder.
- Your own Jupyter Notebook server
  - Install the Azure Machine Learning SDK.
  - Create a workspace configuration file.
  - Download the sample script files `mnist-keras.py` and `utils.py`

  You can also find a completed Jupyter Notebook version of this guide on the GitHub samples page. The notebook includes expanded sections covering intelligent hyperparameter tuning, model deployment, and notebook widgets.

Set up the experiment

This section sets up the training experiment by loading the required python packages, initializing a workspace, creating an experiment, and uploading the training data and training scripts.

Import packages

First, import the necessary Python libraries.

```
import os
import azureml
from azureml.core import Experiment
from azureml.core import Workspace, Run
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
```

Initialize a workspace

The Azure Machine Learning workspace is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the prerequisites section.

```
ws = Workspace.from_config()
```

Create an experiment

Create an experiment called "keras-mnist" in your workspace.

```
exp = Experiment(workspace=ws, name='keras-mnist')
```

Create a file dataset

A `FileDataset` object references one or multiple files in your workspace datastore or public urls. The files can be of any format, and the class provides you with the ability to download or mount the files to your compute. By creating a `FileDataset`, you create a reference to the data source location. If you applied any transformations to the data set, they will be stored in the data set as well. The data remains in its existing location, so no extra storage cost is incurred. See the how-to guide on the `Dataset` package for more information.

```
from azureml.core.dataset import Dataset

web_paths = [
            'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
            ]
dataset = Dataset.File.from_files(path=web_paths)
```

Use the `register()` method to register the data set to your workspace so they can be shared with others, reused across various experiments, and referred to by name in your training script.

```
dataset = dataset.register(workspace=ws,
                           name='mnist dataset',
                           description='training and test dataset',
                           create_new_version=True)
```

Create a compute target

Create a compute target for your TensorFlow job to run on. In this example, create a GPU-enabled Azure Machine Learning compute cluster.

```
cluster_name = "gpucluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                           max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

For more information on compute targets, see the what is a compute target article.

Create a TensorFlow estimator and import Keras

The TensorFlow estimator provides a simple way of launching TensorFlow training jobs on compute target. Since Keras runs on top of TensorFlow, you can use the TensorFlow estimator and import the Keras library using the `pip_packages` argument.

First get the data from the workspace datastore using the `Dataset` class.

```
dataset = Dataset.get_by_name(ws, 'mnist dataset')

# list the files referenced by mnist dataset
dataset.to_path()
```

The TensorFlow estimator is implemented through the generic `estimator` class, which can be used to support any framework. Additionally, create a dictionary `script_params` that contains the DNN hyperparameter settings. For more information about training models using the generic estimator, see train models with Azure Machine Learning using estimator

```
from azureml.train.dnn import TensorFlow

script_params = {
    '--data-folder': dataset.as_named_input('mnist').as_mount(),
    '--batch-size': 50,
    '--first-layer-neurons': 300,
    '--second-layer-neurons': 100,
    '--learning-rate': 0.001
}

est = TensorFlow(source_directory=script_folder,
                 entry_script='keras_mnist.py',
                 script_params=script_params,
                 compute_target=compute_target,
                 pip_packages=['keras', 'matplotlib'],
                 use_gpu=True)
```

Submit a run

The Run object provides the interface to the run history while the job is running and after it has completed.

```
run = exp.submit(est)
run.wait_for_completion(show_output=True)
```

As the Run is executed, it goes through the following stages:

- **Preparing**: A docker image is created according to the TensorFlow estimator. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress.
- **Scaling**: The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running**: All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the entry_script is executed. Outputs from stdout and the ./logs folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing**: The ./outputs folder of the run is copied over to the run history.

Register the model

Once you've trained the DNN model, you can register it to your workspace. Model registration lets you store and version your models in your workspace to simplify model management and deployment.

```
model = run.register_model(model_name='keras-dnn-mnist', model_path='outputs/model')
```

> **TIP**
> The model you just registered is deployed the exact same way as any other registered model in Azure Machine Learning, regardless of which estimator you used for training. The deployment how-to contains a section on registering models, but you can skip directly to creating a compute target for deployment, since you already have a registered model.

You can also download a local copy of the model. This can be useful for doing additional model validation work locally. In the training script, `mnist-keras.py`, a TensorFlow saver object persists the model to a local folder (local to the compute target). You can use the Run object to download a copy from datastore.

```
# Create a model folder in the current directory
os.makedirs('./model', exist_ok=True)

for f in run.get_file_names():
    if f.startswith('outputs/model'):
        output_file_path = os.path.join('./model', f.split('/')[-1])
        print('Downloading from {} to {} ...'.format(f, output_file_path))
        run.download_file(name=f, output_file_path=output_file_path)
```

Next steps

In this article, you trained and registered a Keras model on Azure Machine Learning. To learn how to deploy a model, continue on to our model deployment article.

How and where to deploy models

- Track run metrics during training
- Tune hyperparameters
- Deploy a trained model
- Reference architecture for distributed deep learning training in Azure

Train Pytorch deep learning models at scale with Azure Machine Learning

1/9/2020 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, learn how to run your PyTorch training scripts at enterprise scale using Azure Machine Learning's PyTorch estimator class.

The example scripts in this article are used to classify chicken and turkey images to build a deep learning neural network based on PyTorch's transfer learning tutorial.

Whether you're training a deep learning PyTorch model from the ground-up or you're bringing an existing model into the cloud, you can use Azure Machine Learning to scale out open-source training jobs using elastic cloud compute resources. You can build, deploy, version, and monitor production-grade models with Azure Machine Learning.

Learn more about deep learning vs machine learning.

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
  - Complete the Tutorial: Setup environment and workspace to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
  - In the samples deep learning folder on the notebook server, find a completed and expanded notebook by navigating to this directory: **how-to-use-azureml > training-with-deep-learning > train-hyperparameter-tune-deploy-with-pytorch** folder.
- Your own Jupyter Notebook server
  - Install the Azure Machine Learning SDK.
  - Create a workspace configuration file.
  - Download the sample script files `pytorch_train.py`

  You can also find a completed Jupyter Notebook version of this guide on the GitHub samples page. The notebook includes expanded sections covering intelligent hyperparameter tuning, model deployment, and notebook widgets.

Set up the experiment

This section sets up the training experiment by loading the required python packages, initializing a workspace, creating an experiment, and uploading the training data and training scripts.

Import packages

First, import the necessary Python libraries.

```
import os
import shutil

from azureml.core.workspace import Workspace
from azureml.core import Experiment

from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
from azureml.train.dnn import PyTorch
```

Initialize a workspace

The Azure Machine Learning workspace is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the prerequisites section.

```
ws = Workspace.from_config()
```

Create a deep learning experiment

Create an experiment and a folder to hold your training scripts. In this example, create an experiment called "pytorch-birds".

```
project_folder = './pytorch-birds'
os.makedirs(project_folder, exist_ok=True)

experiment_name = 'pytorch-birds'
experiment = Experiment(ws, name=experiment_name)
```

Get the data

The dataset consists of about 120 training images each for turkeys and chickens, with 100 validation images for each class. We will download and extract the dataset as part of our training script `pytorch_train.py`. The images are a subset of the Open Images v5 Dataset.

Prepare training scripts

In this tutorial, the training script, `pytorch_train.py`, is already provided. In practice, you can take any custom training script, as is, and run it with Azure Machine Learning.

Upload the Pytorch training script, `pytorch_train.py`.

```
shutil.copy('pytorch_train.py', project_folder)
```

However, if you would like to use Azure Machine Learning tracking and metrics capabilities, you will have to add a small amount code inside your training script. Examples of metrics tracking can be found in `pytorch_train.py`.

Create a compute target

Create a compute target for your PyTorch job to run on. In this example, create a GPU-enabled Azure Machine Learning compute cluster.

```
cluster_name = "gpucluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                           max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

For more information on compute targets, see the what is a compute target article.

Create a PyTorch estimator

The PyTorch estimator provides a simple way of launching a PyTorch training job on a compute target.

The PyTorch estimator is implemented through the generic `estimator` class, which can be used to support any framework. For more information about training models using the generic estimator, see train models with Azure Machine Learning using estimator

If your training script needs additional pip or conda packages to run, you can have the packages installed on the resulting docker image by passing their names through the `pip_packages` and `conda_packages` arguments.

```
script_params = {
    '--num_epochs': 30,
    '--output_dir': './outputs'
}

estimator = PyTorch(source_directory=project_folder,
                    script_params=script_params,
                    compute_target=compute_target,
                    entry_script='pytorch_train.py',
                    use_gpu=True,
                    pip_packages=['pillow==5.4.1'])
```

For more information on customizing your Python environment, see Create and manage environments for training and deployment.

Submit a run

The Run object provides the interface to the run history while the job is running and after it has completed.

```
run = experiment.submit(estimator)
run.wait_for_completion(show_output=True)
```

As the Run is executed, it goes through the following stages:

- **Preparing**: A docker image is created according to the PyTorch estimator. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress.
- **Scaling**: The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.

- **Running**: All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the entry_script is executed. Outputs from stdout and the ./logs folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing**: The ./outputs folder of the run is copied over to the run history.

Register or download a model

Once you've trained the model, you can register it to your workspace. Model registration lets you store and version your models in your workspace to simplify model management and deployment.

```
model = run.register_model(model_name='pt-dnn', model_path='outputs/')
```

> **TIP**
> The model you just registered is deployed the exact same way as any other registered model in Azure Machine Learning, regardless of which estimator you used for training. The deployment how-to contains a section on registering models, but you can skip directly to creating a compute target for deployment, since you already have a registered model.

You can also download a local copy of the model by using the Run object. In the training script `pytorch_train.py`, a PyTorch save object persists the model to a local folder (local to the compute target). You can use the Run object to download a copy.

```
# Create a model folder in the current directory
os.makedirs('./model', exist_ok=True)

for f in run.get_file_names():
    if f.startswith('outputs/model'):
        output_file_path = os.path.join('./model', f.split('/')[-1])
        print('Downloading from {} to {} ...'.format(f, output_file_path))
        run.download_file(name=f, output_file_path=output_file_path)
```

Distributed training

The `PyTorch` estimator also supports distributed training across CPU and GPU clusters. You can easily run distributed PyTorch jobs and Azure Machine Learning will manage the orchestration for you.

Horovod

Horovod is an open-source, all reduce framework for distributed training developed by Uber. It offers an easy path to distributed GPU PyTorch jobs.

To use Horovod, specify an `MpiConfiguration` object for the `distributed_training` parameter in the PyTorch constructor. This parameter ensures that Horovod library is installed for you to use in your training script.

```
from azureml.train.dnn import PyTorch

estimator= PyTorch(source_directory=project_folder,
                   compute_target=compute_target,
                   script_params=script_params,
                   entry_script='script.py',
                   node_count=2,
                   process_count_per_node=1,
                   distributed_training=MpiConfiguration(),
                   framework_version='1.13',
                   use_gpu=True)
```

Horovod and its dependencies will be installed for you, so you can import it in your training script `train.py` as follows:

```
import torch
import horovod
```

Export to ONNX

To optimize inference with the ONNX Runtime, convert your trained PyTorch model to the ONNX format. Inference, or model scoring, is the phase where the deployed model is used for prediction, most commonly on production data. See the tutorial for an example.

Next steps

In this article, you trained and registered a deep learning, neural network using PyTorch on Azure Machine Learning. To learn how to deploy a model, continue on to our model deployment article.

How and where to deploy models

- Track run metrics during training
- Tune hyperparameters
- Deploy a trained model
- Reference architecture for distributed deep learning training in Azure

Use the interpretability package to explain ML models & predictions in Python

4/21/2020 • 11 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this how-to guide, you learn to use the interpretability package of the Azure Machine Learning Python SDK to perform the following tasks:

- Explain the entire model behavior or individual predictions on your personal machine locally.
- Enable interpretability techniques for engineered features.
- Explain the behavior for the entire model and individual predictions in Azure.
- Use a visualization dashboard to interact with your model explanations.
- Deploy a scoring explainer alongside your model to observe explanations during inferencing.

For more information on the supported interpretability techniques and machine learning models, see Model interpretability in Azure Machine Learning and sample notebooks.

Generate feature importance value on your personal machine

The following example shows how to use the interpretability package on your personal machine without contacting Azure services.

1. Install `azureml-interpret` and `azureml-contrib-interpret` packages.

```
pip install azureml-interpret
pip install azureml-contrib-interpret
```

2. Train a sample model in a local Jupyter notebook.

```
# load breast cancer dataset, a well-known small dataset that comes with scikit-learn
from sklearn.datasets import load_breast_cancer
from sklearn import svm
from sklearn.model_selection import train_test_split
breast_cancer_data = load_breast_cancer()
classes = breast_cancer_data.target_names.tolist()

# split data into train and test
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(breast_cancer_data.data,
                                                    breast_cancer_data.target,
                                                    test_size=0.2,
                                                    random_state=0)
clf = svm.SVC(gamma=0.001, C=100., probability=True)
model = clf.fit(x_train, y_train)
```

3. Call the explainer locally.

   - To initialize an explainer object, pass your model and some training data to the explainer's constructor.
   - To make your explanations and visualizations more informative, you can choose to pass in feature names

and output class names if doing classification.

The following code blocks show how to instantiate an explainer object with `TabularExplainer`, `MimicExplainer`, and `PFIExplainer` locally.

- `TabularExplainer` calls one of the three SHAP explainers underneath (`TreeExplainer`, `DeepExplainer`, or `KernelExplainer`).
- `TabularExplainer` automatically selects the most appropriate one for your use case, but you can call each of its three underlying explainers directly.

```
from interpret.ext.blackbox import TabularExplainer

# "features" and "classes" fields are optional
explainer = TabularExplainer(model,
                             x_train,
                             features=breast_cancer_data.feature_names,
                             classes=classes)
```

or

```
from interpret.ext.blackbox import MimicExplainer

# you can use one of the following four interpretable models as a global surrogate to the black box
model

from interpret.ext.glassbox import LGBMExplainableModel
from interpret.ext.glassbox import LinearExplainableModel
from interpret.ext.glassbox import SGDExplainableModel
from interpret.ext.glassbox import DecisionTreeExplainableModel

# "features" and "classes" fields are optional
# augment_data is optional and if true, oversamples the initialization examples to improve surrogate
model accuracy to fit original model.  Useful for high-dimensional data where the number of rows is less
than the number of columns.
# max_num_of_augmentations is optional and defines max number of times we can increase the input data
size.
# LGBMExplainableModel can be replaced with LinearExplainableModel, SGDExplainableModel, or
DecisionTreeExplainableModel
explainer = MimicExplainer(model,
                           x_train,
                           LGBMExplainableModel,
                           augment_data=True,
                           max_num_of_augmentations=10,
                           features=breast_cancer_data.feature_names,
                           classes=classes)
```

or

```
from interpret.ext.blackbox import PFIExplainer

# "features" and "classes" fields are optional
explainer = PFIExplainer(model,
                         features=breast_cancer_data.feature_names,
                         classes=classes)
```

Explain the entire model behavior (global explanation)

Refer to the following example to help you get the aggregate (global) feature importance values.

```
# you can use the training data or the test data here
global_explanation = explainer.explain_global(x_train)

# if you used the PFIExplainer in the previous step, use the next line of code instead
# global_explanation = explainer.explain_global(x_train, true_labels=y_test)

# sorted feature importance values and feature names
sorted_global_importance_values = global_explanation.get_ranked_global_values()
sorted_global_importance_names = global_explanation.get_ranked_global_names()
dict(zip(sorted_global_importance_names, sorted_global_importance_values))

# alternatively, you can print out a dictionary that holds the top K feature names and values
global_explanation.get_feature_importance_dict()
```

Explain an individual prediction (local explanation)

Get the individual feature importance values of different datapoints by calling explanations for an individual instance or a group of instances.

> **NOTE**
>
> `PFIExplainer` does not support local explanations.

```
# get explanation for the first data point in the test set
local_explanation = explainer.explain_local(x_test[0:5])

# sorted feature importance values and feature names
sorted_local_importance_names = local_explanation.get_ranked_local_names()
sorted_local_importance_values = local_explanation.get_ranked_local_values()
```

Raw feature transformations

You can opt to get explanations in terms of raw, untransformed features rather than engineered features. For this option, you pass your feature transformation pipeline to the explainer in `train_explain.py`. Otherwise, the explainer provides explanations in terms of engineered features.

The format of supported transformations is the same as described in sklearn-pandas. In general, any transformations are supported as long as they operate on a single column so that it's clear they're one-to-many.

Get an explanation for raw features by using a `sklearn.compose.ColumnTransformer` or with a list of fitted transformer tuples. The following example uses `sklearn.compose.ColumnTransformer`.

```
from sklearn.compose import ColumnTransformer

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# append classifier to preprocessing pipeline.
# now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])


# clf.steps[-1][1] returns the trained classification model
# pass transformation as an input to create the explanation object
# "features" and "classes" fields are optional
tabular_explainer = TabularExplainer(clf.steps[-1][1],
                                     initialization_examples=x_train,
                                     features=dataset_feature_names,
                                     classes=dataset_classes,
                                     transformations=preprocessor)
```

In case you want to run the example with the list of fitted transformer tuples, use the following code:

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn_pandas import DataFrameMapper

# assume that we have created two arrays, numerical and categorical, which holds the numerical and categorical
feature names

numeric_transformations = [([f], Pipeline(steps=[('imputer', SimpleImputer(
    strategy='median')), ('scaler', StandardScaler())])) for f in numerical]

categorical_transformations = [([f], OneHotEncoder(
    handle_unknown='ignore', sparse=False)) for f in categorical]

transformations = numeric_transformations + categorical_transformations

# append model to preprocessing pipeline.
# now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', DataFrameMapper(transformations)),
                      ('classifier', LogisticRegression(solver='lbfgs'))])

# clf.steps[-1][1] returns the trained classification model
# pass transformation as an input to create the explanation object
# "features" and "classes" fields are optional
tabular_explainer = TabularExplainer(clf.steps[-1][1],
                                     initialization_examples=x_train,
                                     features=dataset_feature_names,
                                     classes=dataset_classes,
                                     transformations=transformations)
```

Generate feature importance values via remote runs

The following example shows how you can use the `ExplanationClient` class to enable model interpretability for remote runs. It is conceptually similar to the local process, except you:

- Use the `ExplanationClient` in the remote run to upload the interpretability context.
- Download the context later in a local environment.

1. Install `azureml-interpret` and `azureml-interpret-contrib` packages.

   ```
   pip install azureml-interpret
   pip install azureml-interpret-contrib
   ```

2. Create a training script in a local Jupyter notebook. For example, `train_explain.py`.

   ```
   from azureml.contrib.interpret.explanation.explanation_client import ExplanationClient
   from azureml.core.run import Run
   from interpret.ext.blackbox import TabularExplainer

   run = Run.get_context()
   client = ExplanationClient.from_run(run)

   # write code to get and split your data into train and test sets here
   # write code to train your model here

   # explain predictions on your local machine
   # "features" and "classes" fields are optional
   explainer = TabularExplainer(model,
                                x_train,
                                features=feature_names,
                                classes=classes)

   # explain overall model predictions (global explanation)
   global_explanation = explainer.explain_global(x_test)

   # uploading global model explanation data for storage or visualization in webUX
   # the explanation can then be downloaded on any compute
   # multiple explanations can be uploaded
   client.upload_model_explanation(global_explanation, comment='global explanation: all features')
   # or you can only upload the explanation object with the top k feature info
   #client.upload_model_explanation(global_explanation, top_k=2, comment='global explanation: Only top 2
   features')
   ```

3. Set up an Azure Machine Learning Compute as your compute target and submit your training run. See setting up compute targets for model training for instructions. You might also find the example notebooks helpful.

4. Download the explanation in your local Jupyter notebook.

```
from azureml.contrib.interpret.explanation.explanation_client import ExplanationClient

client = ExplanationClient.from_run(run)

# get model explanation data
explanation = client.download_model_explanation()
# or only get the top k (e.g., 4) most important features with their importance values
explanation = client.download_model_explanation(top_k=4)

global_importance_values = explanation.get_ranked_global_values()
global_importance_names = explanation.get_ranked_global_names()
print('global importance values: {}'.format(global_importance_values))
print('global importance names: {}'.format(global_importance_names))
```

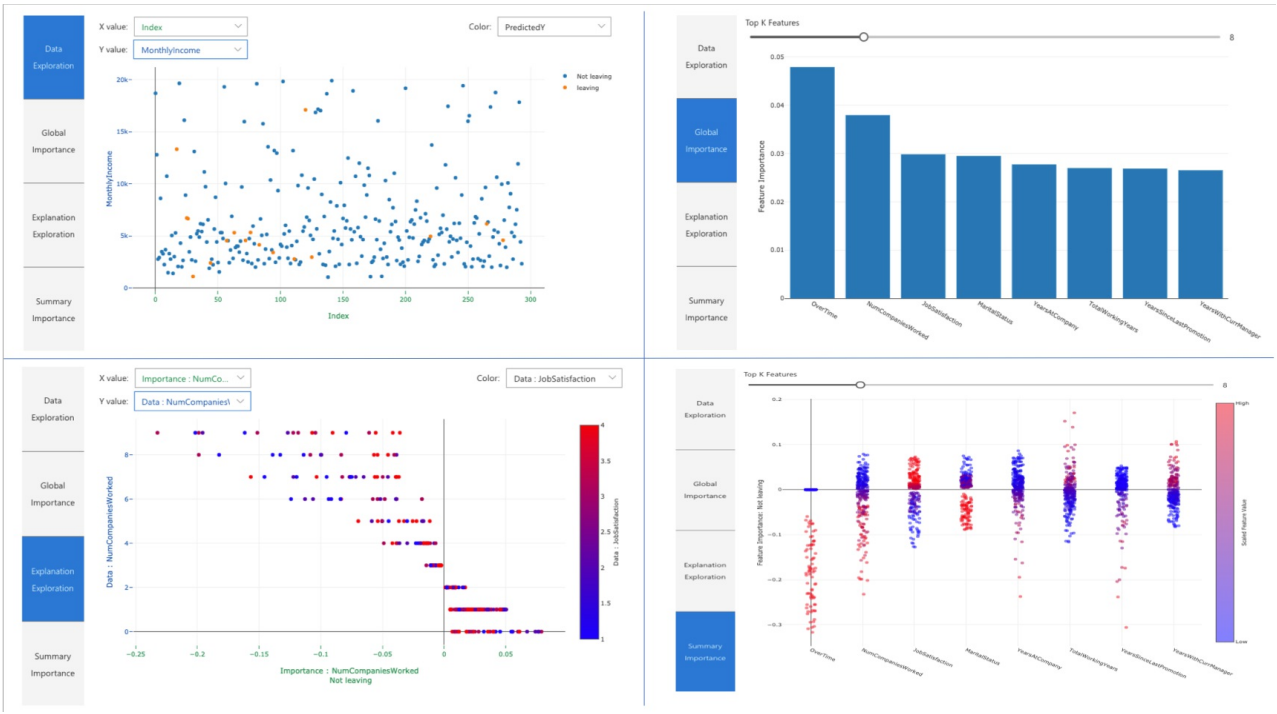
Visualizations

After you download the explanations in your local Jupyter notebook, you can use the visualization dashboard to understand and interpret your model.

Understand entire model behavior (global explanation)

The following plots provide an overall view of the trained model along with its predictions and explanations.

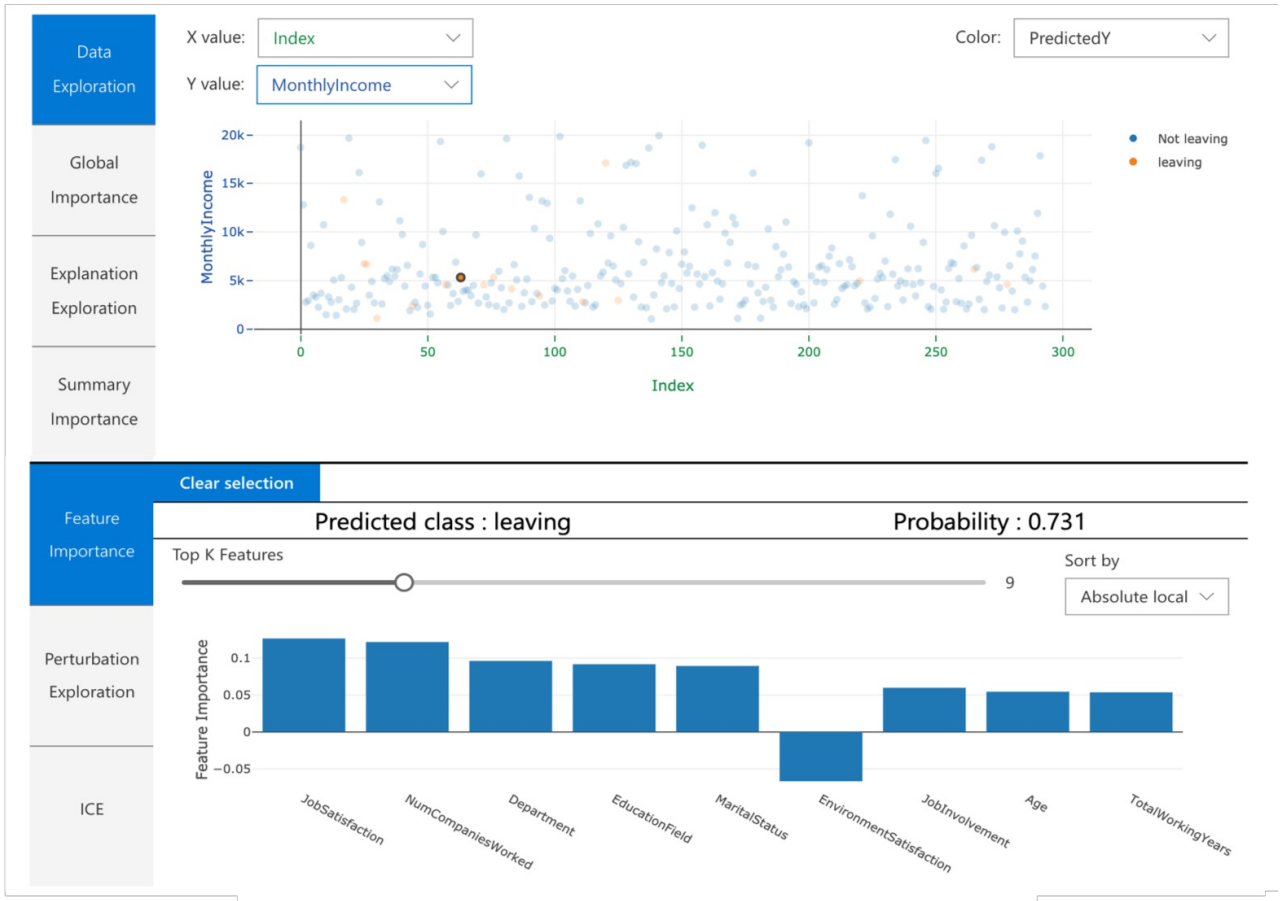
PLOT	DESCRIPTION
Data Exploration	Displays an overview of the dataset along with prediction values.
Global Importance	Aggregates feature importance values of individual datapoints to show the model's overall top K (configurable K) important features. Helps understanding of underlying model's overall behavior.
Explanation Exploration	Demonstrates how a feature affects a change in model's prediction values, or the probability of prediction values. Shows impact of feature interaction.
Summary Importance	Uses individual feature importance values across all data points to show the distribution of each feature's impact on the prediction value. Using this diagram, you investigate in what direction the feature values affects the prediction values.

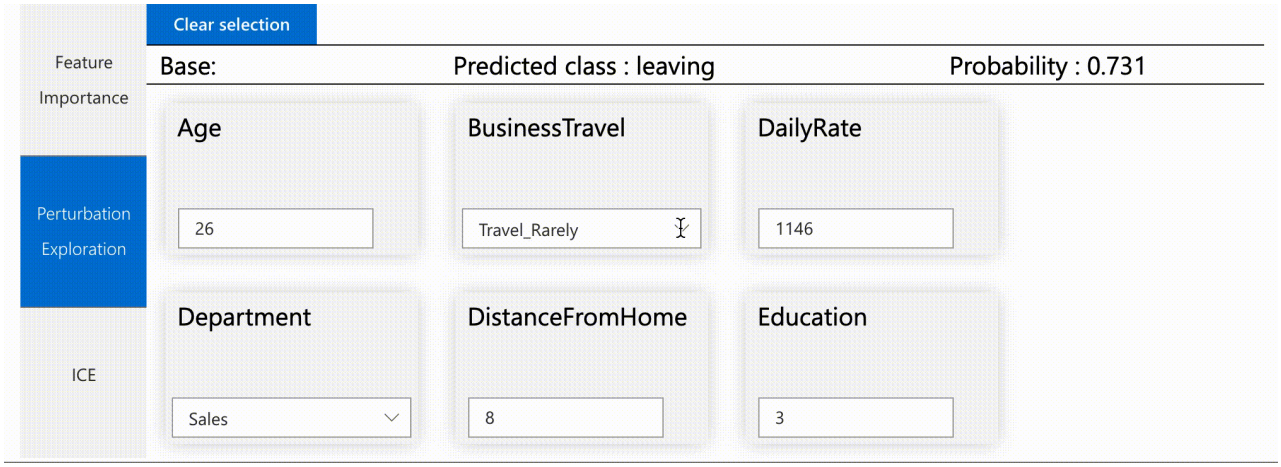


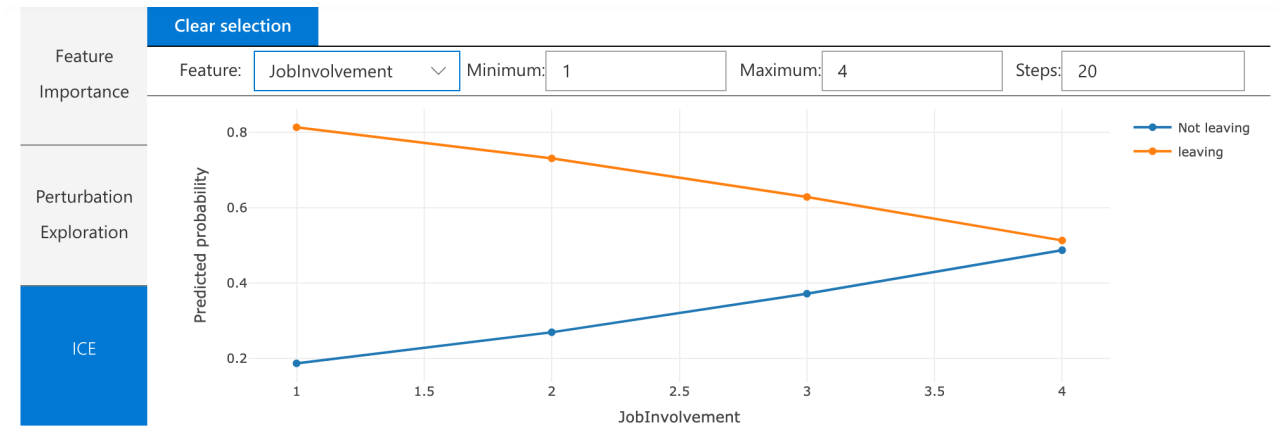
Understand individual predictions (local explanation)

You can load the individual feature importance plot for any data point by clicking on any of the individual data points in any of the overall plots.

PLOT	DESCRIPTION
Local Importance	Shows the top K (configurable K) important features for an individual prediction. Helps illustrate the local behavior of the underlying model on a specific data point.
Perturbation Exploration (what if analysis)	Allows changes to feature values of the selected data point and observe resulting changes to prediction value.
Individual Conditional Expectation (ICE)	Allows feature value changes from a minimum value to a maximum value. Helps illustrate how the data point's prediction changes when a feature changes.







NOTE

Before the Jupyter kernel starts, make sure you enable widget extensions for the visualization dashboard.

- Jupyter notebooks

```
jupyter nbextension install --py --sys-prefix azureml.contrib.interpret.visualize
jupyter nbextension enable --py --sys-prefix azureml.contrib.interpret.visualize
```

- JupyterLab

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
jupyter labextension install microsoft-mli-widget
```

To load the visualization dashboard, use the following code.

```
from interpret_community.widget import ExplanationDashboard

ExplanationDashboard(global_explanation, model, x_test)
```

Visualization in Azure Machine Learning studio

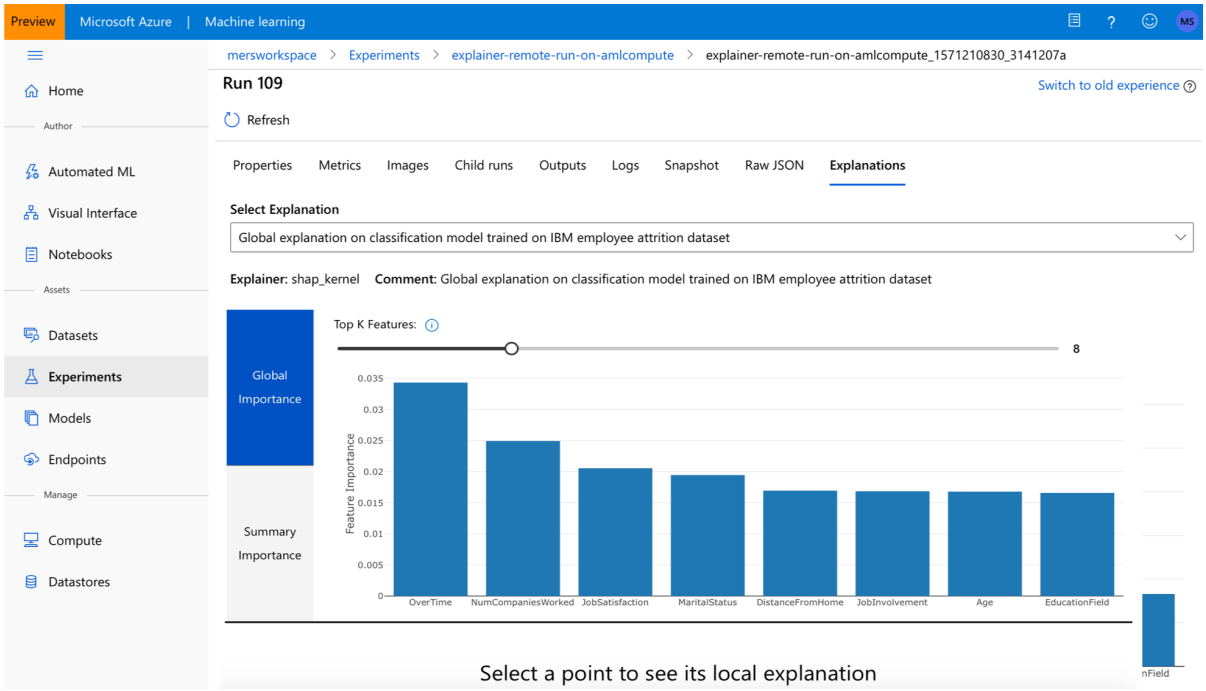
If you complete the remote interpretability steps (uploading generated explanation to Azure Machine Learning Run History), you can view the visualization dashboard in Azure Machine Learning studio. This dashboard is a simpler version of the visualization dashboard explained above (explanation exploration and ICE plots are disabled as there is no active compute in studio that can perform their real time computations).

If the dataset, global, and local explanations are available, data populates all of the tabs (except Perturbation Exploration and ICE). If only a global explanation is available, the Summary Importance tab and all local explanation tabs are disabled.

Follow one of these paths to access the visualization dashboard in Azure Machine Learning studio:

- **Experiments** pane (Preview)

  1. Select **Experiments** in the left pane to see a list of experiments that you've run on Azure Machine Learning.
  2. Select a particular experiment to view all the runs in that experiment.
  3. Select a run, and then the **Explanations** tab to the explanation visualization dashboard.



- **Models** pane

1. If you registered your original model by following the steps in Deploy models with Azure Machine Learning, you can select **Models** in the left pane to view it.
2. Select a model, and then the **Explanations** tab to view the explanation visualization dashboard.

Interpretability at inference time

You can deploy the explainer along with the original model and use it at inference time to provide the individual feature importance values (local explanation) for new any new datapoint. We also offer lighter-weight scoring explainers to improve interpretability performance at inference time. The process of deploying a lighter-weight scoring explainer is similar to deploying a model and includes the following steps:

1. Create an explanation object. For example, you can use `TabularExplainer`:

```
 from interpret.ext.blackbox import TabularExplainer


explainer = TabularExplainer(model,
                             initialization_examples=x_train,
                             features=dataset_feature_names,
                             classes=dataset_classes,
                             transformations=transformations)
```

2. Create a scoring explainer with the explanation object.

```
from azureml.interpret.scoring.scoring_explainer import KernelScoringExplainer, save

# create a lightweight explainer at scoring time
scoring_explainer = KernelScoringExplainer(explainer)

# pickle scoring explainer
# pickle scoring explainer locally
OUTPUT_DIR = 'my_directory'
save(scoring_explainer, directory=OUTPUT_DIR, exist_ok=True)
```

3. Configure and register an image that uses the scoring explainer model.

```
# register explainer model using the path from ScoringExplainer.save - could be done on remote compute
# scoring_explainer.pkl is the filename on disk, while my_scoring_explainer.pkl will be the filename in
cloud storage
run.upload_file('my_scoring_explainer.pkl', os.path.join(OUTPUT_DIR, 'scoring_explainer.pkl'))

scoring_explainer_model = run.register_model(model_name='my_scoring_explainer',
                                             model_path='my_scoring_explainer.pkl')
print(scoring_explainer_model.name, scoring_explainer_model.id, scoring_explainer_model.version, sep =
'\t')
```

4. As an optional step, you can retrieve the scoring explainer from cloud and test the explanations.

```
from azureml.interpret.scoring.scoring_explainer import load

# retrieve the scoring explainer model from cloud"
scoring_explainer_model = Model(ws, 'my_scoring_explainer')
scoring_explainer_model_path = scoring_explainer_model.download(target_dir=os.getcwd(), exist_ok=True)

# load scoring explainer from disk
scoring_explainer = load(scoring_explainer_model_path)

# test scoring explainer locally
preds = scoring_explainer.explain(x_test)
print(preds)
```

5. Deploy the image to a compute target, by following these steps:

   a. If needed, register your original prediction model by following the steps in Deploy models with Azure Machine Learning.

   b. Create a scoring file.

```
%%writefile score.py
import json
import numpy as np
import pandas as pd
import os
import pickle
from sklearn.externals import joblib
from sklearn.linear_model import LogisticRegression
from azureml.core.model import Model

def init():

   global original_model
   global scoring_model

   # retrieve the path to the model file using the model name
   # assume original model is named original_prediction_model
   original_model_path = Model.get_model_path('original_prediction_model')
   scoring_explainer_path = Model.get_model_path('my_scoring_explainer')

   original_model = joblib.load(original_model_path)
   scoring_explainer = joblib.load(scoring_explainer_path)

def run(raw_data):
   # get predictions and explanations for each data point
   data = pd.read_json(raw_data)
   # make prediction
   predictions = original_model.predict(data)
   # retrieve model explanations
   local_importance_values = scoring_explainer.explain(data)
   # you can return any data type as long as it is JSON-serializable
   return {'predictions': predictions.tolist(), 'local_importance_values':
local_importance_values}
```

   c. Define the deployment configuration.

      This configuration depends on the requirements of your model. The following example defines a configuration that uses one CPU core and one GB of memory.

```
from azureml.core.webservice import AciWebservice

 aciconfig = AciWebservice.deploy_configuration(cpu_cores=1,
                                                memory_gb=1,
                                                tags={"data": "NAME_OF_THE_DATASET",
                                                      "method" : "local_explanation"},
                                                description='Get local explanations for
NAME_OF_THE_PROBLEM')
```

d. Create a file with environment dependencies.

```
from azureml.core.conda_dependencies import CondaDependencies

# WARNING: to install this, g++ needs to be available on the Docker image and is not by default
(look at the next cell)

azureml_pip_packages = ['azureml-defaults', 'azureml-contrib-interpret', 'azureml-core',
'azureml-telemetry', 'azureml-interpret']


# specify CondaDependencies obj
myenv = CondaDependencies.create(conda_packages=['scikit-learn', 'pandas'],
                                 pip_packages=['sklearn-pandas'] + azureml_pip_packages,
                                 pin_sdk_version=False)


with open("myenv.yml","w") as f:
   f.write(myenv.serialize_to_string())

with open("myenv.yml","r") as f:
   print(f.read())
```

e. Create a custom dockerfile with g++ installed.

```
%%writefile dockerfile
RUN apt-get update && apt-get install -y g++
```

f. Deploy the created image.

This process takes approximately five minutes.

```
from azureml.core.webservice import Webservice
from azureml.core.image import ContainerImage

# use the custom scoring, docker, and conda files we created above
image_config = ContainerImage.image_configuration(execution_script="score.py",
                                                  docker_file="dockerfile",
                                                  runtime="python",
                                                  conda_file="myenv.yml")

# use configs and models generated above
service = Webservice.deploy_from_model(workspace=ws,
                                       name='model-scoring-service',
                                       deployment_config=aciconfig,
                                       models=[scoring_explainer_model, original_model],
                                       image_config=image_config)

service.wait_for_deployment(show_output=True)
```

6. Test the deployment.

```
import requests

# create data to test service with
examples = x_list[:4]
input_data = examples.to_json()

headers = {'Content-Type':'application/json'}

# send request to service
resp = requests.post(service.scoring_uri, input_data, headers=headers)

print("POST to url", service.scoring_uri)
# can covert back to Python objects from json string if desired
print("prediction:", resp.text)
```

7. Clean up.

   To delete a deployed web service, use `service.delete()`.

Next steps

Learn more about model interpretability

Check out Azure Machine Learning Interpretability sample notebooks

Interpretability: model explanations in automated machine learning

4/22/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to get explanations for automated machine learning (ML) in Azure Machine Learning. Automated ML helps you understand engineered feature importance.

All SDK versions after 1.0.85 set `model_explainability=True` by default. In SDK version 1.0.85 and earlier versions users need to set `model_explainability=True` in the `AutoMLConfig` object in order to use model interpretability.

In this article, you learn how to:

- Perform interpretability during training for best model or any model.
- Enable visualizations to help you see patterns in data and explanations.
- Implement interpretability during inference or scoring.

Prerequisites

- Interpretability features. Run `pip install azureml-interpret azureml-contrib-interpret` to get the necessary packages.
- Knowledge of building automated ML experiments. For more information on how to use the Azure Machine Learning SDK, complete this regression model tutorial or see how to configure automated ML experiments.

Interpretability during training for the best model

Retrieve the explanation from the `best_run`, which includes explanations for engineered features.

Download engineered feature importance from artifact store

You can use `ExplanationClient` to download the engineered feature explanations from the artifact store of the `best_run`.

```
from azureml.explain.model._internal.explanation_client import ExplanationClient

client = ExplanationClient.from_run(best_run)
engineered_explanations = client.download_model_explanation(raw=False)
print(engineered_explanations.get_feature_importance_dict())
```

Interpretability during training for any model

When you compute model explanations and visualize them, you're not limited to an existing model explanation for an automated ML model. You can also get an explanation for your model with different test data. The steps in this section show you how to compute and visualize engineered feature importance based on your test data.

Retrieve any other AutoML model from training

```
automl_run, fitted_model = local_run.get_output(metric='accuracy')
```

Set up the model explanations

Use `automl_setup_model_explanations` to get the engineered explanations. The `fitted_model` can generate the following items:

- Featured data from trained or test samples
- Engineered feature name lists
- Findable classes in your labeled column in classification scenarios

The `automl_explainer_setup_obj` contains all the structures from above list.

```
from azureml.train.automl.runtime.automl_explain_utilities import automl_setup_model_explanations

automl_explainer_setup_obj = automl_setup_model_explanations(fitted_model, X=X_train,
                                                             X_test=X_test, y=y_train,
                                                             task='classification')
```

Initialize the Mimic Explainer for feature importance

To generate an explanation for AutoML models, use the `MimicWrapper` class. You can initialize the MimicWrapper with these parameters:

- The explainer setup object
- Your workspace
- A LightGBM model, which acts as a surrogate to the `fitted_model` automated ML model

The MimicWrapper also takes the `automl_run` object where the engineered explanations will be uploaded.

```
from azureml.explain.model.mimic.models.lightgbm_model import LGBMExplainableModel
from azureml.explain.model.mimic_wrapper import MimicWrapper

# Initialize the Mimic Explainer
explainer = MimicWrapper(ws, automl_explainer_setup_obj.automl_estimator, LGBMExplainableModel,
                         init_dataset=automl_explainer_setup_obj.X_transform, run=automl_run,
                         features=automl_explainer_setup_obj.engineered_feature_names,
                         feature_maps=[automl_explainer_setup_obj.feature_map],
                         classes=automl_explainer_setup_obj.classes)
```

Use MimicExplainer for computing and visualizing engineered feature importance

You can call the `explain()` method in MimicWrapper with the transformed test samples to get the feature importance for the generated engineered features. You can also use `ExplanationDashboard` to view the dashboard visualization of the feature importance values of the generated engineered features by automated ML featurizers.

```
engineered_explanations = explainer.explain(['local', 'global'],
eval_dataset=automl_explainer_setup_obj.X_test_transform)
print(engineered_explanations.get_feature_importance_dict())
```

Interpretability during inference

In this section, you learn how to operationalize an automated ML model with the explainer that was used to compute the explanations in the previous section.

Register the model and the scoring explainer

Use the `TreeScoringExplainer` to create the scoring explainer that'll compute the engineered feature importance values at inference time. You initialize the scoring explainer with the `feature_map` that was computed previously.

Save the scoring explainer, and then register the model and the scoring explainer with the Model Management Service. Run the following code:

```
from azureml.interpret.scoring.scoring_explainer import TreeScoringExplainer, save

# Initialize the ScoringExplainer
scoring_explainer = TreeScoringExplainer(explainer.explainer, feature_maps=
[automl_explainer_setup_obj.feature_map])

# Pickle scoring explainer locally
save(scoring_explainer, exist_ok=True)

# Register trained automl model present in the 'outputs' folder in the artifacts
original_model = automl_run.register_model(model_name='automl_model',
                                           model_path='outputs/model.pkl')

# Register scoring explainer
automl_run.upload_file('scoring_explainer.pkl', 'scoring_explainer.pkl')
scoring_explainer_model = automl_run.register_model(model_name='scoring_explainer',
model_path='scoring_explainer.pkl')
```

Create the conda dependencies for setting up the service

Next, create the necessary environment dependencies in the container for the deployed model. Please note that azureml-defaults with version >= 1.0.45 must be listed as a pip dependency, because it contains the functionality needed to host the model as a web service.

```
from azureml.core.conda_dependencies import CondaDependencies

azureml_pip_packages = [
    'azureml-interpret', 'azureml-train-automl', 'azureml-defaults'
]

myenv = CondaDependencies.create(conda_packages=['scikit-learn', 'pandas', 'numpy', 'py-xgboost<=0.80'],
                                 pip_packages=azureml_pip_packages,
                                 pin_sdk_version=True)

with open("myenv.yml","w") as f:
    f.write(myenv.serialize_to_string())

with open("myenv.yml","r") as f:
    print(f.read())
```

Deploy the service

Deploy the service using the conda file and the scoring file from the previous steps.

```
from azureml.core.webservice import Webservice
from azureml.core.webservice import AciWebservice
from azureml.core.model import Model, InferenceConfig
from azureml.core.environment import Environment

aciconfig = AciWebservice.deploy_configuration(cpu_cores=1,
                                               memory_gb=1,
                                               tags={"data": "Bank Marketing",
                                                     "method" : "local_explanation"},
                                               description='Get local explanations for Bank marketing test
data')
myenv = Environment.from_conda_specification(name="myenv", file_path="myenv.yml")
inference_config = InferenceConfig(entry_script="score_local_explain.py", environment=myenv)

# Use configs and models generated above
service = Model.deploy(ws,
                       'model-scoring',
                       [scoring_explainer_model, original_model],
                       inference_config,
                       aciconfig)
service.wait_for_deployment(show_output=True)
```

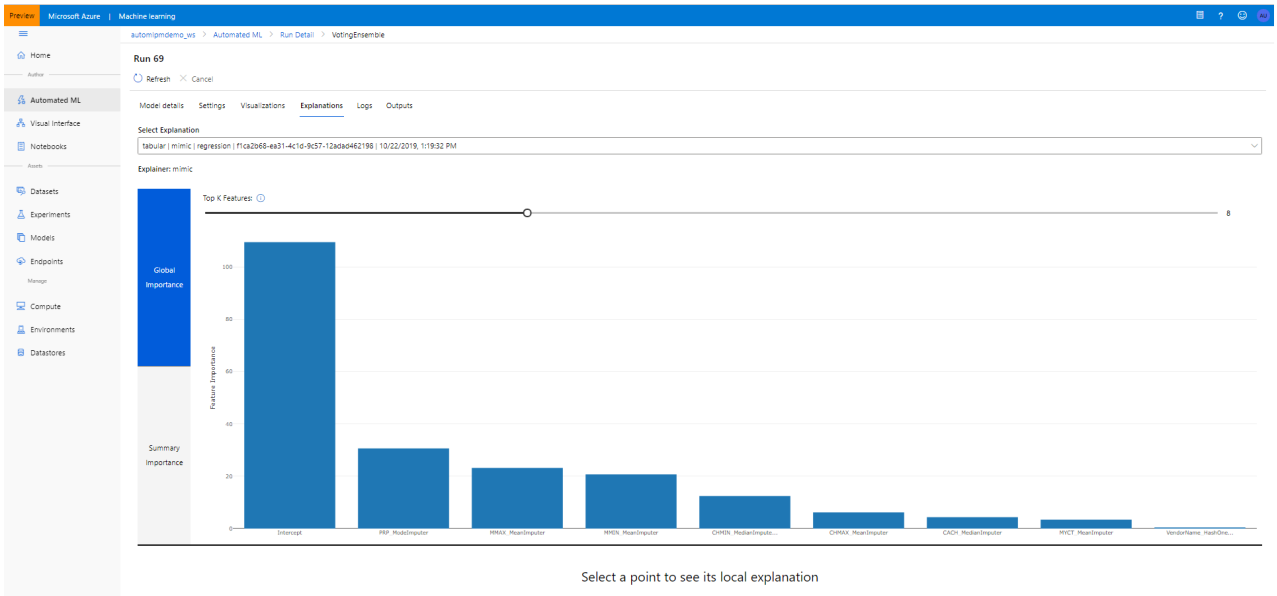
Inference with test data

Inference with some test data to see the predicted value from automated ML model. View the engineered feature importance for the predicted value.

```
if service.state == 'Healthy':
    # Serialize the first row of the test data into json
    X_test_json = X_test[:1].to_json(orient='records')
    print(X_test_json)
    # Call the service to get the predictions and the engineered explanations
    output = service.run(X_test_json)
    # Print the predicted value
    print(output['predictions'])
    # Print the engineered feature importances for the predicted value
    print(output['engineered_local_importance_values'])
```

Visualize to discover patterns in data and explanations at training time

You can visualize the feature importance chart in your workspace in Azure Machine Learning studio. After your automated ML run is complete, select **View model details** to view a specific run. Select the **Explanations** tab to see the explanation visualization dashboard.



Next steps

For more information about how you can enable model explanations and feature importance in areas of the Azure Machine Learning SDK other than automated machine learning, see the concept article on interpretability.

Configure automated ML experiments in Python

4/14/2020 • 14 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this guide, learn how to define various configuration settings of your automated machine learning experiments with the Azure Machine Learning SDK. Automated machine learning picks an algorithm and hyperparameters for you and generates a model ready for deployment. There are several options that you can use to configure automated machine learning experiments.

To view examples of an automated machine learning experiments, see Tutorial: Train a classification model with automated machine learning or Train models with automated machine learning in the cloud.

Configuration options available in automated machine learning:

- Select your experiment type: Classification, Regression, or Time Series Forecasting
- Data source, formats, and fetch data
- Choose your compute target: local or remote
- Automated machine learning experiment settings
- Run an automated machine learning experiment
- Explore model metrics
- Register and deploy model

If you prefer a no code experience, you can also Create your automated machine learning experiments in Azure Machine Learning studio.

Select your experiment type

Before you begin your experiment, you should determine the kind of machine learning problem you are solving. Automated machine learning supports task types of classification, regression, and forecasting. Learn more about task types.

Automated machine learning supports the following algorithms during the automation and tuning process. As a user, there is no need for you to specify the algorithm.

> **NOTE**
> If you plan to export your auto ML created models to an ONNX model, only those algorithms indicated with an * are able to be converted to the ONNX format. Learn more about converting models to ONNX.
>
> Also note, ONNX only supports classification and regression tasks at this time.

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
Logistic Regression*	Elastic Net*	Elastic Net
Light GBM*	Light GBM*	Light GBM
Gradient Boosting*	Gradient Boosting*	Gradient Boosting

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
Decision Tree*	Decision Tree*	Decision Tree
K Nearest Neighbors*	K Nearest Neighbors*	K Nearest Neighbors
Linear SVC*	LARS Lasso*	LARS Lasso
Support Vector Classification (SVC)*	Stochastic Gradient Descent (SGD)*	Stochastic Gradient Descent (SGD)
Random Forest*	Random Forest*	Random Forest
Extremely Randomized Trees*	Extremely Randomized Trees*	Extremely Randomized Trees
Xgboost*	Xgboost*	Xgboost
DNN Classifier	DNN Regressor	DNN Regressor
DNN Linear Classifier	Linear Regressor	Linear Regressor
Naive Bayes*	Fast Linear Regressor	Auto-ARIMA
Stochastic Gradient Descent (SGD)*	Online Gradient Descent Regressor	Prophet
Averaged Perceptron Classifier		ForecastTCN
Linear SVM Classifier*		

Use the `task` parameter in the `AutoMLConfig` constructor to specify your experiment type.

```
from azureml.train.automl import AutoMLConfig

# task can be one of classification, regression, forecasting
automl_config = AutoMLConfig(task = "classification")
```

Data source and format

Automated machine learning supports data that resides on your local desktop or in the cloud such as Azure Blob Storage. The data can be read into a **Pandas DataFrame** or an **Azure Machine Learning TabularDataset**. Learn more about datasets.

Requirements for training data:

- Data must be in tabular form.
- The value to predict, target column, must be in the data.

The following code examples demonstrate how to store the data in these formats.

- TabularDataset

```
from azureml.core.dataset import Dataset
from azureml.opendatasets import Diabetes

tabular_dataset = Diabetes.get_tabular_dataset()
train_dataset, test_dataset = tabular_dataset.random_split(percentage=0.1, seed=42)
label = "Y"
```

- Pandas dataframe

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv("your-local-file.csv")
train_data, test_data = train_test_split(df, test_size=0.1, random_state=42)
label = "label-col-name"
```

Fetch data for running experiment on remote compute

For remote executions, training data must be accessible from the remote compute. The class `Datasets` in the SDK exposes functionality to:

- easily transfer data from static files or URL sources into your workspace
- make your data available to training scripts when running on cloud compute resources

See the how-to for an example of using the `Dataset` class to mount data to your compute target.

Train and validation data

You can specify separate train and validation sets directly in the `AutoMLConfig` constructor.

K-Folds Cross Validation

Use `n_cross_validations` setting to specify the number of cross validations. The training data set will be randomly split into `n_cross_validations` folds of equal size. During each cross validation round, one of the folds will be used for validation of the model trained on the remaining folds. This process repeats for `n_cross_validations` rounds until each fold is used once as validation set. The average scores across all `n_cross_validations` rounds will be reported, and the corresponding model will be retrained on the whole training data set.

Monte Carlo Cross Validation (Repeated Random Sub-Sampling)

Use `validation_size` to specify the percentage of the training dataset that should be used for validation, and use `n_cross_validations` to specify the number of cross validations. During each cross validation round, a subset of size `validation_size` will be randomly selected for validation of the model trained on the remaining data. Finally, the average scores across all `n_cross_validations` rounds will be reported, and the corresponding model will be retrained on the whole training data set. Monte Carlo is not supported for time series forecasting.

Custom validation dataset

Use custom validation dataset if random split is not acceptable, usually time series data or imbalanced data. You can specify your own validation dataset. The model will be evaluated against the validation dataset specified instead of random dataset.

Compute to run experiment

Next determine where the model will be trained. An automated machine learning training experiment can run on the following compute options:

- Your local machine such as a local desktop or laptop – Generally when you have small dataset and you are still in the exploration stage.
- A remote machine in the cloud – Azure Machine Learning Managed Compute is a managed service that enables the ability to train machine learning models on clusters of Azure virtual machines.

  See this GitHub site for examples of notebooks with local and remote compute targets.
- An Azure Databricks cluster in your Azure subscription. You can find more details here - Setup Azure Databricks cluster for Automated ML

  See this GitHub site for examples of notebooks with Azure Databricks.

Configure your experiment settings

There are several options that you can use to configure your automated machine learning experiment. These parameters are set by instantiating an `AutoMLConfig` object. See the AutoMLConfig class for a full list of parameters.

Some examples include:

1. Classification experiment using AUC weighted as the primary metric with experiment timeout minutes set to 30 minutes and 2 cross-validation folds.

```
automl_classifier=AutoMLConfig(
task='classification',
primary_metric='AUC_weighted',
experiment_timeout_minutes=30,
blacklist_models=['XGBoostClassifier'],
training_data=train_data,
label_column_name=label,
n_cross_validations=2)
```

2. Below is an example of a regression experiment set to end after 60 minutes with five validation cross folds.

```
automl_regressor = AutoMLConfig(
task='regression',
experiment_timeout_minutes=60,
whitelist_models=['KNN'],
primary_metric='r2_score',
training_data=train_data,
label_column_name=label,
n_cross_validations=5)
```

The three different `task` parameter values (the third task-type is `forecasting`, and uses a similar algorithm pool as `regression` tasks) determine the list of models to apply. Use the `whitelist` or `blacklist` parameters to further modify iterations with the available models to include or exclude. The list of supported models can be found on SupportedModels Class for (Classification, Forecasting, and Regression).

To help avoid experiment timeout failures, Automated ML's validation service will require that `experiment_timeout_minutes` be set to a minimum of 15 minutes, or 60 minutes if your row by column size exceeds 10 million.

Primary Metric

The primary metric determines the metric to be used during model training for optimization. The available metrics you can select is determined by the task type you choose, and the following table shows valid primary metrics for each task type.

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
accuracy	spearman_correlation	spearman_correlation
AUC_weighted	normalized_root_mean_squared_error	normalized_root_mean_squared_error
average_precision_score_weighted	r2_score	r2_score
norm_macro_recall	normalized_mean_absolute_error	normalized_mean_absolute_error
precision_score_weighted		

Learn about the specific definitions of these metrics in Understand automated machine learning results.

Data featurization

In every automated machine learning experiment, your data is automatically scaled and normalized to help *certain* algorithms that are sensitive to features that are on different scales. However, you can also enable additional featurization, such as missing values imputation, encoding, and transforms. Learn more about what featurization is included.

When configuring your experiments, you can enable the advanced setting `featurization`. The following table shows the accepted settings for featurization in the AutoMLConfig class.

FEATURIZATION CONFIGURATION	DESCRIPTION
`"featurization": 'FeaturizationConfig'`	Indicates customized featurization step should be used. Learn how to customize featurization.
`"featurization": 'off'`	Indicates featurization step should not be done automatically.
`"featurization": 'auto'`	Indicates that as part of preprocessing, data guardrails and featurization steps are performed automatically.

> **NOTE**
> Automated machine learning featurization steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same featurization steps applied during training are applied to your input data automatically.

Time Series Forecasting

The time series `forecasting` task requires additional parameters in the configuration object:

1. `time_column_name`: Required parameter that defines the name of the column in your training data containing a valid time-series.
2. `max_horizon`: Defines the length of time you want to predict out based on the periodicity of the training data. For example if you have training data with daily time grains, you define how far out in days you want the model to train for.
3. `grain_column_names`: Defines the name of columns that contain individual time series data in your training data. For example, if you are forecasting sales of a particular brand by store, you would define store and brand columns as your grain columns. Separate time-series and forecasts will be created for each grain/grouping.

For examples of the settings used below, see the sample notebook.

```
# Setting Store and Brand as grains for training.
grain_column_names = ['Store', 'Brand']
nseries = data.groupby(grain_column_names).ngroups

# View the number of time series data with defined grains
print('Data contains {0} individual time-series.'.format(nseries))
```

```
time_series_settings = {
    'time_column_name': time_column_name,
    'grain_column_names': grain_column_names,
    'drop_column_names': ['logQuantity'],
    'max_horizon': n_test_periods
}

automl_config = AutoMLConfig(task = 'forecasting',
                             debug_log='automl_oj_sales_errors.log',
                             primary_metric='normalized_root_mean_squared_error',
                             experiment_timeout_minutes=20,
                             training_data=train_data,
                             label_column_name=label,
                             n_cross_validations=5,
                             path=project_folder,
                             verbosity=logging.INFO,
                             **time_series_settings)
```

Ensemble configuration

Ensemble models are enabled by default, and appear as the final run iterations in an automated machine learning run. Currently supported ensemble methods are voting and stacking. Voting is implemented as soft-voting using weighted averages, and the stacking implementation is using a two layer implementation, where the first layer has the same models as the voting ensemble, and the second layer model is used to find the optimal combination of the models from the first layer. If you are using ONNX models, **or** have model-explainability enabled, stacking will be disabled and only voting will be utilized.

There are multiple default arguments that can be provided as `kwargs` in an `AutoMLConfig` object to alter the default stack ensemble behavior.

- `stack_meta_learner_type`: the meta-learner is a model trained on the output of the individual heterogeneous models. Default meta-learners are `LogisticRegression` for classification tasks (or `LogisticRegressionCV` if cross-validation is enabled) and `ElasticNet` for regression/forecasting tasks (or `ElasticNetCV` if cross-validation is enabled). This parameter can be one of the following strings: `LogisticRegression`, `LogisticRegressionCV`, `LightGBMClassifier`, `ElasticNet`, `ElasticNetCV`, `LightGBMRegressor`, or `LinearRegression`.
- `stack_meta_learner_train_percentage`: specifies the proportion of the training set (when choosing train and validation type of training) to be reserved for training the meta-learner. Default value is `0.2`.
- `stack_meta_learner_kwargs`: optional parameters to pass to the initializer of the meta-learner. These parameters and parameter types mirror the parameters and parameter types from the corresponding model constructor, and are forwarded to the model constructor.

The following code shows an example of specifying custom ensemble behavior in an `AutoMLConfig` object.

```
ensemble_settings = {
    "stack_meta_learner_type": "LogisticRegressionCV",
    "stack_meta_learner_train_percentage": 0.3,
    "stack_meta_learner_kwargs": {
        "refit": True,
        "fit_intercept": False,
        "class_weight": "balanced",
        "multi_class": "auto",
        "n_jobs": -1
    }
}

automl_classifier = AutoMLConfig(
        task='classification',
        primary_metric='AUC_weighted',
        experiment_timeout_minutes=30,
        training_data=train_data,
        label_column_name=label,
        n_cross_validations=5,
        **ensemble_settings
        )
```

Ensemble training is enabled by default, but it can be disabled by using the `enable_voting_ensemble` and `enable_stack_ensemble` boolean parameters.

```
automl_classifier = AutoMLConfig(
        task='classification',
        primary_metric='AUC_weighted',
        experiment_timeout_minutes=30,
        training_data=data_train,
        label_column_name=label,
        n_cross_validations=5,
        enable_voting_ensemble=False,
        enable_stack_ensemble=False
        )
```

Run experiment

For automated ML, you create an `Experiment` object, which is a named object in a `Workspace` used to run experiments.

```
from azureml.core.experiment import Experiment

ws = Workspace.from_config()

# Choose a name for the experiment and specify the project folder.
experiment_name = 'automl-classification'
project_folder = './sample_projects/automl-classification'

experiment = Experiment(ws, experiment_name)
```

Submit the experiment to run and generate a model. Pass the `AutoMLConfig` to the `submit` method to generate the model.

```
run = experiment.submit(automl_config, show_output=True)
```

> **NOTE**
>
> Dependencies are first installed on a new machine. It may take up to 10 minutes before output is shown. Setting `show_output` to `True` results in output being shown on the console.

Exit Criteria

There are a few options you can define to end your experiment.

1. No Criteria: If you do not define any exit parameters the experiment will continue until no further progress is made on your primary metric.
2. Exit after a length of time: Using `experiment_timeout_minutes` in your settings allows you to define how long in minutes should an experiment continue in run.
3. Exit after a score has been reached: Using `experiment_exit_score` will complete the experiment after a primary metric score has been reached.

Explore model metrics

You can view your training results in a widget or inline if you are in a notebook. See Track and evaluate models for more details.

Understand automated ML models

Any model produced using automated ML includes the following steps:

- Automated feature engineering (if `"featurization": 'auto'`)
- Scaling/Normalization and algorithm with hyperparameter values

We make it transparent to get this information from the fitted_model output from automated ML.

```
automl_config = AutoMLConfig(…)
automl_run = experiment.submit(automl_config …)
best_run, fitted_model = automl_run.get_output()
```

Automated feature engineering

See the list of preprocessing and automated feature engineering that happens when `"featurization": 'auto'`.

Consider this example:

- There are four input features: A (Numeric), B (Numeric), C (Numeric), D (DateTime)
- Numeric feature C is dropped because it is an ID column with all unique values
- Numeric features A and B have missing values and hence are imputed by the mean
- DateTime feature D is featurized into 11 different engineered features

Use these 2 APIs on the first step of fitted model to understand more. See this sample notebook.

- API 1: `get_engineered_feature_names()` returns a list of engineered feature names.

  Usage:

  ```
  fitted_model.named_steps['timeseriestransformer']. get_engineered_feature_names ()
  ```

  ```
  Output: ['A', 'B', 'A_WASNULL', 'B_WASNULL', 'year', 'half', 'quarter', 'month', 'day', 'hour',
  'am_pm', 'hour12', 'wday', 'qday', 'week']
  ```

This list includes all engineered feature names.

> **NOTE**
> Use 'timeseriestransformer' for task='forecasting', else use 'datatransformer' for 'regression' or 'classification' task.

- API 2: `get_featurization_summary()` returns featurization summary for all the input features.

  Usage:

```
fitted_model.named_steps['timeseriestransformer'].get_featurization_summary()
```

> **NOTE**
> Use 'timeseriestransformer' for task='forecasting', else use 'datatransformer' for 'regression' or 'classification' task.

  Output:

```
[{'RawFeatureName': 'A',
  'TypeDetected': 'Numeric',
  'Dropped': 'No',
  'EngineeredFeatureCount': 2,
  'Tranformations': ['MeanImputer', 'ImputationMarker']},
{'RawFeatureName': 'B',
  'TypeDetected': 'Numeric',
  'Dropped': 'No',
  'EngineeredFeatureCount': 2,
  'Tranformations': ['MeanImputer', 'ImputationMarker']},
{'RawFeatureName': 'C',
  'TypeDetected': 'Numeric',
  'Dropped': 'Yes',
  'EngineeredFeatureCount': 0,
  'Tranformations': []},
{'RawFeatureName': 'D',
  'TypeDetected': 'DateTime',
  'Dropped': 'No',
  'EngineeredFeatureCount': 11,
  'Tranformations':
['DateTime','DateTime','DateTime','DateTime','DateTime','DateTime','DateTime','DateTime','DateTime','D
ateTime','DateTime']}]
```

  Where:

OUTPUT	DEFINITION
RawFeatureName	Input feature/column name from the dataset provided.
TypeDetected	Detected datatype of the input feature.
Dropped	Indicates if the input feature was dropped or used.
EngineeringFeatureCount	Number of features generated through automated feature engineering transforms.
Transformations	List of transformations applied to input features to generate engineered features.

Customize feature engineering

To customize feature engineering, specify `"featurization": FeaturizationConfig`.

Supported customization includes:

CUSTOMIZATION	DEFINITION
Column purpose update	Override feature type for the specified column.
Transformer parameter update	Update parameters for the specified transformer. Currently supports Imputer (mean, most frequent & median) and HashOneHotEncoder.
Drop columns	Columns to drop from being featurized.
Block transformers	Block transformers to be used on featurization process.

Create the FeaturizationConfig object using API calls:

```
featurization_config = FeaturizationConfig()
featurization_config.blocked_transformers = ['LabelEncoder']
featurization_config.drop_columns = ['aspiration', 'stroke']
featurization_config.add_column_purpose('engine-size', 'Numeric')
featurization_config.add_column_purpose('body-style', 'CategoricalHash')
#default strategy mean, add transformer param for for 3 columns
featurization_config.add_transformer_params('Imputer', ['engine-size'], {"strategy": "median"})
featurization_config.add_transformer_params('Imputer', ['city-mpg'], {"strategy": "median"})
featurization_config.add_transformer_params('Imputer', ['bore'], {"strategy": "most_frequent"})
featurization_config.add_transformer_params('HashOneHotEncoder', [], {"number_of_bits": 3})
```

Scaling/Normalization and algorithm with hyperparameter values:

To understand the scaling/normalization and algorithm/hyperparameter values for a pipeline, use fitted_model.steps. Learn more about scaling/normalization. Here is a sample output:

```
[('RobustScaler', RobustScaler(copy=True, quantile_range=[10, 90], with_centering=True, with_scaling=True)),
('LogisticRegression', LogisticRegression(C=0.18420699693267145, class_weight='balanced', dual=False,
fit_intercept=True, intercept_scaling=1, max_iter=100, multi_class='multinomial', n_jobs=1, penalty='l2',
random_state=None, solver='newton-cg', tol=0.0001, verbose=0, warm_start=False))
```

To get more details, use this helper function:

```
from pprint import pprint


def print_model(model, prefix=""):
    for step in model.steps:
        print(prefix + step[0])
        if hasattr(step[1], 'estimators') and hasattr(step[1], 'weights'):
            pprint({'estimators': list(
                e[0] for e in step[1].estimators), 'weights': step[1].weights})
            print()
            for estimator in step[1].estimators:
                print_model(estimator[1], estimator[0] + ' - ')
        else:
            pprint(step[1].get_params())
            print()


print_model(model)
```

The following sample output is for a pipeline using a specific algorithm (LogisticRegression with RobustScalar, in this case).

```
RobustScaler
{'copy': True,
'quantile_range': [10, 90],
'with_centering': True,
'with_scaling': True}

LogisticRegression
{'C': 0.18420699693267145,
'class_weight': 'balanced',
'dual': False,
'fit_intercept': True,
'intercept_scaling': 1,
'max_iter': 100,
'multi_class': 'multinomial',
'n_jobs': 1,
'penalty': 'l2',
'random_state': None,
'solver': 'newton-cg',
'tol': 0.0001,
'verbose': 0,
'warm_start': False}
```

Predict class probability

Models produced using automated ML all have wrapper objects that mirror functionality from their open-source origin class. Most classification model wrapper objects returned by automated ML implement the `predict_proba()` function, which accepts an array-like or sparse matrix data sample of your features (X values), and returns an n-dimensional array of each sample and its respective class probability.

Assuming you have retrieved the best run and fitted model using the same calls from above, you can call `predict_proba()` directly from the fitted model, supplying an `X_test` sample in the appropriate format depending on the model type.

```
best_run, fitted_model = automl_run.get_output()
class_prob = fitted_model.predict_proba(X_test)
```

If the underlying model does not support the `predict_proba()` function or the format is incorrect, a model class-specific exception will be thrown. See the RandomForestClassifier and XGBoost reference docs for examples of

how this function is implemented for different model types.

Model interpretability

Model interpretability allows you to understand why your models made predictions, and the underlying feature importance values. The SDK includes various packages for enabling model interpretability features, both at training and inference time, for local and deployed models.

See the how-to for code samples on how to enable interpretability features specifically within automated machine learning experiments.

For general information on how model explanations and feature importance can be enabled in other areas of the SDK outside of automated machine learning, see the concept article on interpretability.

Next steps

Learn more about how and where to deploy a model.

Learn more about how to train a regression model with Automated machine learning or how to train using Automated machine learning on a remote resource.

Create, review, and deploy automated machine learning models with Azure Machine Learning

4/12/2020 • 13 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this article, you learn how to create, explore, and deploy automated machine learning models without a single line of code in Azure Machine Learning's studio interface. Automated machine learning is a process in which the best machine learning algorithm to use for your specific data is selected for you. This process enables you to generate machine learning models quickly. Learn more about automated machine learning.

For an end to end example, try the tutorial for creating a classification model with Azure Machine Learning's automated ML interface.

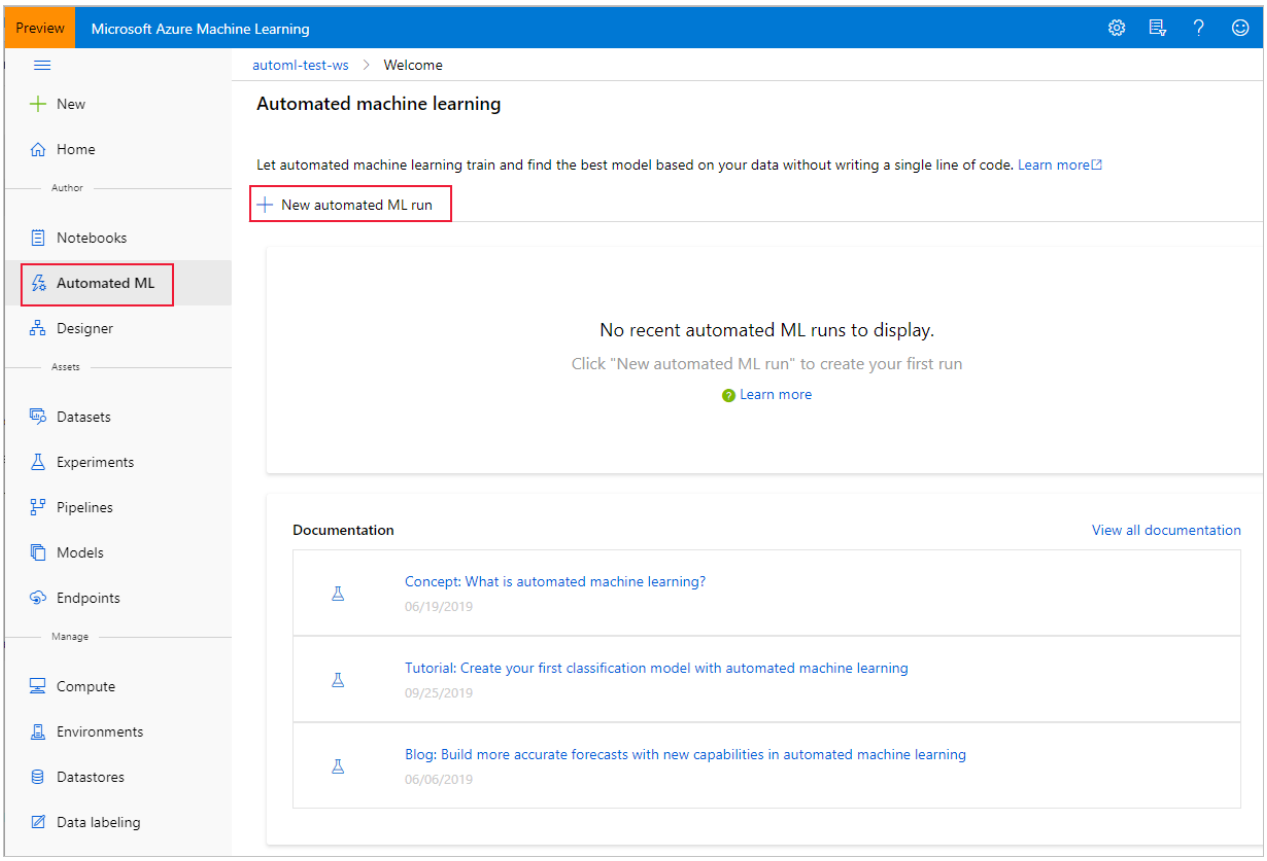
For a Python code-based experience, configure your automated machine learning experiments with the Azure Machine Learning SDK.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- An Azure Machine Learning workspace with a type of **Enterprise edition**. See Create an Azure Machine Learning workspace. To upgrade an existing workspace to Enterprise edition, see Upgrade to Enterprise edition.

Get started

1. Sign in to Azure Machine Learning at https://ml.azure.com.
2. Select your subscription and workspace.
3. Navigate to the left pane. Select **Automated ML** under the **Author** section.



If this is your first time doing any experiments, you'll see an empty list and links to documentation.

Otherwise, you'll see a list of your recent automated machine learning experiments, including those created with the SDK.

Create and run experiment

1. Select **+ New automated ML run** and populate the form.
2. Select a dataset from your storage container, or create a new dataset. Datasets can be created from local files, web urls, datastores, or Azure open datasets.

   > **IMPORTANT**
   > Requirements for training data:
   > - Data must be in tabular form.
   > - The value you want to predict (target column) must be present in the data.

   a. To create a new dataset from a file on your local computer, select **Browse** and then select the file.

   b. Give your dataset a unique name and provide an optional description.

   c. Select **Next** to open the **Datastore and file selection form**. On this form you select where to upload your dataset; the default storage container that's automatically created with your workspace, or choose a storage container that you want to use for the experiment.

   d. Review the **Settings and preview** form for accuracy. The form is intelligently populated based on the file type.

FIELD	DESCRIPTION
File format	Defines the layout and type of data stored in a file.
Delimiter	One or more characters for specifying the boundary between separate, independent regions in plain text or other data streams.
Encoding	Identifies what bit to character schema table to use to read your dataset.
Column headers	Indicates how the headers of the dataset, if any, will be treated.
Skip rows	Indicates how many, if any, rows are skipped in the dataset.

Select **Next**.

e. The **Schema** form is intelligently populated based on the selections in the **Settings and preview** form. Here configure the data type for each column, review the column names, and select which columns to **Not include** for your experiment.

Select **Next.**

f. The **Confirm details** form is a summary of the information previously populated in the **Basic info** and **Settings and preview** forms. You also have the option to create a data profile for your dataset using a profiling enabled compute. Learn more about data profiling.

Select **Next**.

3. Select your newly created dataset once it appears. You are also able to view a preview of the dataset and sample statistics.
4. On the **Configure run** form, enter a unique experiment name.
5. Select a target column; this is the column that you would like to do predictions on.
6. Select a compute for the data profiling and training job. A list of your existing computes is available in the dropdown. To create a new compute, follow the instructions in step 7.
7. Select **Create a new compute** to configure your compute context for this experiment.

FIELD	DESCRIPTION
Compute name	Enter a unique name that identifies your compute context.
Virtual machine size	Select the virtual machine size for your compute.
Min / Max nodes (in Advanced Settings)	To profile data, you must specify 1 or more nodes. Enter the maximum number of nodes for your compute. The default is 6 nodes for an AML Compute.

Select **Create**. Creation of a new compute can take a few minutes.

> **NOTE**
> Your compute name will indicate if the compute you select/create is *profiling enabled*. (See the section data profiling for more details).

Select **Next**.

8. On the **Task type and settings** form, select the task type: classification, regression, or forecasting. See how to define task types for more information.

   a. For classification, you can also enable deep learning which is used for text featurizations.

   b. For forecasting:

      a. Select time column: This column contains the time data to be used.

      b. Select forecast horizon: Indicate how many time units (minutes/hours/days/weeks/months/years) will the model be able to predict to the future. The further the model is required to predict into the future, the less accurate it will become. Learn more about forecasting and forecast horizon.

9. (Optional) View addition configuration settings: additional settings you can use to better control the training job. Otherwise, defaults are applied based on experiment selection and data.

ADDITIONAL CONFIGURATIONS	DESCRIPTION
Primary metric	Main metric used for scoring your model. Learn more about model metrics.
Automatic featurization	Select to enable or disable the preprocessing done by automated machine learning. Preprocessing includes automatic data cleansing, preparing, and transformation to generate synthetic features. Not supported for the time series forecasting task type. Learn more about preprocessing.
Explain best model	Select to enable or disable to show explainability of the recommended best model
Blocked algorithm	Select algorithms you want to exclude from the training job.
Exit criterion	When any of these criteria are met, the training job is stopped. *Training job time (hours)*: How long to allow the training job to run. *Metric score threshold*: Minimum metric score for all pipelines. This ensures that if you have a defined target metric you want to reach, you do not spend more time on the training job than necessary.
Validation	Select one of the cross validation options to use in the training job. Learn more about cross validation.
Concurrency	*Max concurrent iterations*: Maximum number of pipelines (iterations) to test in the training job. The job will not run more than the specified number of iterations.

10. (Optional) View featurization settings: if you choose to enable **Automatic featurization** in the **Additional configuration settings** form, this form is where you specify which columns to perform those featurizations on, and select which statistical value to use for missing value imputations.

Data profiling & summary stats

You can get a vast variety of summary statistics across your data set to verify whether your data set is ML-ready. For non-numeric columns, they include only basic statistics like min, max, and error count. For numeric columns, you can also review their statistical moments and estimated quantiles. Specifically, our data profile includes:

> **NOTE**
> Blank entries appear for features with irrelevant types.

STATISTIC	DESCRIPTION
Feature	Name of the column that is being summarized.
Profile	In-line visualization based on the type inferred. For example, strings, booleans, and dates will have value counts, while decimals (numerics) have approximated histograms. This allows you to gain a quick understanding of the distribution of the data.
Type distribution	In-line value count of types within a column. Nulls are their own type, so this visualization is useful for detecting odd or missing values.
Type	Inferred type of the column. Possible values include: strings, booleans, dates, and decimals.
Min	Minimum value of the column. Blank entries appear for features whose type does not have an inherent ordering (e.g. booleans).
Max	Maximum value of the column.
Count	Total number of missing and non-missing entries in the column.
Not missing count	Number of entries in the column that are not missing. Empty strings and errors are treated as values, so they will not contribute to the "not missing count."
Quantiles	Approximated values at each quantile to provide a sense of the distribution of the data.
Mean	Arithmetic mean or average of the column.
Standard deviation	Measure of the amount of dispersion or variation of this column's data.
Variance	Measure of how far spread out this column's data is from its average value.

STATISTIC	DESCRIPTION
Skewness	Measure of how different this column's data is from a normal distribution.
Kurtosis	Measure of how heavily tailed this column's data is compared to a normal distribution.

Advanced featurization options

Automated machine learning offers preprocessing and data guardrails automatically, to help you identify and manage potential issues with your data.

Preprocessing

> **NOTE**
> If you plan to export your auto ML created models to an ONNX model, only the featurization options indicated with an * are supported in the ONNX format. Learn more about converting models to ONNX.

PREPROCESSING STEPS	DESCRIPTION
Drop high cardinality or no variance features*	Drop these from training and validation sets, including features with all values missing, same value across all rows or with extremely high cardinality (for example, hashes, IDs, or GUIDs).
Impute missing values*	For numerical features, impute with average of values in the column. For categorical features, impute with most frequent value.
Generate additional features*	For DateTime features: Year, Month, Day, Day of week, Day of year, Quarter, Week of the year, Hour, Minute, Second. For Text features: Term frequency based on unigrams, bi-grams, and tri-character-grams.
Transform and encode *	Numeric features with few unique values are transformed into categorical features. One-hot encoding is performed for low cardinality categorical; for high cardinality, one-hot-hash encoding.
Word embeddings	Text featurizer that converts vectors of text tokens into sentence vectors using a pre-trained model. Each word's embedding vector in a document is aggregated together to produce a document feature vector.
Target encodings	For categorical features, maps each category with averaged target value for regression problems, and to the class probability for each class for classification problems. Frequency-based weighting and k-fold cross validation is applied to reduce over fitting of the mapping and noise caused by sparse data categories.

PREPROCESSING STEPS	DESCRIPTION
Text target encoding	For text input, a stacked linear model with bag-of-words is used to generate the probability of each class.
Weight of Evidence (WoE)	Calculates WoE as a measure of correlation of categorical columns to the target column. It is calculated as the log of the ratio of in-class vs out-of-class probabilities. This step outputs one numerical feature column per class and removes the need to explicitly impute missing values and outlier treatment.
Cluster Distance	Trains a k-means clustering model on all numerical columns. Outputs k new features, one new numerical feature per cluster, containing the distance of each sample to the centroid of each cluster.

Data guardrails

Data guardrails are applied when automatic featurization is enabled or validation is set to auto. Data guardrails help you identify potential issues with your data (e.g., missing values, class imbalance) and help take corrective actions for improved results.

Users can review data guardrails in the studio within the **Data guardrails** tab of an automated ML run or by setting `show_output=True` when submitting an experiment using the Python SDK.

Data Guardrail States

Data guardrails will display one of three states: **Passed**, **Done**, or **Alerted**.

STATE	DESCRIPTION
Passed	No data problems were detected and no user action is required.
Done	Changes were applied to your data. We encourage users to review the corrective actions Automated ML took to ensure the changes align with the expected results.
Alerted	A data issue that could not be remedied was detected. We encourage users to revise and fix the issue.

> **NOTE**
> Previous versions of automated ML experiments displayed a fourth state: **Fixed**. Newer experiments will not display this state, and all guardrails which displayed the **Fixed** state will now display **Done**.

The following table describes the data guardrails currently supported, and the associated statuses that users may come across when submitting their experiment.

GUARDRAIL	STATUS	CONDITION FOR TRIGGER
Missing feature values imputation	**Passed** **Done**	No missing feature values were detected in your training data. Learn more about missing value imputation. Missing feature values were detected in your training data and imputed.

GUARDRAIL	STATUS	CONDITION FOR TRIGGER
High cardinality feature handling	**Passed** **Done**	Your inputs were analyzed, and no high cardinality features were detected. Learn more about high cardinality feature detection. High cardinality features were detected in your inputs and were handled.
Validation split handling	**Done**	*The validation configuration was set to 'auto' and the training data contained **less** than 20,000 rows.* Each iteration of the trained model was validated through cross-validation. Learn more about validation data. *The validation configuration was set to 'auto' and the training data contained **more** than 20,000 rows.* The input data has been split into a training dataset and a validation dataset for validation of the model.
Class balancing detection	**Passed** **Alerted**	Your inputs were analyzed, and all classes are balanced in your training data. A dataset is considered balanced if each class has good representation in the dataset, as measured by number and ratio of samples. Imbalanced classes were detected in your inputs. To fix model bias fix the balancing problem. Learn more about imbalanced data.
Memory issues detection	**Passed** **Done**	The selected {horizon, lag, rolling window} value(s) were analyzed, and no potential out-of-memory issues were detected. Learn more about time-series forecasting configurations. The selected {horizon, lag, rolling window} values were analyzed and will potentially cause your experiment to run out of memory. The lag or rolling window configurations have been turned off.

GUARDRAIL	STATUS	CONDITION FOR TRIGGER
Frequency detection	**Passed** **Done**	The time series was analyzed and all data points are aligned with the detected frequency. The time series was analyzed and data points that do not align with the detected frequency were detected. These data points were removed from the dataset. Learn more about data preparation for time-series forecasting.

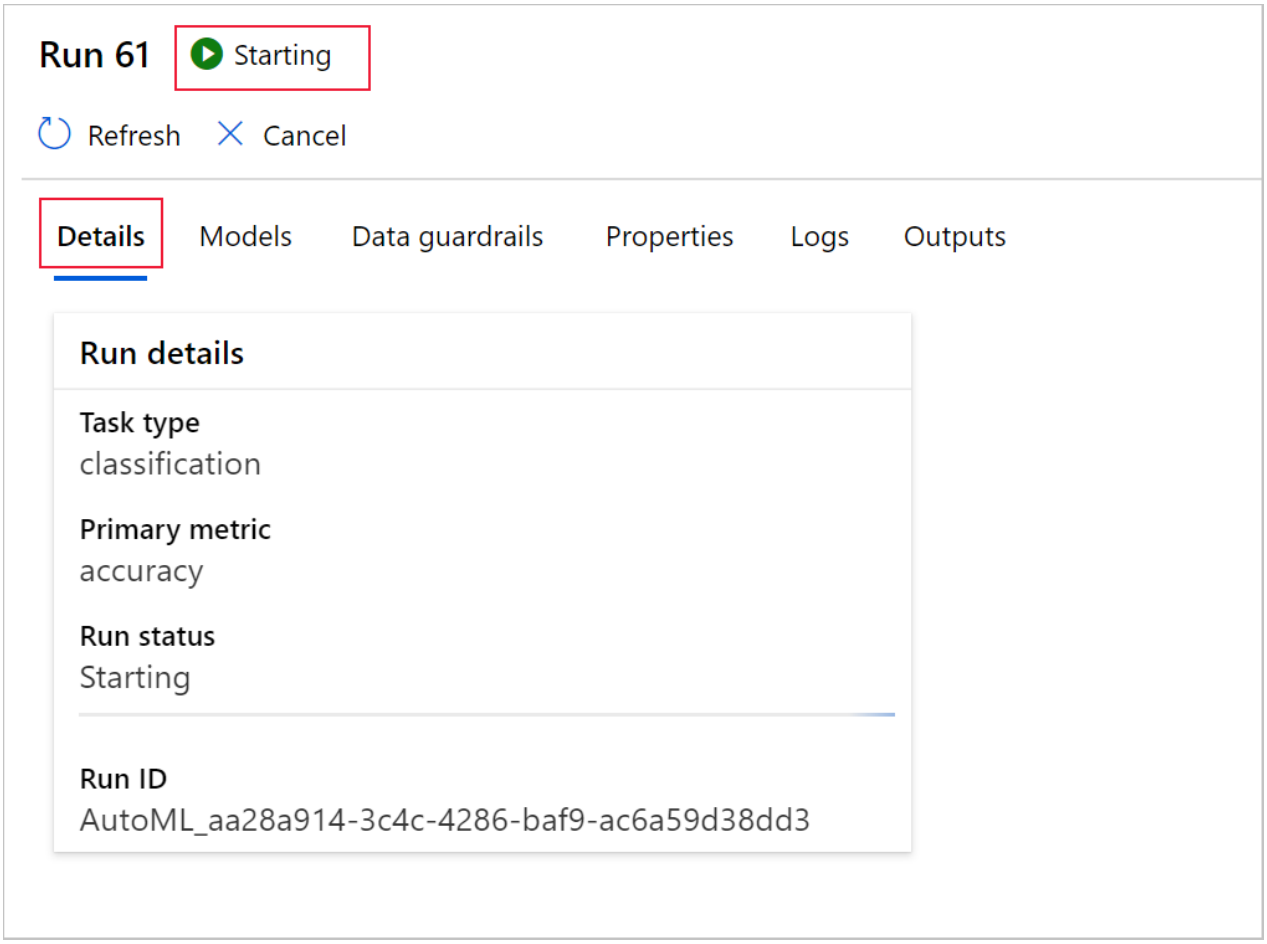
Run experiment and view results

Select **Finish** to run your experiment. The experiment preparing process can take up to 10 minutes. Training jobs can take an additional 2-3 minutes more for each pipeline to finish running.

View experiment details

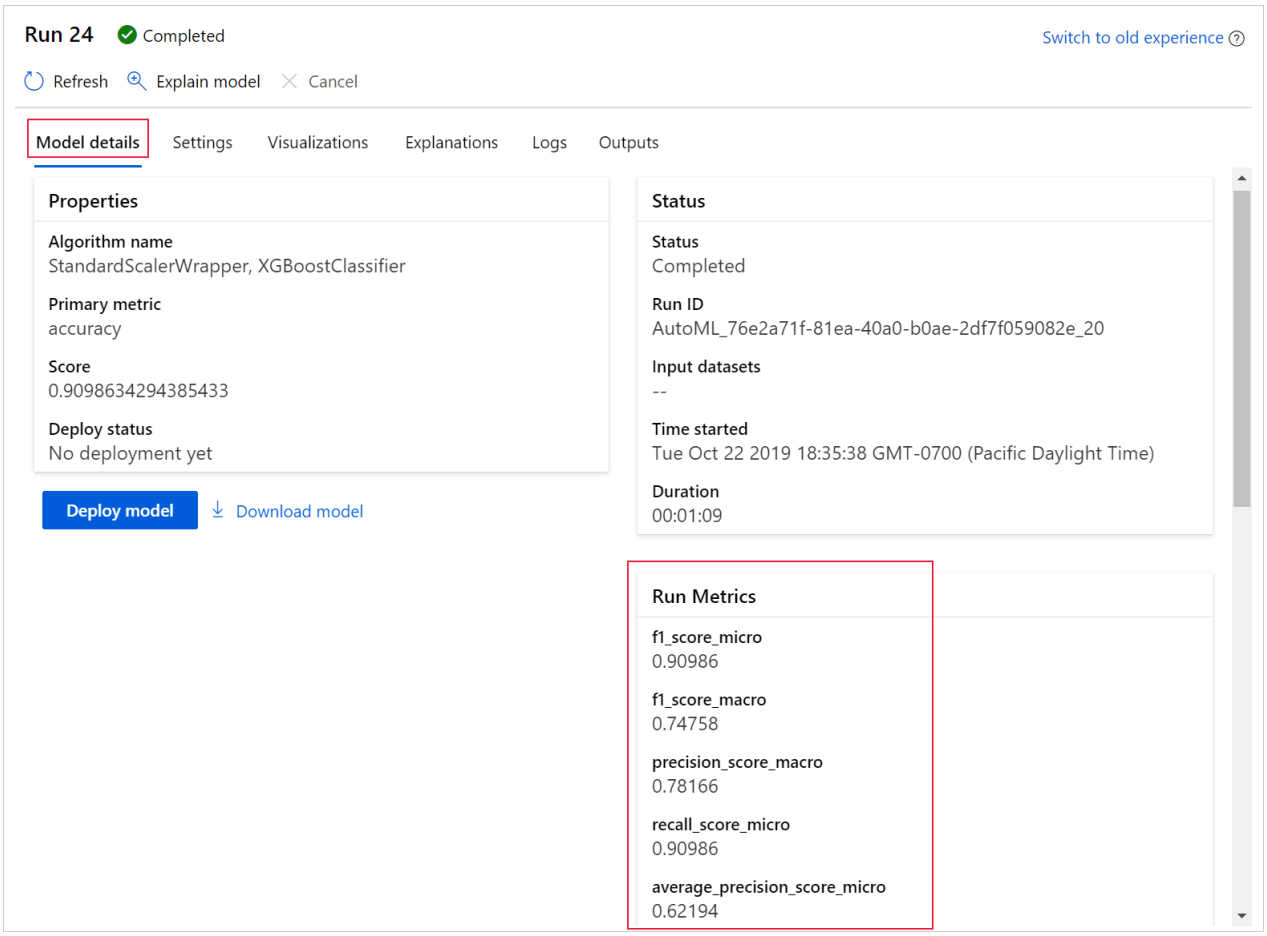
The **Run Detail** screen opens to the **Details** tab. This screen shows you a summary of the experiment run including a status bar at the top next to the run number.

The **Models** tab contains a list of the models created ordered by the metric score. By default, the model that scores the highest based on the chosen metric is at the top of the list. As the training job tries out more models, they are added to the list. Use this to get a quick comparison of the metrics for the models produced so far.



View training run details

Drill down on any of the completed models to see training run details, like run metrics on the **Model details** tab or performance charts on the **Visualizations** tab. Learn more about charts.



Deploy your model

Once you have the best model at hand, it is time to deploy it as a web service to predict on new data.

Automated ML helps you with deploying the model without writing code:

1. You have a couple options for deployment.
    - Option 1: To deploy the best model (according to the metric criteria you defined), select the **Deploy best model** button on the **Details** tab.
    - Option 2: To deploy a specific model iteration from this experiment, drill down on the model to open its **Model details** tab and select **Deploy model**.
2. Populate the **Deploy model** pane.

FIELD	VALUE
Name	Enter a unique name for your deployment.
Description	Enter a description to better identify what this deployment is for.
Compute type	Select the type of endpoint you want to deploy: *Azure Kubernetes Service (AKS)* or *Azure Container Instance (ACI)*.
Compute name	*Applies to AKS only:* Select the name of the AKS cluster you wish to deploy to.

FIELD	VALUE
Enable authentication	Select to allow for token-based or key-based authentication.
Use custom deployment assets	Enable this feature if you want to upload your own scoring script and environment file. Learn more about scoring scripts.

> **IMPORTANT**
>
> File names must be under 32 characters and must begin and end with alphanumerics. May include dashes, underscores, dots, and alphanumerics between. Spaces are not allowed.

The *Advanced* menu offers default deployment features such as data collection and resource utilization settings. If you wish to override these defaults do so in this menu.

3. Select **Deploy**. Deployment can take about 20 minutes to complete.

Now you have an operational web service to generate predictions! You can test the predictions by querying the service from Power BI's built in Azure Machine Learning support.

Next steps

- Learn how to consume a web service.
- Understand automated machine learning results.
- Learn more about automated machine learning and Azure Machine Learning.

Train models with automated machine learning in the cloud

3/10/2020 • 5 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In Azure Machine Learning, you train your model on different types of compute resources that you manage. The compute target could be a local computer or a resource in the cloud.

You can easily scale up or scale out your machine learning experiment by adding additional compute targets, such as Azure Machine Learning Compute (AmlCompute). AmlCompute is a managed-compute infrastructure that allows you to easily create a single or multi-node compute.

In this article, you learn how to build a model using automated ML with AmlCompute.

How does remote differ from local?

The tutorial "Train a classification model with automated machine learning" teaches you how to use a local computer to train a model with automated ML. The workflow when training locally also applies to remote targets as well. However, with remote compute, automated ML experiment iterations are executed asynchronously. This functionality allows you to cancel a particular iteration, watch the status of the execution, or continue to work on other cells in the Jupyter notebook. To train remotely, you first create a remote compute target such as AmlCompute. Then you configure the remote resource and submit your code there.

This article shows the extra steps needed to run an automated ML experiment on a remote AmlCompute target. The workspace object, `ws`, from the tutorial is used throughout the code here.

```
ws = Workspace.from_config()
```

Create resource

Create the `AmlCompute` target in your workspace (`ws`) if it doesn't already exist.

Time estimate: Creation of the AmlCompute target takes approximately 5 minutes.

```
from azureml.core.compute import AmlCompute
from azureml.core.compute import ComputeTarget
import os

# choose a name for your cluster
compute_name = os.environ.get("AML_COMPUTE_CLUSTER_NAME", "cpu-cluster")
compute_min_nodes = os.environ.get("AML_COMPUTE_CLUSTER_MIN_NODES", 0)
compute_max_nodes = os.environ.get("AML_COMPUTE_CLUSTER_MAX_NODES", 4)

# This example uses CPU VM. For using GPU VM, set SKU to STANDARD_NC6
vm_size = os.environ.get("AML_COMPUTE_CLUSTER_SKU", "STANDARD_D2_V2")


if compute_name in ws.compute_targets:
    compute_target = ws.compute_targets[compute_name]
    if compute_target and type(compute_target) is AmlCompute:
        print('found compute target. just use it. ' + compute_name)
else:
    print('creating a new compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size = vm_size,
                                                                 min_nodes = compute_min_nodes,
                                                                 max_nodes = compute_max_nodes)

    # create the cluster
    compute_target = ComputeTarget.create(ws, compute_name, provisioning_config)

    # can poll for a minimum number of nodes and for a specific timeout.
    # if no min node count is provided it will use the scale settings for the cluster
    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

     # For a more detailed view of current AmlCompute status, use get_status()
    print(compute_target.get_status().serialize())
```

You can now use the `compute_target` object as the remote compute target.

Cluster name restrictions include:

- Must be shorter than 64 characters.
- Cannot include any of the following characters: `\` ~ ! @ # $ % ^ & * ( ) = + _ [ ] { } \\ | ; : ' \" , < > / ?.`

Access data using TabularDataset function

Defined training_data as `TabularDataset` and the label, which are passed to Automated ML in the `AutoMLConfig`. The `TabularDataset` method `from_delimited_files`, by default, sets the `infer_column_types` to true, which will infer the columns type automatically.

If you do wish to manually set the column types, you can set the `set_column_types` argument to manually set the type of each column. In the following code sample, the data comes from the sklearn package.

```
from sklearn import datasets
from azureml.core.dataset import Dataset
from scipy import sparse
import numpy as np
import pandas as pd
import os

# Create a project_folder if it doesn't exist
if not os.path.isdir('data'):
    os.mkdir('data')

if not os.path.exists('project_folder'):
    os.makedirs('project_folder')

X = pd.DataFrame(data_train.data[100:,:])
y = pd.DataFrame(data_train.target[100:])

# merge X and y
label = "digit"
X[label] = y

training_data = X

training_data.to_csv('data/digits.csv')
ds = ws.get_default_datastore()
ds.upload(src_dir='./data', target_path='digitsdata', overwrite=True, show_progress=True)

training_data = Dataset.Tabular.from_delimited_files(path=ds.path('digitsdata/digits.csv'))
```

Configure experiment

Specify the settings for `AutoMLConfig`. (See a full list of parameters and their possible values.)

```
from azureml.train.automl import AutoMLConfig
import time
import logging

automl_settings = {
    "name": "AutoML_Demo_Experiment_{0}".format(time.time()),
    "experiment_timeout_minutes" : 20,
    "enable_early_stopping" : True,
    "iteration_timeout_minutes": 10,
    "n_cross_validations": 5,
    "primary_metric": 'AUC_weighted',
    "max_concurrent_iterations": 10,
}

automl_config = AutoMLConfig(task='classification',
                             debug_log='automl_errors.log',
                             path=project_folder,
                             compute_target=compute_target,
                             training_data=training_data,
                             label_column_name=label,
                             **automl_settings,
                             )
```

Submit training experiment

Now submit the configuration to automatically select the algorithm, hyper parameters, and train the model.

```
from azureml.core.experiment import Experiment
experiment = Experiment(ws, 'automl_remote')
remote_run = experiment.submit(automl_config, show_output=True)
```

You will see output similar to the following example:

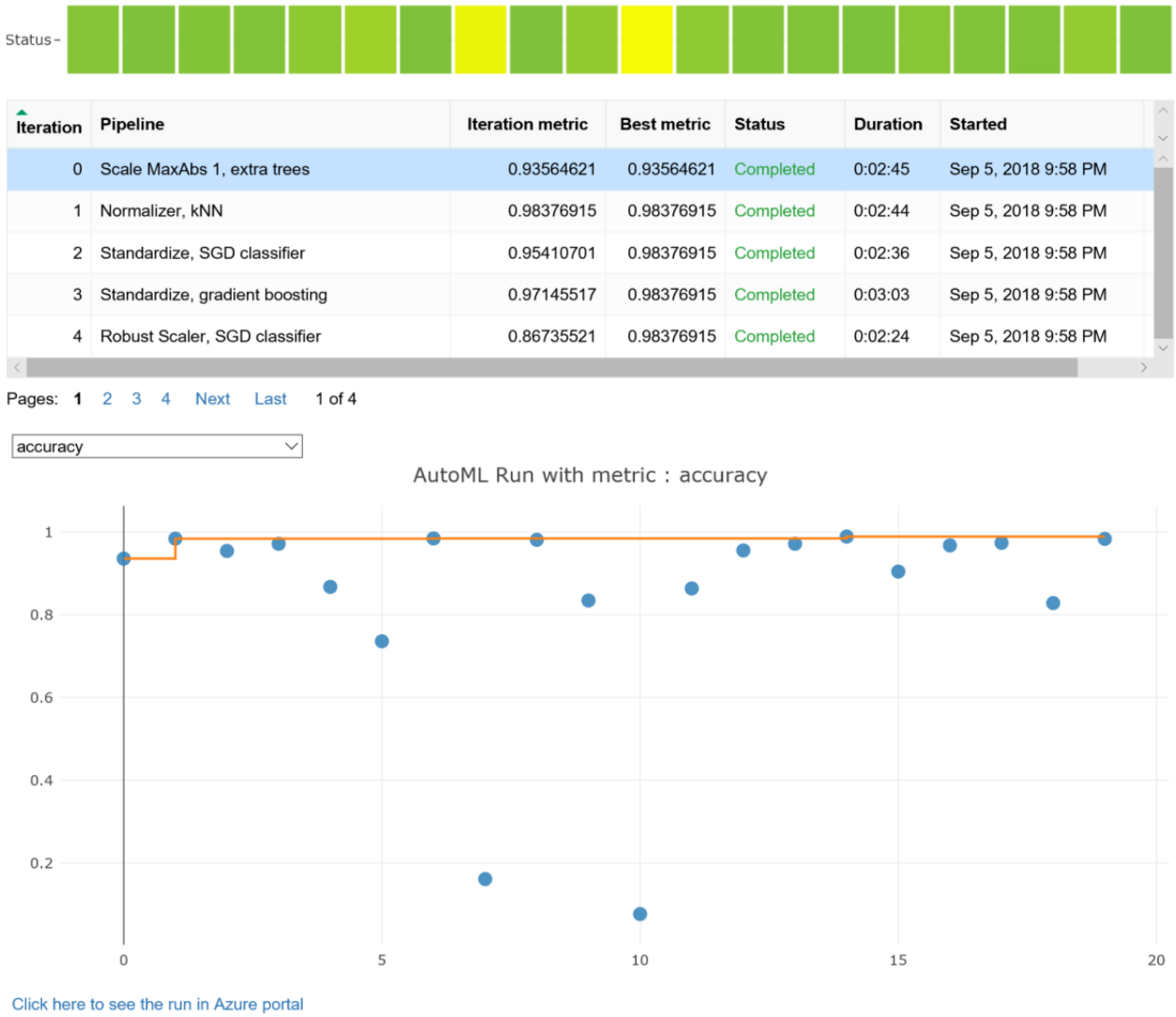
```
Running on remote compute: mydsvmParent Run ID: AutoML_015ffe76-c331-406d-9bfd-0fd42d8ab7f6
*********************************************************************************************
ITERATION: The iteration being evaluated.
PIPELINE:  A summary description of the pipeline being evaluated.
DURATION: Time taken for the current iteration.
METRIC: The result of computing score on the fitted pipeline.
BEST: The best observed score thus far.
*********************************************************************************************

 ITERATION     PIPELINE                                DURATION                METRIC      BEST
         2      Standardize SGD classifier            0:02:36                  0.954     0.954
         7      Normalizer DT                         0:02:22                  0.161     0.954
         0      Scale MaxAbs 1 extra trees            0:02:45                  0.936     0.954
         4      Robust Scaler SGD classifier          0:02:24                  0.867     0.954
         1      Normalizer kNN                        0:02:44                  0.984     0.984
         9      Normalizer extra trees                0:03:15                  0.834     0.984
         5      Robust Scaler DT                      0:02:18                  0.736     0.984
         8      Standardize kNN                       0:02:05                  0.981     0.984
         6      Standardize SVM                       0:02:18                  0.984     0.984
        10      Scale MaxAbs 1 DT                     0:02:18                  0.077     0.984
        11      Standardize SGD classifier            0:02:24                  0.863     0.984
         3      Standardize gradient boosting         0:03:03                  0.971     0.984
        12      Robust Scaler logistic regression     0:02:32                  0.955     0.984
        14      Scale MaxAbs 1 SVM                    0:02:15                  0.989     0.989
        13      Scale MaxAbs 1 gradient boosting      0:02:15                  0.971     0.989
        15      Robust Scaler kNN                     0:02:28                  0.904     0.989
        17      Standardize kNN                       0:02:22                  0.974     0.989
        16      Scale 0/1 gradient boosting           0:02:18                  0.968     0.989
        18      Scale 0/1 extra trees                 0:02:18                  0.828     0.989
        19      Robust Scaler kNN                     0:02:32                  0.983     0.989
```

Explore results

You can use the same Jupyter widget as shown in the training tutorial to see a graph and table of results.

```
from azureml.widgets import RunDetails
RunDetails(remote_run).show()
```

Here is a static image of the widget. In the notebook, you can click on any line in the table to see run properties and output logs for that run. You can also use the dropdown above the graph to view a graph of each available metric for each iteration.



The widget displays a URL you can use to see and explore the individual run details.

If you aren't in a Jupyter notebook, you can display the URL from the run itself:

```
remote_run.get_portal_url()
```

The same information is available in your workspace. To learn more about these results, see Understand automated machine learning results.

Example

The following notebook demonstrates concepts in this article.

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Next steps

- Learn how to configure settings for automatic training.
- See the how-to on enabling model interpretability features in automated ML experiments.

How to define a machine learning task

3/31/2020 • 2 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn the supported machine learning tasks and how to define them for an automated machine learning (automated ML) experiment run.

What is a machine learning task?

A machine learning task represents the type of problem being solved by creating a predictive model. Automated ML supports three different types of tasks including classification, regression, and time series forecasting.

TASK TYPE	DESCRIPTION	EXAMPLE
Classification	Task for predicting the category of a particular row in a dataset.	Fraud detection on a credit card. The target column would be **Fraud Detected** with categories of *True* or *False*. In this case, we are classifying each row in the data as either true or false.
Regression	Task for predicting a continuous quantity output.	Automobile cost based on its features, the target column would be **price**.
Forecasting	Task for making informed estimates in determining the direction of future trends.	Forecasting energy demand for them next 48 hours. The target column would be **demand** and the predicted values would be used to show patterns in the energy demand.

Automated ML supports the following algorithms during the automation and tuning process. As a user, there is no need for you to specify the algorithm.

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
Logistic Regression	Elastic Net	Elastic Net
Light GBM	Light GBM	Light GBM
Gradient Boosting	Gradient Boosting	Gradient Boosting
Decision Tree	Decision Tree	Decision Tree
K Nearest Neighbors	K Nearest Neighbors	K Nearest Neighbors
Linear SVC	LARS Lasso	LARS Lasso
C-Support Vector Classification (SVC)	Stochastic Gradient Descent (SGD)	Stochastic Gradient Descent (SGD)
Random Forest	Random Forest	Random Forest

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
Extremely Randomized Trees	Extremely Randomized Trees	Extremely Randomized Trees
Xgboost	Xgboost	Xgboost
DNN Classifier	DNN Regressor	DNN Regressor
DNN Linear Classifier	Linear Regressor	Linear Regressor
Naive Bayes		
Stochastic Gradient Descent (SGD)		

Set the task type

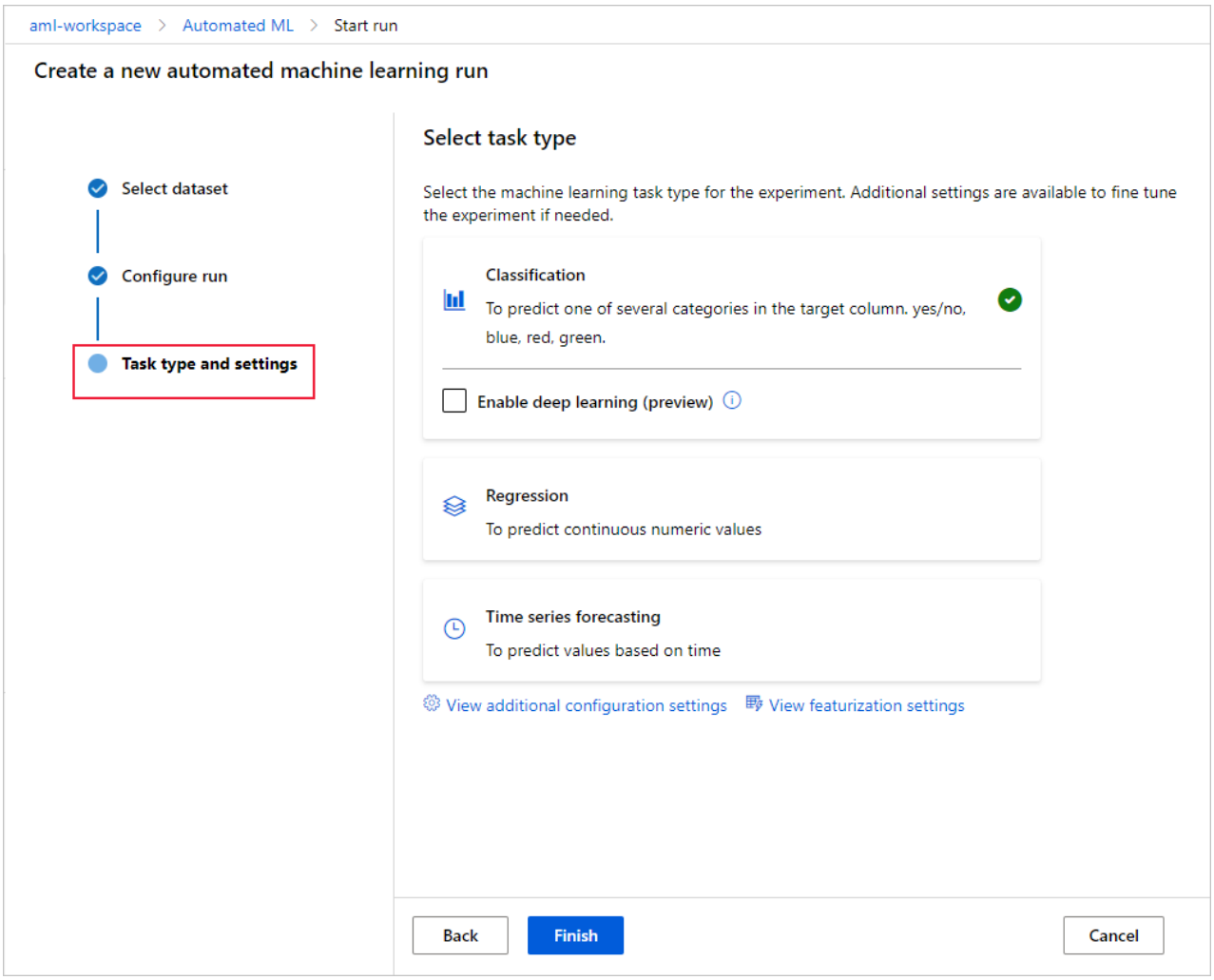
You can set the task type for your automated ML experiments with the SDK or the Azure Machine Learning studio.

Use the `task` parameter in the `AutoMLConfig` constructor to specify your experiment type.

```
from azureml.train.automl import AutoMLConfig

# task can be one of classification, regression, forecasting
automl_config = AutoMLConfig(task="classification")
```

You can set the task as part of your automated ML experiment run creation in the Azure Machine Learning studio.

> **IMPORTANT**
> The functionality in this studio, https://ml.azure.com, is **accessible from Enterprise workspaces only**. Learn more about editions and upgrading.



Next steps

- Learn more about automated ml in Azure Machine Learning.
- Learn more about auto-training a time-series forecast model in Azure Machine Learning
- Try the Automated Machine Learning Classification tutorial.
- Try the Automated Machine Learning Regression sample notebook.

Auto-train a time-series forecast model

4/24/2020 • 12 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to configure and train a time-series forecasting regression model using automated machine learning in Azure Machine Learning.

Configuring a forecasting model is similar to setting up a standard regression model using automated machine learning, but certain configuration options and pre-processing steps exist for working with time-series data.

For example, you can configure how far into the future the forecast should extend (the forecast horizon), as well as lags and more. Automated ML learns a single, but often internally branched model for all items in the dataset and prediction horizons. More data is thus available to estimate model parameters and generalization to unseen series becomes possible.

The following examples show you how to:

- Prepare data for time series modeling
- Configure specific time-series parameters in an `AutoMLConfig` object
- Run predictions with time-series data

Unlike classical time series methods, in automated ML past time-series values are "pivoted" to become additional dimensions for the regressor together with other predictors. This approach incorporates multiple contextual variables and their relationship to one another during training. Since multiple factors can influence a forecast, this method aligns itself well with real world forecasting scenarios. For example, when forecasting sales, interactions of historical trends, exchange rate and price all jointly drive the sales outcome.

Features extracted from the training data play a critical role. And, automated ML performs standard pre-processing steps and generates additional time-series features to capture seasonal effects and maximize predictive accuracy

Time-series and deep learning models

Automated ML's deep learning allows for forecasting univariate and multivariate time series data.

Deep learning models have three intrinsic capabilities:

1. They can learn from arbitrary mappings from inputs to outputs
2. They support multiple inputs and outputs
3. They can automatically extract patterns in input data that spans over long sequences

Given larger data, deep learning models, such as Microsoft's ForecastTCN, can improve the scores of the resulting model. Learn how to configure your experiment for deep learning.

Automated ML provides users with both native time-series and deep learning models as part of the recommendation system.

MODELS	DESCRIPTION	BENEFITS

MODELS	DESCRIPTION	BENEFITS
Prophet (Preview)	Prophet works best with time series that have strong seasonal effects and several seasons of historical data.	Accurate & fast, robust to outliers, missing data, and dramatic changes in your time series.
Auto-ARIMA (Preview)	AutoRegressive Integrated Moving Average (ARIMA) performs best, when the data is stationary. This means that its statistical properties like the mean and variance are constant over the entire set. For example, if you flip a coin, then the probability of you getting heads is 50%, regardless if you flip today, tomorrow or next year.	Great for univariate series, since the past values are used to predict the future values.
ForecastTCN (Preview)	ForecastTCN is a neural network model designed to tackle the most demanding forecasting tasks, capturing nonlinear local and global trends in your data as well as relationships between time series.	Capable of leveraging complex trends in your data and readily scales to the largest of datasets.

Prerequisites

- An Azure Machine Learning workspace. To create the workspace, see Create an Azure Machine Learning workspace.
- This article assumes basic familiarity with setting up an automated machine learning experiment. Follow the tutorial or how-to to see the basic automated machine learning experiment design patterns.

Preparing data

The most important difference between a forecasting regression task type and regression task type within automated machine learning is including a feature in your data that represents a valid time series. A regular time series has a well-defined and consistent frequency and has a value at every sample point in a continuous time span. Consider the following snapshot of a file `sample.csv`.

```
day_datetime,store,sales_quantity,week_of_year
9/3/2018,A,2000,36
9/3/2018,B,600,36
9/4/2018,A,2300,36
9/4/2018,B,550,36
9/5/2018,A,2100,36
9/5/2018,B,650,36
9/6/2018,A,2400,36
9/6/2018,B,700,36
9/7/2018,A,2450,36
9/7/2018,B,650,36
```

This data set is a simple example of daily sales data for a company that has two different stores, A and B. Additionally, there is a feature for `week_of_year` that will allow the model to detect weekly seasonality. The field `day_datetime` represents a clean time series with daily frequency, and the field `sales_quantity` is the target column for running predictions. Read the data into a Pandas dataframe, then use the `to_datetime` function to ensure the time series is a `datetime` type.

```
import pandas as pd
data = pd.read_csv("sample.csv")
data["day_datetime"] = pd.to_datetime(data["day_datetime"])
```

In this case, the data is already sorted ascending by the time field `day_datetime`. However, when setting up an experiment, ensure the desired time column is sorted in ascending order to build a valid time series. Assume the data contains 1,000 records, and make a deterministic split in the data to create training and test data sets. Identify the label column name and set it to label. In this example, the label will be `sales_quantity`. Then separate the label field from `test_data` to form the `test_target` set.

```
train_data = data.iloc[:950]
test_data = data.iloc[-50:]

label =  "sales_quantity"

test_labels = test_data.pop(label).values
```

> **NOTE**
> When training a model for forecasting future values, ensure all the features used in training can be used when running predictions for your intended horizon. For example, when creating a demand forecast, including a feature for current stock price could massively increase training accuracy. However, if you intend to forecast with a long horizon, you may not be able to accurately predict future stock values corresponding to future time-series points, and model accuracy could suffer.

Train and validation data

You can specify separate train and validation sets directly in the `AutoMLConfig` constructor.

Rolling Origin Cross Validation

For time series forecasting Rolling Origin Cross Validation (ROCV) is used to split time series in a temporally consistent way. ROCV divides the series into training and validation data using an origin time point. Sliding the origin in time generates the cross-validation folds.

This strategy will preserve the time series data integrity and eliminate the risk of data leakage. ROCV is automatically used for forecasting tasks by passing the training and validation data together and setting the number of cross validation folds using `n_cross_validations`.

```
automl_config = AutoMLConfig(task='forecasting',
                             n_cross_validations=3,
                             ...
                             **time_series_settings)
```

Learn more about the AutoMLConfig.

Configure and run experiment

For forecasting tasks, automated machine learning uses pre-processing and estimation steps that are specific to time-series data. The following pre-processing steps will be executed:

- Detect time-series sample frequency (for example, hourly, daily, weekly) and create new records for absent time points to make the series continuous.
- Impute missing values in the target (via forward-fill) and feature columns (using median column values)

- Create grain-based features to enable fixed effects across different series
- Create time-based features to assist in learning seasonal patterns
- Encode categorical variables to numeric quantities

The `AutoMLConfig` object defines the settings and data necessary for an automated machine learning task. Similar to a regression problem, you define standard training parameters like task type, number of iterations, training data, and number of cross-validations. For forecasting tasks, there are additional parameters that must be set that affect the experiment. The following table explains each parameter and its usage.

PARAMETER NAME	DESCRIPTION	REQUIRED
`time_column_name`	Used to specify the datetime column in the input data used for building the time series and inferring its frequency.	✓
`grain_column_names`	Name(s) defining individual series groups in the input data. If grain is not defined, the data set is assumed to be one time-series.	
`max_horizon`	Defines the maximum desired forecast horizon in units of time-series frequency. Units are based on the time interval of your training data, for example, monthly, weekly that the forecaster should predict out.	✓
`target_lags`	Number of rows to lag the target values based on the frequency of the data. The lag is represented as a list or single integer. Lag should be used when the relationship between the independent variables and dependent variable doesn't match up or correlate by default. For example, when trying to forecast demand for a product, the demand in any month may depend on the price of specific commodities 3 months prior. In this example, you may want to lag the target (demand) negatively by 3 months so that the model is training on the correct relationship.	
`target_rolling_window_size`	*n* historical periods to use to generate forecasted values, <= training set size. If omitted, *n* is the full training set size. Specify this parameter when you only want to consider a certain amount of history when training the model.	
`enable_dnn`	Enable Forecasting DNNs.	

See the reference documentation for more information.

Create the time-series settings as a dictionary object. Set the `time_column_name` to the `day_datetime` field in the data set. Define the `grain_column_names` parameter to ensure that **two separate time-series groups** are created for the data; one for store A and B. Lastly, set the `max_horizon` to 50 in order to predict for the entire test set. Set a forecast window to 10 periods with `target_rolling_window_size`, and specify a single lag on the target

values for two periods ahead with the `target_lags` parameter. It is recommended to set `max_horizon`, `target_rolling_window_size` and `target_lags` to "auto" which will automatically detect these values for you. In the example below, "auto" settings have been used for these parameters.

```
time_series_settings = {
    "time_column_name": "day_datetime",
    "grain_column_names": ["store"],
    "max_horizon": "auto",
    "target_lags": "auto",
    "target_rolling_window_size": "auto",
    "preprocess": True,
}
```

NOTE

Automated machine learning pre-processing steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same pre-processing steps applied during training are applied to your input data automatically.

By defining the `grain_column_names` in the code snippet above, AutoML will create two separate time-series groups, also known as multiple time-series. If no grain is defined, AutoML will assume that the dataset is a single time-series. To learn more about single time-series, see the energy_demand_notebook.

Now create a standard `AutoMLConfig` object, specifying the `forecasting` task type, and submit the experiment. After the model finishes, retrieve the best run iteration.

```
from azureml.core.workspace import Workspace
from azureml.core.experiment import Experiment
from azureml.train.automl import AutoMLConfig
import logging

automl_config = AutoMLConfig(task='forecasting',
                             primary_metric='normalized_root_mean_squared_error',
                             experiment_timeout_minutes=15,
                             enable_early_stopping=True,
                             training_data=train_data,
                             label_column_name=label,
                             n_cross_validations=5,
                             enable_ensembling=False,
                             verbosity=logging.INFO,
                             **time_series_settings)

ws = Workspace.from_config()
experiment = Experiment(ws, "forecasting_example")
local_run = experiment.submit(automl_config, show_output=True)
best_run, fitted_model = local_run.get_output()
```

See the forecasting sample notebooks for detailed code examples of advanced forecasting configuration including:

- holiday detection and featurization
- rolling-origin cross validation
- configurable lags
- rolling window aggregate features
- DNN

Configure a DNN enable Forecasting experiment

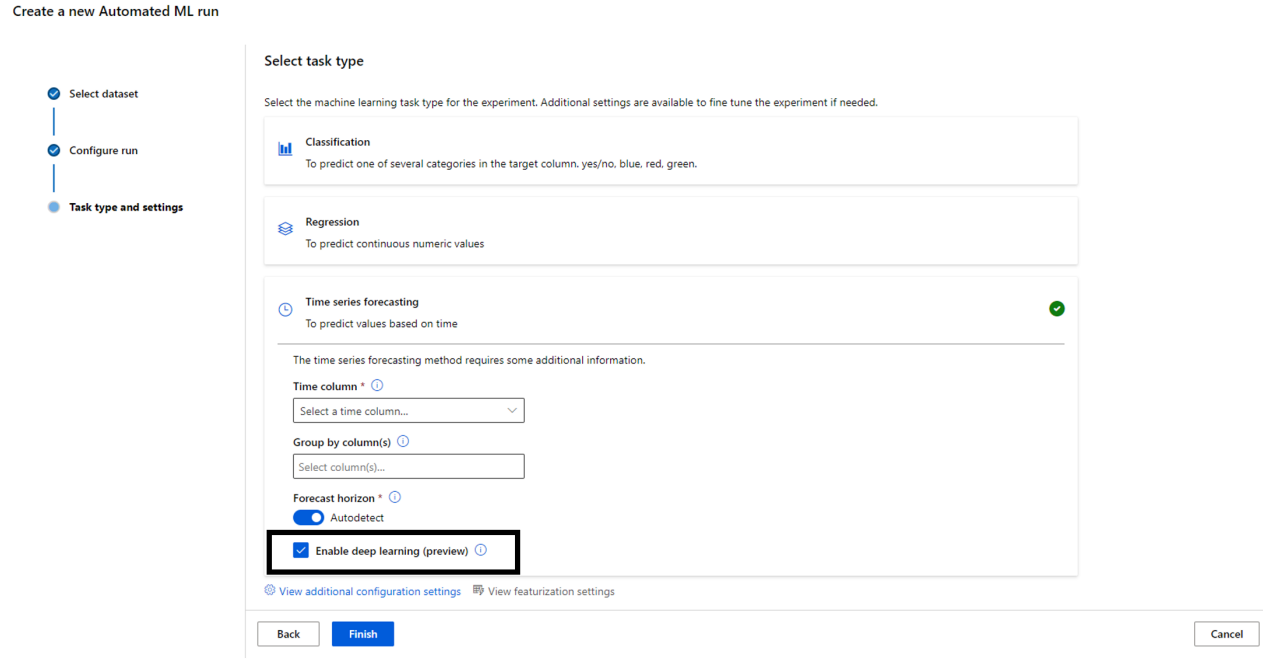
> **NOTE**
> DNN support for forecasting in Automated Machine Learning is in Preview and not supported for local runs.

In order to leverage DNNs for forecasting, you will need to set the `enable_dnn` parameter in the AutoMLConfig to true.

```
automl_config = AutoMLConfig(task='forecasting',
                             enable_dnn=True,
                             ...
                             **time_series_settings)
```

Learn more about the AutoMLConfig.

Alternatively, you can select the `Enable deep learning` option in the studio.



We recommend using an AML Compute cluster with GPU SKUs and at least two nodes as the compute target. To allow sufficient time for the DNN training to complete, we recommend setting the experiment timeout to a minimum of a couple of hours. For more information on AML compute and VM sizes that include GPU's, see the AML Compute documentation and GPU optimized virtual machine sizes documentation.

View the Beverage Production Forecasting notebook for a detailed code example leveraging DNNs.

Target Rolling Window Aggregation

Often the best information a forecaster can have is the recent value of the target. Creating cumulative statistics of the target may increase the accuracy of your predictions. Target rolling window aggregations allows you to add a rolling aggregation of data values as features. To enable target rolling windows set the `target_rolling_window_size` to your desired integer window size.

An example of this can be seen when predicting energy demand. You might add a rolling window feature of three days to account for thermal changes of heated spaces. In the example below, we've created this window of size three by setting `target_rolling_window_size=3` in the `AutoMLConfig` constructor. The table shows feature engineering that occurs when window aggregation is applied. Columns for minimum, maximum, and sum are generated on a sliding window of three based on the defined settings. Each row has a new calculated feature, in the case of the time-stamp for September 8, 2017 4:00am the maximum, minimum, and sum values are calculated using the demand values for September 8, 2017 1:00AM - 3:00AM. This window of three shifts along to populate data for the remaining rows.

Generating and using these additional features as extra contextual data helps with the accuracy of the train model.

View a Python code example leveraging the target rolling window aggregate feature.

View feature engineering summary

For time-series task types in automated machine learning, you can view details from the feature engineering process. The following code shows each raw feature along with the following attributes:

- Raw feature name
- Number of engineered features formed out of this raw feature
- Type detected
- Whether feature was dropped
- List of feature transformations for the raw feature

```
fitted_model.named_steps['timeseriestransformer'].get_featurization_summary()
```

Forecasting with best model

Use the best model iteration to forecast values for the test data set.

The `forecast()` function should be used instead of `predict()`, this will allow specifications of when predictions should start. In the following example, you first replace all values in `y_pred` with `NaN`. The forecast origin will be at the end of training data in this case, as it would normally be when using `predict()`. However, if you replaced only the second half of `y_pred` with `NaN`, the function would leave the numerical values in the first half unmodified, but forecast the `NaN` values in the second half. The function returns both the forecasted values and the aligned features.

You can also use the `forecast_destination` parameter in the `forecast()` function to forecast values up until a specified date.

```
label_query = test_labels.copy().astype(np.float)
label_query.fill(np.nan)
label_fcst, data_trans = fitted_pipeline.forecast(
    test_data, label_query, forecast_destination=pd.Timestamp(2019, 1, 8))
```

Calculate RMSE (root mean squared error) between the `actual_labels` actual values, and the forecasted values in `predict_labels`.

```
from sklearn.metrics import mean_squared_error
from math import sqrt

rmse = sqrt(mean_squared_error(actual_labels, predict_labels))
rmse
```

Now that the overall model accuracy has been determined, the most realistic next step is to use the model to forecast unknown future values. Supply a data set in the same format as the test set `test_data` but with future datetimes, and the resulting prediction set is the forecasted values for each time-series step. Assume the last time-series records in the data set were for 12/31/2018. To forecast demand for the next day (or as many periods as you need to forecast, <= `max_horizon`), create a single time series record for each store for 01/01/2019.

```
day_datetime,store,week_of_year
01/01/2019,A,1
01/01/2019,A,1
```

Repeat the necessary steps to load this future data to a dataframe and then run `best_run.predict(test_data)` to predict future values.

> **NOTE**
>
> Values cannot be predicted for number of periods greater than the `max_horizon`. The model must be re-trained with a larger horizon to predict future values beyond the current horizon.

Next steps

- Follow the tutorial to learn how to create experiments with automated machine learning.
- View the Azure Machine Learning SDK for Python reference documentation.

Understand automated machine learning results

3/11/2020 • 13 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to view and understand the charts and metrics for each of your automated machine learning runs.

Learn more about:

- Metrics, charts, and curves for classification models
- Metrics, charts, and graphs for regression models
- Model interpretability and feature importance

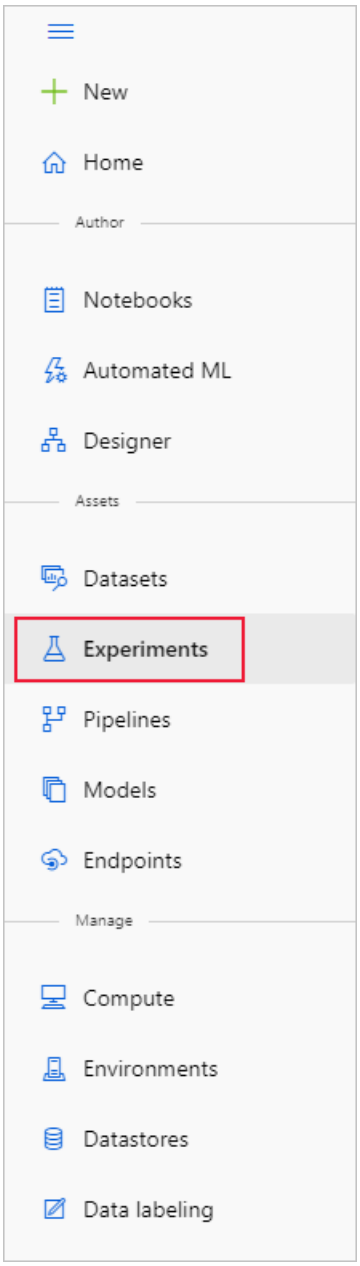
Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- Create an experiment for your automated machine learning run, either with the SDK, or in Azure Machine Learning studio.
  - Use the SDK to build a classification model or regression model
  - Use Azure Machine Learning studio to create a classification or regression model by uploading the appropriate data.

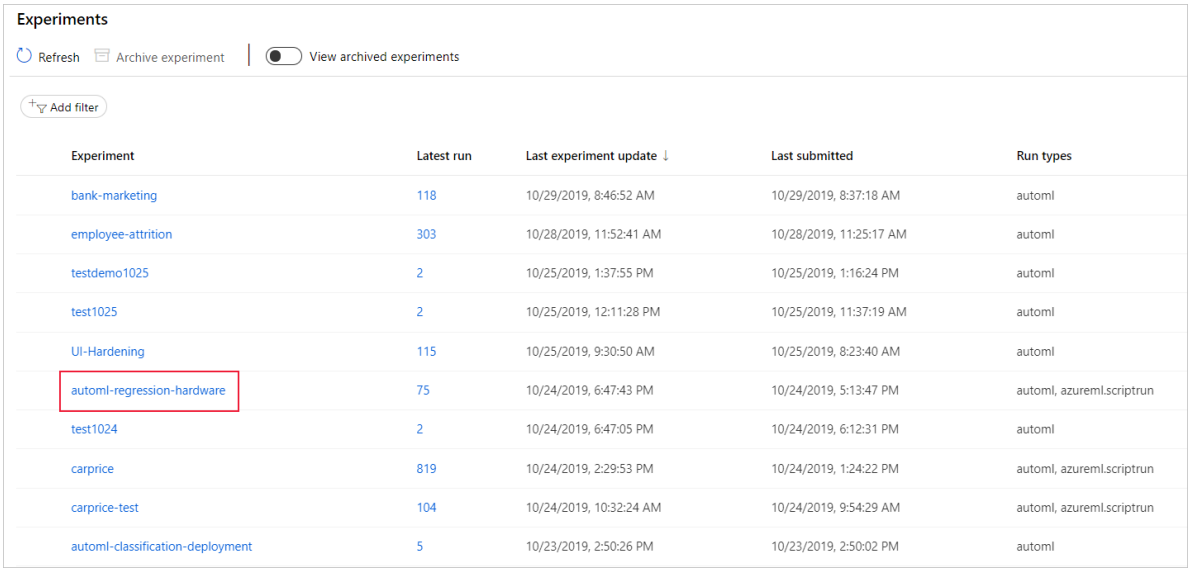
View the run

After running an automated machine learning experiment, a history of the runs can be found in your machine learning workspace.

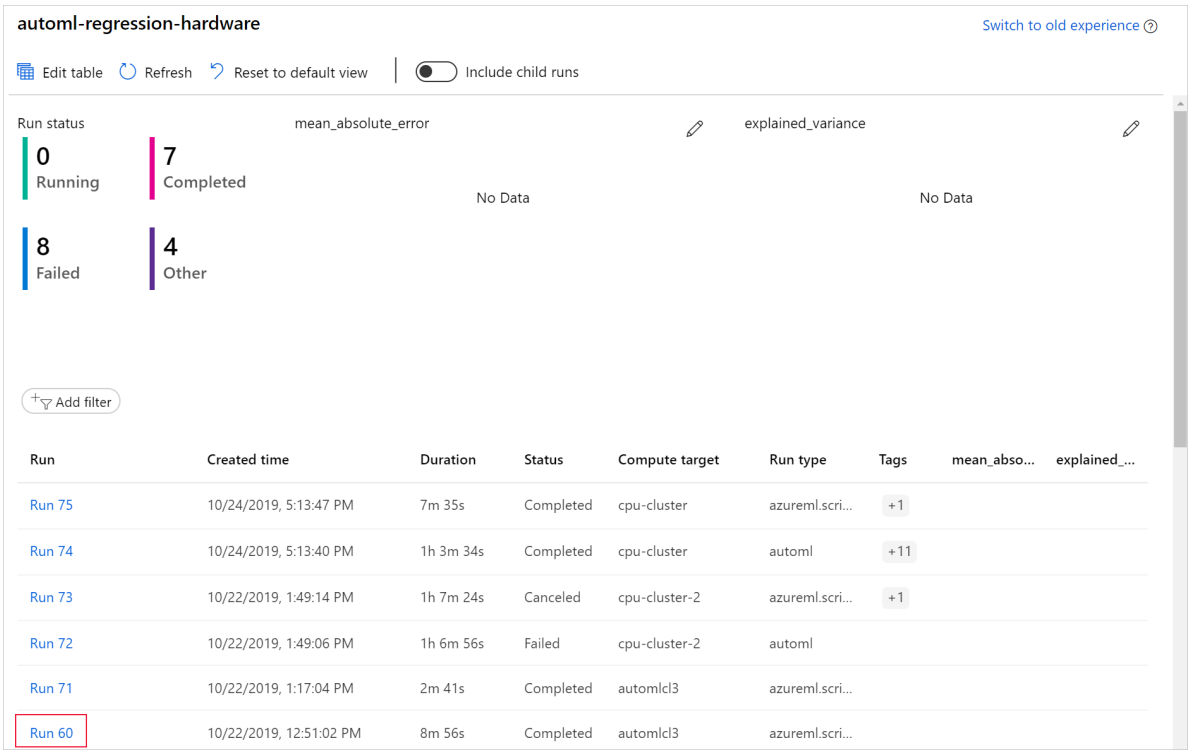
1. Go to your workspace.
2. In the left panel of the workspace, select **Experiments**.



3. In the list of experiments, select the one you want to explore.

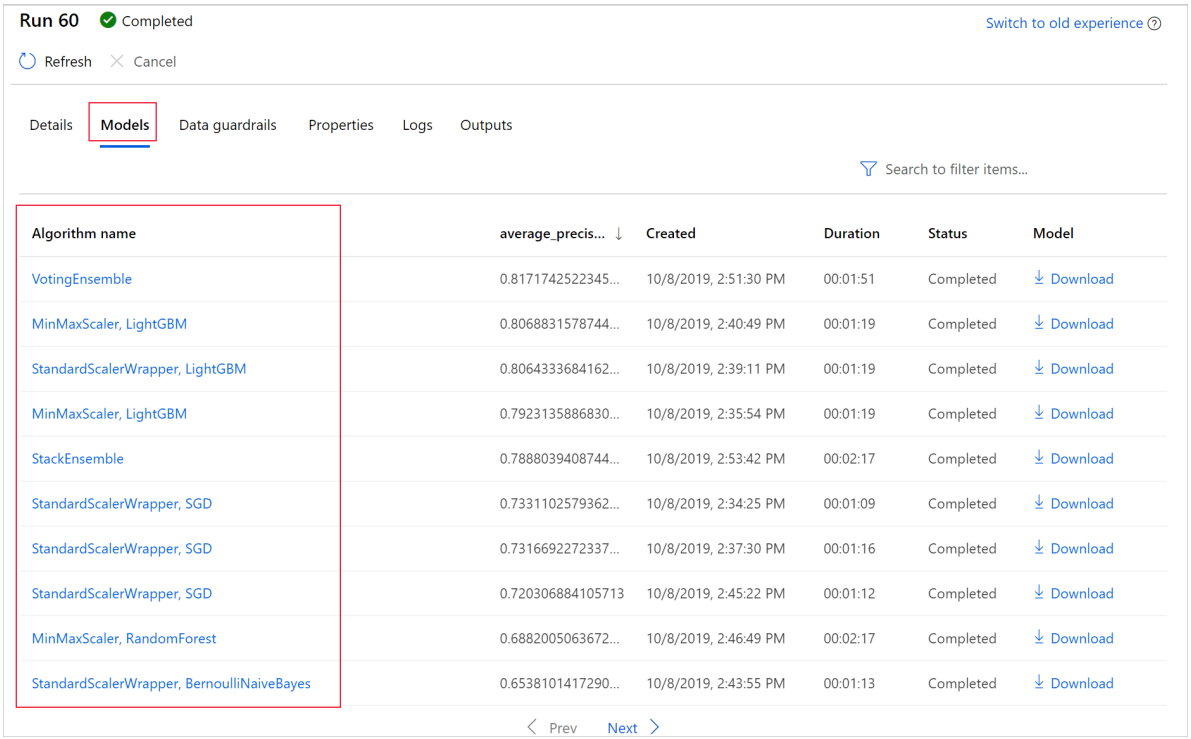


4. In the bottom table, select the **Run**.



)

5. In the Models, select the **Algorithm name** for the model that you want to explore further.



You also see these same results during a run when you use the `RunDetails` Jupyter widget.

Classification results

Thee following metrics and charts are available for every classification model that you build using the automated machine learning capabilities of Azure Machine Learning

- Metrics
- Confusion matrix
- Precision-Recall chart
- Receiver operating characteristics (or ROC)

- Lift curve
- Gains curve
- Calibration plot

Classification metrics

The following metrics are saved in each run iteration for a classification task.

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS
AUC_Macro	AUC is the Area under the Receiver Operating Characteristic Curve. Macro is the arithmetic mean of the AUC for each class.	Calculation	average="macro"
AUC_Micro	AUC is the Area under the Receiver Operating Characteristic Curve. Micro is computed globally by combining the true positives and false positives from each class.	Calculation	average="micro"
AUC_Weighted	AUC is the Area under the Receiver Operating Characteristic Curve. Weighted is the arithmetic mean of the score for each class, weighted by the number of true instances in each class.	Calculation	average="weighted"
accuracy	Accuracy is the percent of predicted labels that exactly match the true labels.	Calculation	None
average_precision_score_macro	Average precision summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight. Macro is the arithmetic mean of the average precision score of each class.	Calculation	average="macro"
average_precision_score_micro	Average precision summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight. Micro is computed globally by combining the true positives and false positives at each cutoff.	Calculation	average="micro"

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS
average_precision_score_weighted	Average precision summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight. Weighted is the arithmetic mean of the average precision score for each class, weighted by the number of true instances in each class.	Calculation	average="weighted"
balanced_accuracy	Balanced accuracy is the arithmetic mean of recall for each class.	Calculation	average="macro"
f1_score_macro	F1 score is the harmonic mean of precision and recall. Macro is the arithmetic mean of F1 score for each class.	Calculation	average="macro"
f1_score_micro	F1 score is the harmonic mean of precision and recall. Micro is computed globally by counting the total true positives, false negatives, and false positives.	Calculation	average="micro"
f1_score_weighted	F1 score is the harmonic mean of precision and recall. Weighted mean by class frequency of F1 score for each class	Calculation	average="weighted"
log_loss	This is the loss function used in (multinomial) logistic regression and extensions of it such as neural networks, defined as the negative log-likelihood of the true labels given a probabilistic classifier's predictions. For a single sample with true label yt in {0,1} and estimated probability yp that yt = 1, the log loss is -log P(yt|yp) = -(yt log(yp) + (1 - yt) log(1 - yp)).	Calculation	None

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS
norm_macro_recall	Normalized Macro Recall is Macro Recall normalized so that random performance has a score of 0 and perfect performance has a score of 1. This is achieved by norm_macro_recall := (recall_score_macro - R)/(1 - R), where R is the expected value of recall_score_macro for random predictions (i.e., R=0.5 for binary classification and R=(1/C) for C-class classification problems).	Calculation	average = "macro"
precision_score_macro	Precision is the percent of positively predicted elements that are correctly labeled. Macro is the arithmetic mean of precision for each class.	Calculation	average="macro"
precision_score_micro	Precision is the percent of positively predicted elements that are correctly labeled. Micro is computed globally by counting the total true positives and false positives.	Calculation	average="micro"
precision_score_weighted	Precision is the percent of positively predicted elements that are correctly labeled. Weighted is the arithmetic mean of precision for each class, weighted by number of true instances in each class.	Calculation	average="weighted"
recall_score_macro	Recall is the percent of correctly labeled elements of a certain class. Macro is the arithmetic mean of recall for each class.	Calculation	average="macro"
recall_score_micro	Recall is the percent of correctly labeled elements of a certain class. Micro is computed globally by counting the total true positives, false negatives and false positives	Calculation	average="micro"

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS
recall_score_weighted	Recall is the percent of correctly labeled elements of a certain class. Weighted is the arithmetic mean of recall for each class, weighted by number of true instances in each class.	Calculation	average="weighted"
weighted_accuracy	Weighted accuracy is accuracy where the weight given to each example is equal to the proportion of true instances in that example's true class.	Calculation	sample_weight is a vector equal to the proportion of that class for each element in the target

Confusion matrix

What is a confusion matrix?

A confusion matrix is used to describe the performance of a classification model. Each row displays the instances of the true, or actual class in your dataset, and each column represents the instances of the class that was predicted by the model.

What does automated ML do with the confusion matrix?

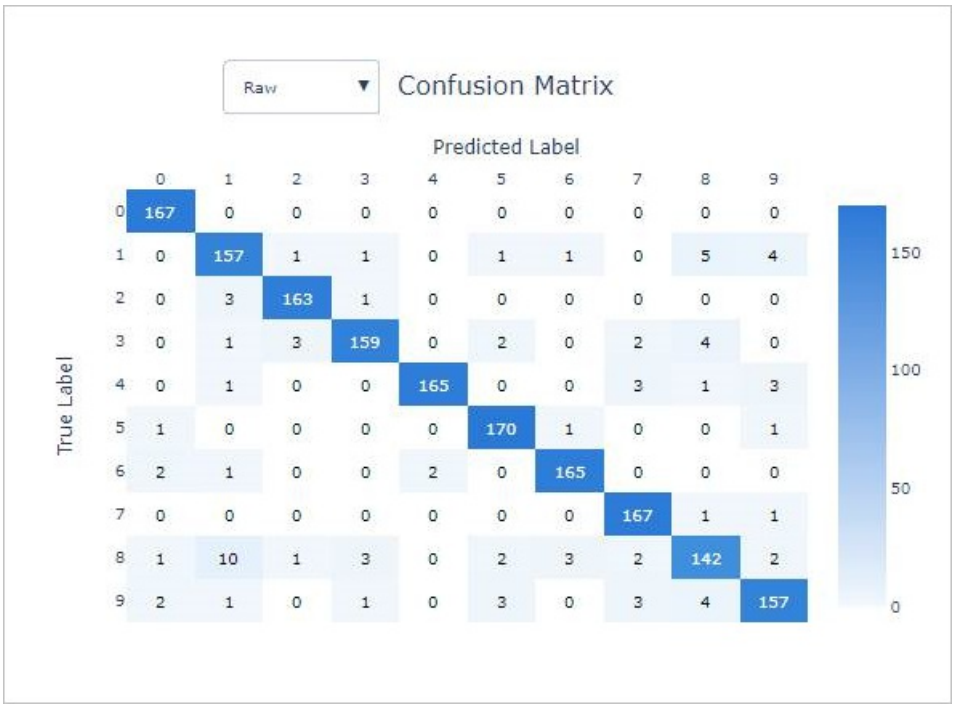
For classification problems, Azure Machine Learning automatically provides a confusion matrix for each model that is built. For each confusion matrix, automated ML will show the frequency of each predicted label (column) compared against the true label (row). The darker the color, the higher the count in that particular part of the matrix.

What does a good model look like?

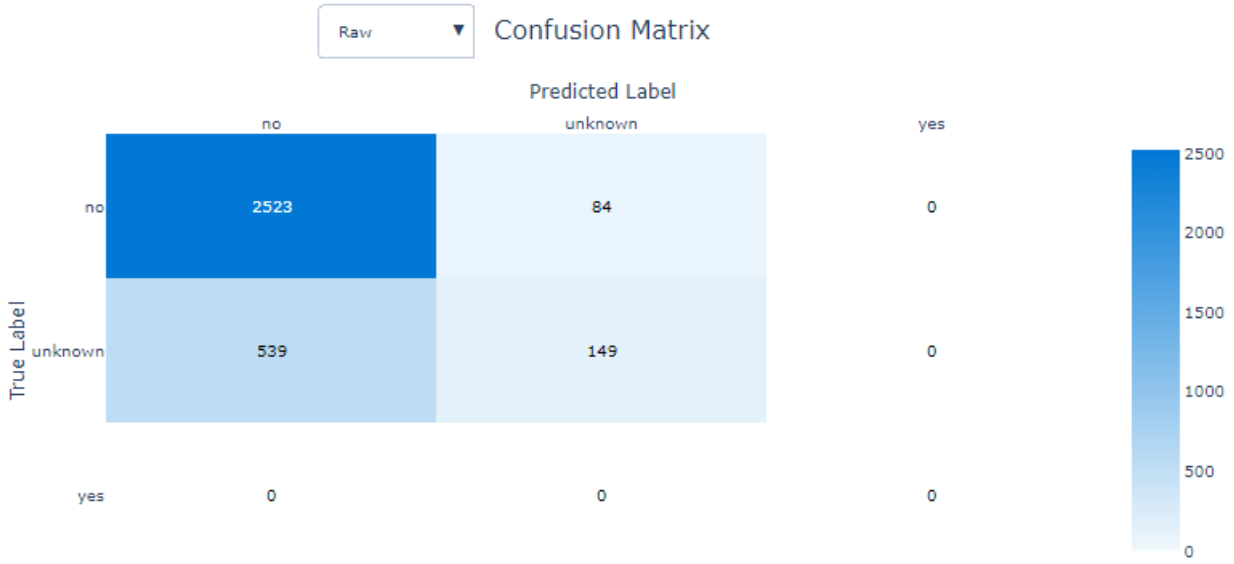
We are comparing the actual value of the dataset against the predicted values that the model gave. Because of this, machine learning models have higher accuracy if the model has most of its values along the diagonal, meaning the model predicted the correct value. If a model has class imbalance, the confusion matrix will help to detect a biased model.

Example 1: A classification model with poor accuracy

Example 2: A classification model with high accuracy



Example 3: A classification model with high accuracy and high bias in model predictions



Precision-recall chart

What is a precision-recall chart?

The precision-recall curve shows the relationship between precision and recall from a model. The term precision represents that ability for a model to label all instances correctly. Recall represents the ability for a classifier to find all instances of a particular label.

What does automated ML do with the precision-recall chart?

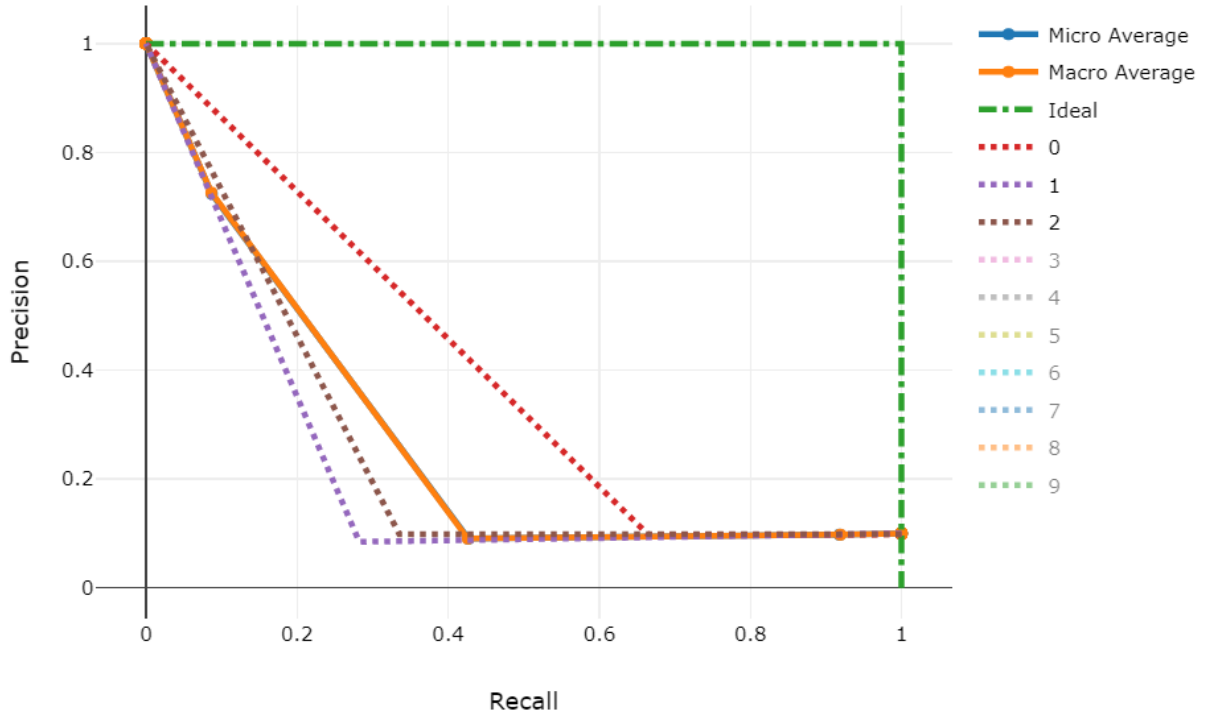
With this chart, you can compare the precision-recall curves for each model to determine which model has an acceptable relationship between precision and recall for your particular business problem. This chart shows Macro Average Precision-Recall, Micro Average Precision-Recall, and the precision-recall associated with all classes for a model.

Macro-average will compute the metric independently of each class and then take the average, treating all classes equally. However, micro-average will aggregate the contributions of all the classes to compute the average. Micro-average is preferable if there is class imbalance present in the dataset.

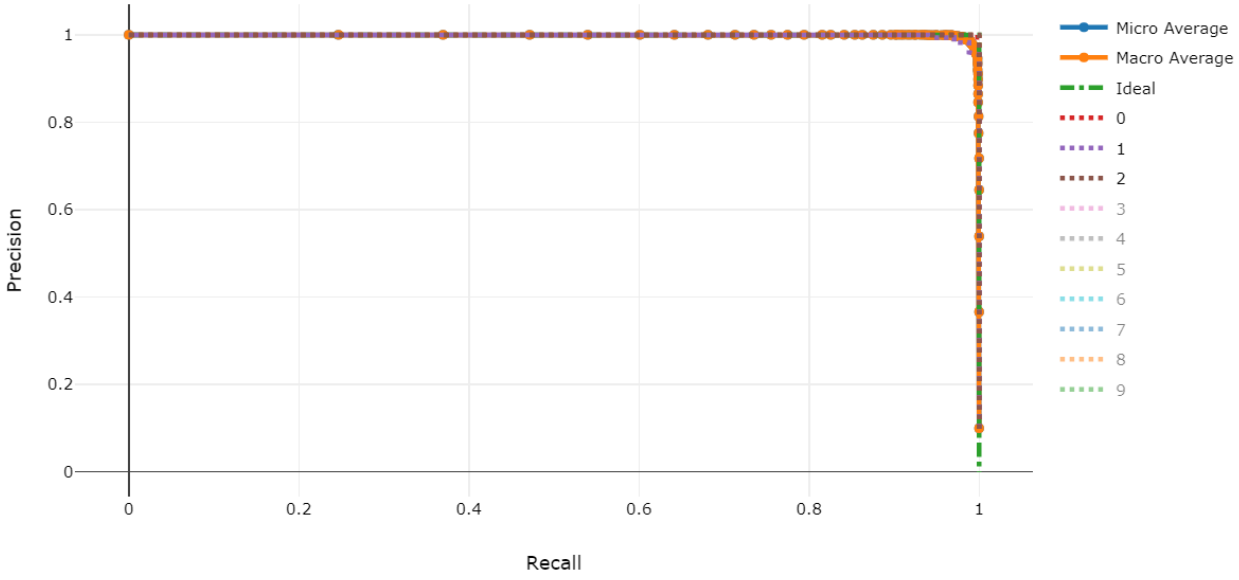
What does a good model look like?

Depending on the goal of the business problem, the ideal precision-recall curve could differ. Some examples are given below

Example 1: A classification model with low precision and low recall



Example 2: A classification model with ~100% precision and ~100% recall



ROC chart

What is a ROC chart?

Receiver operating characteristic (or ROC) is a plot of the correctly classified labels vs. the incorrectly classified labels for a particular model. The ROC curve can be less informative when training models on datasets with high bias, as it will not show the false positive labels.

What does automated ML do with the ROC chart?

Automated ML generates Macro Average Precision-Recall, Micro Average Precision-Recall, and the precision-recall associated with all classes for a model.

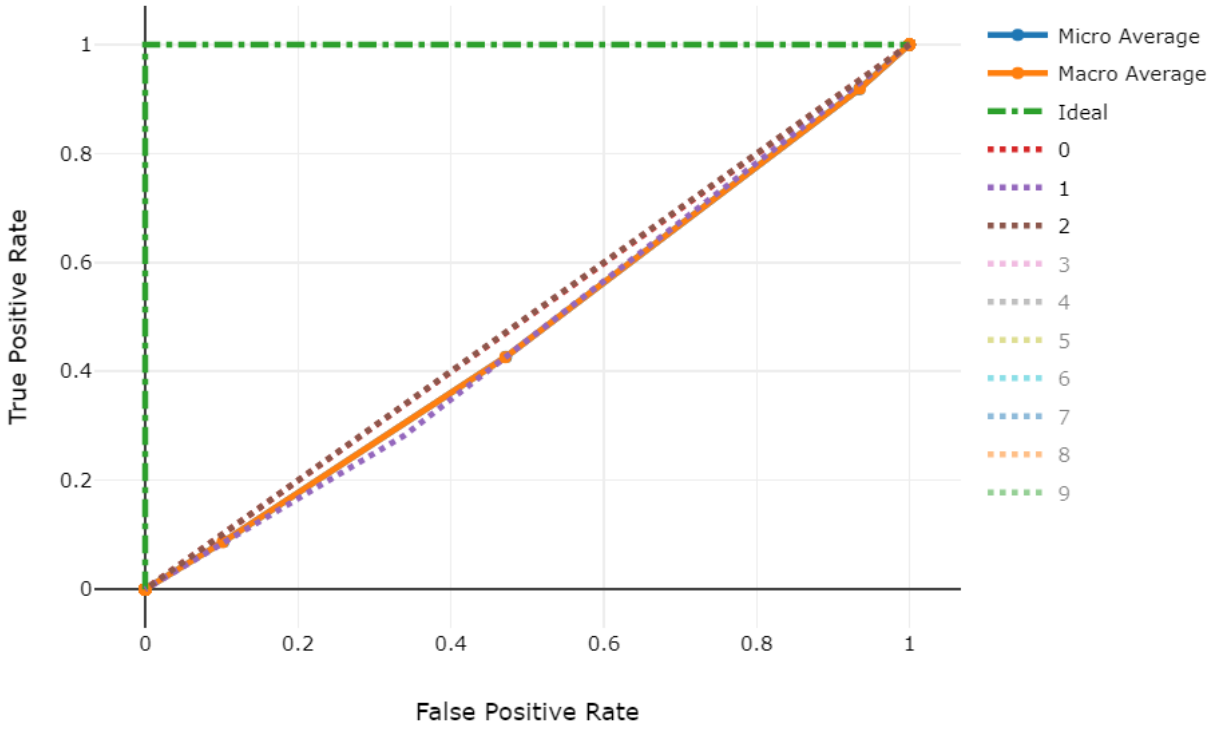
Macro-average will compute the metric independently of each class and then take the average, treating all classes equally. However, micro-average will aggregate the contributions of all the classes to compute the average. Micro-

average is preferable if there is class imbalance present in the dataset.

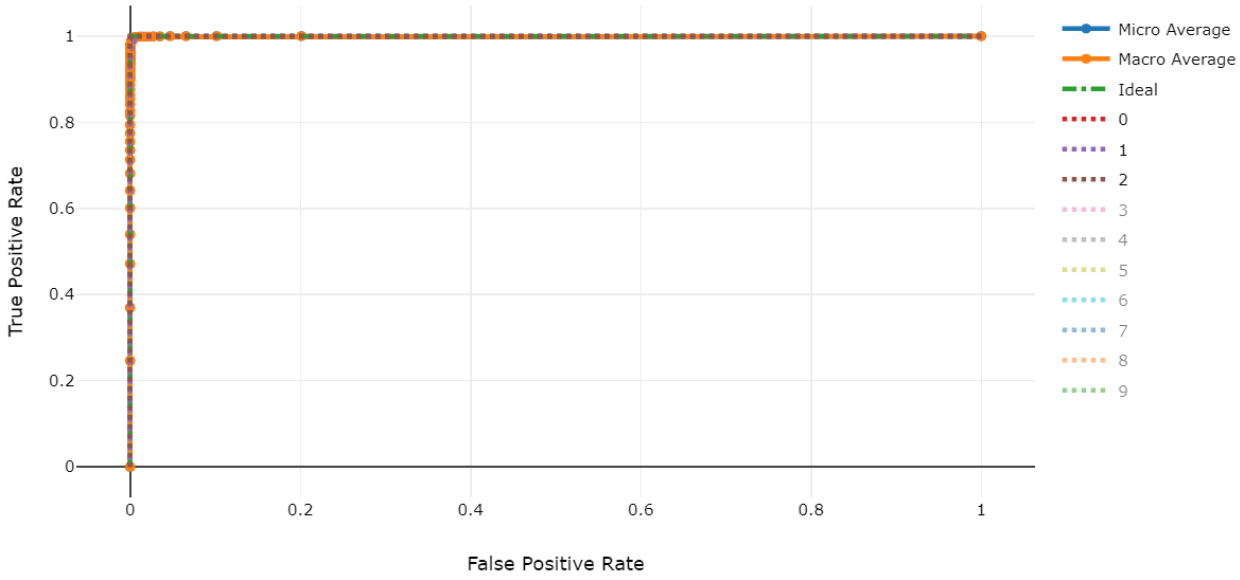
What does a good model look like?

Ideally, the model will have closer to 100% true positive rate and closer to 0% false positive rate.

Example 1: A classification model with low true labels and high false labels



Example 2: A classification model with high true labels and low false labels



Lift chart

What is a lift chart?

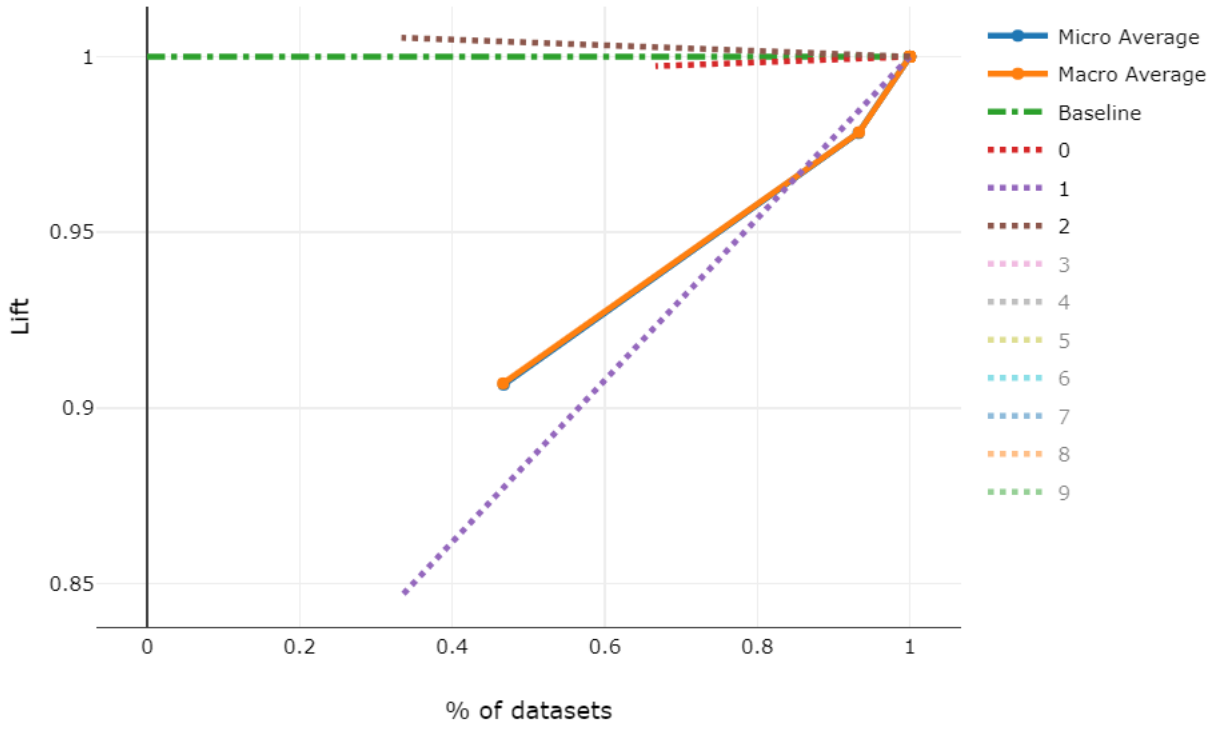
Lift charts are used to evaluate the performance of a classification model. It shows how much better you can expect to do with the generated model compared to without a model in terms of accuracy.

What does automated ML do with the lift chart?

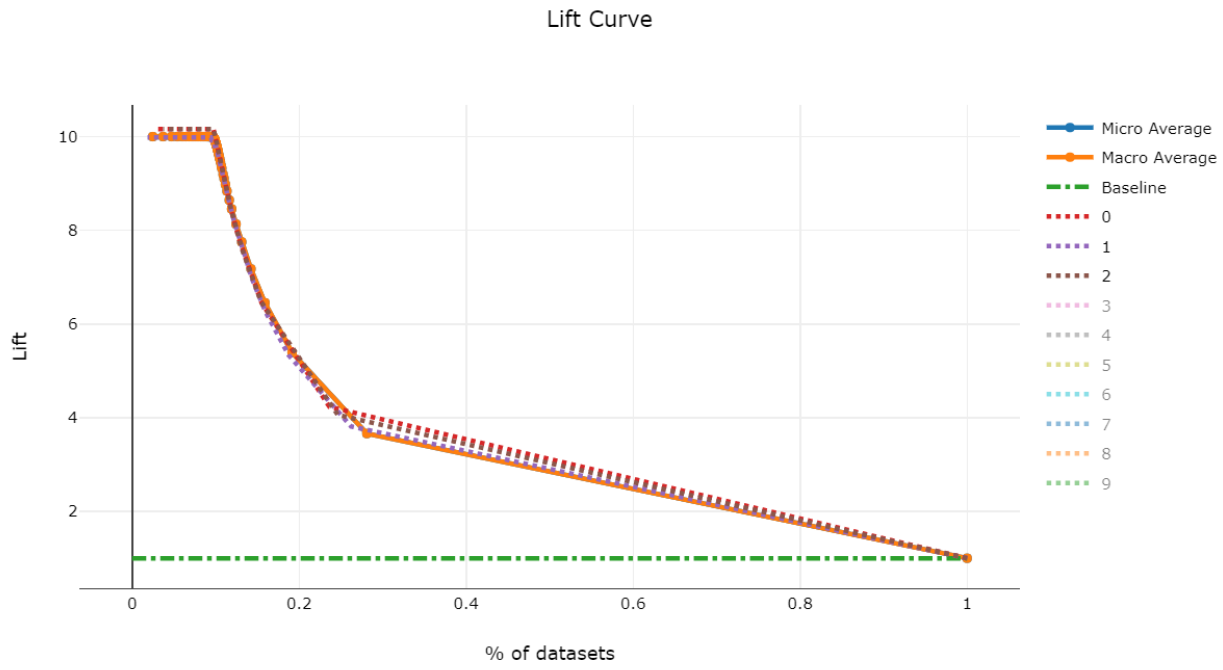
You can compare the lift of the model built automatically with Azure Machine Learning to the baseline in order to view the value gain of that particular model.

What does a good model look like?

Example 1: A classification model that does worse than a random selection model



Example 2: A classification model that performs better than a random selection model



Gains chart

What is a gains chart?

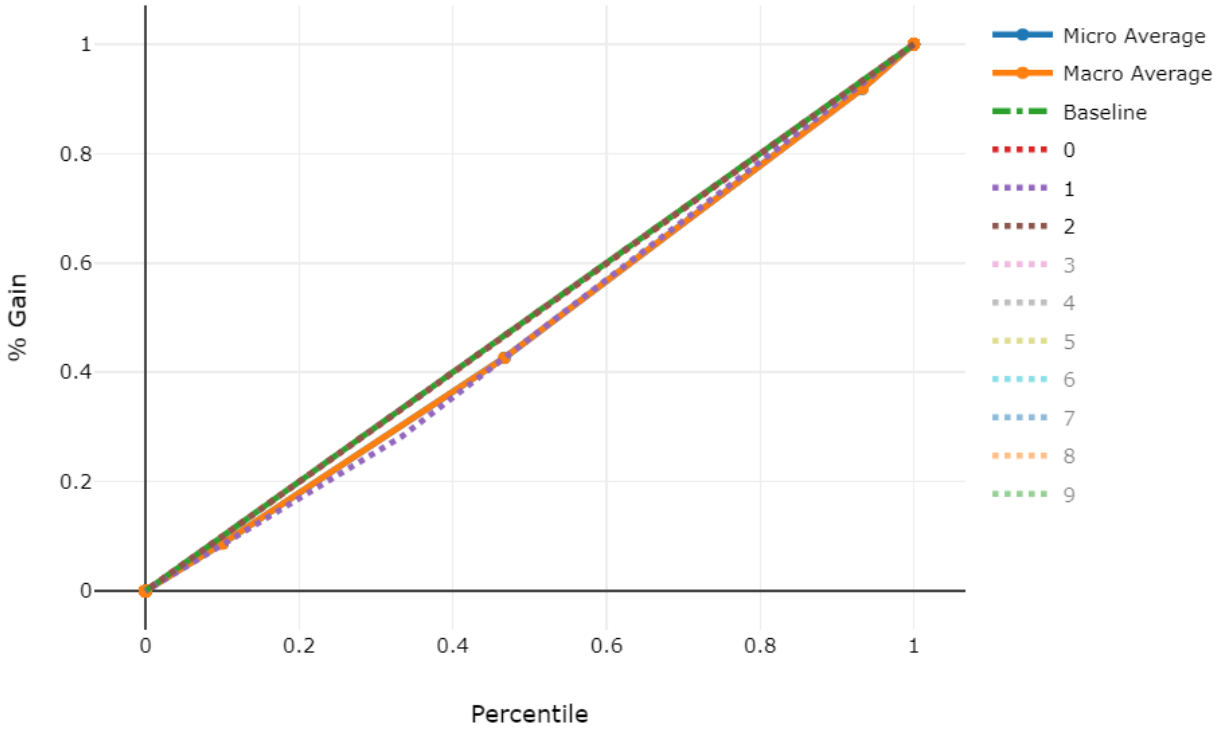
A gains chart evaluates the performance of a classification model by each portion of the data. It shows for each percentile of the data set, how much better you can expect to perform compared against a random selection model.

What does automated ML do with the gains chart?

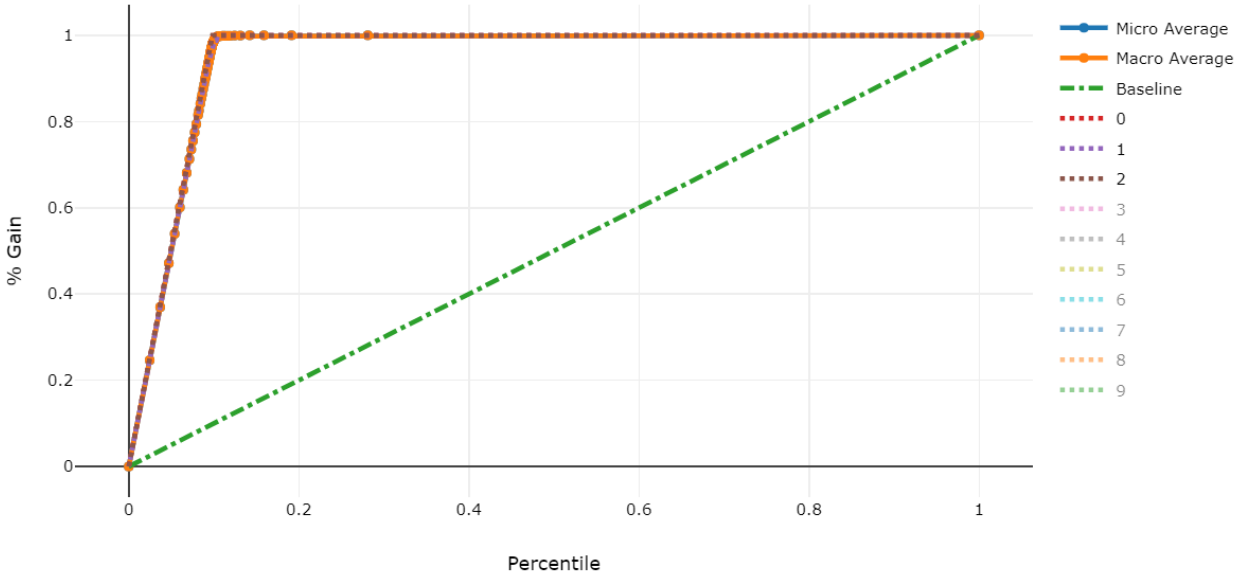
Use the cumulative gains chart to help you choose the classification cutoff using a percentage that corresponds to a desired gain from the model. This information provides another way of looking at the results in the accompanying lift chart.

What does a good model look like?

Example 1: A classification model with minimal gain



Example 2: A classification model with significant gain



Calibration chart

What is a calibration chart?

A calibration plot is used to display the confidence of a predictive model. It does this by showing the relationship between the predicted probability and the actual probability, where "probability" represents the likelihood that a particular instance belongs under some label.

What does automated ML do with the calibration chart?

For all classification problems, you can review the calibration line for micro-average, macro-average, and each class in a given predictive model.

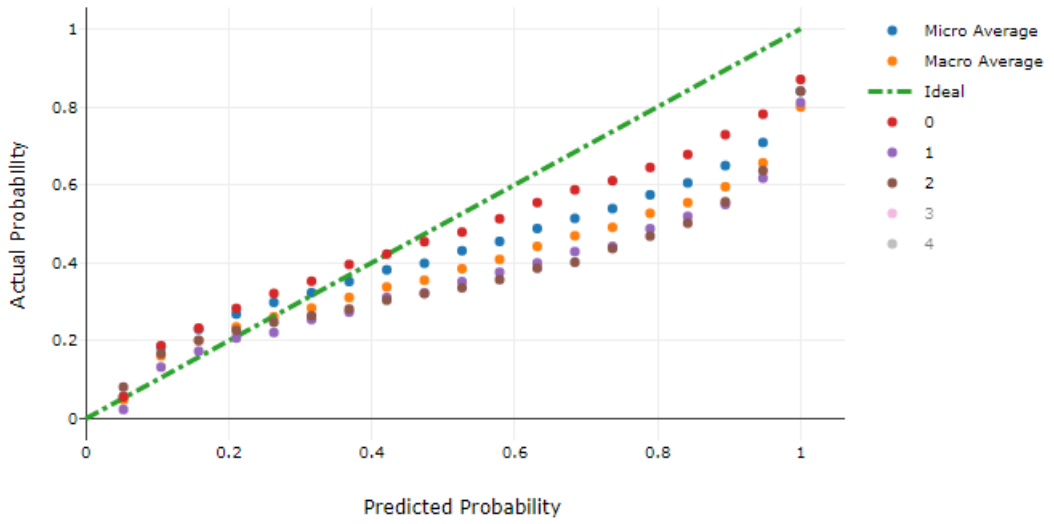
Macro-average will compute the metric independently of each class and then take the average, treating all classes equally. However, micro-average will aggregate the contributions of all the classes to compute the average.

What does a good model look like?

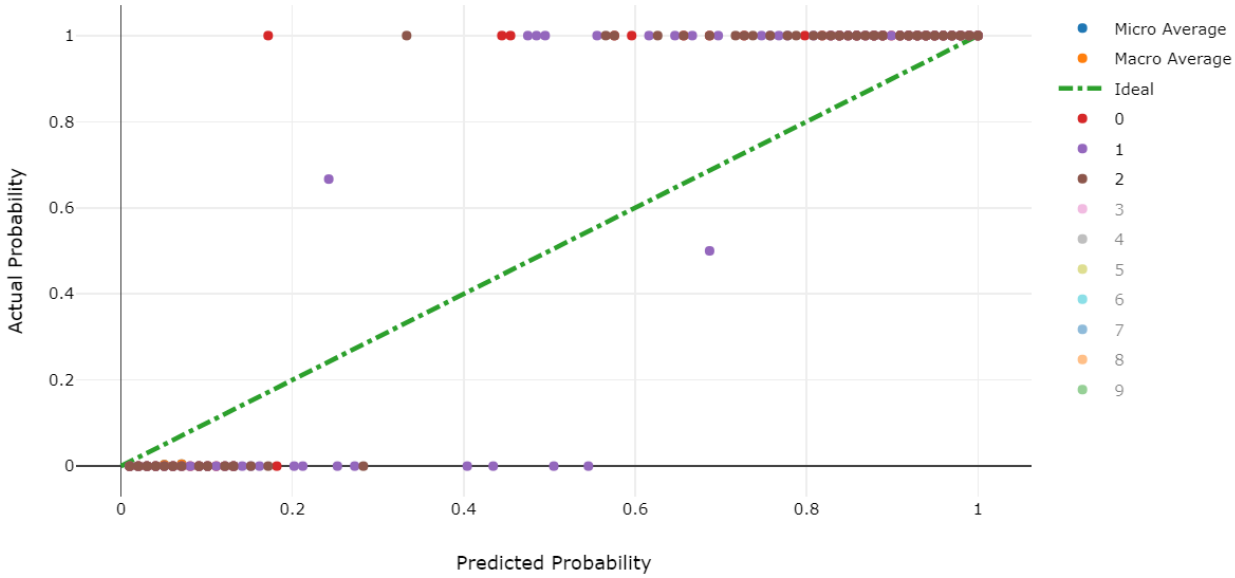
A well-calibrated model aligns with the y=x line, where it is reasonably confident in its predictions. An over-confident model aligns with the y=0 line, where the predicted probability is present but there is no actual

probability.

Example 1: A well-calibrated model



Example 2: An over-confident model



Regression results

Thee following metrics and charts are available for every regression model that you build using the automated machine learning capabilities of Azure Machine Learning

- Metrics
- Predicted vs. True
- Histogram of residuals

Regression metrics

The following metrics are saved in each run iteration for a regression or forecasting task.

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS
explained_variance	Explained variance is the proportion to which a mathematical model accounts for the variation of a given data set. It is the percent decrease in variance of the original data to the variance of the errors. When the mean of the errors is 0, it is equal to explained variance.	Calculation	None
r2_score	R2 is the coefficient of determination or the percent reduction in squared errors compared to a baseline model that outputs the mean.	Calculation	None
spearman_correlation	Spearman correlation is a nonparametric measure of the monotonicity of the relationship between two datasets. Unlike the Pearson correlation, the Spearman correlation does not assume that both datasets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.	Calculation	None
mean_absolute_error	Mean absolute error is the expected value of absolute value of difference between the target and the prediction	Calculation	None
normalized_mean_absolute_error	Normalized mean absolute error is mean Absolute Error divided by the range of the data	Calculation	Divide by range of the data
median_absolute_error	Median absolute error is the median of all absolute differences between the target and the prediction. This loss is robust to outliers.	Calculation	None

METRIC	DESCRIPTION	CALCULATION	EXTRA PARAMETERS
normalized_median_absolute_error	Normalized median absolute error is median absolute error divided by the range of the data	Calculation	Divide by range of the data
root_mean_squared_error	Root mean squared error is the square root of the expected squared difference between the target and the prediction	Calculation	None
normalized_root_mean_squared_error	Normalized root mean squared error is root mean squared error divided by the range of the data	Calculation	Divide by range of the data
root_mean_squared_log_error	Root mean squared log error is the square root of the expected squared logarithmic error	Calculation	None
normalized_root_mean_squared_log_error	Normalized Root mean squared log error is root mean squared log error divided by the range of the data	Calculation	Divide by range of the data

Predicted vs. True chart

What is a Predicted vs. True chart?

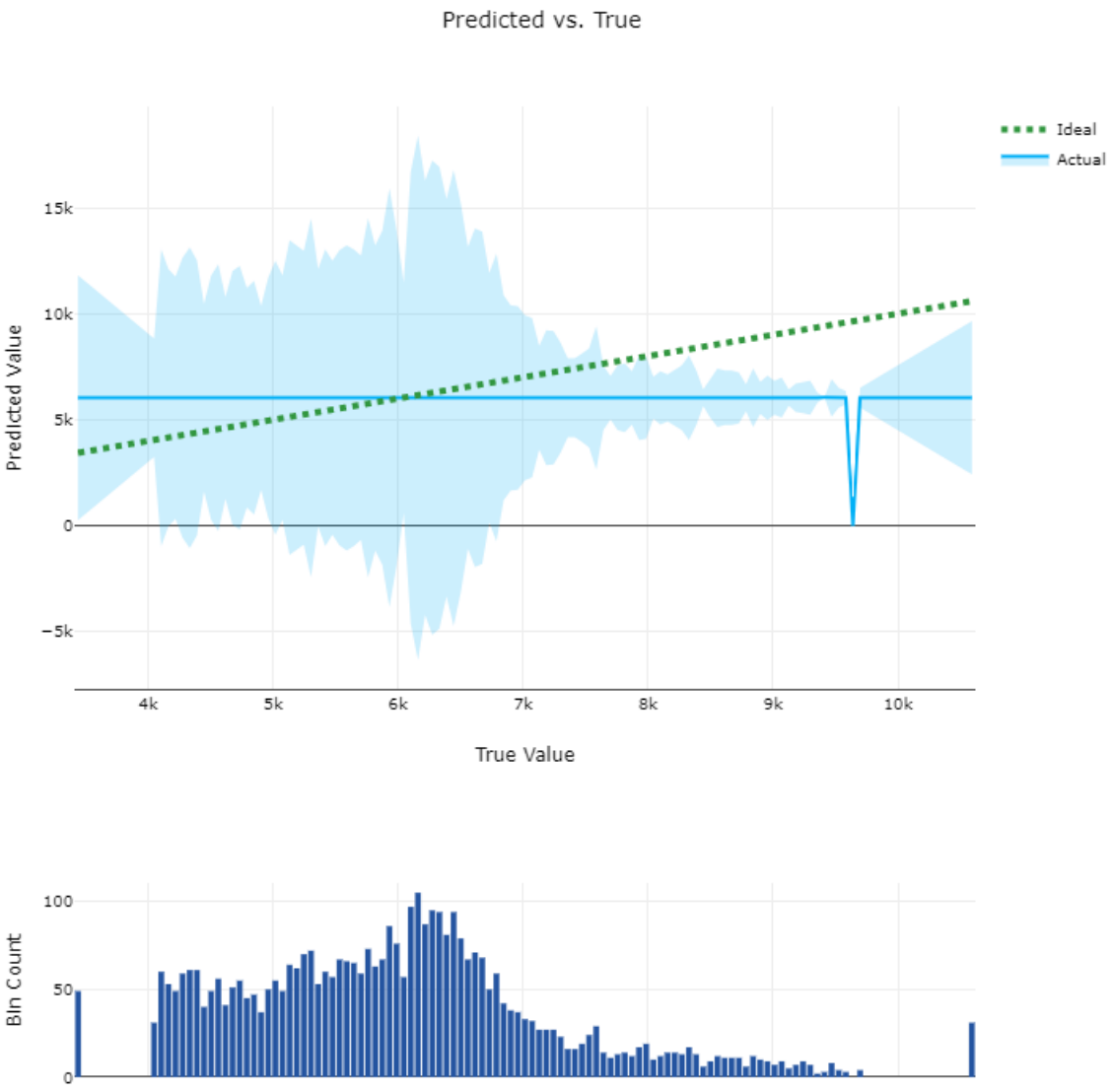
Predicted vs. True shows the relationship between a predicted value and its correlating true value for a regression problem. This graph can be used to measure performance of a model as the closer to the y=x line the predicted values are, the better the accuracy of a predictive model.

What does automated ML do with the Predicted vs. True chart?

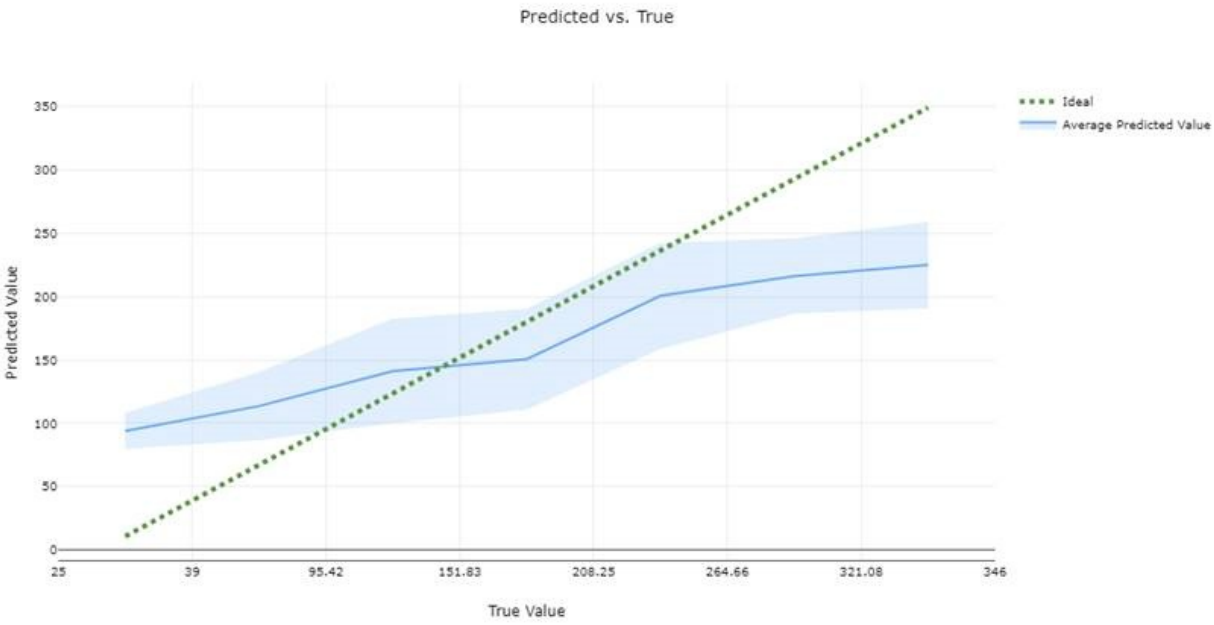
After each run, you can see a predicted vs. true graph for each regression model. To protect data privacy, values are binned together and the size of each bin is shown as a bar graph on the bottom portion of the chart area. You can compare the predictive model, with the lighter shade area showing error margins, against the ideal value of where the model should be.

What does a good model look like?

Example 1: A classification model with low accuracy



Example 2: A regression model with high accuracy



Histogram of residuals chart

What is a residuals chart?

A residual represents an observed y – the predicted y. To show a margin of error with low bias, the histogram of residuals should be shaped as a bell curve, centered around 0.

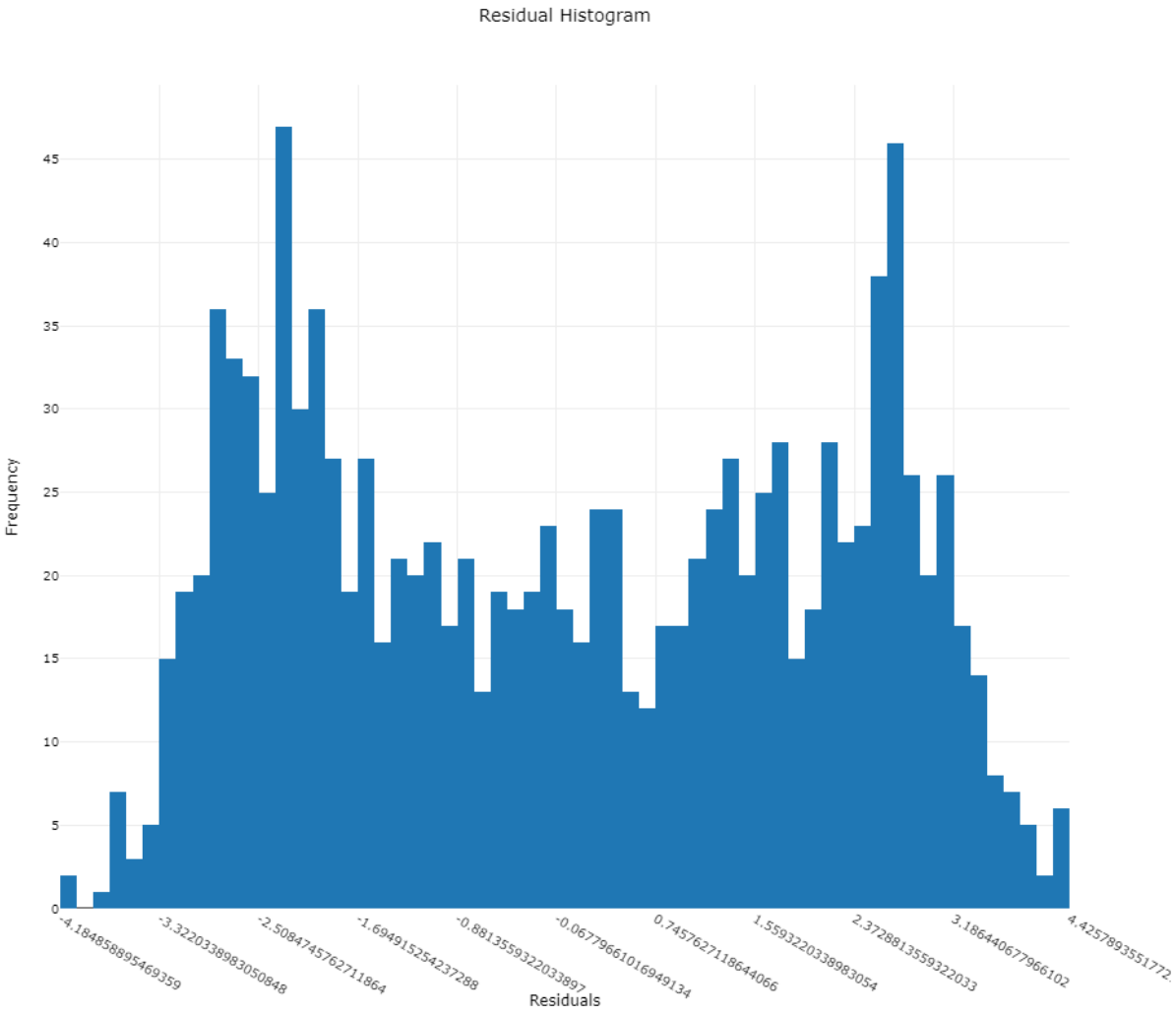
What does automated ML do with the residuals chart?

Automated ML automatically provides a residuals chart to show the distribution of errors in the predictions.

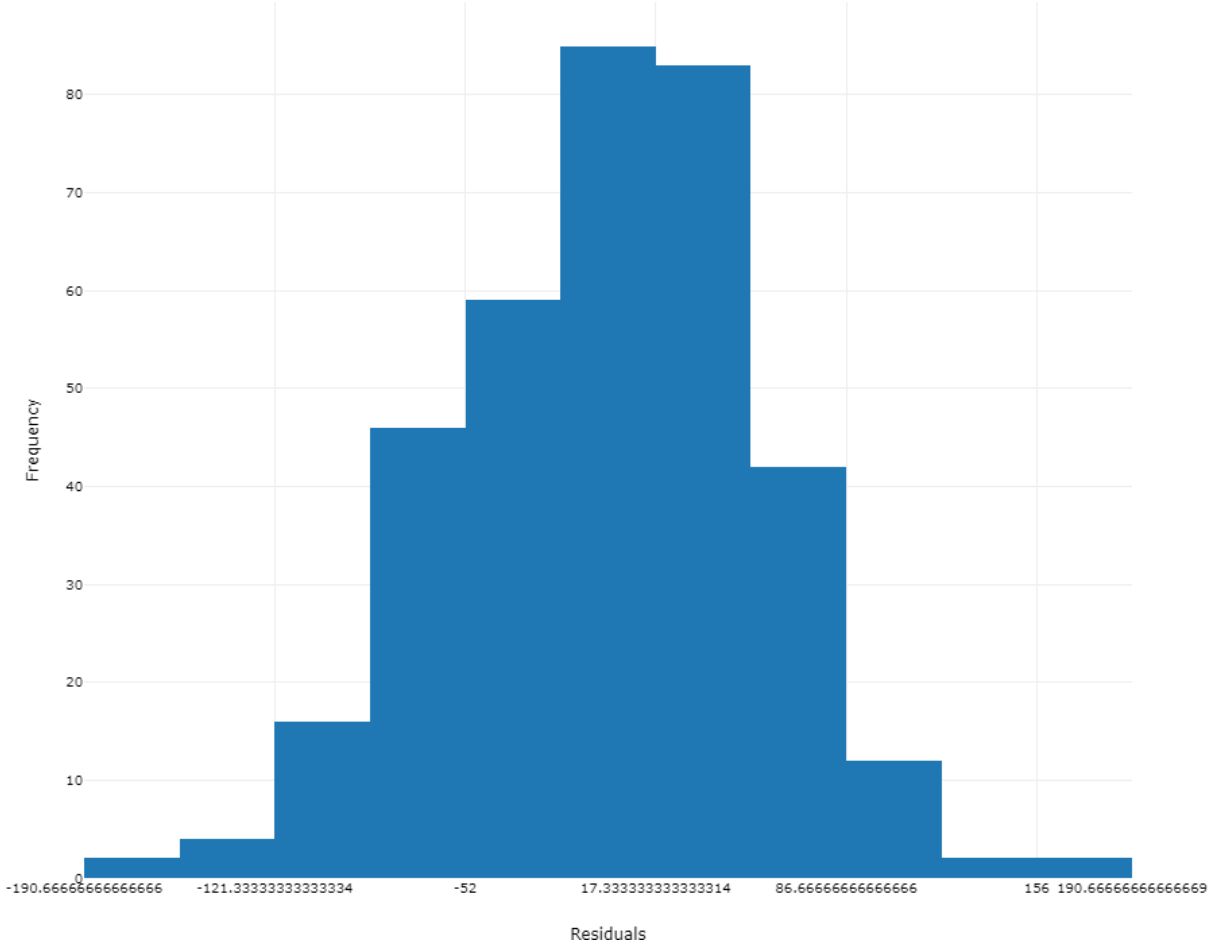
What does a good model look like?

A good model will typically have a bell curve or errors around zero.

Example 1: A regression model with bias in its errors



Example 2: A regression model with more even distribution of errors



Model interpretability and feature importance

Automated ML provides a machine learning interpretability dashboard for your runs. For more information on enabling interpretability features, see the how-to on enabling interpretability in automated ML experiments.

Next steps

- Learn more about automated ml in Azure Machine Learning.
- Try the Automated Machine Learning Model Explanation sample notebooks.

Start, monitor, and cancel training runs in Python

4/17/2020 • 9 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

The Azure Machine Learning SDK for Python, Machine Learning CLI, and Azure Machine Learning studio provide various methods to monitor, organize, and manage your runs for training and experimentation.

This article shows examples of the following tasks:

- Monitor run performance.
- Cancel or fail runs.
- Create child runs.
- Tag and find runs.

Prerequisites

You'll need the following items:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- An Azure Machine Learning workspace.
- The Azure Machine Learning SDK for Python (version 1.0.21 or later). To install or update to the latest version of the SDK, see Install or update the SDK.

  To check your version of the Azure Machine Learning SDK, use the following code:

  ```
  print(azureml.core.VERSION)
  ```

- The Azure CLI and CLI extension for Azure Machine Learning.

Start a run and its logging process

Using the SDK

Set up your experiment by importing the Workspace, Experiment, Run, and ScriptRunConfig classes from the azureml.core package.

```
import azureml.core
from azureml.core import Workspace, Experiment, Run
from azureml.core import ScriptRunConfig

ws = Workspace.from_config()
exp = Experiment(workspace=ws, name="explore-runs")
```

Start a run and its logging process with the `start_logging()` method.

```
notebook_run = exp.start_logging()
notebook_run.log(name="message", value="Hello from run!")
```

Using the CLI

To start a run of your experiment, use the following steps:

1. From a shell or command prompt, use the Azure CLI to authenticate to your Azure subscription:

   ```
   az login
   ```

   TIP

   After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.

   To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

2. Attach a workspace configuration to the folder that contains your training script. Replace `myworkspace` with your Azure Machine Learning workspace. Replace `myresourcegroup` with the Azure resource group that contains your workspace:

   ```
   az ml folder attach -w myworkspace -g myresourcegroup
   ```

   This command creates a `.azureml` subdirectory that contains example runconfig and conda environment files. It also contains a `config.json` file that is used to communicate with your Azure Machine Learning workspace.

   For more information, see az ml folder attach.

3. To start the run, use the following command. When using this command, specify the name of the runconfig file (the text before *.runconfig if you are looking at your file system) against the -c parameter.

   ```
   az ml run submit-script -c sklearn -e testexperiment train.py
   ```

   TIP

   The `az ml folder attach` command created a `.azureml` subdirectory, which contains two example runconfig files.

   If you have a Python script that creates a run configuration object programmatically, you can use RunConfig.save() to save it as a runconfig file.

   For more example runconfig files, see https://github.com/MicrosoftDocs/pipelines-azureml/tree/master/.azureml.

   For more information, see az ml run submit-script.

Using Azure Machine Learning studio

To start a submit a pipeline run in the designer (preview), use the following steps:

1. Set a default compute target for your pipeline.
2. Select **Run** at the top of the pipeline canvas.

3. Select an Experiment to group your pipeline runs.

Monitor the status of a run

Using the SDK

Get the status of a run with the `get_status()` method.

```
print(notebook_run.get_status())
```

To get the run ID, execution time, and additional details about the run, use the `get_details()` method.

```
print(notebook_run.get_details())
```

When your run finishes successfully, use the `complete()` method to mark it as completed.

```
notebook_run.complete()
print(notebook_run.get_status())
```

If you use Python's `with...as` design pattern, the run will automatically mark itself as completed when the run is out of scope. You don't need to manually mark the run as completed.

```
with exp.start_logging() as notebook_run:
    notebook_run.log(name="message", value="Hello from run!")
    print(notebook_run.get_status())

print(notebook_run.get_status())
```

Using the CLI

1. To view a list of runs for your experiment, use the following command. Replace `experiment` with the name of your experiment:

   ```
   az ml run list --experiment-name experiment
   ```

   This command returns a JSON document that lists information about runs for this experiment.

   For more information, see az ml experiment list.

2. To view information on a specific run, use the following command. Replace `runid` with the ID of the run:

   ```
   az ml run show -r runid
   ```

   This command returns a JSON document that lists information about the run.

   For more information, see az ml run show.

Using Azure Machine Learning studio

To view the number of active runs for your experiment in the studio.

1. Navigate to the **Experiments** section..
2. Select an experiment.

   In the experiment page, you can see the number of active compute targets and the duration for each run.

3. Select a specific run number.

4. In the **Logs** tab, you can find diagnostic and error logs for your pipeline run.

Cancel or fail runs

If you notice a mistake or if your run is taking too long to finish, you can cancel the run.

Using the SDK

To cancel a run using the SDK, use the `cancel()` method:

```
run_config = ScriptRunConfig(source_directory='.', script='hello_with_delay.py')
local_script_run = exp.submit(run_config)
print(local_script_run.get_status())

local_script_run.cancel()
print(local_script_run.get_status())
```

If your run finishes, but it contains an error (for example, the incorrect training script was used), you can use the `fail()` method to mark it as failed.

```
local_script_run = exp.submit(run_config)
local_script_run.fail()
print(local_script_run.get_status())
```

Using the CLI

To cancel a run using the CLI, use the following command. Replace `runid` with the ID of the run

```
az ml run cancel -r runid -w workspace_name -e experiment_name
```

For more information, see az ml run cancel.

Using Azure Machine Learning studio

To cancel a run in the studio, using the following steps:

1. Go to the running pipeline in either the **Experiments** or **Pipelines** section.

2. Select the pipeline run number you want to cancel.

3. In the toolbar, select **Cancel**

Create child runs

Create child runs to group together related runs, such as for different hyperparameter-tuning iterations.

> **NOTE**
> Child runs can only be created using the SDK.

This code example uses the `hello_with_children.py` script to create a batch of five child runs from within a submitted run by using the `child_run()` method:

```
!more hello_with_children.py
run_config = ScriptRunConfig(source_directory='.', script='hello_with_children.py')

local_script_run = exp.submit(run_config)
local_script_run.wait_for_completion(show_output=True)
print(local_script_run.get_status())

with exp.start_logging() as parent_run:
    for c,count in enumerate(range(5)):
        with parent_run.child_run() as child:
            child.log(name="Hello from child run", value=c)
```

> **NOTE**
> As they move out of scope, child runs are automatically marked as completed.

To create many child runs efficiently, use the `create_children()` method. Because each creation results in a network call, creating a batch of runs is more efficient than creating them one by one.

Submit child runs

Child runs can also be submitted from a parent run. This allows you to create hierarchies of parent and child runs.

You may wish your child runs to use a different run configuration than the parent run. For instance, you might use a less-powerful, CPU-based configuration for the parent, while using GPU-based configurations for your children. Another common desire is to pass each child different arguments and data. To customize a child run, pass a `RunConfiguration` object to the child's `ScriptRunConfig` constructor. This code example, which would be part of the parent `ScriptRunConfig` object's script:

- Creates a `RunConfiguration` retrieving a named compute resource `"gpu-compute"`
- Iterates over different argument values to be passed to the children `ScriptRunConfig` objects
- Creates and submits a new child run, using the custom compute resource and argument
- Blocks until all of the child runs complete

```
# parent.py
# This script controls the launching of child scripts
from azureml.core import Run, ScriptRunConfig, RunConfiguration

run_config_for_aml_compute = RunConfiguration()
run_config_for_aml_compute.target = "gpu-compute"
run_config_for_aml_compute.environment.docker.enabled = True

run = Run.get_context()

child_args = ['Apple', 'Banana', 'Orange']
for arg in child_args:
    run.log('Status', f'Launching {arg}')
    child_config = ScriptRunConfig(source_directory=".", script='child.py', arguments=['--fruit', arg],
run_config = run_config_for_aml_compute)
    # Starts the run asynchronously
    run.submit_child(child_config)

# Experiment will "complete" successfully at this point.
# Instead of returning immediately, block until child runs complete

for child in run.get_children():
    child.wait_for_completion()
```

To create many child runs with identical configurations, arguments, and inputs efficiently, use the

`create_children()` method. Because each creation results in a network call, creating a batch of runs is more efficient than creating them one by one.

Within a child run, you can view the parent run ID:

```
## In child run script
child_run = Run.get_context()
child_run.parent.id
```

Query child runs

To query the child runs of a specific parent, use the `get_children()` method. The `recursive = True` argument allows you to query a nested tree of children and grandchildren.

```
print(parent_run.get_children())
```

Tag and find runs

In Azure Machine Learning, you can use properties and tags to help organize and query your runs for important information.

Add properties and tags

Using the SDK

To add searchable metadata to your runs, use the `add_properties()` method. For example, the following code adds the `"author"` property to the run:

```
local_script_run.add_properties({"author":"azureml-user"})
print(local_script_run.get_properties())
```

Properties are immutable, so they create a permanent record for auditing purposes. The following code example results in an error, because we already added `"azureml-user"` as the `"author"` property value in the preceding code:

```
try:
    local_script_run.add_properties({"author":"different-user"})
except Exception as e:
    print(e)
```

Unlike properties, tags are mutable. To add searchable and meaningful information for consumers of your experiment, use the `tag()` method.

```
local_script_run.tag("quality", "great run")
print(local_script_run.get_tags())

local_script_run.tag("quality", "fantastic run")
print(local_script_run.get_tags())
```

You can also add simple string tags. When these tags appear in the tag dictionary as keys, they have a value of `None`.

```
local_script_run.tag("worth another look")
print(local_script_run.get_tags())
```

Using the CLI

> **NOTE**
> Using the CLI, you can only add or update tags.

To add or update a tag, use the following command:

```
az ml run update -r runid --add-tag quality='fantastic run'
```

For more information, see az ml run update.

Query properties and tags

You can query runs within an experiment to return a list of runs that match specific properties and tags.

Using the SDK

```
list(exp.get_runs(properties={"author":"azureml-user"},tags={"quality":"fantastic run"}))
list(exp.get_runs(properties={"author":"azureml-user"},tags="worth another look"))
```

Using the CLI

The Azure CLI supports JMESPath queries, which can be used to filter runs based on properties and tags. To use a JMESPath query with the Azure CLI, specify it with the `--query` parameter. The following examples show basic queries using properties and tags:

```
# list runs where the author property = 'azureml-user'
az ml run list --experiment-name experiment [?properties.author=='azureml-user']
# list runs where the tag contains a key that starts with 'worth another look'
az ml run list --experiment-name experiment [?tags.keys(@)[?starts_with(@, 'worth another look')]]
# list runs where the author property = 'azureml-user' and the 'quality' tag starts with 'fantastic run'
az ml run list --experiment-name experiment [?properties.author=='azureml-user' && tags.quality=='fantastic
run']
```

For more information on querying Azure CLI results, see Query Azure CLI command output.

Using Azure Machine Learning studio

1. Navigate to the **Pipelines** section.
2. Use the search bar to filter pipelines using tags, descriptions, experiment names, and submitter name.

Example notebooks

The following notebooks demonstrate the concepts in this article:

- To learn more about the logging APIs, see the logging API notebook.
- For more information about managing runs with the Azure Machine Learning SDK, see the manage runs notebook.

Next steps

- To learn how to log metrics for your experiments, see Log metrics during training runs.
- To learn how to monitor resources and logs from Azure Machine Learning, see Monitoring Azure Machine Learning.

Monitor Azure ML experiment runs and metrics

3/13/2020 • 11 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Enhance the model creation process by tracking your experiments and monitoring run metrics. In this article, learn how to add logging code to your training script, submit an experiment run, monitor that run, and inspect the results in Azure Machine Learning.

NOTE

Azure Machine Learning may also log information from other sources during training, such as automated machine learning runs, or the Docker container that runs the training job. These logs are not documented. If you encounter problems and contact Microsoft support, they may be able to use these logs during troubleshooting.

TIP

The information in this document is primarily for data scientists and developers who want to monitor the model training process. If you are an administrator interested in monitoring resource usage and events from Azure Machine learning, such as quotas, completed training runs, or completed model deployments, see Monitoring Azure Machine Learning.

Available metrics to track

The following metrics can be added to a run while training an experiment. To view a more detailed list of what can be tracked on a run, see the Run class reference documentation.

TYPE	PYTHON FUNCTION	NOTES
Scalar values	Function: `run.log(name, value, description='')` Example: run.log("accuracy", 0.95)	Log a numerical or string value to the run with the given name. Logging a metric to a run causes that metric to be stored in the run record in the experiment. You can log the same metric multiple times within a run, the result being considered a vector of that metric.
Lists	Function: `run.log_list(name, value, description='')` Example: run.log_list("accuracies", [0.6, 0.7, 0.87])	Log a list of values to the run with the given name.
Row	Function: `run.log_row(name, description=None, **kwargs)` Example: run.log_row("Y over X", x=1, y=0.4)	Using *log_row* creates a metric with multiple columns as described in kwargs. Each named parameter generates a column with the value specified. *log_row* can be called once to log an arbitrary tuple, or multiple times in a loop to generate a complete table.

TYPE	PYTHON FUNCTION	NOTES
Table	Function: `run.log_table(name, value, description='')` Example: run.log_table("Y over X", {"x":[1, 2, 3], "y":[0.6, 0.7, 0.89]})	Log a dictionary object to the run with the given name.

Images	Function:
	`run.log_image(name, path=None, plot=None)`
	
	Example:
	`run.log_image("ROC", plot=plt)`
		Log an image to the run record. Use log_image to log an image file or a matplotlib plot to the run. These images will be visible and comparable in the run record.

Tag a run	Function: `run.tag(key, value=None)` Example: run.tag("selected", "yes")	Tag the run with a string key and optional string value.
Upload file or directory	Function: `run.upload_file(name, path_or_stream)` Example: run.upload_file("best_model.pkl", "./model.pkl")	Upload a file to the run record. Runs automatically capture file in the specified output directory, which defaults to "./outputs" for most run types. Use upload_file only when additional files need to be uploaded or an output directory is not specified. We suggest adding `outputs` to the name so that it gets uploaded to the outputs directory. You can list all of the files that are associated with this run record by called `run.get_file_names()`

NOTE

Metrics for scalars, lists, rows, and tables can have type: float, integer, or string.

Choose a logging option

If you want to track or monitor your experiment, you must add code to start logging when you submit the run. The following are ways to trigger the run submission:

- **Run.start_logging** - Add logging functions to your training script and start an interactive logging session in the specified experiment. **start_logging** creates an interactive run for use in scenarios such as notebooks. Any metrics that are logged during the session are added to the run record in the experiment.
- **ScriptRunConfig** - Add logging functions to your training script and load the entire script folder with the run. **ScriptRunConfig** is a class for setting up configurations for script runs. With this option, you can add monitoring code to be notified of completion or to get a visual widget to monitor.

Set up the workspace

Before adding logging and submitting an experiment, you must set up the workspace.

1. Load the workspace. To learn more about setting the workspace configuration, see workspace configuration

file.

```
import azureml.core
from azureml.core import Experiment, Workspace

# Check core SDK version number
print("This notebook was created using version 1.0.2 of the Azure ML SDK")
print("You are currently using version", azureml.core.VERSION, "of the Azure ML SDK")
print("")


ws = Workspace.from_config()
print('Workspace name: ' + ws.name,
      'Azure region: ' + ws.location,
      'Subscription id: ' + ws.subscription_id,
      'Resource group: ' + ws.resource_group, sep='\n')
```

Option 1: Use start_logging

start_logging creates an interactive run for use in scenarios such as notebooks. Any metrics that are logged during the session are added to the run record in the experiment.

The following example trains a simple sklearn Ridge model locally in a local Jupyter notebook. To learn more about submitting experiments to different environments, see Set up compute targets for model training with Azure Machine Learning.

Load the data

This example uses the diabetes dataset, a well-known small dataset that comes with scikit-learn. This cell loads the dataset and splits it into random training and testing sets.

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.externals import joblib

X, y = load_diabetes(return_X_y = True)
columns = ['age', 'gender', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
data = {
    "train":{"X": X_train, "y": y_train},
    "test":{"X": X_test, "y": y_test}
}

print ("Data contains", len(data['train']['X']), "training samples and",len(data['test']['X']), "test
samples")
```

Add tracking

Add experiment tracking using the Azure Machine Learning SDK, and upload a persisted model into the experiment run record. The following code adds tags, logs, and uploads a model file to the experiment run.

```
# Get an experiment object from Azure Machine Learning
experiment = Experiment(workspace=ws, name="train-within-notebook")

# Create a run object in the experiment
run =  experiment.start_logging()
# Log the algorithm parameter alpha to the run
run.log('alpha', 0.03)

# Create, fit, and test the scikit-learn Ridge regression model
regression_model = Ridge(alpha=0.03)
regression_model.fit(data['train']['X'], data['train']['y'])
preds = regression_model.predict(data['test']['X'])

# Output the Mean Squared Error to the notebook and to the run
print('Mean Squared Error is', mean_squared_error(data['test']['y'], preds))
run.log('mse', mean_squared_error(data['test']['y'], preds))

# Save the model to the outputs directory for capture
model_file_name = 'outputs/model.pkl'

joblib.dump(value = regression_model, filename = model_file_name)

# upload the model file explicitly into artifacts
run.upload_file(name = model_file_name, path_or_stream = model_file_name)

# Complete the run
run.complete()
```

The script ends with `run.complete()`, which marks the run as completed. This function is typically used in interactive notebook scenarios.

Option 2: Use ScriptRunConfig

ScriptRunConfig is a class for setting up configurations for script runs. With this option, you can add monitoring code to be notified of completion or to get a visual widget to monitor.

This example expands on the basic sklearn Ridge model from above. It does a simple parameter sweep to sweep over alpha values of the model to capture metrics and trained models in runs under the experiment. The example runs locally against a user-managed environment.

1. Create a training script `train.py`.

```
# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license.

from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from azureml.core.run import Run
from sklearn.externals import joblib
import os
import numpy as np
import mylib

os.makedirs('./outputs', exist_ok=True)

X, y = load_diabetes(return_X_y=True)

run = Run.get_context()

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=0)
data = {"train": {"X": X_train, "y": y_train},
        "test": {"X": X_test, "y": y_test}}

# list of numbers from 0.0 to 1.0 with a 0.05 interval
alphas = mylib.get_alphas()

for alpha in alphas:
    # Use Ridge algorithm to create a regression model
    reg = Ridge(alpha=alpha)
    reg.fit(data["train"]["X"], data["train"]["y"])

    preds = reg.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    run.log('alpha', alpha)
    run.log('mse', mse)

    model_file_name = 'ridge_{0:.2f}.pkl'.format(alpha)
    # save model in the outputs folder so it automatically get uploaded
    with open(model_file_name, "wb") as file:
        joblib.dump(value=reg, filename=os.path.join('./outputs/',
                                                     model_file_name))

    print('alpha is {0:.2f}, and mse is {1:0.2f}'.format(alpha, mse))
```

2. The `train.py` script references `mylib.py` which allows you to get the list of alpha values to use in the ridge model.

```
# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license.

import numpy as np


def get_alphas():
    # list of numbers from 0.0 to 1.0 with a 0.05 interval
    return np.arange(0.0, 1.0, 0.05)
```

3. Configure a user-managed local environment.

```
from azureml.core import Environment

# Editing a run configuration property on-fly.
user_managed_env = Environment("user-managed-env")

user_managed_env.python.user_managed_dependencies = True

# You can choose a specific Python environment by pointing to a Python path
#user_managed_env.python.interpreter_path = '/home/johndoe/miniconda3/envs/myenv/bin/python'
```

4. Submit the `train.py` script to run in the user-managed environment. This whole script folder is submitted for training, including the `mylib.py` file.

```
from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory='./', script='train.py')
src.run_config.environment = user_managed_env
```

```
run = exp.submit(src)
```

Manage a run

The Start, monitor, and cancel training runs article highlights specific Azure Machine Learning workflows for how to manage your experiments.

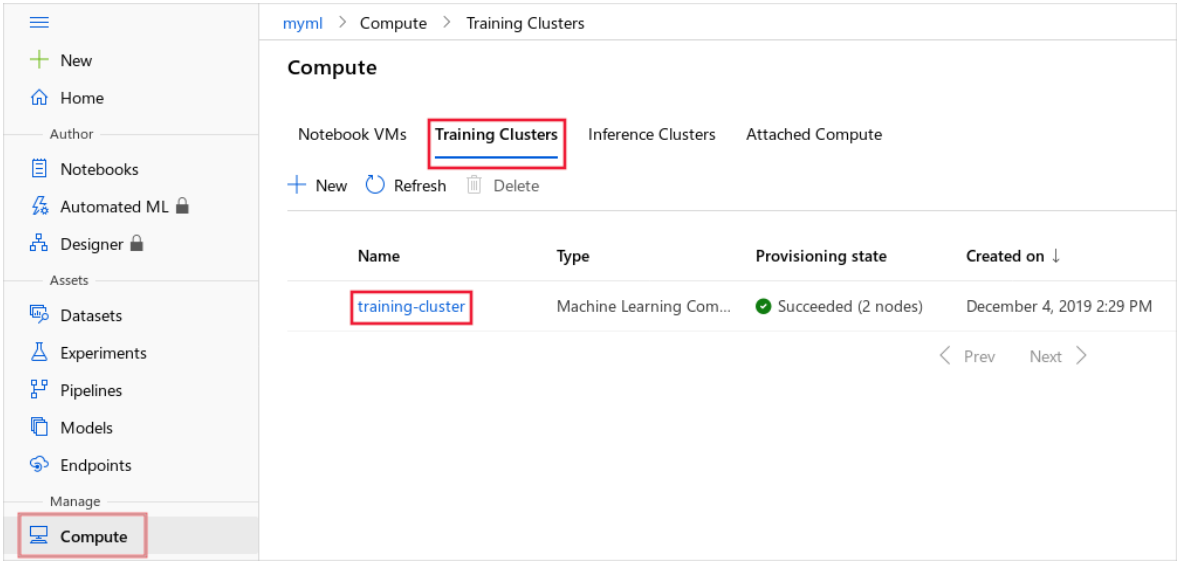
View run details

View active/queued runs from the browser

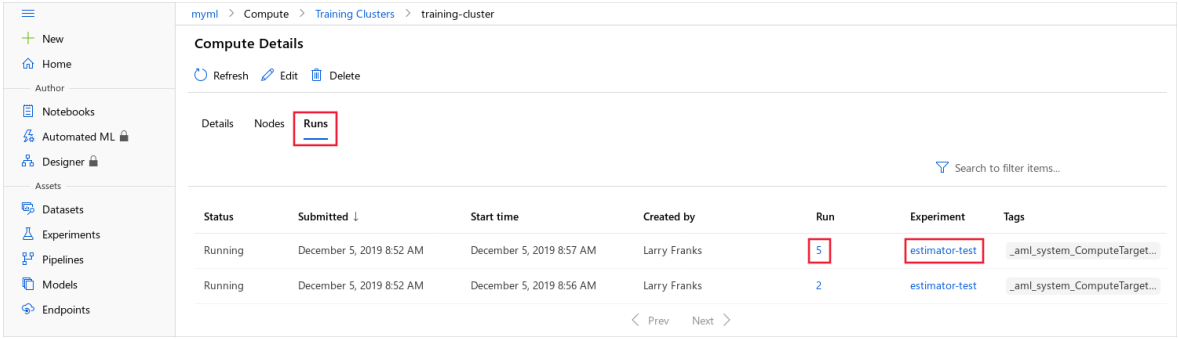
Compute targets used to train models are a shared resource. As such, they may have multiple runs queued or active at a given time. To see the runs for a specific compute target from your browser, use the following steps:

1. From the Azure Machine Learning studio, select your workspace, and then select **Compute** from the left side of the page.

2. Select **Training Clusters** to display a list of compute targets used for training. Then select the cluster.



3. Select **Runs**. The list of runs that use this cluster is displayed. To view details for a specific run, use the link in the **Run** column. To view details for the experiment, use the link in the **Experiment** column.



> **TIP**
>
> A run can contain child runs, so one training job can result in multiple entries.

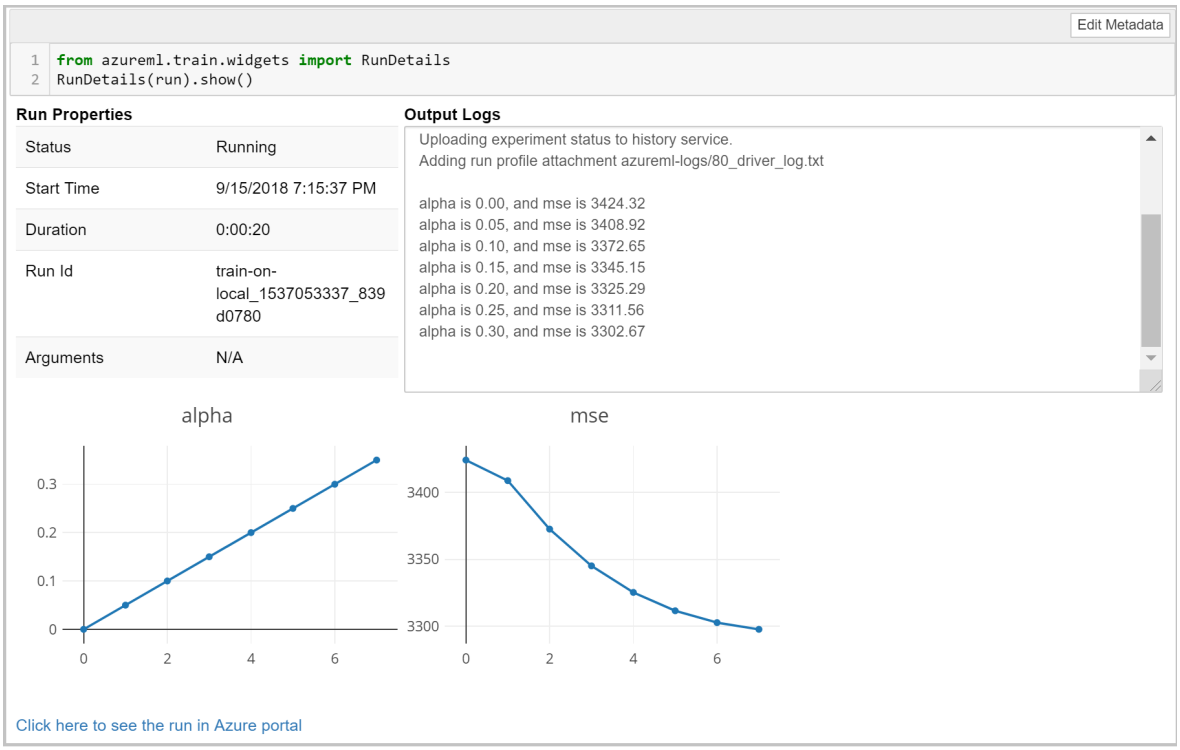
Once a run completes, it is no longer displayed on this page. To view information on completed runs, visit the **Experiments** section of the studio and select the experiment and run. For more information, see the Query run metrics section.

Monitor run with Jupyter notebook widget

When you use the **ScriptRunConfig** method to submit runs, you can watch the progress of the run with a Jupyter widget. Like the run submission, the widget is asynchronous and provides live updates every 10-15 seconds until the job completes.

1. View the Jupyter widget while waiting for the run to complete.

```
from azureml.widgets import RunDetails
RunDetails(run).show()
```



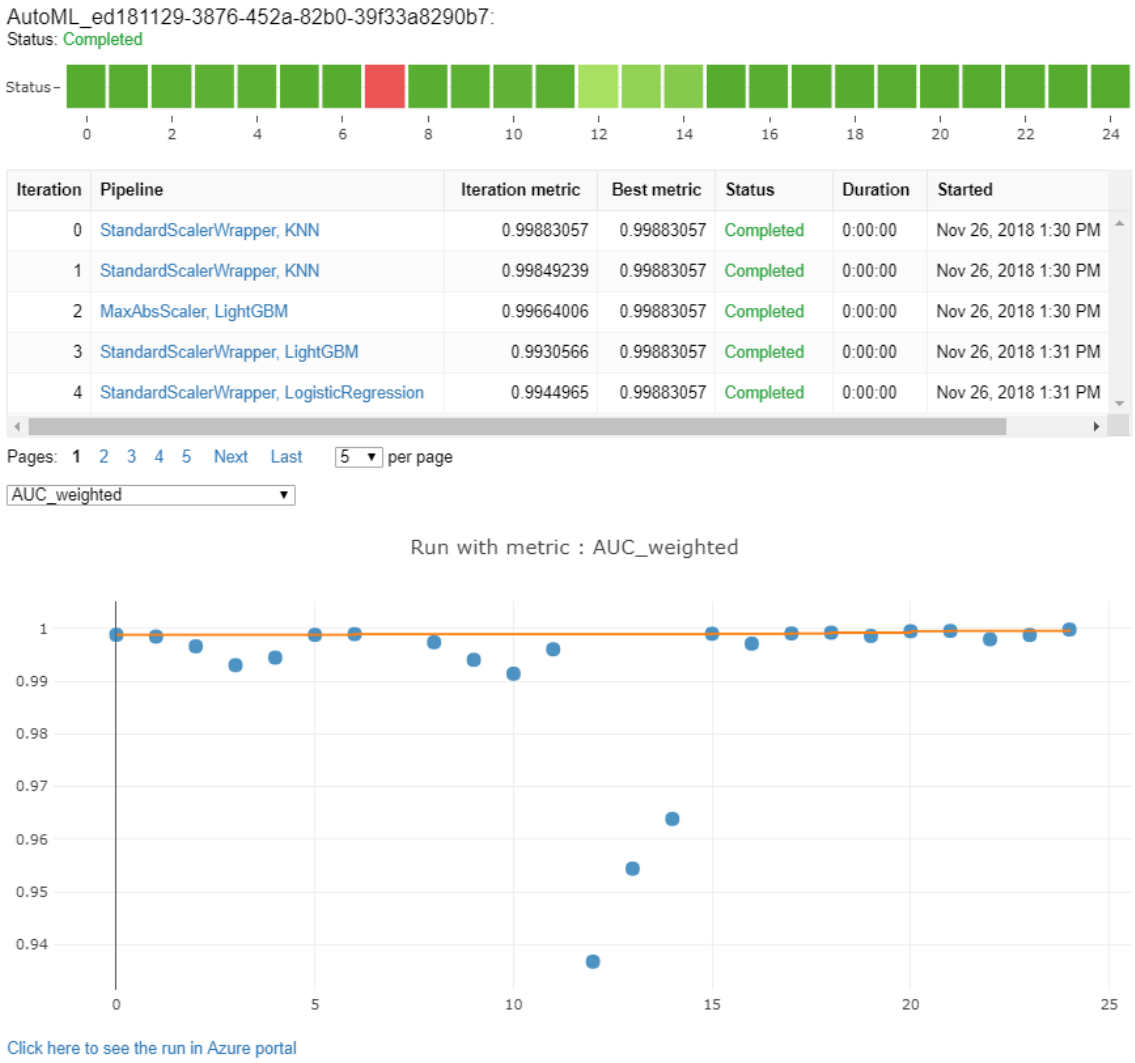
You can also get a link to the same display in your workspace.

```
print(run.get_portal_url())
```

2. **[For automated machine learning runs]** To access the charts from a previous run. Replace

`<<experiment_name>>` with the appropriate experiment name:

```
from azureml.widgets import RunDetails
from azureml.core.run import Run

experiment = Experiment (workspace, <<experiment_name>>)
run_id = 'autoML_my_runID' #replace with run_ID
run = Run(experiment, run_id)
RunDetails(run).show()
```



To view further details of a pipeline click on the Pipeline you would like to explore in the table, and the charts will render in a pop-up from the Azure Machine Learning studio.

Get log results upon completion

Model training and monitoring occur in the background so that you can run other tasks while you wait. You can also wait until the model has completed training before running more code. When you use **ScriptRunConfig**, you can use `run.wait_for_completion(show_output = True)` to show when the model training is complete. The `show_output` flag gives you verbose output.

Query run metrics

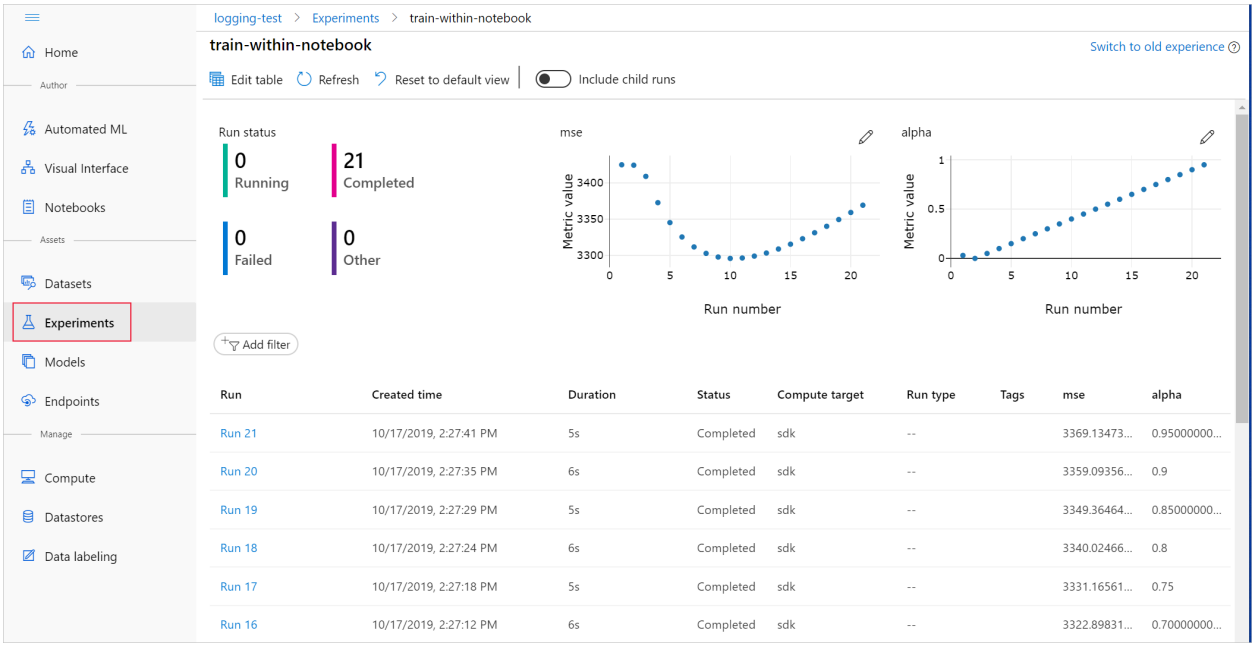
You can view the metrics of a trained model using `run.get_metrics()`. You can now get all of the metrics that were logged in the example above to determine the best model.

View the experiment in your workspace in Azure Machine Learning

studio

When an experiment has finished running, you can browse to the recorded experiment run record. You can access the history from the Azure Machine Learning studio.

Navigate to the Experiments tab and select your experiment. You are brought to the experiment run dashboard, where you can see tracked metrics and charts that are logged for each run. In this case, we logged MSE and the alpha values.



You can drill down to a specific run to view its outputs or logs, or download the snapshot of the experiment you submitted so you can share the experiment folder with others.

Viewing charts in run details

There are various ways to use the logging APIs to record different types of metrics during a run and view them as charts in Azure Machine Learning studio.

LOGGED VALUE	EXAMPLE CODE	VIEW IN PORTAL
Log an array of numeric values	`run.log_list(name='Fibonacci', value=[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89])`	single-variable line chart
Log a single numeric value with the same metric name repeatedly used (like from within a for loop)	`for i in tqdm(range(-10, 10)): run.log(name='Sigmoid', value=1 / (1 + np.exp(-i))) angle = i / 2.0`	Single-variable line chart
Log a row with 2 numerical columns repeatedly	`run.log_row(name='Cosine Wave', angle=angle, cos=np.cos(angle)) sines['angle'].append(angle) sines['sine'].append(np.sin(angle))`	Two-variable line chart
Log table with 2 numerical columns	`run.log_table(name='Sine Wave', value=sines)`	Two-variable line chart

Example notebooks

The following notebooks demonstrate concepts in this article:

- how-to-use-azureml/training/train-within-notebook

- how-to-use-azureml/training/train-on-local
- how-to-use-azureml/track-and-monitor-experiments/logging-api

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Next steps

Try these next steps to learn how to use the Azure Machine Learning SDK for Python:

- See an example of how to register the best model and deploy it in the tutorial, Train an image classification model with Azure Machine Learning.
- Learn how to Train PyTorch Models with Azure Machine Learning.

Track models metrics with MLflow and Azure Machine Learning (preview)

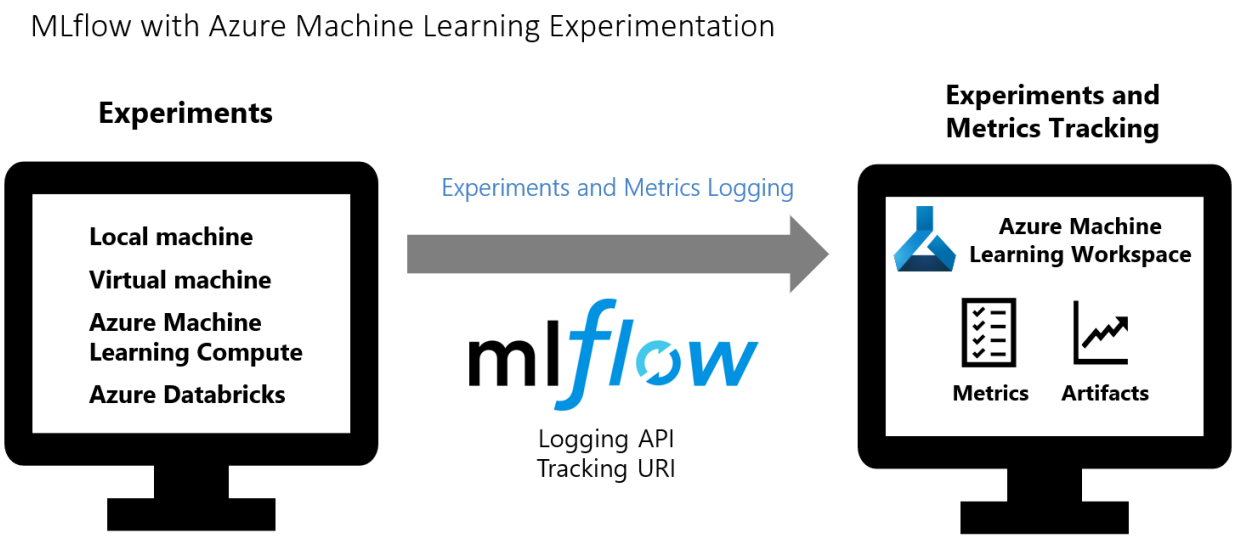
2/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article demonstrates how to enable MLflow's tracking URI and logging API, collectively known as MLflow Tracking, to connect your MLflow experiments and Azure Machine Learning. Doing so enables you to track and log experiment metrics and artifacts in your Azure Machine Learning workspace. If you already use MLflow Tracking for your experiments, the workspace provides a centralized, secure, and scalable location to store training metrics and models.

MLflow is an open-source library for managing the life cycle of your machine learning experiments. MLFlow Tracking is a component of MLflow that logs and tracks your training run metrics and model artifacts, no matter your experiment's environment--locally on your computer, on a remote compute target, a virtual machine, or an Azure Databricks cluster.

The following diagram illustrates that with MLflow Tracking, you track an experiment's run metrics and store model artifacts in your Azure Machine Learning workspace.



TIP

The information in this document is primarily for data scientists and developers who want to monitor the model training process. If you are an administrator interested in monitoring resource usage and events from Azure Machine Learning, such as quotas, completed training runs, or completed model deployments, see Monitoring Azure Machine Learning.

Compare MLflow and Azure Machine Learning clients

The below table summarizes the different clients that can use Azure Machine Learning, and their respective function capabilities.

MLflow Tracking offers metric logging and artifact storage functionalities that are only otherwise available via the Azure Machine Learning Python SDK.

	MLFLOW TRACKING	AZURE MACHINE LEARNING PYTHON SDK	AZURE MACHINE LEARNING CLI	AZURE MACHINE LEARNING STUDIO
Manage workspace		✓	✓	✓
Use data stores		✓	✓	
Log metrics	✓	✓		
Upload artifacts	✓	✓		
View metrics	✓	✓	✓	✓
Manage compute		✓	✓	✓

Prerequisites

- Install MLflow.
- Install the Azure Machine Learning SDK on your local computer The SDK provides the connectivity for MLflow to access your workspace.
- Create an Azure Machine Learning Workspace.

Track local runs

MLflow Tracking with Azure Machine Learning lets you store the logged metrics and artifacts from your local runs into your Azure Machine Learning workspace.

Install the `azureml-mlflow` package to use MLflow Tracking with Azure Machine Learning on your experiments locally run in a Jupyter Notebook or code editor.

```
pip install azureml-mlflow
```

> **NOTE**
> The azureml.contrib namespace changes frequently, as we work to improve the service. As such, anything in this namespace should be considered as a preview, and not fully supported by Microsoft.

Import the `mlflow` and `Workspace` classes to access MLflow's tracking URI and configure your workspace.

In the following code, the `get_mlflow_tracking_uri()` method assigns a unique tracking URI address to the workspace, `ws`, and `set_tracking_uri()` points the MLflow tracking URI to that address.

```
import mlflow
from azureml.core import Workspace

ws = Workspace.from_config()

mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())
```

> **NOTE**
> The tracking URI is valid up to an hour or less. If you restart your script after some idle time, use the get_mlflow_tracking_uri API to get a new URI.

Set the MLflow experiment name with `set_experiment()` and start your training run with `start_run()`. Then use `log_metric()` to activate the MLflow logging API and begin logging your training run metrics.

```
experiment_name = 'experiment_with_mlflow'
mlflow.set_experiment(experiment_name)

with mlflow.start_run():
    mlflow.log_metric('alpha', 0.03)
```

Track remote runs

MLflow Tracking with Azure Machine Learning lets you store the logged metrics and artifacts from your remote runs into your Azure Machine Learning workspace.

Remote runs let you train your models on more powerful computes, such as GPU enabled virtual machines, or Machine Learning Compute clusters. See Set up compute targets for model training to learn about different compute options.

Configure your compute and training run environment with the `Environment` class. Include `mlflow` and `azureml-mlflow` pip packages in environment's `CondaDependencies` section. Then construct `ScriptRunConfig` with your remote compute as the compute target.

```
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core import ScriptRunConfig

exp = Experiment(workspace = 'my_workspace',
                 name='my_experiment')

mlflow_env = Environment(name='mlflow-env')

cd = CondaDependencies.create(pip_packages=['mlflow', 'azureml-mlflow'])

mlflow_env.python.conda_dependencies = cd

src = ScriptRunConfig(source_directory='./my_script_location', script='my_training_script.py')

src.run_config.target = 'my-remote-compute-compute'
src.run_config.environment = mlflow_env
```

In your training script, import `mlflow` to use the MLflow logging APIs, and start logging your run metrics.

```
import mlflow

with mlflow.start_run():
    mlflow.log_metric('example', 1.23)
```

With this compute and training run configuration, use the `Experiment.submit('train.py')` method to submit a run. This method automatically sets the MLflow tracking URI and directs the logging from MLflow to your Workspace.

```
run = exp.submit(src)
```

Track Azure Databricks runs

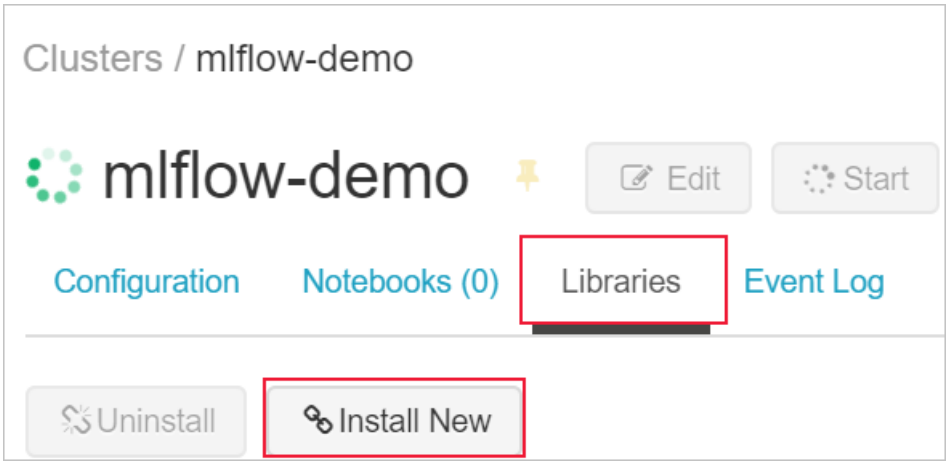
MLflow Tracking with Azure Machine Learning lets you store the logged metrics and artifacts from your Azure Databricks runs in your Azure Machine Learning workspace.

To run your Mlflow experiments with Azure Databricks, you need to first create an Azure Databricks workspace and cluster. In your cluster, be sure to install the *azureml-mlflow* library from PyPi, to ensure that your cluster has access to the necessary functions and classes.

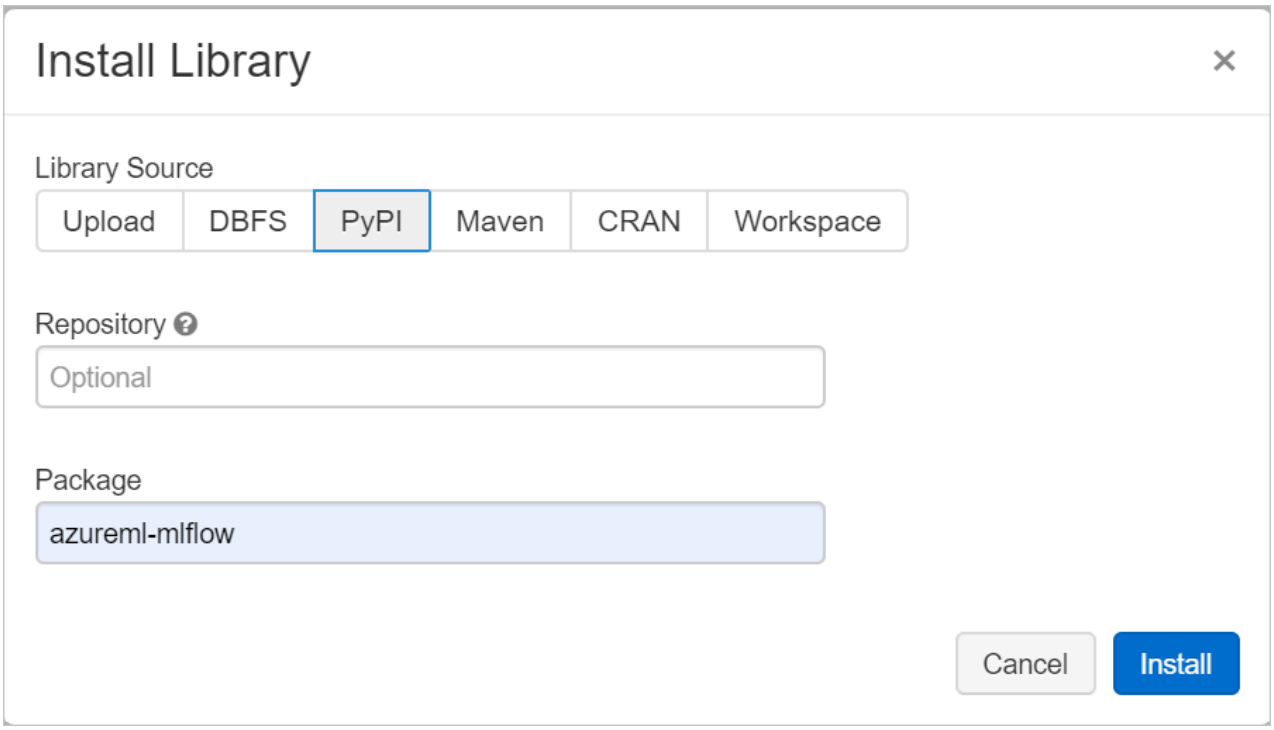
From here, import your experiment notebook, attach it to your Azure Databricks cluster and run your experiment.

Install libraries

To install libraries on your cluster, navigate to the **Libraries** tab and click **Install New**



In the **Package** field, type azureml-mlflow and then click install. Repeat this step as necessary to install other additional packages to your cluster for your experiment.



Set up your notebook and workspace

Once your cluster is set up, import your experiment notebook, open it and attach your cluster to it.

The following code should be in your experiment notebook. This code gets the details of your Azure subscription to instantiate your workspace. This code assumes you have an existing resource group and Azure Machine Learning workspace, otherwise you can create them.

```
import mlflow
import mlflow.azureml
import azureml.mlflow
import azureml.core

from azureml.core import Workspace
from azureml.mlflow import get_portal_url

subscription_id = 'subscription_id'

# Azure Machine Learning resource group NOT the managed resource group
resource_group = 'resource_group_name'

#Azure Machine Learning workspace name, NOT Azure Databricks workspace
workspace_name = 'workspace_name'

# Instantiate Azure Machine Learning workspace
ws = Workspace.get(name=workspace_name,
                   subscription_id=subscription_id,
                   resource_group=resource_group)
```

Connect your Azure Databricks and Azure Machine Learning workspaces

On the Azure portal, you can link your Azure Databricks (ADB) workspace to a new or existing Azure Machine Learning workspace. To do so, navigate to your ADB workspace and select the **Link Azure Machine Learning workspace** button on the bottom right. Linking your workspaces enables you to track your experiment data in the Azure Machine Learning workspace.

Link MLflow tracking to your workspace

After you instantiate your workspace, set the MLflow tracking URI. By doing so, you link the MLflow tracking to Azure Machine Learning workspace. After linking, all your experiments will land in the managed Azure Machine Learning tracking service.

Directly set MLflow Tracking in your notebook

```
uri = ws.get_mlflow_tracking_uri()
mlflow.set_tracking_uri(uri)
```

In your training script, import mlflow to use the MLflow logging APIs, and start logging your run metrics. The following example, logs the epoch loss metric.

```
import mlflow
mlflow.log_metric('epoch_loss', loss.item())
```

Automate setting MLflow Tracking

Instead of manually setting the tracking URI in every subsequent experiment notebook session on your clusters, do so automatically using this Azure Machine Learning Tracking Cluster Init script.

When configured correctly, you are able to see your MLflow tracking data in the Azure Machine Learning REST API and all clients, and in Azure Databricks via the MLflow user interface or by using the MLflow client.

View metrics and artifacts in your workspace

The metrics and artifacts from MLflow logging are kept in your workspace. To view them anytime, navigate to your workspace and find the experiment by name in your workspace in Azure Machine Learning studio. Or run the

below code.

```
run.get_metrics()
ws.get_details()
```

Example notebooks

The MLflow with Azure ML notebooks demonstrate and expand upon concepts presented in this article.

Next steps

- Manage your models.
- Monitor your production models for data drift.

Visualize experiment runs and metrics with TensorBoard and Azure Machine Learning

2/28/2020 • 7 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to view your experiment runs and metrics in TensorBoard using the `tensorboard` package in the main Azure Machine Learning SDK. Once you've inspected your experiment runs, you can better tune and retrain your machine learning models.

TensorBoard is a suite of web applications for inspecting and understanding your experiment structure and performance.

How you launch TensorBoard with Azure Machine Learning experiments depends on the type of experiment:

- If your experiment natively outputs log files that are consumable by TensorBoard, such as PyTorch, Chainer and TensorFlow experiments, then you can launch TensorBoard directly from experiment's run history.
- For experiments that don't natively output TensorBoard consumable files, such as like Scikit-learn or Azure Machine Learning experiments, use the `export_to_tensorboard()` method to export the run histories as TensorBoard logs and launch TensorBoard from there.

> **TIP**
> The information in this document is primarily for data scientists and developers who want to monitor the model training process. If you are an administrator interested in monitoring resource usage and events from Azure Machine learning, such as quotas, completed training runs, or completed model deployments, see Monitoring Azure Machine Learning.

Prerequisites

- To launch TensorBoard and view your experiment run histories, your experiments need to have previously enabled logging to track its metrics and performance.
- The code in this document can be run in either of the following environments:
  - Azure Machine Learning compute instance - no downloads or installation necessary
    - Complete the Tutorial: Setup environment and workspace to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
    - In the samples folder on the notebook server, find two completed and expanded notebooks by navigating to these directories:
      - **how-to-use-azureml > training-with-deep-learning > export-run-history-to-tensorboard > export-run-history-to-tensorboard.ipynb**
      - **how-to-use-azureml > track-and-monitor-experiments > tensorboard.ipynb**
  - Your own Juptyer notebook server
    - Install the Azure Machine Learning SDK with the `tensorboard` extra
    - Create an Azure Machine Learning workspace.
    - Create a workspace configuration file.

Option 1: Directly view run history in TensorBoard

This option works for experiments that natively outputs log files consumable by TensorBoard, such as PyTorch, Chainer, and TensorFlow experiments. If that is not the case of your experiment, use the `export_to_tensorboard()` method instead.

The following example code uses the MNIST demo experiment from TensorFlow's repository in a remote compute target, Azure Machine Learning Compute. Next, we train our model with the SDK's custom TensorFlow estimator, and then start TensorBoard against this TensorFlow experiment, that is, an experiment that natively outputs TensorBoard event files.

Set experiment name and create project folder

Here we name the experiment and create its folder.

```
from os import path, makedirs
experiment_name = 'tensorboard-demo'

# experiment folder
exp_dir = './sample_projects/' + experiment_name

if not path.exists(exp_dir):
    makedirs(exp_dir)
```

Download TensorFlow demo experiment code

TensorFlow's repository has an MNIST demo with extensive TensorBoard instrumentation. We do not, nor need to, alter any of this demo's code for it to work with Azure Machine Learning. In the following code, we download the MNIST code and save it in our newly created experiment folder.

```
import requests
import os

tf_code =
requests.get("https://raw.githubusercontent.com/tensorflow/tensorflow/r1.8/tensorflow/examples/tutorials/mnist
/mnist_with_summaries.py")
with open(os.path.join(exp_dir, "mnist_with_summaries.py"), "w") as file:
    file.write(tf_code.text)
```

Throughout the MNIST code file, mnist_with_summaries.py, notice that there are lines that call `tf.summary.scalar()`, `tf.summary.histogram()`, `tf.summary.FileWriter()` etc. These methods group, log, and tag key metrics of your experiments into run history. The `tf.summary.FileWriter()` is especially important as it serializes the data from your logged experiment metrics, which allows for TensorBoard to generate visualizations off of them.

Configure experiment

In the following, we configure our experiment and set up directories for logs and data. These logs will be uploaded to the Artifact Service, which TensorBoard accesses later.

> **NOTE**
>
> For this TensorFlow example, you will need to install TensorFlow on your local machine. Further, the TensorBoard module (that is, the one included with TensorFlow) must be accessible to this notebook's kernel, as the local machine is what runs TensorBoard.

```
import azureml.core
from azureml.core import Workspace
from azureml.core import Experiment

ws = Workspace.from_config()

# create directories for experiment logs and dataset
logs_dir = os.path.join(os.curdir, "logs")
data_dir = os.path.abspath(os.path.join(os.curdir, "mnist_data"))

if not path.exists(data_dir):
    makedirs(data_dir)

os.environ["TEST_TMPDIR"] = data_dir

# Writing logs to ./logs results in their being uploaded to Artifact Service,
# and thus, made accessible to our TensorBoard instance.
arguments_list = ["--log_dir", logs_dir]

# Create an experiment
exp = Experiment(ws, experiment_name)
```

Create a cluster for your experiment

We create an AmlCompute cluster for this experiment, however your experiments can be created in any environment and you are still able to launch TensorBoard against the experiment run history.

```
from azureml.core.compute import ComputeTarget, AmlCompute

cluster_name = "cpucluster"

cts = ws.compute_targets
found = False
if cluster_name in cts and cts[cluster_name].type == 'AmlCompute':
   found = True
   print('Found existing compute target.')
   compute_target = cts[cluster_name]
if not found:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                           max_nodes=4)

    # create the cluster
    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

compute_target.wait_for_completion(show_output=True, min_node_count=None)

# use get_status() to get a detailed status for the current cluster.
# print(compute_target.get_status().serialize())
```

Submit run with TensorFlow estimator

The TensorFlow estimator provides a simple way of launching a TensorFlow training job on a compute target. It's implemented through the generic `estimator` class, which can be used to support any framework. For more information about training models using the generic estimator, see train models with Azure Machine Learning using estimator

```
from azureml.train.dnn import TensorFlow
script_params = {"--log_dir": "./logs"}

tf_estimator = TensorFlow(source_directory=exp_dir,
                          compute_target=compute_target,
                          entry_script='mnist_with_summaries.py',
                          script_params=script_params)

run = exp.submit(tf_estimator)
```

Launch TensorBoard

You can launch TensorBoard during your run or after it completes. In the following, we create a TensorBoard object instance, `tb`, that takes the experiment run history loaded in the `run`, and then launches TensorBoard with the `start()` method.

The TensorBoard constructor takes an array of runs, so be sure and pass it in as a single-element array.

```
from azureml.tensorboard import Tensorboard

tb = Tensorboard([run])

# If successful, start() returns a string with the URI of the instance.
tb.start()

# After your job completes, be sure to stop() the streaming otherwise it will continue to run.
tb.stop()
```

NOTE

While this example used TensorFlow, TensorBoard can be used as easily with PyTorch or Chainer models. TensorFlow must be available on the machine running TensorBoard, but is not necessary on the machine doing PyTorch or Chainer computations.

Option 2: Export history as log to view in TensorBoard

The following code sets up a sample experiment, begins the logging process using the Azure Machine Learning run history APIs, and exports the experiment run history into logs consumable by TensorBoard for visualization.

Set up experiment

The following code sets up a new experiment and names the run directory `root_run`.

```
from azureml.core import Workspace, Experiment
import azuremml.core

# set experiment name and run name
ws = Workspace.from_config()
experiment_name = 'export-to-tensorboard'
exp = Experiment(ws, experiment_name)
root_run = exp.start_logging()
```

Here we load the diabetes dataset-- a built-in small dataset that comes with scikit-learn, and split it into test and training sets.

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
X, y = load_diabetes(return_X_y=True)
columns = ['age', 'gender', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
data = {
    "train":{"x":x_train, "y":y_train},
    "test":{"x":x_test, "y":y_test}
}
```

Run experiment and log metrics

For this code, we train a linear regression model and log key metrics, the alpha coefficient, `alpha`, and mean squared error, `mse`, in run history.

```
from tqdm import tqdm
alphas = [.1, .2, .3, .4, .5, .6 , .7]
# try a bunch of alpha values in a Linear Regression (aka Ridge regression) mode
for alpha in tqdm(alphas):
  # create child runs and fit lines for the resulting models
  with root_run.child_run("alpha" + str(alpha)) as run

   reg = Ridge(alpha=alpha)
   reg.fit(data["train"]["x"], data["train"]["y"])

   preds = reg.predict(data["test"]["x"])
   mse = mean_squared_error(preds, data["test"]["y"])
   # End train and eval

# log alpha, mean_squared_error and feature names in run history
   root_run.log("alpha", alpha)
   root_run.log("mse", mse)
```

Export runs to TensorBoard

With the SDK's export_to_tensorboard() method, we can export the run history of our Azure machine learning experiment into TensorBoard logs, so we can view them via TensorBoard.

In the following code, we create the folder `logdir` in our current working directory. This folder is where we will export our experiment run history and logs from `root_run` and then mark that run as completed.

```
from azureml.tensorboard.export import export_to_tensorboard
import os

logdir = 'exportedTBlogs'
log_path = os.path.join(os.getcwd(), logdir)
try:
    os.stat(log_path)
except os.error:
    os.mkdir(log_path)
print(logdir)

# export run history for the project
export_to_tensorboard(root_run, logdir)

root_run.complete()
```

NOTE

You can also export a particular run to TensorBoard by specifying the name of the run

`export_to_tensorboard(run_name, logdir)`

Start and stop TensorBoard

Once our run history for this experiment is exported, we can launch TensorBoard with the start() method.

```
from azureml.tensorboard import Tensorboard

# The TensorBoard constructor takes an array of runs, so be sure and pass it in as a single-element array here
tb = Tensorboard([], local_root=logdir, port=6006)

# If successful, start() returns a string with the URI of the instance.
tb.start()
```

When you're done, make sure to call the stop() method of the TensorBoard object. Otherwise, TensorBoard will continue to run until you shut down the notebook kernel.

```
tb.stop()
```

Next steps

In this how-to you, created two experiments and learned how to launch TensorBoard against their run histories to identify areas for potential tuning and retraining.

- If you are satisfied with your model, head over to our How to deploy a model article.
- Learn more about hyperparameter tuning.

Deploy models with Azure Machine Learning

4/6/2020 • 37 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to deploy your machine learning model as a web service in the Azure cloud or to Azure IoT Edge devices.

The workflow is similar no matter where you deploy your model:

1. Register the model.
2. Prepare to deploy. (Specify assets, usage, compute target.)
3. Deploy the model to the compute target.
4. Test the deployed model, also called a web service.

For more information on the concepts involved in the deployment workflow, see Manage, deploy, and monitor models with Azure Machine Learning.

Prerequisites

- An Azure Machine Learning workspace. For more information, see Create an Azure Machine Learning workspace.
- A model. If you don't have a trained model, you can use the model and dependency files provided in this tutorial.
- The Azure CLI extension for the Machine Learning service, the Azure Machine Learning SDK for Python, or the Azure Machine Learning Visual Studio Code extension.

Connect to your workspace

The following code shows how to connect to an Azure Machine Learning workspace by using information cached to the local development environment:

- **Using the SDK**

  ```
  from azureml.core import Workspace
  ws = Workspace.from_config(path=".file-path/ws_config.json")
  ```

  For more information on using the SDK to connect to a workspace, see the Azure Machine Learning SDK for Python documentation.

- **Using the CLI**

  When using the CLI, use the `-w` or `--workspace-name` parameter to specify the workspace for the command.

- **Using Visual Studio Code**

  When you use Visual Studio Code, you select the workspace by using a graphical interface. For more information, see Deploy and manage models in the Visual Studio Code extension documentation.

Register your model

A registered model is a logical container for one or more files that make up your model. For example, if you have a model that's stored in multiple files, you can register them as a single model in the workspace. After you register the files, you can then download or deploy the registered model and receive all the files that you registered.

> **TIP**
> When you register a model, you provide the path of either a cloud location (from a training run) or a local directory. This path is just to locate the files for upload as part of the registration process. It doesn't need to match the path used in the entry script. For more information, see Locate model files in your entry script.

Machine learning models are registered in your Azure Machine Learning workspace. The model can come from Azure Machine Learning or from somewhere else. When registering a model, you can optionally provide metadata about the model. The `tags` and `properties` dictionaries that you apply to a model registration can then be used to filter models.

The following examples demonstrate how to register a model.

Register a model from an experiment run

The code snippets in this section demonstrate how to register a model from a training run:

> **IMPORTANT**
> To use these snippets, you need to have previously performed a training run and you need to have access to the `Run` object (SDK example) or the run ID value (CLI example). For more information on training models, see Set up compute targets for model training.

- Using the SDK

  When you use the SDK to train a model, you can receive either a Run object or an AutoMLRun object, depending on how you trained the model. Each object can be used to register a model created by an experiment run.

  - Register a model from an `azureml.core.Run` object:

    ```
    model = run.register_model(model_name='sklearn_mnist',
                               tags={'area': 'mnist'},
                               model_path='outputs/sklearn_mnist_model.pkl')
    print(model.name, model.id, model.version, sep='\t')
    ```

    The `model_path` parameter refers to the cloud location of the model. In this example, the path of a single file is used. To include multiple files in the model registration, set `model_path` to the path of a folder that contains the files. For more information, see the Run.register_model documentation.

  - Register a model from an `azureml.train.automl.run.AutoMLRun` object:

    ```
    description = 'My AutoML Model'
    model = run.register_model(description = description,
                               tags={'area': 'mnist'})

    print(run.model_id)
    ```

In this example, the `metric` and `iteration` parameters aren't specified, so the iteration with the best primary metric will be registered. The `model_id` value returned from the run is used instead of a model name.

For more information, see the AutoMLRun.register_model documentation.

- Using the CLI

```
az ml model register -n sklearn_mnist  --asset-path outputs/sklearn_mnist_model.pkl  --
experiment-name myexperiment --run-id myrunid --tag area=mnist
```

> **TIP**
> If you get an error message stating that the ml extension isn't installed, use the following command to install it:
>
> ```
> az extension add -n azure-cli-ml
> ```

The `--asset-path` parameter refers to the cloud location of the model. In this example, the path of a single file is used. To include multiple files in the model registration, set `--asset-path` to the path of a folder that contains the files.

- Using Visual Studio Code

Register models using any model files or folders by using the Visual Studio Code extension.

Register a model from a local file

You can register a model by providing the local path of the model. You can provide the path of either a folder or a single file. You can use this method to register models trained with Azure Machine Learning and then downloaded. You can also use this method to register models trained outside of Azure Machine Learning.

> **IMPORTANT**
> You should use only models that you create or obtain from a trusted source. You should treat serialized models as code, because security vulnerabilities have been discovered in a number of popular formats. Also, models might be intentionally trained with malicious intent to provide biased or inaccurate output.

- Using the SDK and ONNX

```
import os
import urllib.request
from azureml.core.model import Model
# Download model
onnx_model_url = "https://www.cntk.ai/OnnxModels/mnist/opset_7/mnist.tar.gz"
urllib.request.urlretrieve(onnx_model_url, filename="mnist.tar.gz")
os.system('tar xvzf mnist.tar.gz')
# Register model
model = Model.register(workspace = ws,
                       model_path ="mnist/model.onnx",
                       model_name = "onnx_mnist",
                       tags = {"onnx": "demo"},
                       description = "MNIST image classification CNN from ONNX Model Zoo",)
```

To include multiple files in the model registration, set `model_path` to the path of a folder that

contains the files.

- **Using the CLI**

  ```
  az ml model register -n onnx_mnist -p mnist/model.onnx
  ```

  To include multiple files in the model registration, set `-p` to the path of a folder that contains the files.

Time estimate: Approximately 10 seconds.

For more information, see the documentation for the Model class.

For more information on working with models trained outside Azure Machine Learning, see How to deploy an existing model.

Single versus multi-model endpoints

Azure ML supports deploying single or multiple models behind a single endpoint.

Multi-model endpoints use a shared container to host multiple models. This helps to reduce overhead costs, improves utilization, and enables you to chain modules together into ensembles. Models you specify in your deployment script are mounted and made available on the disk of the serving container - you can load them into memory on demand and score based on the specific model being requested at scoring time.

For an E2E example, which shows how to use multiple models behind a single containerized endpoint, see this example

Prepare to deploy

To deploy the model as a service, you need the following components:

- **Define inference environment**. This environment encapsulates the dependencies required to run your model for inference.
- **Define scoring code**. This script accepts requests, scores the requests by using the model, and returns the results.
- **Define inference configuration**. The inference configuration specifies the environment configuration, entry script, and other components needed to run the model as a service.

Once you have the necessary components, you can profile the service that will be created as a result of deploying your model to understand its CPU and memory requirements.

1. Define inference environment

An inference configuration describes how to set up the web-service containing your model. It's used later, when you deploy the model.

Inference configuration uses Azure Machine Learning environments to define the software dependencies needed for your deployment. Environments allow you to create, manage, and reuse the software dependencies required for training and deployment. You can create an environment from custom dependency files or use one of the curated Azure Machine Learning environments. The following YAML is an example of a Conda dependencies file for inference. Note that you must indicate azureml-defaults with verion >= 1.0.45 as a pip dependency, because it contains the functionality needed to host the model as a web service. If you want to use automatic schema generation, your entry script must also import the `inference-schema` packages.

```
name: project_environment
dependencies:
  - python=3.6.2
  - scikit-learn=0.20.0
  - pip:
      # You must list azureml-defaults as a pip dependency
    - azureml-defaults>=1.0.45
    - inference-schema[numpy-support]
```

IMPORTANT

If your dependency is available through both Conda and pip (from PyPi), Microsoft recommends using the Conda version, as Conda packages typically come with pre-built binaries that make installation more reliable.

For more information, see Understanding Conda and Pip.

To check if your dependency is available through Conda, use the `conda search <package-name>` command, or use the package indexes at https://anaconda.org/anaconda/repo and https://anaconda.org/conda-forge/repo.

You can use the dependencies file to create an environment object and save it to your workspace for future use:

```
from azureml.core.environment import Environment
myenv = Environment.from_conda_specification(name = 'myenv',
                                             file_path = 'path-to-conda-specification-file'
myenv.register(workspace=ws)
```

2. Define scoring code

The entry script receives data submitted to a deployed web service and passes it to the model. It then takes the response returned by the model and returns that to the client. *The script is specific to your model*. It must understand the data that the model expects and returns.

The script contains two functions that load and run the model:

- `init()`: Typically, this function loads the model into a global object. This function is run only once, when the Docker container for your web service is started.
- `run(input_data)`: This function uses the model to predict a value based on the input data. Inputs and outputs of the run typically use JSON for serialization and deserialization. You can also work with raw binary data. You can transform the data before sending it to the model or before returning it to the client.

Load model files in your entry script

There are two ways to locate models in your entry script:

- `AZUREML_MODEL_DIR`: An environment variable containing the path to the model location.
- `Model.get_model_path`: An API that returns the path to model file using the registered model name.

AZUREML_MODEL_DIR

AZUREML_MODEL_DIR is an environment variable created during service deployment. You can use this environment variable to find the location of the deployed model(s).

The following table describes the value of AZUREML_MODEL_DIR depending on the number of models deployed:

DEPLOYMENT	ENVIRONMENT VARIABLE VALUE
Single model	The path to the folder containing the model.
Multiple models	The path to the folder containing all models. Models are located by name and version in this folder (`$MODEL_NAME/$VERSION`)

During model registration and deployment, Models are placed in the AZUREML_MODEL_DIR path, and their original filenames are preserved.

To get the path to a model file in your entry script, combine the environment variable with the file path you're looking for.

Single model example

```
# Example when the model is a file
model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_regression_model.pkl')

# Example when the model is a folder containing a file
file_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'my_model_folder',
'sklearn_regression_model.pkl')
```

Multiple model example

```
# Example when the model is a file, and the deployment contains multiple models
model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_model', '1',
'sklearn_regression_model.pkl')
```

`get_model_path`

When you register a model, you provide a model name that's used for managing the model in the registry. You use this name with the Model.get_model_path() method to retrieve the path of the model file or files on the local file system. If you register a folder or a collection of files, this API returns the path of the directory that contains those files.

When you register a model, you give it a name. The name corresponds to where the model is placed, either locally or during service deployment.

(Optional) Define model web service schema

To automatically generate a schema for your web service, provide a sample of the input and/or output in the constructor for one of the defined type objects. The type and sample are used to automatically create the schema. Azure Machine Learning then creates an OpenAPI (Swagger) specification for the web service during deployment.

These types are currently supported:

- `pandas`
- `numpy`
- `pyspark`
- Standard Python object

To use schema generation, include the open-source `inference-schema` package in your dependencies file. For more information on this package, see https://github.com/Azure/InferenceSchema. Define the input and output sample formats in the `input_sample` and `output_sample` variables, which represent the request and response formats for the web service. Use these samples in the input and output function decorators on the `run()` function. The following scikit-learn example uses schema

generation.

Example entry script

The following example demonstrates how to accept and return JSON data:

```
#Example: scikit-learn and Swagger
import json
import numpy as np
import os
from sklearn.externals import joblib
from sklearn.linear_model import Ridge

from inference_schema.schema_decorators import input_schema, output_schema
from inference_schema.parameter_types.numpy_parameter_type import NumpyParameterType


def init():
    global model
    # AZUREML_MODEL_DIR is an environment variable created during deployment. Join this path with
the filename of the model file.
    # It holds the path to the directory that contains the deployed model (./azureml-
models/$MODEL_NAME/$VERSION).
    # If there are multiple models, this value is the path to the directory containing all
deployed models (./azureml-models).
    # Alternatively: model_path = Model.get_model_path('sklearn_mnist')
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_mnist_model.pkl')
    # Deserialize the model file back into a sklearn model
    model = joblib.load(model_path)


input_sample = np.array([[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]])
output_sample = np.array([3726.995])


@input_schema('data', NumpyParameterType(input_sample))
@output_schema(NumpyParameterType(output_sample))
def run(data):
    try:
        result = model.predict(data)
        # You can return any data type, as long as it is JSON serializable.
        return result.tolist()
    except Exception as e:
        error = str(e)
        return error
```

The following example demonstrates how to define the input data as a `<key: value>` dictionary by using a DataFrame. This method is supported for consuming the deployed web service from Power BI. (Learn more about how to consume the web service from Power BI.)

```
import json
import pickle
import numpy as np
import pandas as pd
import azureml.train.automl
from sklearn.externals import joblib
from azureml.core.model import Model

from inference_schema.schema_decorators import input_schema, output_schema
from inference_schema.parameter_types.numpy_parameter_type import NumpyParameterType
from inference_schema.parameter_types.pandas_parameter_type import PandasParameterType


def init():
    global model
    # Replace filename if needed.
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'model_file.pkl')
    # Deserialize the model file back into a sklearn model.
    model = joblib.load(model_path)


input_sample = pd.DataFrame(data=[{
    # This is a decimal type sample. Use the data type that reflects this column in your data.
    "input_name_1": 5.1,
    # This is a string type sample. Use the data type that reflects this column in your data.
    "input_name_2": "value2",
    # This is an integer type sample. Use the data type that reflects this column in your data.
    "input_name_3": 3
}])

# This is an integer type sample. Use the data type that reflects the expected result.
output_sample = np.array([0])


@input_schema('data', PandasParameterType(input_sample))
@output_schema(NumpyParameterType(output_sample))
def run(data):
    try:
        result = model.predict(data)
        # You can return any data type, as long as it is JSON serializable.
        return result.tolist()
    except Exception as e:
        error = str(e)
        return error
```

For more examples, see the following scripts:

- PyTorch
- TensorFlow
- Keras
- AutoML
- ONNX
- Binary Data
- CORS

3. Define inference configuration

The following example demonstrates loading an environment from your workspace and then using it with the inference configuration:

```
from azureml.core.environment import Environment
from azureml.core.model import InferenceConfig


myenv = Environment.get(workspace=ws, name='myenv', version='1')
inference_config = InferenceConfig(entry_script='path-to-score.py',
                                   environment=myenv)
```

For more information on environments, see Create and manage environments for training and deployment.

For more information on inference configuration, see the InferenceConfig class documentation.

For information on using a custom Docker image with an inference configuration, see How to deploy a model using a custom Docker image.

CLI example of InferenceConfig

The entries in the `inferenceconfig.json` document map to the parameters for the InferenceConfig class. The following table describes the mapping between entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`entryScript`	`entry_script`	Path to a local file that contains the code to run for the image.
`sourceDirectory`	`source_directory`	Optional. Path to folders that contain all files to create the image, which makes it easy to access any files within this folder or subfolder. You can upload an entire folder from your local machine as dependencies for the Webservice. Note: your entry_script, conda_file, and extra_docker_file_steps paths are relative paths to the source_directory path.
`environment`	`environment`	Optional. Azure Machine Learning environment.

You can include full specifications of an Azure Machine Learning environment in the inference configuration file. If this environment doesn't exist in your workspace, Azure Machine Learning will create it. Otherwise, Azure Machine Learning will update the environment if necessary. The following JSON is an example:

```
{
    "entryScript": "score.py",
    "environment": {
        "docker": {
            "arguments": [],
            "baseDockerfile": null,
            "baseImage": "mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04",
            "enabled": false,
            "sharedVolumes": true,
            "shmSize": null
        },
        "environmentVariables": {
            "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
        },
        "name": "my-deploy-env",
        "python": {
            "baseCondaEnvironment": null,
            "condaDependencies": {
                "channels": [
                    "conda-forge"
                ],
                "dependencies": [
                    "python=3.6.2",
                    {
                        "pip": [
                            "azureml-defaults",
                            "azureml-telemetry",
                            "scikit-learn",
                            "inference-schema[numpy-support]"
                        ]
                    }
                ],
                "name": "project_environment"
            },
            "condaDependenciesFile": null,
            "interpreterPath": "python",
            "userManagedDependencies": false
        },
        "version": "1"
    }
}
```

You can also use an existing Azure Machine Learning environment in separated CLI parameters and remove the "environment" key from the inference configuration file. Use -e for the environment name, and --ev for the environment version. If you don't specify --ev, the latest version will be used. Here is an example of an inference configuration file:

```
{
    "entryScript": "score.py",
    "sourceDirectory": null
}
```

The following command demonstrates how to deploy a model using the previous inference configuration file (named myInferenceConfig.json).

It also uses the latest version of an existing Azure Machine Learning environment (named AzureML-Minimal).

```
az ml model deploy -m mymodel:1 --ic myInferenceConfig.json -e AzureML-Minimal --dc
deploymentconfig.json
```

The following command demonstrates how to deploy a model by using the CLI:

```
az ml model deploy -n myservice -m mymodel:1 --ic inferenceconfig.json
```

In this example, the configuration specifies the following settings:

- That the model requires Python.
- The entry script, which is used to handle web requests sent to the deployed service.
- The Conda file that describes the Python packages needed for inference.

For information on using a custom Docker image with an inference configuration, see How to deploy a model using a custom Docker image.

4. (Optional) Profile your model to determine resource utilization

Once you have registered your model and prepared the other components necessary for its deployment, you can determine the CPU and memory the deployed service will need. Profiling tests the service that runs your model and returns information such as the CPU usage, memory usage, and response latency. It also provides a recommendation for the CPU and memory based on resource usage.

In order to profile your model, you will need:

- A registered model.
- An inference configuration based on your entry script and inference environment definition.
- A single column tabular dataset, where each row contains a string representing sample request data.

> **IMPORTANT**
> At this point we only support profiling of services that expect their request data to be a string, for example: string serialized json, text, string serialized image, etc. The content of each row of the dataset (string) will be put into the body of the HTTP request and sent to the service encapsulating the model for scoring.

Below is an example of how you can construct an input dataset to profile a service that expects its incoming request data to contain serialized json. In this case, we created a dataset based one hundred instances of the same request data content. In real world scenarios we suggest that you use larger datasets containing various inputs, especially if your model resource usage/behavior is input dependent.

```
import json
from azureml.core import Datastore
from azureml.core.dataset import Dataset
from azureml.data import dataset_type_definitions

input_json = {'data': [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                       [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]]}
# create a string that can be utf-8 encoded and
# put in the body of the request
serialized_input_json = json.dumps(input_json)
dataset_content = []
for i in range(100):
    dataset_content.append(serialized_input_json)
dataset_content = '\n'.join(dataset_content)
file_name = 'sample_request_data.txt'
f = open(file_name, 'w')
f.write(dataset_content)
f.close()

# upload the txt file created above to the Datastore and create a dataset from it
data_store = Datastore.get_default(ws)
data_store.upload_files(['./' + file_name], target_path='sample_request_data')
datastore_path = [(data_store, 'sample_request_data' +'/' + file_name)]
sample_request_data = Dataset.Tabular.from_delimited_files(
    datastore_path, separator='\n',
    infer_column_types=True,
    header=dataset_type_definitions.PromoteHeadersBehavior.NO_HEADERS)
sample_request_data = sample_request_data.register(workspace=ws,
                                                   name='sample_request_data',
                                                   create_new_version=True)
```

Once you have the dataset containing sample request data ready, create an inference configuration. Inference configuration is based on the score.py and the environment definition. The following example demonstrates how to create the inference configuration and run profiling:

```
from azureml.core.model import InferenceConfig, Model
from azureml.core.dataset import Dataset


model = Model(ws, id=model_id)
inference_config = InferenceConfig(entry_script='path-to-score.py',
                                   environment=myenv)
input_dataset = Dataset.get_by_name(workspace=ws, name='sample_request_data')
profile = Model.profile(ws,
            'unique_name',
            [model],
            inference_config,
            input_dataset=input_dataset)

profile.wait_for_completion(True)

# see the result
details = profile.get_details()
```

The following command demonstrates how to profile a model by using the CLI:

```
az ml model profile -g <resource-group-name> -w <workspace-name> --inference-config-file <path-to-
inf-config.json> -m <model-id> --idi <input-dataset-id> -n <unique-name>
```

TIP

To persist the information returned by profiling, use tags or properties for the model. Using tags or properties stores the data with the model in the model registry. The following examples demonstrate adding a new tag containing the `requestedCpu` and `requestedMemoryInGb` information:

```
model.add_tags({'requestedCpu': details['requestedCpu'],
                'requestedMemoryInGb': details['requestedMemoryInGb']})
```

```
az ml model profile -g <resource-group-name> -w <workspace-name> --i <model-id> --add-tag
requestedCpu=1 --add-tag requestedMemoryInGb=0.5
```

Deploy to target

Deployment uses the inference configuration deployment configuration to deploy the models. The deployment process is similar regardless of the compute target. Deploying to AKS is slightly different because you must provide a reference to the AKS cluster.

Choose a compute target

You can use the following compute targets, or compute resources, to host your web service deployment:

COMPUTE TARGET	USED FOR	GPU SUPPORT	FPGA SUPPORT	DESCRIPTION
Local web service	Testing/debugging			Use for limited testing and troubleshooting. Hardware acceleration depends on use of libraries in the local system.
Azure Machine Learning compute instance web service	Testing/debugging			Use for limited testing and troubleshooting.

COMPUTE TARGET	USED FOR	GPU SUPPORT	FPGA SUPPORT	DESCRIPTION
Azure Kubernetes Service (AKS)	Real-time inference	Yes (web service deployment)	Yes	Use for high-scale production deployments. Provides fast response time and autoscaling of the deployed service. Cluster autoscaling isn't supported through the Azure Machine Learning SDK. To change the nodes in the AKS cluster, use the UI for your AKS cluster in the Azure portal. AKS is the only option available for the designer.
Azure Container Instances	Testing or development			Use for low-scale CPU-based workloads that require less than 48 GB of RAM.
Azure Machine Learning compute clusters	(Preview) Batch inference	Yes (machine learning pipeline)		Run batch scoring on serverless compute. Supports normal and low-priority VMs.
Azure Functions	(Preview) Real-time inference			
Azure IoT Edge	(Preview) IoT module			Deploy and serve ML models on IoT devices.
Azure Data Box Edge	Via IoT Edge		Yes	Deploy and serve ML models on IoT devices.

> **NOTE**
>
> Although compute targets like local, Azure Machine Learning compute instance, and Azure Machine Learning compute clusters support GPU for training and experimentation, using GPU for inference **when deployed as a web service** is supported only on Azure Kubernetes Service.
>
> Using a GPU for inference **when scoring with a machine learning pipeline** is supported only on Azure Machine Learning Compute.

Define your deployment configuration

Before deploying your model, you must define the deployment configuration. *The deployment configuration is specific to the compute target that will host the web service.* For example, when you

deploy a model locally, you must specify the port where the service accepts requests. The deployment configuration isn't part of your entry script. It's used to define the characteristics of the compute target that will host the model and entry script.

You might also need to create the compute resource, if, for example, you don't already have an Azure Kubernetes Service (AKS) instance associated with your workspace.

The following table provides an example of creating a deployment configuration for each compute target:

COMPUTE TARGET	DEPLOYMENT CONFIGURATION EXAMPLE
Local	`deployment_config = LocalWebservice.deploy_configuration(port=8890)`
Azure Container Instances	`deployment_config = AciWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)`
Azure Kubernetes Service	`deployment_config = AksWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)`

The classes for local, Azure Container Instances, and AKS web services can be imported from `azureml.core.webservice`:

```
from azureml.core.webservice import AciWebservice, AksWebservice, LocalWebservice
```

Securing deployments with TLS

For more information on how to secure a web service deployment, see Enable TLS and deploy.

Local deployment

To deploy a model locally, you need to have Docker installed on your local machine.

Using the SDK

```
from azureml.core.webservice import LocalWebservice, Webservice

deployment_config = LocalWebservice.deploy_configuration(port=8890)
service = Model.deploy(ws, "myservice", [model], inference_config, deployment_config)
service.wait_for_deployment(show_output = True)
print(service.state)
```

For more information, see the documentation for LocalWebservice, Model.deploy(), and Webservice.

Using the CLI

To deploy a model by using the CLI, use the following command. Replace `mymodel:1` with the name and version of the registered model:

```
az ml model deploy -m mymodel:1 --ic inferenceconfig.json --dc deploymentconfig.json
```

The entries in the `deploymentconfig.json` document map to the parameters for LocalWebservice.deploy_configuration. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`computeType`	NA	The compute target. For local targets, the value must be `local`.
`port`	`port`	The local port on which to expose the service's HTTP endpoint.

This JSON is an example deployment configuration for use with the CLI:

```
{
    "computeType": "local",
    "port": 32267
}
```

For more information, see the az ml model deploy documentation.

Understanding service state

During model deployment, you may see the service state change while it fully deploys.

The following table describes the different service states:

WEBSERVICE STATE	DESCRIPTION	FINAL STATE?
Transitioning	The service is in the process of deployment.	No
Unhealthy	The service has deployed but is currently unreachable.	No
Unschedulable	The service cannot be deployed at this time due to lack of resources.	No
Failed	The service has failed to deploy due to an error or crash.	Yes
Healthy	The service is healthy and the endpoint is available.	Yes

Compute instance web service (dev/test)

See Deploy a model to Azure Machine Learning compute instance.

Azure Container Instances (dev/test)

See Deploy to Azure Container Instances.

Azure Kubernetes Service (dev/test and production)

See Deploy to Azure Kubernetes Service.

A/B Testing (controlled rollout)

See Controlled rollout of ML models for more information.

Consume web services

Every deployed web service provides a REST endpoint, so you can create client applications in any programming language. If you've enabled key-based authentication for your service, you need to

provide a service key as a token in your request header. If you've enabled token-based authentication for your service, you need to provide an Azure Machine Learning JSON Web Token (JWT) as a bearer token in your request header.

The primary difference is that **keys are static and can be regenerated manually**, and **tokens need to be refreshed upon expiration**. Key-based auth is supported for Azure Container Instance and Azure Kubernetes Service deployed web-services, and token-based auth is **only** available for Azure Kubernetes Service deployments. See the how-to on authentication for more information and specific code samples.

> **TIP**
> You can retrieve the schema JSON document after you deploy the service. Use the swagger_uri property from the deployed web service (for example, `service.swagger_uri`) to get the URI to the local web service's Swagger file.

Request-response consumption

Here's an example of how to invoke your service in Python:

```
import requests
import json

headers = {'Content-Type': 'application/json'}

if service.auth_enabled:
    headers['Authorization'] = 'Bearer '+service.get_keys()[0]
elif service.token_auth_enabled:
    headers['Authorization'] = 'Bearer '+service.get_token()[0]

print(headers)

test_sample = json.dumps({'data': [
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
]})

response = requests.post(
    service.scoring_uri, data=test_sample, headers=headers)
print(response.status_code)
print(response.elapsed)
print(response.json())
```

For more information, see Create client applications to consume web services.

Web service schema (OpenAPI specification)

If you used automatic schema generation with your deployment, you can get the address of the OpenAPI specification for the service by using the swagger_uri property. (For example, `print(service.swagger_uri)`.) Use a GET request or open the URI in a browser to retrieve the specification.

The following JSON document is an example of a schema (OpenAPI specification) generated for a deployment:

```
{
    "swagger": "2.0",
    "info": {
        "title": "myservice",
        "description": "API specification for Azure Machine Learning myservice",
        "version": "1.0"
```

```
    },
    "schemes": [
        "https"
    ],
    "consumes": [
        "application/json"
    ],
    "produces": [
        "application/json"
    ],
    "securityDefinitions": {
        "Bearer": {
            "type": "apiKey",
            "name": "Authorization",
            "in": "header",
            "description": "For example: Bearer abc123"
        }
    },
    "paths": {
        "/": {
            "get": {
                "operationId": "ServiceHealthCheck",
                "description": "Simple health check endpoint to ensure the service is up at any
given point.",
                "responses": {
                    "200": {
                        "description": "If service is up and running, this response will be
returned with the content 'Healthy'",
                        "schema": {
                            "type": "string"
                        },
                        "examples": {
                            "application/json": "Healthy"
                        }
                    },
                    "default": {
                        "description": "The service failed to execute due to an error.",
                        "schema": {
                            "$ref": "#/definitions/ErrorResponse"
                        }
                    }
                }
            }
        },
        "/score": {
            "post": {
                "operationId": "RunMLService",
                "description": "Run web service's model and get the prediction output",
                "security": [
                    {
                        "Bearer": []
                    }
                ],
                "parameters": [
                    {
                        "name": "serviceInputPayload",
                        "in": "body",
                        "description": "The input payload for executing the real-time machine
learning service.",
                        "schema": {
                            "$ref": "#/definitions/ServiceInput"
                        }
                    }
                ],
                "responses": {
                    "200": {
                        "description": "The service processed the input correctly and provided a
result prediction, if applicable.",
                        "schema": {
```

```
                            "$ref": "#/definitions/ServiceOutput"
                        }
                    },
                    "default": {
                        "description": "The service failed to execute due to an error.",
                        "schema": {
                            "$ref": "#/definitions/ErrorResponse"
                        }
                    }
                }
            }
        }
    },
    "definitions": {
        "ServiceInput": {
            "type": "object",
            "properties": {
                "data": {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "format": "int64"
                        }
                    }
                }
            },
            "example": {
                "data": [
                    [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
                ]
            }
        },
        "ServiceOutput": {
            "type": "array",
            "items": {
                "type": "number",
                "format": "double"
            },
            "example": [
                3726.995
            ]
        },
        "ErrorResponse": {
            "type": "object",
            "properties": {
                "status_code": {
                    "type": "integer",
                    "format": "int32"
                },
                "message": {
                    "type": "string"
                }
            }
        }
    }
}
```

For more information, see OpenAPI specification.

For a utility that can create client libraries from the specification, see swagger-codegen.

Batch inference

Azure Machine Learning Compute targets are created and managed by Azure Machine Learning. They can be used for batch prediction from Azure Machine Learning pipelines.

For a walkthrough of batch inference with Azure Machine Learning Compute, see How to run batch predictions.

IoT Edge inference

Support for deploying to the edge is in preview. For more information, see Deploy Azure Machine Learning as an IoT Edge module.

Update web services

To update a web service, use the `update` method. You can update the web service to use a new model, a new entry script, or new dependencies that can be specified in an inference configuration. For more information, see the documentation for Webservice.update.

> **IMPORTANT**
>
> When you create a new version of a model, you must manually update each service that you want to use it.
>
> You can not use the SDK to update a web service published from the Azure Machine Learning designer.

Using the SDK

The following code shows how to use the SDK to update the model, environment, and entry script for a web service:

```
from azureml.core import Environment
from azureml.core.webservice import Webservice
from azureml.core.model import Model, InferenceConfig

# Register new model.
new_model = Model.register(model_path="outputs/sklearn_mnist_model.pkl",
                           model_name="sklearn_mnist",
                           tags={"key": "0.1"},
                           description="test",
                           workspace=ws)

# Use version 3 of the environment.
deploy_env = Environment.get(workspace=ws,name="myenv",version="3")
inference_config = InferenceConfig(entry_script="score.py",
                                   environment=deploy_env)

service_name = 'myservice'
# Retrieve existing service.
service = Webservice(name=service_name, workspace=ws)



# Update to new model(s).
service.update(models=[new_model], inference_config=inference_config)
print(service.state)
print(service.get_logs())
```

Using the CLI

You can also update a web service by using the ML CLI. The following example demonstrates registering a new model and then updating a web service to use the new model:

```
az ml model register -n sklearn_mnist  --asset-path outputs/sklearn_mnist_model.pkl  --experiment-
name myexperiment --output-metadata-file modelinfo.json
az ml service update -n myservice --model-metadata-file modelinfo.json
```

TIP

In this example, a JSON document is used to pass the model information from the registration command into the update command.

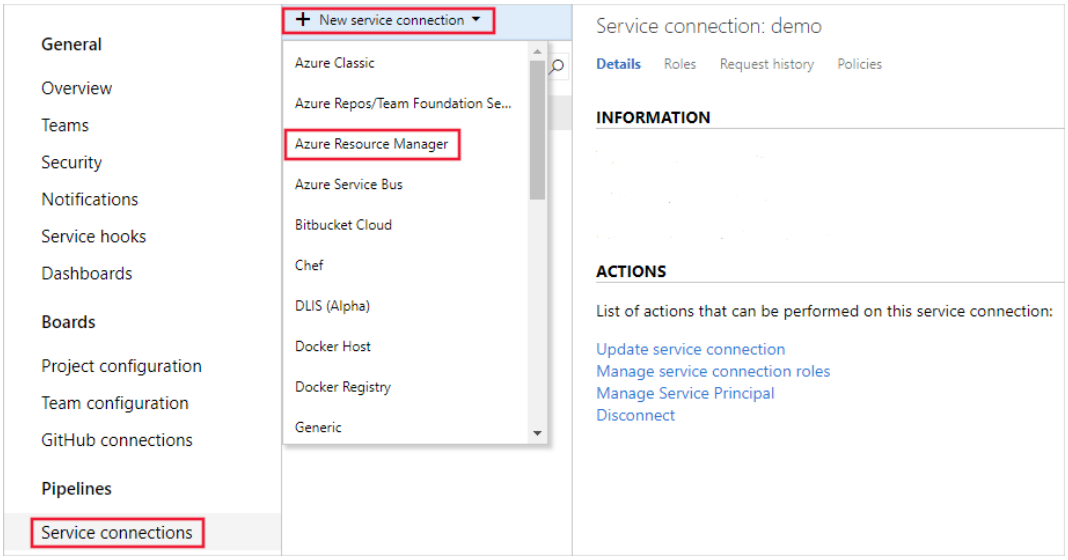
To update the service to use a new entry script or environment, create an inference configuration file and specify it with the `ic` parameter.

For more information, see the az ml service update documentation.

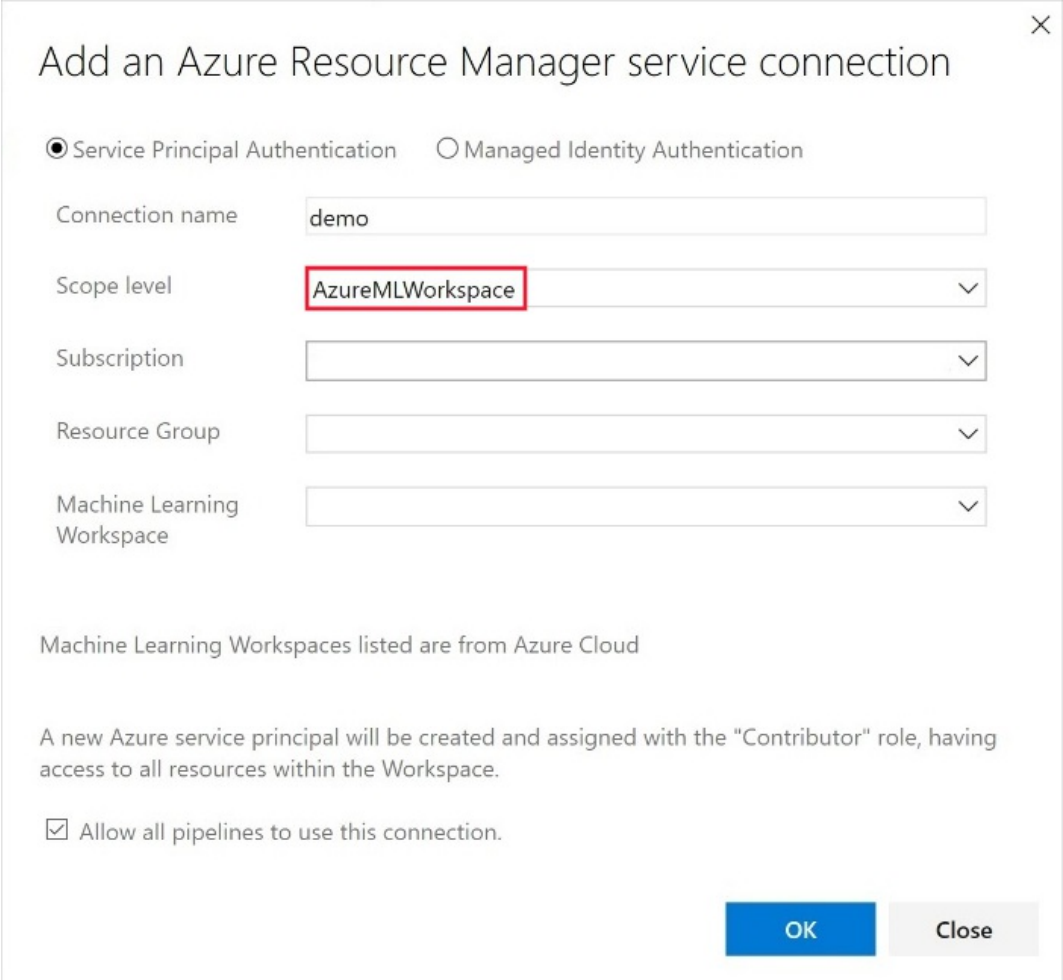
Continuously deploy models

You can continuously deploy models by using the Machine Learning extension for Azure DevOps. You can use the Machine Learning extension for Azure DevOps to trigger a deployment pipeline when a new machine learning model is registered in an Azure Machine Learning workspace.

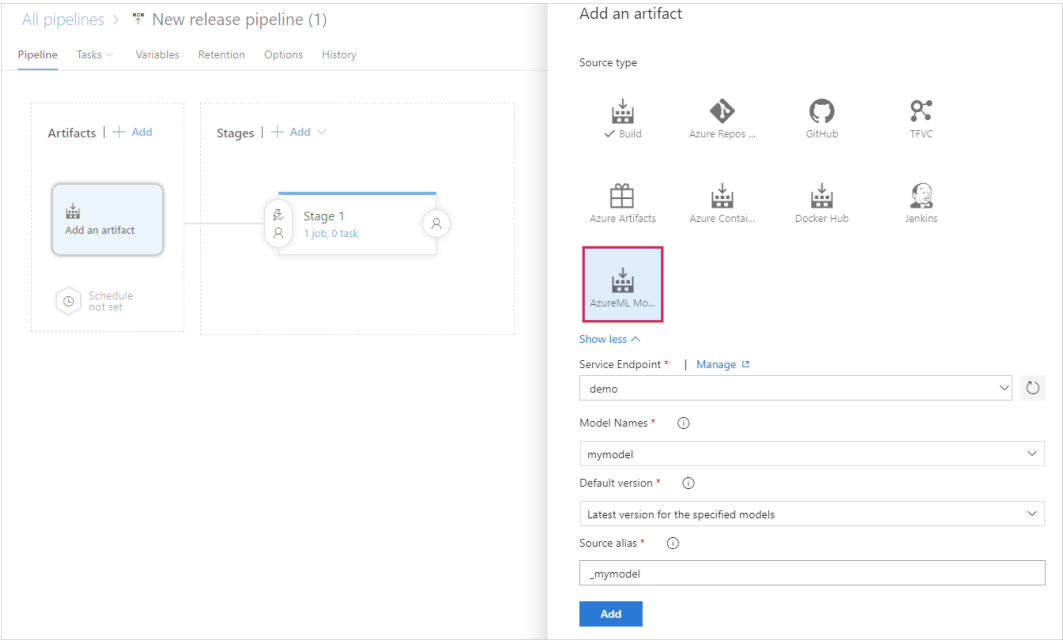
1. Sign up for Azure Pipelines, which makes continuous integration and delivery of your application to any platform or cloud possible. (Note that Azure Pipelines isn't the same as Machine Learning pipelines.)
2. Create an Azure DevOps project.
3. Install the Machine Learning extension for Azure Pipelines.
4. Use service connections to set up a service principal connection to your Azure Machine Learning workspace so you can access your artifacts. Go to project settings, select **Service connections**, and then select **Azure Resource Manager**:



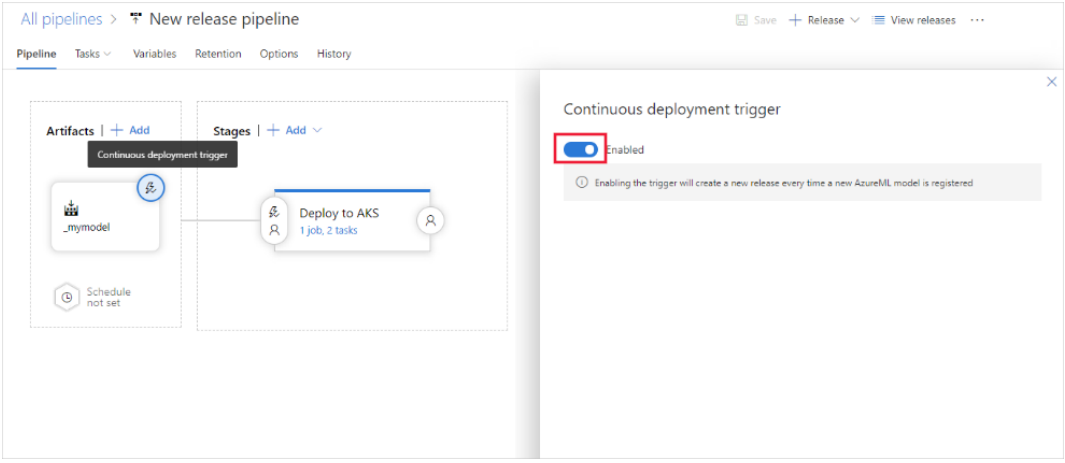
5. In the **Scope level** list, select **AzureMLWorkspace**, and then enter the rest of the values:



6. To continuously deploy your machine learning model by using Azure Pipelines, under pipelines, select **release**. Add a new artifact, and then select the **AzureML Model** artifact and the service connection that you created earlier. Select the model and version to trigger a deployment:



7. Enable the model trigger on your model artifact. When you turn on the trigger, every time the specified version (that is, the newest version) of that model is registered in your workspace, an Azure DevOps release pipeline is triggered.



For more sample projects and examples, see these sample repos in GitHub:

- Microsoft/MLOps
- Microsoft/MLOpsPython

Download a model

If you want to download your model to use it in your own execution environment, you can do so with the following SDK / CLI commands:

SDK:

```
model_path = Model(ws,'mymodel').download()
```

CLI:

```
az ml model download --model-id mymodel:1 --target-dir model_folder
```

(Preview) No-code model deployment

No-code model deployment is currently in preview and supports the following machine learning frameworks:

Tensorflow SavedModel format

Tensorflow models need to be registered in **SavedModel format** to work with no-code model deployment.

Please see this link for information on how to create a SavedModel.

```
from azureml.core import Model

model = Model.register(workspace=ws,
                       model_name='flowers',                          # Name of the registered model
in your workspace.
                       model_path='./flowers_model',                  # Local Tensorflow SavedModel
folder to upload and register as a model.
                       model_framework=Model.Framework.TENSORFLOW,  # Framework used to create the
model.
                       model_framework_version='1.14.0',            # Version of Tensorflow used
to create the model.
                       description='Flowers model')

service_name = 'tensorflow-flower-service'
service = Model.deploy(ws, service_name, [model])
```

ONNX models

ONNX model registration and deployment is supported for any ONNX inference graph. Preprocess and postprocess steps are not currently supported.

Here is an example of how to register and deploy an MNIST ONNX model:

```
from azureml.core import Model

model = Model.register(workspace=ws,
                       model_name='mnist-sample',                   # Name of the registered model
in your workspace.
                       model_path='mnist-model.onnx',               # Local ONNX model to upload
and register as a model.
                       model_framework=Model.Framework.ONNX ,       # Framework used to create the
model.
                       model_framework_version='1.3',               # Version of ONNX used to
create the model.
                       description='Onnx MNIST model')

service_name = 'onnx-mnist-service'
service = Model.deploy(ws, service_name, [model])
```

If you're using Pytorch, Exporting models from PyTorch to ONNX has the details on conversion and limitations.

Scikit-learn models

No code model deployment is supported for all built-in scikit-learn model types.

Here is an example of how to register and deploy a sklearn model with no extra code:

```
from azureml.core import Model
from azureml.core.resource_configuration import ResourceConfiguration

model = Model.register(workspace=ws,
                       model_name='my-sklearn-model',                # Name of the registered
model in your workspace.
                       model_path='./sklearn_regression_model.pkl',  # Local file to upload and
register as a model.
                       model_framework=Model.Framework.SCIKITLEARN,  # Framework used to create
the model.
                       model_framework_version='0.19.1',             # Version of scikit-learn
used to create the model.
                       resource_configuration=ResourceConfiguration(cpu=1, memory_in_gb=0.5),
                       description='Ridge regression model to predict diabetes progression.',
                       tags={'area': 'diabetes', 'type': 'regression'})

service_name = 'my-sklearn-service'
service = Model.deploy(ws, service_name, [model])
```

NOTE: Models which support predict_proba will use that method by default. To override this to use predict you can modify the POST body as below:

```
import json


input_payload = json.dumps({
    'data': [
        [ 0.03807591,  0.05068012,  0.06169621, 0.02187235, -0.0442235,
         -0.03482076, -0.04340085, -0.00259226, 0.01990842, -0.01764613]
    ],
    'method': 'predict'  # If you have a classification model, the default behavior is to run
'predict_proba'.
})

output = service.run(input_payload)

print(output)
```

NOTE: These dependencies are included in the prebuilt sklearn inference container:

```
    - azureml-defaults
    - inference-schema[numpy-support]
    - scikit-learn
    - numpy
```

Package models

In some cases, you might want to create a Docker image without deploying the model (if, for example, you plan to deploy to Azure App Service). Or you might want to download the image and run it on a local Docker installation. You might even want to download the files used to build the image, inspect them, modify them, and build the image manually.

Model packaging enables you to do these things. It packages all the assets needed to host a model as a web service and allows you to download either a fully built Docker image or the files needed to build one. There are two ways to use model packaging:

Download a packaged model: Download a Docker image that contains the model and other files needed to host it as a web service.

Generate a Dockerfile: Download the Dockerfile, model, entry script, and other assets needed to build a Docker image. You can then inspect the files or make changes before you build the image locally.

Both packages can be used to get a local Docker image.

> **TIP**
> Creating a package is similar to deploying a model. You use a registered model and an inference configuration.

> **IMPORTANT**
> To download a fully built image or build an image locally, you need to have Docker installed in your development environment.

Download a packaged model

The following example builds an image, which is registered in the Azure container registry for your workspace:

```
package = Model.package(ws, [model], inference_config)
package.wait_for_creation(show_output=True)
```

After you create a package, you can use `package.pull()` to pull the image to your local Docker environment. The output of this command will display the name of the image. For example:

`Status: Downloaded newer image for myworkspacef78fd10.azurecr.io/package:20190822181338`.

After you download the model, use the `docker images` command to list the local images:

```
REPOSITORY                                TAG                 IMAGE ID            CREATED
SIZE
myworkspacef78fd10.azurecr.io/package     20190822181338      7ff48015d5bd        4 minutes ago
1.43 GB
```

To start a local container based on this image, use the following command to start a named container from the shell or command line. Replace the `<imageid>` value with the image ID returned by the `docker images` command.

```
docker run -p 6789:5001 --name mycontainer <imageid>
```

This command starts the latest version of the image named `myimage`. It maps local port 6789 to the port in the container on which the web service is listening (5001). It also assigns the name `mycontainer` to the container, which makes the container easier to stop. After the container is started, you can submit requests to `http://localhost:6789/score`.

Generate a Dockerfile and dependencies

The following example shows how to download the Dockerfile, model, and other assets needed to build an image locally. The `generate_dockerfile=True` parameter indicates that you want the files, not a fully built image.

```
package = Model.package(ws, [model], inference_config, generate_dockerfile=True)
package.wait_for_creation(show_output=True)
# Download the package.
package.save("./imagefiles")
# Get the Azure container registry that the model/Dockerfile uses.
acr=package.get_container_registry()
print("Address:", acr.address)
print("Username:", acr.username)
print("Password:", acr.password)
```

This code downloads the files needed to build the image to the `imagefiles` directory. The Dockerfile included in the saved files references a base image stored in an Azure container registry. When you build the image on your local Docker installation, you need to use the address, user name, and password to authenticate to the registry. Use the following steps to build the image by using a local Docker installation:

1. From a shell or command-line session, use the following command to authenticate Docker with the Azure container registry. Replace `<address>`, `<username>`, and `<password>` with the values retrieved by `package.get_container_registry()`.

   ```
   docker login <address> -u <username> -p <password>
   ```

2. To build the image, use the following command. Replace `<imagefiles>` with the path of the directory where `package.save()` saved the files.

   ```
   docker build --tag myimage <imagefiles>
   ```

   This command sets the image name to `myimage`.

To verify that the image is built, use the `docker images` command. You should see the `myimage` image in the list:

```
REPOSITORY      TAG                 IMAGE ID            CREATED             SIZE
<none>          <none>              2d5ee0bf3b3b        49 seconds ago      1.43 GB
myimage         latest              739f22498d64        3 minutes ago       1.43 GB
```

To start a new container based on this image, use the following command:

```
docker run -p 6789:5001 --name mycontainer myimage:latest
```

This command starts the latest version of the image named `myimage`. It maps local port 6789 to the port in the container on which the web service is listening (5001). It also assigns the name `mycontainer` to the container, which makes the container easier to stop. After the container is started, you can submit requests to `http://localhost:6789/score`.

Example client to test the local container

The following code is an example of a Python client that can be used with the container:

```
import requests
import json

# URL for the web service.
scoring_uri = 'http://localhost:6789/score'

# Two sets of data to score, so we get two results back.
data = {"data":
        [
            [ 1,2,3,4,5,6,7,8,9,10 ],
            [ 10,9,8,7,6,5,4,3,2,1 ]
        ]
        }
# Convert to JSON string.
input_data = json.dumps(data)

# Set the content type.
headers = {'Content-Type': 'application/json'}

# Make the request and display the response.
resp = requests.post(scoring_uri, input_data, headers=headers)
print(resp.text)
```

For example clients in other programming languages, see Consume models deployed as web services.

Stop the Docker container

To stop the container, use the following command from a different shell or command line:

```
docker kill mycontainer
```

Clean up resources

To delete a deployed web service, use `service.delete()`. To delete a registered model, use `model.delete()`.

For more information, see the documentation for WebService.delete() and Model.delete().

Advanced entry script authoring

Binary data

If your model accepts binary data, like an image, you must modify the `score.py` file used for your deployment to accept raw HTTP requests. To accept raw data, use the `AMLRequest` class in your entry script and add the `@rawhttp` decorator to the `run()` function.

Here's an example of a `score.py` that accepts binary data:

```
from azureml.contrib.services.aml_request import AMLRequest, rawhttp
from azureml.contrib.services.aml_response import AMLResponse


def init():
    print("This is init()")


@rawhttp
def run(request):
    print("This is run()")
    print("Request: [{0}]".format(request))
    if request.method == 'GET':
        # For this example, just return the URL for GETs.
        respBody = str.encode(request.full_path)
        return AMLResponse(respBody, 200)
    elif request.method == 'POST':
        reqBody = request.get_data(False)
        # For a real-world solution, you would load the data from reqBody
        # and send it to the model. Then return the response.

        # For demonstration purposes, this example just returns the posted data as the response.
        return AMLResponse(reqBody, 200)
    else:
        return AMLResponse("bad request", 500)
```

IMPORTANT

The `AMLRequest` class is in the `azureml.contrib` namespace. Entities in this namespace change frequently as we work to improve the service. Anything in this namespace should be considered a preview that's not fully supported by Microsoft.

If you need to test this in your local development environment, you can install the components by using the following command:

```
pip install azureml-contrib-services
```

The `AMLRequest` class only allows you to access the raw posted data in the score.py, there is no client-side component. From a client, you post data as normal. For example, the following Python code reads an image file and posts the data:

```
import requests
# Load image data
data = open('example.jpg', 'rb').read()
# Post raw data to scoring URI
res = request.post(url='<scoring-uri>', data=data, headers={'Content-Type': 'application/octet-stream'})
```

Cross-origin resource sharing (CORS)

Cross-origin resource sharing is a way to allow resources on a webpage to be requested from another domain. CORS works via HTTP headers sent with the client request and returned with the service response. For more information on CORS and valid headers, see Cross-origin resource sharing in Wikipedia.

To configure your model deployment to support CORS, use the `AMLResponse` class in your entry script. This class allows you to set the headers on the response object.

The following example sets the `Access-Control-Allow-Origin` header for the response from the entry

script:

```
from azureml.contrib.services.aml_response import AMLResponse

def init():
    print("This is init()")

def run(request):
    print("This is run()")
    print("Request: [{0}]".format(request))
    if request.method == 'GET':
        # For this example, just return the URL for GETs.
        respBody = str.encode(request.full_path)
        return AMLResponse(respBody, 200)
    elif request.method == 'POST':
        reqBody = request.get_data(False)
        # For a real-world solution, you would load the data from reqBody
        # and send it to the model. Then return the response.

        # For demonstration purposes, this example
        # adds a header and returns the request body.
        resp = AMLResponse(reqBody, 200)
        resp.headers['Access-Control-Allow-Origin'] = "http://www.example.com"
        return resp
    else:
        return AMLResponse("bad request", 500)
```

IMPORTANT

The `AMLResponse` class is in the `azureml.contrib` namespace. Entities in this namespace change frequently as we work to improve the service. Anything in this namespace should be considered a preview that's not fully supported by Microsoft.

If you need to test this in your local development environment, you can install the components by using the following command:

```
pip install azureml-contrib-services
```

WARNING

Azure Machine Learning will route only POST and GET requests to the containers running the scoring service. This can cause errors due to browsers using OPTIONS requests to pre-flight CORS requests.

Next steps

- How to deploy a model using a custom Docker image
- Deployment troubleshooting
- Use TLS to secure a web service through Azure Machine Learning
- Consume an Azure Machine Learning model deployed as a web service
- Monitor your Azure Machine Learning models with Application Insights
- Collect data for models in production
- Create event alerts and triggers for model deployments

Deploy a model to Azure Machine Learning compute instances

4/6/2020 • 2 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to use Azure Machine Learning to deploy a model as a web service on your Azure Machine Learning compute instance. Use compute instances if one of the following conditions is true:

- You need to quickly deploy and validate your model.
- You are testing a model that is under development.

> **TIP**
> Deploying a model from a Jupyter Notebook on a compute instance, to a web service on the same VM is a *local deployment*. In this case, the 'local' computer is the compute instance. For more information on deployments, see Deploy models with Azure Machine Learning.

Prerequisites

- An Azure Machine Learning workspace with a compute instance running. For more information, see Setup environment and workspace.

Deploy to the compute instances

An example notebook that demonstrates local deployments is included on your compute instance. Use the following steps to load the notebook and deploy the model as a web service on the VM:

1. From Azure Machine Learning studio, select your Azure Machine Learning compute instances.
2. Open the `samples-*` subdirectory, and then open `how-to-use-azureml/deploy-to-local/register-model-deploy-local.ipynb`. Once open, run the notebook.

```
from azureml.core.webservice import LocalWebservice

#this is optional, if not provided we choose random port
deployment_config = LocalWebservice.deploy_configuration(port=6789)

local_service = Model.deploy(ws, "test", [model], inference_config, deployment_config)

local_service.wait_for_deployment()
```

```
Requirement already satisfied: PyJWT>=1.0.0 in /opt/miniconda/lib/python3.6/site-packages (from adal>=0.4.5->azureml-model-ma
nagement-sdk==1.0.1b6.post1->-r /var/azureml-app/requirements.txt (line 1)) (1.7.1)

Requirement already satisfied: cffi!=1.11.3,>=1.8 in /opt/miniconda/lib/python3.6/site-packages (from cryptography>=1.1.0->ad
al>=0.4.5->azureml-model-management-sdk==1.0.1b6.post1->-r /var/azureml-app/requirements.txt (line 1)) (1.12.3)
Requirement already satisfied: asn1crypto>=0.21.0 in /opt/miniconda/lib/python3.6/site-packages (from cryptography>=1.1.0->ad
al>=0.4.5->azureml-model-management-sdk==1.0.1b6.post1->-r /var/azureml-app/requirements.txt (line 1)) (0.24.0)
Requirement already satisfied: pycparser in /opt/miniconda/lib/python3.6/site-packages (from cffi!=1.11.3,>=1.8->cryptography
>=1.1.0->adal>=0.4.5->azureml-model-management-sdk==1.0.1b6.post1->-r /var/azureml-app/requirements.txt (line 1)) (2.19)
 ---> fe3a3661fb0b
Step 7/7 : CMD ["runsvdir","/var/runit"]
 ---> Running in 3611ce505d62
 ---> ece2403f94cc
Successfully built ece2403f94cc
Successfully tagged test:latest
Downloading model sklearn_regression_model.pkl:1 to /tmp/azureml_models_test/sklearn_regression_model.pkl/1
Starting Docker container...
Docker container running.
Checking container health...
Local webservice is running at http://localhost:6789
```

3. The notebook displays the URL and port that the service is running on. For example,

`https://localhost:6789`. You can also run the cell containing `print('Local service port: {}'.format(local_service.port))` to display the port.

```
print('Local service port: {}'.format(local_service.port))
```

```
Local service port: 6789
```

4. To test the service from a compute instance, use the `https://localhost:<local_service.port>` URL. To test from a remote client, get the public URL of the service running on the compute instance. The public URL can be determined use the following formula;

   - Notebook VM: `https://<vm_name>-<local_service_port>.<azure_region_of_workspace>.notebooks.azureml.net/score`.
   - Compute instance: `https://<vm_name>-<local_service_port>.<azure_region_of_workspace>.instances.azureml.net/score`.

   For example,

   - Notebook VM: `https://vm-name-6789.northcentralus.notebooks.azureml.net/score`
   - Compute instance: `https://vm-name-6789.northcentralus.instances.azureml.net/score`

Test the service

To submit sample data to the running service, use the following code. Replace the value of `service_url` with the URL of from the previous step:

> **NOTE**
>
> When authenticating to a deployment on the compute instance, the authentication is made using Azure Active Directory. The call to `interactive_auth.get_authentication_header()` in the example code authenticates you using AAD, and returns a header that can then be used to authenticate to the service on the compute instance. For more information, see Set up authentication for Azure Machine Learning resources and workflows.
>
> When authenticating to a deployment on Azure Kubernetes Service or Azure Container Instances, a different authentication method is used. For more information on, see Set up authentication for Azure Machine Learning resources and workflows.

```
import requests
import json
from azureml.core.authentication import InteractiveLoginAuthentication

# Get a token to authenticate to the compute instance from remote
interactive_auth = InteractiveLoginAuthentication()
auth_header = interactive_auth.get_authentication_header()

# Create and submit a request using the auth header
headers = auth_header
# Add content type header
headers.update({'Content-Type':'application/json'})

# Sample data to send to the service
test_sample = json.dumps({'data': [
    [1,2,3,4,5,6,7,8,9,10],
    [10,9,8,7,6,5,4,3,2,1]
]})
test_sample = bytes(test_sample,encoding = 'utf8')

# Replace with the URL for your compute instance, as determined from the previous section
service_url = "https://vm-name-6789.northcentralus.notebooks.azureml.net/score"
# for a compute instance, the url would be https://vm-name-6789.northcentralus.instances.azureml.net/score
resp = requests.post(service_url, test_sample, headers=headers)
print("prediction:", resp.text)
```

Next steps

- How to deploy a model using a custom Docker image
- Deployment troubleshooting
- Use TLS to secure a web service through Azure Machine Learning
- Consume a ML Model deployed as a web service
- Monitor your Azure Machine Learning models with Application Insights
- Collect data for models in production

Deploy a model to an Azure Kubernetes Service cluster

4/7/2020 • 17 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to use Azure Machine Learning to deploy a model as a web service on Azure Kubernetes Service (AKS). Azure Kubernetes Service is good for high-scale production deployments. Use Azure Kubernetes service if you need one or more of the following capabilities:

- **Fast response time**.
- **Autoscaling** of the deployed service.
- **Hardware acceleration** options such as GPU and field-programmable gate arrays (FPGA).

> **IMPORTANT**
> Cluster scaling is not provided through the Azure Machine Learning SDK. For more information on scaling the nodes in an AKS cluster, see Scale the node count in an AKS cluster.

When deploying to Azure Kubernetes Service, you deploy to an AKS cluster that is **connected to your workspace**. There are two ways to connect an AKS cluster to your workspace:

- Create the AKS cluster using the Azure Machine Learning SDK, the Machine Learning CLI, or Azure Machine Learning studio. This process automatically connects the cluster to the workspace.
- Attach an existing AKS cluster to your Azure Machine Learning workspace. A cluster can be attached using the Azure Machine Learning SDK, Machine Learning CLI, or Azure Machine Learning studio.

> **IMPORTANT**
> The creation or attachment process is a one time task. Once an AKS cluster is connected to the workspace, you can use it for deployments. You can detach or delete the AKS cluster if you no longer need it. Once detached or deleted, you will no longer be able to deploy to the cluster.

Prerequisites

- An Azure Machine Learning workspace. For more information, see Create an Azure Machine Learning workspace.
- A machine learning model registered in your workspace. If you don't have a registered model, see How and where to deploy models.
- The Azure CLI extension for Machine Learning service, Azure Machine Learning Python SDK, or the Azure Machine Learning Visual Studio Code extension.
- The **Python** code snippets in this article assume that the following variables are set:
  - `ws` - Set to your workspace.
  - `model` - Set to your registered model.
  - `inference_config` - Set to the inference configuration for the model.

For more information on setting these variables, see How and where to deploy models.

- The **CLI** snippets in this article assume that you've created an `inferenceconfig.json` document. For more information on creating this document, see How and where to deploy models.

Create a new AKS cluster

Time estimate: Approximately 20 minutes.

Creating or attaching an AKS cluster is a one time process for your workspace. You can reuse this cluster for multiple deployments. If you delete the cluster or the resource group that contains it, you must create a new cluster the next time you need to deploy. You can have multiple AKS clusters attached to your workspace.

> **TIP**
> If you want to secure your AKS cluster using an Azure Virtual Network, you must create the virtual network first. For more information, see Secure experimentation and inference with Azure Virtual Network.

If you want to create an AKS cluster for **development**, **validation**, and **testing** instead of production, you can specify the **cluster purpose** to **dev test**.

> **WARNING**
> If you set `cluster_purpose = AksCompute.ClusterPurpose.DEV_TEST`, the cluster that is created is not suitable for production level traffic and may increase inference times. Dev/test clusters also do not guarantee fault tolerance. We recommend at least 2 virtual CPUs for dev/test clusters.

The following examples demonstrate how to create a new AKS cluster using the SDK and CLI:

Using the SDK

```
from azureml.core.compute import AksCompute, ComputeTarget

# Use the default configuration (you can also provide parameters to customize this).
# For example, to create a dev/test cluster, use:
# prov_config = AksCompute.provisioning_configuration(cluster_purpose = AksCompute.ClusterPurpose.DEV_TEST)
prov_config = AksCompute.provisioning_configuration()

aks_name = 'myaks'
# Create the cluster
aks_target = ComputeTarget.create(workspace = ws,
                                  name = aks_name,
                                  provisioning_configuration = prov_config)

# Wait for the create process to complete
aks_target.wait_for_completion(show_output = True)
```

IMPORTANT

For `provisioning_configuration()`, if you pick custom values for `agent_count` and `vm_size`, and `cluster_purpose` is not `DEV_TEST`, then you need to make sure `agent_count` multiplied by `vm_size` is greater than or equal to 12 virtual CPUs. For example, if you use a `vm_size` of "Standard_D3_v2", which has 4 virtual CPUs, then you should pick an `agent_count` of 3 or greater.

The Azure Machine Learning SDK does not provide support scaling an AKS cluster. To scale the nodes in the cluster, use the UI for your AKS cluster in the Azure Machine Learning studio. You can only change the node count, not the VM size of the cluster.

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- AksCompute.ClusterPurpose
- AksCompute.provisioning_configuration
- ComputeTarget.create
- ComputeTarget.wait_for_completion

Using the CLI

```
az ml computetarget create aks -n myaks
```

For more information, see the az ml computetarget create aks reference.

Attach an existing AKS cluster

Time estimate: Approximately 5 minutes.

If you already have AKS cluster in your Azure subscription, and it is version 1.17 or lower, you can use it to deploy your image.

TIP

The existing AKS cluster can be in a Azure region other than your Azure Machine Learning workspace.

If you want to secure your AKS cluster using an Azure Virtual Network, you must create the virtual network first. For more information, see Secure experimentation and inference with Azure Virtual Network.

When attaching an AKS cluster to a workspace, you can define how you will use the cluster by setting the `cluster_purpose` parameter.

If you do not set the `cluster_purpose` parameter, or set `cluster_purpose = AksCompute.ClusterPurpose.FAST_PROD`, then the cluster must have at least 12 virtual CPUs available.

If you set `cluster_purpose = AksCompute.ClusterPurpose.DEV_TEST`, then the cluster does not need to have 12 virtual CPUs. We recommend at least 2 virtual CPUs for dev/test. However a cluster that is configured for dev/test is not suitable for production level traffic and may increase inference times. Dev/test clusters also do not guarantee fault tolerance.

> **WARNING**
> Do not create multiple, simultaneous attachments to the same AKS cluster from your workspace. For example, attaching one AKS cluster to a workspace using two different names. Each new attachment will break the previous existing attachment(s).
>
> If you want to re-attach an AKS cluster, for example to change TLS or other cluster configuration setting, you must first remove the existing attachment by using AksCompute.detach().

For more information on creating an AKS cluster using the Azure CLI or portal, see the following articles:

- Create an AKS cluster (CLI)
- Create an AKS cluster (portal)

The following examples demonstrate how to attach an existing AKS cluster to your workspace:

Using the SDK

```
from azureml.core.compute import AksCompute, ComputeTarget
# Set the resource group that contains the AKS cluster and the cluster name
resource_group = 'myresourcegroup'
cluster_name = 'myexistingcluster'

# Attach the cluster to your workgroup. If the cluster has less than 12 virtual CPUs, use the following
instead:
# attach_config = AksCompute.attach_configuration(resource_group = resource_group,
#                                         cluster_name = cluster_name,
#                                         cluster_purpose = AksCompute.ClusterPurpose.DEV_TEST)
attach_config = AksCompute.attach_configuration(resource_group = resource_group,
                                        cluster_name = cluster_name)
aks_target = ComputeTarget.attach(ws, 'myaks', attach_config)
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- AksCompute.attach_configuration()
- AksCompute.ClusterPurpose
- AksCompute.attach

Using the CLI

To attach an existing cluster using the CLI, you need to get the resource ID of the existing cluster. To get this value, use the following command. Replace `myexistingcluster` with the name of your AKS cluster. Replace `myresourcegroup` with the resource group that contains the cluster:

```
az aks show -n myexistingcluster -g myresourcegroup --query id
```

This command returns a value similar to the following text:

```
/subscriptions/{GUID}/resourcegroups/{myresourcegroup}/providers/Microsoft.ContainerService/managedClusters/{
myexistingcluster}
```

To attach the existing cluster to your workspace, use the following command. Replace `aksresourceid` with the value returned by the previous command. Replace `myresourcegroup` with the resource group that contains your workspace. Replace `myworkspace` with your workspace name.

```
az ml computetarget attach aks -n myaks -i aksresourceid -g myresourcegroup -w myworkspace
```

For more information, see the az ml computetarget attach aks reference.

Deploy to AKS

To deploy a model to Azure Kubernetes Service, create a **deployment configuration** that describes the compute resources needed. For example, number of cores and memory. You also need an **inference configuration**, which describes the environment needed to host the model and web service. For more information on creating the inference configuration, see How and where to deploy models.

Using the SDK

```
from azureml.core.webservice import AksWebservice, Webservice
from azureml.core.model import Model

aks_target = AksCompute(ws,"myaks")
# If deploying to a cluster configured for dev/test, ensure that it was created with enough
# cores and memory to handle this deployment configuration. Note that memory is also used by
# things such as dependencies and AML components.
deployment_config = AksWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)
service = Model.deploy(ws, "myservice", [model], inference_config, deployment_config, aks_target)
service.wait_for_deployment(show_output = True)
print(service.state)
print(service.get_logs())
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- AksCompute
- AksWebservice.deploy_configuration
- Model.deploy
- Webservice.wait_for_deployment

Using the CLI

To deploy using the CLI, use the following command. Replace `myaks` with the name of the AKS compute target. Replace `mymodel:1` with the name and version of the registered model. Replace `myservice` with the name to give this service:

```
az ml model deploy -ct myaks -m mymodel:1 -n myservice -ic inferenceconfig.json -dc deploymentconfig.json
```

The entries in the `deploymentconfig.json` document map to the parameters for AksWebservice.deploy_configuration. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`computeType`	NA	The compute target. For AKS, the value must be `aks`.
`autoScaler`	NA	Contains configuration elements for autoscale. See the autoscaler table.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`autoscaleEnabled`	`autoscale_enabled`	Whether to enable autoscaling for the web service. If `numReplicas` = `0`, `True`; otherwise, `False`.
`minReplicas`	`autoscale_min_replicas`	The minimum number of containers to use when autoscaling this web service. Default, `1`.
`maxReplicas`	`autoscale_max_replicas`	The maximum number of containers to use when autoscaling this web service. Default, `10`.
`refreshPeriodInSeconds`	`autoscale_refresh_seconds`	How often the autoscaler attempts to scale this web service. Default, `1`.
`targetUtilization`	`autoscale_target_utilization`	The target utilization (in percent out of 100) that the autoscaler should attempt to maintain for this web service. Default, `70`.
`dataCollection`	NA	Contains configuration elements for data collection.
`storageEnabled`	`collect_model_data`	Whether to enable model data collection for the web service. Default, `False`.
`authEnabled`	`auth_enabled`	Whether or not to enable key authentication for the web service. Both `tokenAuthEnabled` and `authEnabled` cannot be `True`. Default, `True`.
`tokenAuthEnabled`	`token_auth_enabled`	Whether or not to enable token authentication for the web service. Both `tokenAuthEnabled` and `authEnabled` cannot be `True`. Default, `False`.
`containerResourceRequirements`	NA	Container for the CPU and memory entities.
`cpu`	`cpu_cores`	The number of CPU cores to allocate for this web service. Defaults, `0.1`
`memoryInGB`	`memory_gb`	The amount of memory (in GB) to allocate for this web service. Default, `0.5`
`appInsightsEnabled`	`enable_app_insights`	Whether to enable Application Insights logging for the web service. Default, `False`.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`scoringTimeoutMs`	`scoring_timeout_ms`	A timeout to enforce for scoring calls to the web service. Default, `60000`.
`maxConcurrentRequestsPerContainer`	`replica_max_concurrent_requests`	The maximum concurrent requests per node for this web service. Default, `1`.
`maxQueueWaitMs`	`max_request_wait_time`	The maximum time a request will stay in thee queue (in milliseconds) before a 503 error is returned. Default, `500`.
`numReplicas`	`num_replicas`	The number of containers to allocate for this web service. No default value. If this parameter is not set, the autoscaler is enabled by default.
`keys`	NA	Contains configuration elements for keys.
`primaryKey`	`primary_key`	A primary auth key to use for this Webservice
`secondaryKey`	`secondary_key`	A secondary auth key to use for this Webservice
`gpuCores`	`gpu_cores`	The number of GPU cores (per-container replica) to allocate for this Webservice. Default is 1. Only supports whole number values.
`livenessProbeRequirements`	NA	Contains configuration elements for liveness probe requirements.
`periodSeconds`	`period_seconds`	How often (in seconds) to perform the liveness probe. Default to 10 seconds. Minimum value is 1.
`initialDelaySeconds`	`initial_delay_seconds`	Number of seconds after the container has started before liveness probes are initiated. Defaults to 310
`timeoutSeconds`	`timeout_seconds`	Number of seconds after which the liveness probe times out. Defaults to 2 seconds. Minimum value is 1
`successThreshold`	`success_threshold`	Minimum consecutive successes for the liveness probe to be considered successful after having failed. Defaults to 1. Minimum value is 1.
`failureThreshold`	`failure_threshold`	When a Pod starts and the liveness probe fails, Kubernetes will try failureThreshold times before giving up. Defaults to 3. Minimum value is 1.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`namespace`	`namespace`	The Kubernetes namespace that the webservice is deployed into. Up to 63 lowercase alphanumeric ('a'-'z', '0'-'9') and hyphen ('-') characters. The first and last characters can't be hyphens.

The following JSON is an example deployment configuration for use with the CLI:

```
{
    "computeType": "aks",
    "autoScaler":
    {
        "autoscaleEnabled": true,
        "minReplicas": 1,
        "maxReplicas": 3,
        "refreshPeriodInSeconds": 1,
        "targetUtilization": 70
    },
    "dataCollection":
    {
        "storageEnabled": true
    },
    "authEnabled": true,
    "containerResourceRequirements":
    {
        "cpu": 0.5,
        "memoryInGB": 1.0
    }
}
```

For more information, see the az ml model deploy reference.

Using VS Code

For information on using VS Code, see deploy to AKS via the VS Code extension.

> **IMPORTANT**
>
> Deploying through VS Code requires the AKS cluster to be created or attached to your workspace in advance.

Deploy models to AKS using controlled rollout (preview)

Analyze and promote model versions in a controlled fashion using endpoints. You can deploy up to six versions behind a single endpoint. Endpoints provide the following capabilities:

- Configure the **percentage of scoring traffic sent to each endpoint**. For example, route 20% of the traffic to endpoint 'test' and 80% to 'production'.

  > **NOTE**
  >
  > If you do not account for 100% of the traffic, any remaining percentage is routed to the **default** endpoint version. For example, if you configure endpoint version 'test' to get 10% of the traffic, and 'prod' for 30%, the remaining 60% is sent to the default endpoint version.
  >
  > The first endpoint version created is automatically configured as the default. You can change this by setting `is_default=True` when creating or updating an endpoint version.

- Tag an endpoint version as either **control** or **treatment**. For example, the current production endpoint version might be the control, while potential new models are deployed as treatment versions. After evaluating performance of the treatment versions, if one outperforms the current control, it might be promoted to the new production/control.

  > **NOTE**
  > You can only have **one** control. You can have multiple treatments.

You can enable app insights to view operational metrics of endpoints and deployed versions.

Create an endpoint

Once you are ready to deploy your models, create a scoring endpoint and deploy your first version. The following example shows how to deploy and create the endpoint using the SDK. The first deployment will be defined as the default version, which means that unspecified traffic percentile across all versions will go to the default version.

> **TIP**
> In the following example, the configuration sets the initial endpoint version to handle 20% of the traffic. Since this is the first endpoint, it's also the default version. And since we don't have any other versions for the other 80% of traffic, it is routed to the default as well. Until other versions that take a percentage of traffic are deployed, this one effectively receives 100% of the traffic.

```
import azureml.core,
from azureml.core.webservice import AksEndpoint
from azureml.core.compute import AksCompute
from azureml.core.compute import ComputeTarget
# select a created compute
compute = ComputeTarget(ws, 'myaks')
namespace_name= endpointnamespace
# define the endpoint and version name
endpoint_name = "mynewendpoint"
version_name= "versiona"
# create the deployment config and define the scoring traffic percentile for the first deployment
endpoint_deployment_config = AksEndpoint.deploy_configuration(cpu_cores = 0.1, memory_gb = 0.2,
                                                              enable_app_insights = True,
                                                              tags = {'sckitlearn':'demo'},
                                                              description = "testing versions",
                                                              version_name = version_name,
                                                              traffic_percentile = 20)
 # deploy the model and endpoint
 endpoint = Model.deploy(ws, endpoint_name, [model], inference_config, endpoint_deployment_config, compute)
 # Wait for he process to complete
 endpoint.wait_for_deployment(True)
```

Update and add versions to an endpoint

Add another version to your endpoint and configure the scoring traffic percentile going to the version. There are two types of versions, a control and a treatment version. There can be multiple treatment versions to help compare against a single control version.

> **TIP**
> The second version, created by the following code snippet, accepts 10% of traffic. The first version is configured for 20%, so only 30% of the traffic is configured for specific versions. The remaining 70% is sent to the first endpoint version, because it is also the default version.

```
from azureml.core.webservice import AksEndpoint

# add another model deployment to the same endpoint as above
version_name_add = "versionb"
endpoint.create_version(version_name = version_name_add,
                        inference_config=inference_config,
                        models=[model],
                        tags = {'modelVersion':'b'},
                        description = "my second version",
                        traffic_percentile = 10)
endpoint.wait_for_deployment(True)
```

Update existing versions or delete them in an endpoint. You can change the version's default type, control type, and the traffic percentile. In the following example, the second version increases its traffic to 40% and is now the default.

> **TIP**
> After the following code snippet, the second version is now default. It is now configured for 40%, while the original version is still configured for 20%. This means that 40% of traffic is not accounted for by version configurations. The leftover traffic will be routed to the second version, because it is now default. It effectively receives 80% of the traffic.

```
from azureml.core.webservice import AksEndpoint

# update the version's scoring traffic percentage and if it is a default or control type
endpoint.update_version(version_name=endpoint.versions["versionb"].name,
                        description="my second version update",
                        traffic_percentile=40,
                        is_default=True,
                        is_control_version_type=True)
# Wait for the process to complete before deleting
endpoint.wait_for_deployment(true)
# delete a version in an endpoint
endpoint.delete_version(version_name="versionb")
```

Web service authentication

When deploying to Azure Kubernetes Service, **key-based** authentication is enabled by default. You can also enable **token-based** authentication. Token-based authentication requires clients to use an Azure Active Directory account to request an authentication token, which is used to make requests to the deployed service.

To **disable** authentication, set the `auth_enabled=False` parameter when creating the deployment configuration. The following example disables authentication using the SDK:

```
deployment_config = AksWebservice.deploy_configuration(cpu_cores=1, memory_gb=1, auth_enabled=False)
```

For information on authenticating from a client application, see the Consume an Azure Machine Learning model deployed as a web service.

Authentication with keys

If key authentication is enabled, you can use the `get_keys` method to retrieve a primary and secondary authentication key:

```
primary, secondary = service.get_keys()
print(primary)
```

IMPORTANT

If you need to regenerate a key, use `service.regen_key`

Authentication with tokens

To enable token authentication, set the `token_auth_enabled=True` parameter when you are creating or updating a deployment. The following example enables token authentication using the SDK:

```
deployment_config = AksWebservice.deploy_configuration(cpu_cores=1, memory_gb=1, token_auth_enabled=True)
```

If token authentication is enabled, you can use the `get_token` method to retrieve a JWT token and that token's expiration time:

```
token, refresh_by = service.get_token()
print(token)
```

IMPORTANT

You will need to request a new token after the token's `refresh_by` time.

Microsoft strongly recommends that you create your Azure Machine Learning workspace in the same region as your Azure Kubernetes Service cluster. To authenticate with a token, the web service will make a call to the region in which your Azure Machine Learning workspace is created. If your workspace's region is unavailable, then you will not be able to fetch a token for your web service even, if your cluster is in a different region than your workspace. This effectively results in Token-based Authentication being unavailable until your workspace's region is available again. In addition, the greater the distance between your cluster's region and your workspace's region, the longer it will take to fetch a token.

Update the web service

To update a web service, use the `update` method. You can update the web service to use a new model, a new entry script, or new dependencies that can be specified in an inference configuration. For more information, see the documentation for Webservice.update.

IMPORTANT

When you create a new version of a model, you must manually update each service that you want to use it.

You can not use the SDK to update a web service published from the Azure Machine Learning designer.

Using the SDK

The following code shows how to use the SDK to update the model, environment, and entry script for a web service:

```
from azureml.core import Environment
from azureml.core.webservice import Webservice
from azureml.core.model import Model, InferenceConfig

# Register new model.
new_model = Model.register(model_path="outputs/sklearn_mnist_model.pkl",
                           model_name="sklearn_mnist",
                           tags={"key": "0.1"},
                           description="test",
                           workspace=ws)

# Use version 3 of the environment.
deploy_env = Environment.get(workspace=ws,name="myenv",version="3")
inference_config = InferenceConfig(entry_script="score.py",
                                   environment=deploy_env)

service_name = 'myservice'
# Retrieve existing service.
service = Webservice(name=service_name, workspace=ws)



# Update to new model(s).
service.update(models=[new_model], inference_config=inference_config)
print(service.state)
print(service.get_logs())
```

Using the CLI

You can also update a web service by using the ML CLI. The following example demonstrates registering a new model and then updating a web service to use the new model:

```
az ml model register -n sklearn_mnist  --asset-path outputs/sklearn_mnist_model.pkl  --experiment-name
myexperiment --output-metadata-file modelinfo.json
az ml service update -n myservice --model-metadata-file modelinfo.json
```

TIP

In this example, a JSON document is used to pass the model information from the registration command into the update command.

To update the service to use a new entry script or environment, create an inference configuration file and specify it with the `ic` parameter.

For more information, see the az ml service update documentation.

Next steps

- Secure experimentation and inference in a virtual network
- How to deploy a model using a custom Docker image
- Deployment troubleshooting
- Use TLS to secure a web service through Azure Machine Learning
- Consume a ML Model deployed as a web service
- Monitor your Azure Machine Learning models with Application Insights
- Collect data for models in production

Deploy a model to Azure Container Instances

3/31/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to use Azure Machine Learning to deploy a model as a web service on Azure Container Instances (ACI). Use Azure Container Instances if one of the following conditions is true:

- You need to quickly deploy and validate your model. You do not need to create ACI containers ahead of time. They are created as part of the deployment process.
- You are testing a model that is under development.

For information on quota and region availability for ACI, see Quotas and region availability for Azure Container Instances article.

Prerequisites

- An Azure Machine Learning workspace. For more information, see Create an Azure Machine Learning workspace.
- A machine learning model registered in your workspace. If you don't have a registered model, see How and where to deploy models.
- The Azure CLI extension for Machine Learning service, Azure Machine Learning Python SDK, or the Azure Machine Learning Visual Studio Code extension.
- The **Python** code snippets in this article assume that the following variables are set:
  - `ws` - Set to your workspace.
  - `model` - Set to your registered model.
  - `inference_config` - Set to the inference configuration for the model.

  For more information on setting these variables, see How and where to deploy models.
- The **CLI** snippets in this article assume that you've created an `inferenceconfig.json` document. For more information on creating this document, see How and where to deploy models.

Deploy to ACI

To deploy a model to Azure Container Instances, create a **deployment configuration** that describes the compute resources needed. For example, number of cores and memory. You also need an **inference configuration**, which describes the environment needed to host the model and web service. For more information on creating the inference configuration, see How and where to deploy models.

Using the SDK

```
from azureml.core.webservice import AciWebservice, Webservice
from azureml.core.model import Model

deployment_config = AciWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)
service = Model.deploy(ws, "aciservice", [model], inference_config, deployment_config)
service.wait_for_deployment(show_output = True)
print(service.state)
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- AciWebservice.deploy_configuration
- Model.deploy
- Webservice.wait_for_deployment

Using the CLI

To deploy using the CLI, use the following command. Replace `mymodel:1` with the name and version of the registered model. Replace `myservice` with the name to give this service:

```
az ml model deploy -m mymodel:1 -n myservice -ic inferenceconfig.json -dc deploymentconfig.json
```

The entries in the `deploymentconfig.json` document map to the parameters for AciWebservice.deploy_configuration. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`computeType`	NA	The compute target. For ACI, the value must be `ACI`.
`containerResourceRequirements`	NA	Container for the CPU and memory entities.
`cpu`	`cpu_cores`	The number of CPU cores to allocate. Defaults, `0.1`
`memoryInGB`	`memory_gb`	The amount of memory (in GB) to allocate for this web service. Default, `0.5`
`location`	`location`	The Azure region to deploy this Webservice to. If not specified the Workspace location will be used. More details on available regions can be found here: ACI Regions
`authEnabled`	`auth_enabled`	Whether to enable auth for this Webservice. Defaults to False
`sslEnabled`	`ssl_enabled`	Whether to enable SSL for this Webservice. Defaults to False.
`appInsightsEnabled`	`enable_app_insights`	Whether to enable AppInsights for this Webservice. Defaults to False
`sslCertificate`	`ssl_cert_pem_file`	The cert file needed if SSL is enabled
`sslKey`	`ssl_key_pem_file`	The key file needed if SSL is enabled
`cname`	`ssl_cname`	The cname for if SSL is enabled

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`dnsNameLabel`	`dns_name_label`	The dns name label for the scoring endpoint. If not specified a unique dns name label will be generated for the scoring endpoint.

The following JSON is an example deployment configuration for use with the CLI:

```
{
    "computeType": "aci",
    "containerResourceRequirements":
    {
        "cpu": 0.5,
        "memoryInGB": 1.0
    },
    "authEnabled": true,
    "sslEnabled": false,
    "appInsightsEnabled": false
}
```

For more information, see the az ml model deploy reference.

Using VS Code

See deploy your models with VS Code.

> **IMPORTANT**
> You don't need to create an ACI container to test in advance. ACI containers are created as needed.

Update the web service

To update a web service, use the `update` method. You can update the web service to use a new model, a new entry script, or new dependencies that can be specified in an inference configuration. For more information, see the documentation for Webservice.update.

> **IMPORTANT**
> When you create a new version of a model, you must manually update each service that you want to use it.
>
> You can not use the SDK to update a web service published from the Azure Machine Learning designer.

Using the SDK

The following code shows how to use the SDK to update the model, environment, and entry script for a web service:

```
from azureml.core import Environment
from azureml.core.webservice import Webservice
from azureml.core.model import Model, InferenceConfig

# Register new model.
new_model = Model.register(model_path="outputs/sklearn_mnist_model.pkl",
                           model_name="sklearn_mnist",
                           tags={"key": "0.1"},
                           description="test",
                           workspace=ws)

# Use version 3 of the environment.
deploy_env = Environment.get(workspace=ws,name="myenv",version="3")
inference_config = InferenceConfig(entry_script="score.py",
                                   environment=deploy_env)

service_name = 'myservice'
# Retrieve existing service.
service = Webservice(name=service_name, workspace=ws)



# Update to new model(s).
service.update(models=[new_model], inference_config=inference_config)
print(service.state)
print(service.get_logs())
```

Using the CLI

You can also update a web service by using the ML CLI. The following example demonstrates registering a new model and then updating a web service to use the new model:

```
az ml model register -n sklearn_mnist  --asset-path outputs/sklearn_mnist_model.pkl  --experiment-name
myexperiment --output-metadata-file modelinfo.json
az ml service update -n myservice --model-metadata-file modelinfo.json
```

TIP

In this example, a JSON document is used to pass the model information from the registration command into the update command.

To update the service to use a new entry script or environment, create an inference configuration file and specify it with the `ic` parameter.

For more information, see the az ml service update documentation.

Next steps

- How to deploy a model using a custom Docker image
- Deployment troubleshooting
- Use TLS to secure a web service through Azure Machine Learning
- Consume a ML Model deployed as a web service
- Monitor your Azure Machine Learning models with Application Insights
- Collect data for models in production

Deploy a deep learning model for inference with GPU

3/5/2020 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article teaches you how to use Azure Machine Learning to deploy a GPU-enabled model as a web service. The information in this article is based on deploying a model on Azure Kubernetes Service (AKS). The AKS cluster provides a GPU resource that is used by the model for inference.

Inference, or model scoring, is the phase where the deployed model is used to make predictions. Using GPUs instead of CPUs offers performance advantages on highly parallelizable computation.

IMPORTANT

For web service deployments, GPU inference is only supported on Azure Kubernetes Service. For inference using a **machine learning pipeline**, GPUs are only supported on Azure Machine Learning Compute. For more information on using ML pipelines, see Run batch predictions.

TIP

Although the code snippets in this article use a TensorFlow model, you can apply the information to any machine learning framework that supports GPUs.

NOTE

The information in this article builds on the information in the How to deploy to Azure Kubernetes Service article. Where that article generally covers deployment to AKS, this article covers GPU specific deployment.

Prerequisites

- An Azure Machine Learning workspace. For more information, see Create an Azure Machine Learning workspace.
- A Python development environment with the Azure Machine Learning SDK installed. For more information, see Azure Machine Learning SDK.
- A registered model that uses a GPU.
  - To learn how to register models, see Deploy Models.
  - To create and register the Tensorflow model used to create this document, see How to Train a TensorFlow Model.
- A general understanding of How and where to deploy models.

Connect to your workspace

To connect to an existing workspace, use the following code:

IMPORTANT

This code snippet expects the workspace configuration to be saved in the current directory or its parent. For more information on creating a workspace, see Create and manage Azure Machine Learning workspaces. For more information on saving the configuration to file, see Create a workspace configuration file.

```
from azureml.core import Workspace

# Connect to the workspace
ws = Workspace.from_config()
```

Create a Kubernetes cluster with GPUs

Azure Kubernetes Service provides many different GPU options. You can use any of them for model inference. See the list of N-series VMs for a full breakdown of capabilities and costs.

The following code demonstrates how to create a new AKS cluster for your workspace:

```
from azureml.core.compute import ComputeTarget, AksCompute
from azureml.exceptions import ComputeTargetException

# Choose a name for your cluster
aks_name = "aks-gpu"

# Check to see if the cluster already exists
try:
    aks_target = ComputeTarget(workspace=ws, name=aks_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    # Provision AKS cluster with GPU machine
    prov_config = AksCompute.provisioning_configuration(vm_size="Standard_NC6")

    # Create the cluster
    aks_target = ComputeTarget.create(
        workspace=ws, name=aks_name, provisioning_configuration=prov_config
    )

    aks_target.wait_for_completion(show_output=True)
```

IMPORTANT

Azure will bill you as long as the AKS cluster exists. Make sure to delete your AKS cluster when you're done with it.

For more information on using AKS with Azure Machine Learning, see How to deploy to Azure Kubernetes Service.

Write the entry script

The entry script receives data submitted to the web service, passes it to the model, and returns the scoring results. The following script loads the Tensorflow model on startup, and then uses the model to score data.

TIP

The entry script is specific to your model. For example, the script must know the framework to use with your model, data formats, etc.

```
import json
import numpy as np
import os
import tensorflow as tf

from azureml.core.model import Model


def init():
    global X, output, sess
    tf.reset_default_graph()
    model_root = os.getenv('AZUREML_MODEL_DIR')
    # the name of the folder in which to look for tensorflow model files
    tf_model_folder = 'model'
    saver = tf.train.import_meta_graph(
        os.path.join(model_root, tf_model_folder, 'mnist-tf.model.meta'))
    X = tf.get_default_graph().get_tensor_by_name("network/X:0")
    output = tf.get_default_graph().get_tensor_by_name("network/output/MatMul:0")

    sess = tf.Session()
    saver.restore(sess, os.path.join(model_root, tf_model_folder, 'mnist-tf.model'))


def run(raw_data):
    data = np.array(json.loads(raw_data)['data'])
    # make prediction
    out = output.eval(session=sess, feed_dict={X: data})
    y_hat = np.argmax(out, axis=1)
    return y_hat.tolist()
```

This file is named `score.py`. For more information on entry scripts, see How and where to deploy.

Define the conda environment

The conda environment file specifies the dependencies for the service. It includes dependencies required by both the model and the entry script. Please note that you must indicate azureml-defaults with verion >= 1.0.45 as a pip dependency, because it contains the functionality needed to host the model as a web service. The following YAML defines the environment for a Tensorflow model. It specifies `tensorflow-gpu`, which will make use of the GPU used in this deployment:

```
name: project_environment
dependencies:
  # The python interpreter version.
  # Currently Azure ML only supports 3.5.2 and later.
- python=3.6.2

- pip:
  # You must list azureml-defaults as a pip dependency
  - azureml-defaults>=1.0.45
- numpy
- tensorflow-gpu=1.12
channels:
- conda-forge
```

For this example, the file is saved as `myenv.yml`.

Define the deployment configuration

The deployment configuration defines the Azure Kubernetes Service environment used to run the web service:

```
from azureml.core.webservice import AksWebservice

gpu_aks_config = AksWebservice.deploy_configuration(autoscale_enabled=False,
                                                    num_replicas=3,
                                                    cpu_cores=2,
                                                    memory_gb=4)
```

For more information, see the reference documentation for AksService.deploy_configuration.

Define the inference configuration

The inference configuration points to the entry script and an environment object, which uses a docker image with GPU support. Please note that the YAML file used for environment definition must list azureml-defaults with version >= 1.0.45 as a pip dependency, because it contains the functionality needed to host the model as a web service.

```
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment, DEFAULT_GPU_IMAGE

myenv = Environment.from_conda_specification(name="myenv", file_path="myenv.yml")
myenv.docker.base_image = DEFAULT_GPU_IMAGE
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
```

For more information on environments, see Create and manage environments for training and deployment. For more information, see the reference documentation for InferenceConfig.

Deploy the model

Deploy the model to your AKS cluster and wait for it to create your service.

```
from azureml.core.model import Model

# Name of the web service that is deployed
aks_service_name = 'aks-dnn-mnist'
# Get the registerd model
model = Model(ws, "tf-dnn-mnist")
# Deploy the model
aks_service = Model.deploy(ws,
                           models=[model],
                           inference_config=inference_config,
                           deployment_config=gpu_aks_config,
                           deployment_target=aks_target,
                           name=aks_service_name)

aks_service.wait_for_deployment(show_output=True)
print(aks_service.state)
```

NOTE

If the `InferenceConfig` object has `enable_gpu=True`, then the `deployment_target` parameter must reference a cluster that provides a GPU. Otherwise, the deployment will fail.

For more information, see the reference documentation for Model.

Issue a sample query to your service

Send a test query to the deployed model. When you send a jpeg image to the model, it scores the image. The following code sample downloads test data and then selects a random test image to send to the service.

```
# Used to test your webservice
import os
import urllib
import gzip
import numpy as np
import struct
import requests

# load compressed MNIST gz files and return numpy arrays
def load_data(filename, label=False):
    with gzip.open(filename) as gz:
        struct.unpack('I', gz.read(4))
        n_items = struct.unpack('>I', gz.read(4))
        if not label:
            n_rows = struct.unpack('>I', gz.read(4))[0]
            n_cols = struct.unpack('>I', gz.read(4))[0]
            res = np.frombuffer(gz.read(n_items[0] * n_rows * n_cols), dtype=np.uint8)
            res = res.reshape(n_items[0], n_rows * n_cols)
        else:
            res = np.frombuffer(gz.read(n_items[0]), dtype=np.uint8)
            res = res.reshape(n_items[0], 1)
    return res

# one-hot encode a 1-D array
def one_hot_encode(array, num_of_classes):
    return np.eye(num_of_classes)[array.reshape(-1)]

# Download test data
os.makedirs('./data/mnist', exist_ok=True)
urllib.request.urlretrieve('http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
filename='./data/mnist/test-images.gz')
urllib.request.urlretrieve('http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz',
filename='./data/mnist/test-labels.gz')

# Load test data from model training
X_test = load_data('./data/mnist/test-images.gz', False) / 255.0
y_test = load_data('./data/mnist/test-labels.gz', True).reshape(-1)

# send a random row from the test set to score
random_index = np.random.randint(0, len(X_test)-1)
input_data = "{\"data\": [" + str(list(X_test[random_index])) + "]}"

api_key = aks_service.get_keys()[0]
headers = {'Content-Type': 'application/json',
           'Authorization': ('Bearer ' + api_key)}
resp = requests.post(aks_service.scoring_uri, input_data, headers=headers)

print("POST to url", aks_service.scoring_uri)
print("label:", y_test[random_index])
print("prediction:", resp.text)
```

For more information on creating a client application, see Create client to consume deployed web service.

Clean up the resources

If you created the AKS cluster specifically for this example, delete your resources after you're done.

> **IMPORTANT**
>
> Azure bills you based on how long the AKS cluster is deployed. Make sure to clean it up after you are done with it.

```
aks_service.delete()
aks_target.delete()
```

Next steps

- Deploy model on FPGA
- Deploy model with ONNX
- Train Tensorflow DNN Models

Deploy a machine learning model to Azure App Service (preview)

3/25/2020 • 7 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to deploy a model from Azure Machine Learning as a web app in Azure App Service.

> **IMPORTANT**
> While both Azure Machine Learning and Azure App Service are generally available, the ability to deploy a model from the Machine Learning service to App Service is in preview.

With Azure Machine Learning, you can create Docker images from trained machine learning models. This image contains a web service that receives data, submits it to the model, and then returns the response. Azure App Service can be used to deploy the image, and provides the following features:

- Advanced authentication for enhanced security. Authentication methods include both Azure Active Directory and multi-factor auth.
- Autoscale without having to redeploy.
- TLS support for secure communications between clients and the service.

For more information on features provided by Azure App Service, see the App Service overview.

> **IMPORTANT**
> If you need the ability to log the scoring data used with your deployed model, or the results of scoring, you should instead deploy to Azure Kubernetes Service. For more information, see Collect data on your production models.

Prerequisites

- An Azure Machine Learning workspace. For more information, see the Create a workspace article.
- The Azure CLI.
- A trained machine learning model registered in your workspace. If you do not have a model, use the Image classification tutorial: train model to train and register one.

  > **IMPORTANT**
  > The code snippets in this article assume that you have set the following variables:
  >
  > - `ws` - Your Azure Machine Learning workspace.
  > - `model` - The registered model that will be deployed.
  > - `inference_config` - The inference configuration for the model.
  >
  > For more information on setting these variables, see Deploy models with Azure Machine Learning.

Prepare for deployment

Before deploying, you must define what is needed to run the model as a web service. The following list describes the basic items needed for a deployment:

- An **entry script**. This script accepts requests, scores the request using the model, and returns the results.

  > **IMPORTANT**
  >
  > The entry script is specific to your model; it must understand the format of the incoming request data, the format of the data expected by your model, and the format of the data returned to clients.
  >
  > If the request data is in a format that is not usable by your model, the script can transform it into an acceptable format. It may also transform the response before returning to it to the client.

  > **IMPORTANT**
  >
  > The Azure Machine Learning SDK does not provide a way for the web service access your datastore or data sets. If you need the deployed model to access data stored outside the deployment, such as in an Azure Storage account, you must develop a custom code solution using the relevant SDK. For example, the Azure Storage SDK for Python.
  >
  > Another alternative that may work for your scenario is batch predictions, which does provide access to datastores when scoring.

  For more information on entry scripts, see Deploy models with Azure Machine Learning.

- **Dependencies**, such as helper scripts or Python/Conda packages required to run the entry script or model

These entities are encapsulated into an **inference configuration**. The inference configuration references the entry script and other dependencies.

> **IMPORTANT**
>
> When creating an inference configuration for use with Azure App Service, you must use an Environment object. Please note that if you are defining a custom environment, you must add azureml-defaults with version >= 1.0.45 as a pip dependency. This package contains the functionality needed to host the model as a web service. The following example demonstrates creating an environment object and using it with an inference configuration:

```
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.model import InferenceConfig

# Create an environment and add conda dependencies to it
myenv = Environment(name="myenv")
# Enable Docker based environment
myenv.docker.enabled = True
# Build conda dependencies
myenv.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-learn'],
                                                           pip_packages=['azureml-defaults'])
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
```

For more information on environments, see Create and manage environments for training and deployment.

For more information on inference configuration, see Deploy models with Azure Machine Learning.

> **IMPORTANT**
>
> When deploying to Azure App Service, you do not need to create a **deployment configuration**.

Create the image

To create the Docker image that is deployed to Azure App Service, use Model.package. The following code snippet demonstrates how to build a new image from the model and inference configuration:

> **NOTE**
> The code snippet assumes that `model` contains a registered model, and that `inference_config` contains the configuration for the inference environment. For more information, see Deploy models with Azure Machine Learning.

```
from azureml.core import Model

package = Model.package(ws, [model], inference_config)
package.wait_for_creation(show_output=True)
# Display the package location/ACR path
print(package.location)
```

When `show_output=True`, the output of the Docker build process is shown. Once the process finishes, the image has been created in the Azure Container Registry for your workspace. Once the image has been built, the location in your Azure Container Registry is displayed. The location returned is in the format `<acrinstance>.azurecr.io/package:<imagename>`. For example, `myml08024f78fd10.azurecr.io/package:20190827151241`.

> **IMPORTANT**
> Save the location information, as it is used when deploying the image.

Deploy image as a web app

1. Use the following command to get the login credentials for the Azure Container Registry that contains the image. Replace `<acrinstance>` with th e value returned previously from `package.location`:

   ```
   az acr credential show --name <myacr>
   ```

   The output of this command is similar to the following JSON document:

   ```
   {
   "passwords": [
       {
       "name": "password",
       "value": "Iv0lRZQ9762LUJrFiffo3P4sWgk4q+nW"
       },
       {
       "name": "password2",
       "value": "=pKCxHatX96jeoYBWZLsPR6opszr==mg"
       }
   ],
   "username": "myml08024f78fd10"
   }
   ```

   Save the value for **username** and one of the **passwords**.

2. If you do not already have a resource group or app service plan to deploy the service, the following commands demonstrate how to create both:

```
az group create --name myresourcegroup --location "West Europe"
az appservice plan create --name myplanname --resource-group myresourcegroup --sku B1 --is-linux
```

In this example, a **Basic** pricing tier (`--sku B1`) is used.

> **IMPORTANT**
> Images created by Azure Machine Learning use Linux, so you must use the `--is-linux` parameter.

3. To create the web app, use the following command. Replace `<app-name>` with the name you want to use. Replace `<acrinstance>` and `<imagename>` with the values from returned `package.location` earlier:

```
az webapp create --resource-group myresourcegroup --plan myplanname --name <app-name> --deployment-container-image-name <acrinstance>.azurecr.io/package:<imagename>
```

This command returns information similar to the following JSON document:

```
{
"adminSiteName": null,
"appServicePlanName": "myplanname",
"geoRegion": "West Europe",
"hostingEnvironmentProfile": null,
"id": "/subscriptions/0000-0000/resourceGroups/myResourceGroup/providers/Microsoft.Web/serverfarms/myplanname",
"kind": "linux",
"location": "West Europe",
"maximumNumberOfWorkers": 1,
"name": "myplanname",
< JSON data removed for brevity. >
"targetWorkerSizeId": 0,
"type": "Microsoft.Web/serverfarms",
"workerTierName": null
}
```

> **IMPORTANT**
> At this point, the web app has been created. However, since you haven't provided the credentials to the Azure Container Registry that contains the image, the web app is not active. In the next step, you provide the authentication information for the container registry.

4. To provide the web app with the credentials needed to access the container registry, use the following command. Replace `<app-name>` with the name you want to use. Replace `<acrinstance>` and `<imagename>` with the values from returned `package.location` earlier. Replace `<username>` and `<password>` with the ACR login information retrieved earlier:

```
az webapp config container set --name <app-name> --resource-group myresourcegroup --docker-custom-image-name <acrinstance>.azurecr.io/package:<imagename> --docker-registry-server-url https://<acrinstance>.azurecr.io --docker-registry-server-user <username> --docker-registry-server-password <password>
```

This command returns information similar to the following JSON document:

```
[
{
    "name": "WEBSITES_ENABLE_APP_SERVICE_STORAGE",
    "slotSetting": false,
    "value": "false"
},
{
    "name": "DOCKER_REGISTRY_SERVER_URL",
    "slotSetting": false,
    "value": "https://myml08024f78fd10.azurecr.io"
},
{
    "name": "DOCKER_REGISTRY_SERVER_USERNAME",
    "slotSetting": false,
    "value": "myml08024f78fd10"
},
{
    "name": "DOCKER_REGISTRY_SERVER_PASSWORD",
    "slotSetting": false,
    "value": null
},
{
    "name": "DOCKER_CUSTOM_IMAGE_NAME",
    "value": "DOCKER|myml08024f78fd10.azurecr.io/package:20190827195524"
}
]
```

At this point, the web app begins loading the image.

IMPORTANT

It may take several minutes before the image has loaded. To monitor progress, use the following command:

```
az webapp log tail --name <app-name> --resource-group myresourcegroup
```

Once the image has been loaded and the site is active, the log displays a message that states `Container <container name> for site <app-name> initialized successfully and is ready to serve requests`.

Once the image is deployed, you can find the hostname by using the following command:

```
az webapp show --name <app-name> --resource-group myresourcegroup
```

This command returns information similar to the following hostname - `<app-name>.azurewebsites.net`. Use this value as part of the **base url** for the service.

Use the Web App

The web service that passes requests to the model is located at `{baseurl}/score`. For example, `https://<app-name>.azurewebsites.net/score`. The following Python code demonstrates how to submit data to the URL and display the response:

```
import requests
import json

scoring_uri = "https://mywebapp.azurewebsites.net/score"

headers = {'Content-Type':'application/json'}

test_sample = json.dumps({'data': [
    [1,2,3,4,5,6,7,8,9,10],
    [10,9,8,7,6,5,4,3,2,1]
]})

response = requests.post(scoring_uri, data=test_sample, headers=headers)
print(response.status_code)
print(response.elapsed)
print(response.json())
```

Next steps

- Learn to configure your Web App in the App Service on Linux documentation.
- Learn more about scaling in Get started with Autoscale in Azure.
- Use a TLS/SSL certificate in your Azure App Service.
- Configure your App Service app to use Azure Active Directory sign-in.
- Consume a ML Model deployed as a web service

Deploy a machine learning model to Azure Functions (preview)

3/8/2020 • 8 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to deploy a model from Azure Machine Learning as a function app in Azure Functions.

> **IMPORTANT**
> While both Azure Machine Learning and Azure Functions are generally available, the ability to package a model from the Machine Learning service for Functions is in preview.

With Azure Machine Learning, you can create Docker images from trained machine learning models. Azure Machine Learning now has the preview functionality to build these machine learning models into function apps, which can be deployed into Azure Functions.

Prerequisites

- An Azure Machine Learning workspace. For more information, see the Create a workspace article.
- The Azure CLI.
- A trained machine learning model registered in your workspace. If you do not have a model, use the Image classification tutorial: train model to train and register one.

> **IMPORTANT**
> The code snippets in this article assume that you have set the following variables:
>
> - `ws` - Your Azure Machine Learning workspace.
> - `model` - The registered model that will be deployed.
> - `inference_config` - The inference configuration for the model.
>
> For more information on setting these variables, see Deploy models with Azure Machine Learning.

Prepare for deployment

Before deploying, you must define what is needed to run the model as a web service. The following list describes the basic items needed for a deployment:

- An **entry script**. This script accepts requests, scores the request using the model, and returns the results.

> **IMPORTANT**
>
> The entry script is specific to your model; it must understand the format of the incoming request data, the format of the data expected by your model, and the format of the data returned to clients.
>
> If the request data is in a format that is not usable by your model, the script can transform it into an acceptable format. It may also transform the response before returning to it to the client.
>
> By default when packaging for functions, the input is treated as text. If you are interested in consuming the raw bytes of the input (for instance for Blob triggers), you should use AMLRequest to accept raw data.

- **Dependencies**, such as helper scripts or Python/Conda packages required to run the entry script or model

These entities are encapsulated into an **inference configuration**. The inference configuration references the entry script and other dependencies.

> **IMPORTANT**
>
> When creating an inference configuration for use with Azure Functions, you must use an Environment object. Please note that if you are defining a custom environment, you must add azureml-defaults with version >= 1.0.45 as a pip dependency. This package contains the functionality needed to host the model as a web service. The following example demonstrates creating an environment object and using it with an inference configuration:
>
> ```
> from azureml.core.environment import Environment
> from azureml.core.conda_dependencies import CondaDependencies
>
> # Create an environment and add conda dependencies to it
> myenv = Environment(name="myenv")
> # Enable Docker based environment
> myenv.docker.enabled = True
> # Build conda dependencies
> myenv.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-learn'],
>                                                            pip_packages=['azureml-defaults'])
> inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
> ```

For more information on environments, see Create and manage environments for training and deployment.

For more information on inference configuration, see Deploy models with Azure Machine Learning.

> **IMPORTANT**
>
> When deploying to Functions, you do not need to create a **deployment configuration**.

Install the SDK preview package for functions support

To build packages for Azure Functions, you must install the SDK preview package.

```
pip install azureml-contrib-functions
```

Create the image

To create the Docker image that is deployed to Azure Functions, use azureml.contrib.functions.package or the specific package function for the trigger you are interested in using. The following code snippet demonstrates how to create a new package with a blob trigger from the model and inference configuration:

> **NOTE**
>
> The code snippet assumes that `model` contains a registered model, and that `inference_config` contains the configuration for the inference environment. For more information, see Deploy models with Azure Machine Learning.

```
from azureml.contrib.functions import package
from azureml.contrib.functions import BLOB_TRIGGER
blob = package(ws, [model], inference_config, functions_enabled=True, trigger=BLOB_TRIGGER,
input_path="input/{blobname}.json", output_path="output/{blobname}_out.json")
blob.wait_for_creation(show_output=True)
# Display the package location/ACR path
print(blob.location)
```

When `show_output=True`, the output of the Docker build process is shown. Once the process finishes, the image has been created in the Azure Container Registry for your workspace. Once the image has been built, the location in your Azure Container Registry is displayed. The location returned is in the format `<acrinstance>.azurecr.io/package@sha256:<hash>`.

> **NOTE**
>
> Packaging for functions currently supports HTTP Triggers, Blob triggers and Service bus triggers. For more information on triggers, see Azure Functions bindings.

> **IMPORTANT**
>
> Save the location information, as it is used when deploying the image.

Deploy image as a web app

1. Use the following command to get the login credentials for the Azure Container Registry that contains the image. Replace `<myacr>` with the value returned previously from `package.location`:

   ```
   az acr credential show --name <myacr>
   ```

   The output of this command is similar to the following JSON document:

   ```
   {
   "passwords": [
       {
       "name": "password",
       "value": "Iv0lRZQ9762LUJrFiffo3P4sWgk4q+nW"
       },
       {
       "name": "password2",
       "value": "=pKCxHatX96jeoYBWZLsPR6opszr==mg"
       }
   ],
   "username": "myml08024f78fd10"
   }
   ```

   Save the value for **username** and one of the **passwords**.

2. If you do not already have a resource group or app service plan to deploy the service, the following commands demonstrate how to create both:

```
az group create --name myresourcegroup --location "West Europe"
az appservice plan create --name myplanname --resource-group myresourcegroup --sku B1 --is-linux
```

In this example, a *Linux basic* pricing tier (`--sku B1`) is used.

> **IMPORTANT**
> Images created by Azure Machine Learning use Linux, so you must use the `--is-linux` parameter.

3. Create the storage account to use for the web job storage and get its connection string. Replace `<webjobStorage>` with the name you want to use.

```
az storage account create --name <webjobStorage> --location westeurope --resource-group myresourcegroup
--sku Standard_LRS
```

```
az storage account show-connection-string --resource-group myresourcegroup --name <webJobStorage> --
query connectionString --output tsv
```

4. To create the function app, use the following command. Replace `<app-name>` with the name you want to use. Replace `<acrinstance>` and `<imagename>` with the values from returned `package.location` earlier. Replace `<webjobStorage>` with the name of the storage account from the previous step:

```
az functionapp create --resource-group myresourcegroup --plan myplanname --name <app-name> --
deployment-container-image-name <acrinstance>.azurecr.io/package:<imagename> --storage-account
<webjobStorage>
```

> **IMPORTANT**
> At this point, the function app has been created. However, since you haven't provided the connection string for the blob trigger or credentials to the Azure Container Registry that contains the image, the function app is not active. In the next steps, you provide the connection string and the authentication information for the container registry.

5. Create the storage account to use for the blob trigger storage and get its connection string. Replace `<triggerStorage>` with the name you want to use.

```
az storage account create --name <triggerStorage> --location westeurope --resource-group
myresourcegroup --sku Standard_LRS
```

```
az storage account show-connection-string --resource-group myresourcegroup --name <triggerStorage> --
query connectionString --output tsv
```

Record this connection string to provide to the function app. We will use it later when we ask for `<triggerConnectionString>`

6. Create the containers for the input and output in the storage account. Replace `<triggerConnectionString>` with the connection string returned earlier:

```
az storage container create -n input --connection-string <triggerConnectionString>
```

```
az storage container create -n output --connection-string <triggerConnectionString>
```

7. To associate the trigger connection string with the function app, use the following command. Replace `<app-name>` with the name of the function app. Replace `<triggerConnectionString>` with the connection string returned earlier:

   ```
   az functionapp config appsettings set --name <app-name> --resource-group myresourcegroup --settings
   "TriggerConnectionString=<triggerConnectionString>"
   ```

8. You will need to retrieve the tag associated with the created container using the following command. Replace `<username>` with the username returned earlier from the container registry:

   ```
   az acr repository show-tags --repository package --name <username> --output tsv
   ```

   Save the value returned, it will be used as the `imagetag` in the next step.

9. To provide the function app with the credentials needed to access the container registry, use the following command. Replace `<app-name>` with the name of the function app. Replace `<acrinstance>` and `<imagetag>` with the values from the AZ CLI call in the previous step. Replace `<username>` and `<password>` with the ACR login information retrieved earlier:

   ```
   az functionapp config container set --name <app-name> --resource-group myresourcegroup --docker-custom-
   image-name <acrinstance>.azurecr.io/package:<imagetag> --docker-registry-server-url
   https://<acrinstance>.azurecr.io --docker-registry-server-user <username> --docker-registry-server-
   password <password>
   ```

   This command returns information similar to the following JSON document:

   ```
   [
   {
       "name": "WEBSITES_ENABLE_APP_SERVICE_STORAGE",
       "slotSetting": false,
       "value": "false"
   },
   {
       "name": "DOCKER_REGISTRY_SERVER_URL",
       "slotSetting": false,
       "value": "https://myml08024f78fd10.azurecr.io"
   },
   {
       "name": "DOCKER_REGISTRY_SERVER_USERNAME",
       "slotSetting": false,
       "value": "myml08024f78fd10"
   },
   {
       "name": "DOCKER_REGISTRY_SERVER_PASSWORD",
       "slotSetting": false,
       "value": null
   },
   {
       "name": "DOCKER_CUSTOM_IMAGE_NAME",
       "value": "DOCKER|myml08024f78fd10.azurecr.io/package:20190827195524"
   }
   ]
   ```

At this point, the function app begins loading the image.

> **IMPORTANT**
> It may take several minutes before the image has loaded. You can monitor progress using the Azure Portal.

Test the deployment

Once the image has loaded and the app is available, use the following steps to trigger the app:

1. Create a text file that contains the data that the score.py file expects. The following example would work with a score.py that expects an array of 10 numbers:

   ```
   {"data": [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]]}
   ```

   > **IMPORTANT**
   > The format of the data depends on what your score.py and model expects.

2. Use the following command to upload this file to the input container in the trigger storage blob created earlier. Replace `<file>` with the name of the file containing the data. Replace `<triggerConnectionString>` with the connection string returned earlier. In this example, `input` is the name of the input container created earlier. If you used a different name, replace this value:

   ```
   az storage blob upload --container-name input --file <file> --name <file> --connection-string
   <triggerConnectionString>
   ```

   The output of this command is similar to the following JSON:

   ```
   {
   "etag": "\"0x8D7C21528E08844\"",
   "lastModified": "2020-03-06T21:27:23+00:00"
   }
   ```

3. To view the output produced by the function, use the following command to list the output files generated. Replace `<triggerConnectionString>` with the connection string returned earlier. In this example, `output` is the name of the output container created earlier. If you used a different name, replace this value::

   ```
   az storage blob list --container-name output --connection-string <triggerConnectionString> --query
   '[].name' --output tsv
   ```

   The output of this command is similar to `sample_input_out.json`.

4. To download the file and inspect the contents, use the following command. Replace `<file>` with the file name returned by the previous command. Replace `<triggerConnectionString>` with the connection string returned earlier:

   ```
   az storage blob download --container-name output --file <file> --name <file> --connection-string
   <triggerConnectionString>
   ```

   Once the command completes, open the file. It contains the data returned by the model.

For more information on using blob triggers, see the Create a function triggered by Azure Blob storage article.

Next steps

- Learn to configure your Functions App in the Functions documentation.
- Learn more about Blob storage triggers Azure Blob storage bindings.
- Deploy your model to Azure App Service.
- Consume a ML Model deployed as a web service
- API Reference

What are field-programmable gate arrays (FPGA) and how to deploy

4/10/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article provides an introduction to field-programmable gate arrays (FPGA), and shows you how to deploy your models using Azure Machine Learning to an Azure FPGA.

FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects. The interconnects allow these blocks to be configured in various ways after manufacturing. Compared to other chips, FPGAs provide a combination of programmability and performance.

FPGAs vs. CPU, GPU, and ASIC

The following diagram and table show how FPGAs compare to other processors.

PROCESSOR		DESCRIPTION
Application-specific integrated circuits	ASICs	Custom circuits, such as Google's TensorFlow Processor Units (TPU), provide the highest efficiency. They can't be reconfigured as your needs change.
Field-programmable gate arrays	FPGAs	FPGAs, such as those available on Azure, provide performance close to ASICs. They are also flexible and reconfigurable over time, to implement new logic.
Graphics processing units	GPUs	A popular choice for AI computations. GPUs offer parallel processing capabilities, making it faster at image rendering than CPUs.

PROCESSOR		DESCRIPTION
Central processing units	CPUs	General-purpose processors, the performance of which isn't ideal for graphics and video processing.

FPGAs on Azure are based on Intel's FPGA devices, which data scientists and developers use to accelerate real-time AI calculations. This FPGA-enabled architecture offers performance, flexibility, and scale, and is available on Azure.

FPGAs make it possible to achieve low latency for real-time inference (or model scoring) requests. Asynchronous requests (batching) aren't needed. Batching can cause latency, because more data needs to be processed. Implementations of neural processing units don't require batching; therefore the latency can be many times lower, compared to CPU and GPU processors.

Reconfigurable power

You can reconfigure FPGAs for different types of machine learning models. This flexibility makes it easier to accelerate the applications based on the most optimal numerical precision and memory model being used. Because FPGAs are reconfigurable, you can stay current with the requirements of rapidly changing AI algorithms.

What's supported on Azure

Microsoft Azure is the world's largest cloud investment in FPGAs. Using this FPGA-enabled hardware architecture, trained neural networks run quickly and with lower latency. Azure can parallelize pre-trained deep neural networks (DNN) across FPGAs to scale out your service. The DNNs can be pre-trained, as a deep featurizer for transfer learning, or fine-tuned with updated weights.

FPGAs on Azure supports:

- Image classification and recognition scenarios
- TensorFlow deployment (requires Tensorflow 1.x)
- Intel FPGA hardware

These DNN models are currently available:

- ResNet 50
- ResNet 152
- DenseNet-121
- VGG-16
- SSD-VGG

FPGAs are available in these Azure regions:

- East US
- Southeast Asia
- West Europe
- West US 2

> **IMPORTANT**
> To optimize latency and throughput, your client sending data to the FPGA model should be in one of the regions above (the one you deployed the model to).

The **PBS Family of Azure VMs** contains Intel Arria 10 FPGAs. It will show as "Standard PBS Family vCPUs" when you check your Azure quota allocation. The PB6 VM has six vCPUs and one FPGA, and it will automatically be

provisioned by Azure ML as part of deploying a model to an FPGA. It is only used with Azure ML, and it cannot run arbitrary bitstreams. For example, you will not be able to flash the FPGA with bitstreams to do encryption, encoding, etc.

Scenarios and applications

Azure FPGAs are integrated with Azure Machine Learning. Microsoft uses FPGAs for DNN evaluation, Bing search ranking, and software defined networking (SDN) acceleration to reduce latency, while freeing CPUs for other tasks.

The following scenarios use FPGAs:

- Automated optical inspection system
- Land cover mapping

Example: Deploy models on FPGAs

You can deploy a model as a web service on FPGAs with Azure Machine Learning Hardware Accelerated Models. Using FPGAs provides ultra-low latency inference, even with a single batch size. Inference, or model scoring, is the phase where the deployed model is used for prediction, most commonly on production data.

Prerequisites

- An Azure subscription. If you do not have one, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- FPGA quota. Use the Azure CLI to check whether you have quota:

  ```
  az vm list-usage --location "eastus" -o table --query "[?localName=='Standard PBS Family vCPUs']"
  ```

  > **TIP**
  > The other possible locations are `southeastasia`, `westeurope`, and `westus2`.

  The command returns text similar to the following:

  ```
  CurrentValue    Limit    LocalName
  --------------  -------  ------------------------
  0               6        Standard PBS Family vCPUs
  ```

  Make sure you have at least 6 vCPUs under **CurrentValue**.

  If you do not have quota, then submit a request at https://aka.ms/accelerateAI.

- An Azure Machine Learning workspace and the Azure Machine Learning SDK for Python installed. For more information, see Create a workspace.
- The Python SDK for hardware-accelerated models:

  ```
  pip install --upgrade azureml-accel-models[cpu]
  ```

1. Create and containerize models

This document will describe how to create a TensorFlow graph to preprocess the input image, make it a featurizer using ResNet 50 on an FPGA, and then run the features through a classifier trained on the ImageNet data set.

Follow the instructions to:

- Define the TensorFlow model
- Convert the model
- Deploy the model
- Consume the deployed model
- Delete deployed services

Use the Azure Machine Learning SDK for Python to create a service definition. A service definition is a file describing a pipeline of graphs (input, featurizer, and classifier) based on TensorFlow. The deployment command automatically compresses the definition and graphs into a ZIP file, and uploads the ZIP to Azure Blob storage. The DNN is already deployed to run on the FPGA.

Load Azure Machine Learning workspace

Load your Azure Machine Learning workspace.

```
import os
import tensorflow as tf

from azureml.core import Workspace

ws = Workspace.from_config()
print(ws.name, ws.resource_group, ws.location, ws.subscription_id, sep='\n')
```

Preprocess image

The input to the web service is a JPEG image. The first step is to decode the JPEG image and preprocess it. The JPEG images are treated as strings and the result are tensors that will be the input to the ResNet 50 model.

```
# Input images as a two-dimensional tensor containing an arbitrary number of images represented a strings
import azureml.accel.models.utils as utils
tf.reset_default_graph()

in_images = tf.placeholder(tf.string)
image_tensors = utils.preprocess_array(in_images)
print(image_tensors.shape)
```

Load featurizer

Initialize the model and download a TensorFlow checkpoint of the quantized version of ResNet50 to be used as a featurizer. You may replace "QuantizedResnet50" in the code snippet below with by importing other deep neural networks:

- QuantizedResnet152
- QuantizedVgg16
- Densenet121

```
from azureml.accel.models import QuantizedResnet50
save_path = os.path.expanduser('~/models')
model_graph = QuantizedResnet50(save_path, is_frozen=True)
feature_tensor = model_graph.import_graph_def(image_tensors)
print(model_graph.version)
print(feature_tensor.name)
print(feature_tensor.shape)
```

Add classifier

This classifier has been trained on the ImageNet data set. Examples for transfer learning and training your customized weights are available in the set of sample notebooks.

```
classifier_output = model_graph.get_default_classifier(feature_tensor)
print(classifier_output)
```

Save the model

Now that the preprocessor, ResNet 50 featurizer, and the classifier have been loaded, save the graph and associated variables as a model.

```
model_name = "resnet50"
model_save_path = os.path.join(save_path, model_name)
print("Saving model in {}".format(model_save_path))

with tf.Session() as sess:
    model_graph.restore_weights(sess)
    tf.saved_model.simple_save(sess, model_save_path,
                               inputs={'images': in_images},
                               outputs={'output_alias': classifier_output})
```

Save input and output tensors

The input and output tensors that were created during the preprocessing and classifier steps will be needed for model conversion and inference.

```
input_tensors = in_images.name
output_tensors = classifier_output.name

print(input_tensors)
print(output_tensors)
```

IMPORTANT

Save the input and output tensors because you will need them for model conversion and inference requests.

The available models and the corresponding default classifier output tensors are below, which is what you would use for inference if you used the default classifier.

- Resnet50, QuantizedResnet50

  ```
  output_tensors = "classifier_1/resnet_v1_50/predictions/Softmax:0"
  ```

- Resnet152, QuantizedResnet152

  ```
  output_tensors = "classifier/resnet_v1_152/predictions/Softmax:0"
  ```

- Densenet121, QuantizedDensenet121

  ```
  output_tensors = "classifier/densenet121/predictions/Softmax:0"
  ```

- Vgg16, QuantizedVgg16

  ```
  output_tensors = "classifier/vgg_16/fc8/squeezed:0"
  ```

- SsdVgg, QuantizedSsdVgg

```
output_tensors = ['ssd_300_vgg/block4_box/Reshape_1:0', 'ssd_300_vgg/block7_box/Reshape_1:0',
'ssd_300_vgg/block8_box/Reshape_1:0', 'ssd_300_vgg/block9_box/Reshape_1:0',
'ssd_300_vgg/block10_box/Reshape_1:0', 'ssd_300_vgg/block11_box/Reshape_1:0',
'ssd_300_vgg/block4_box/Reshape:0', 'ssd_300_vgg/block7_box/Reshape:0',
'ssd_300_vgg/block8_box/Reshape:0', 'ssd_300_vgg/block9_box/Reshape:0',
'ssd_300_vgg/block10_box/Reshape:0', 'ssd_300_vgg/block11_box/Reshape:0']
```

Register model

Register the model by using the SDK with the ZIP file in Azure Blob storage. Adding tags and other metadata about the model helps you keep track of your trained models.

```
from azureml.core.model import Model

registered_model = Model.register(workspace=ws,
                                  model_path=model_save_path,
                                  model_name=model_name)

print("Successfully registered: ", registered_model.name,
      registered_model.description, registered_model.version, sep='\t')
```

If you've already registered a model and want to load it, you may retrieve it.

```
from azureml.core.model import Model
model_name = "resnet50"
# By default, the latest version is retrieved. You can specify the version, i.e. version=1
registered_model = Model(ws, name="resnet50")
print(registered_model.name, registered_model.description,
      registered_model.version, sep='\t')
```

Convert model

Convert the TensorFlow graph to the Open Neural Network Exchange format (ONNX). You will need to provide the names of the input and output tensors, and these names will be used by your client when you consume the web service.

```
from azureml.accel import AccelOnnxConverter

convert_request = AccelOnnxConverter.convert_tf_model(
    ws, registered_model, input_tensors, output_tensors)

# If it fails, you can run wait_for_completion again with show_output=True.
convert_request.wait_for_completion(show_output=False)

# If the above call succeeded, get the converted model
converted_model = convert_request.result
print("\nSuccessfully converted: ", converted_model.name, converted_model.url, converted_model.version,
      converted_model.id, converted_model.created_time, '\n')
```

Create Docker image

The converted model and all dependencies are added to a Docker image. This Docker image can then be deployed and instantiated. Supported deployment targets include AKS in the cloud or an edge device such as Azure Data Box Edge. You can also add tags and descriptions for your registered Docker image.

```
from azureml.core.image import Image
from azureml.accel import AccelContainerImage

image_config = AccelContainerImage.image_configuration()
# Image name must be lowercase
image_name = "{}-image".format(model_name)

image = Image.create(name=image_name,
                     models=[converted_model],
                     image_config=image_config,
                     workspace=ws)
image.wait_for_creation(show_output=False)
```

List the images by tag and get the detailed logs for any debugging.

```
for i in Image.list(workspace=ws):
    print('{}(v.{} [{}]) stored at {} with build log {}'.format(
        i.name, i.version, i.creation_state, i.image_location, i.image_build_log_uri))
```

2. Deploy to cloud or edge

Deploy to the cloud

To deploy your model as a high-scale production web service, use Azure Kubernetes Service (AKS). You can create a new one using the Azure Machine Learning SDK, CLI, or Azure Machine Learning studio.

```
from azureml.core.compute import AksCompute, ComputeTarget

# Specify the Standard_PB6s Azure VM and location. Values for location may be "eastus", "southeastasia",
"westeurope", or "westus2". If no value is specified, the default is "eastus".
prov_config = AksCompute.provisioning_configuration(vm_size = "Standard_PB6s",
                                                    agent_count = 1,
                                                    location = "eastus")

aks_name = 'my-aks-cluster'
# Create the cluster
aks_target = ComputeTarget.create(workspace=ws,
                                  name=aks_name,
                                  provisioning_configuration=prov_config)
```

The AKS deployment may take around 15 minutes. Check to see if the deployment succeeded.

```
aks_target.wait_for_completion(show_output=True)
print(aks_target.provisioning_state)
print(aks_target.provisioning_errors)
```

Deploy the container to the AKS cluster.

```
from azureml.core.webservice import Webservice, AksWebservice

# For this deployment, set the web service configuration without enabling auto-scaling or authentication for
testing
aks_config = AksWebservice.deploy_configuration(autoscale_enabled=False,
                                                num_replicas=1,
                                                auth_enabled=False)

aks_service_name = 'my-aks-service'

aks_service = Webservice.deploy_from_image(workspace=ws,
                                           name=aks_service_name,
                                           image=image,
                                           deployment_config=aks_config,
                                           deployment_target=aks_target)
aks_service.wait_for_deployment(show_output=True)
```

Test the cloud service

The Docker image supports gRPC and the TensorFlow Serving "predict" API. Use the sample client to call into the Docker image to get predictions from the model. Sample client code is available:

- Python
- C#

If you want to use TensorFlow Serving, you can download a sample client.

```
# Using the grpc client in Azure ML Accelerated Models SDK package
from azureml.accel import PredictionClient

address = aks_service.scoring_uri
ssl_enabled = address.startswith("https")
address = address[address.find('/')+2:].strip('/')
port = 443 if ssl_enabled else 80

# Initialize Azure ML Accelerated Models client
client = PredictionClient(address=address,
                          port=port,
                          use_ssl=ssl_enabled,
                          service_name=aks_service.name)
```

Since this classifier was trained on the ImageNet data set, map the classes to human-readable labels.

```
import requests
classes_entries = requests.get(

"https://raw.githubusercontent.com/Lasagne/Recipes/master/examples/resnet50/imagenet_classes.txt").text.splitl
ines()

# Score image with input and output tensor names
results = client.score_file(path="./snowleopardgaze.jpg",
                            input_name=input_tensors,
                            outputs=output_tensors)

# map results [class_id] => [confidence]
results = enumerate(results)
# sort results by confidence
sorted_results = sorted(results, key=lambda x: x[1], reverse=True)
# print top 5 results
for top in sorted_results[:5]:
    print(classes_entries[top[0]], 'confidence:', top[1])
```

Clean-up the service

Delete your web service, image, and model (must be done in this order since there are dependencies).

```
aks_service.delete()
aks_target.delete()
image.delete()
registered_model.delete()
converted_model.delete()
```

Deploy to a local edge server

All Azure Data Box Edge devices contain an FPGA for running the model. Only one model can be running on the FPGA at one time. To run a different model, just deploy a new container. Instructions and sample code can be found in this Azure Sample.

Secure FPGA web services

To secure your FPGA web services, see the Secure web services document.

Next steps

Check out these notebooks, videos, and blogs:

- Several sample notebooks
- Hyperscale hardware: ML at scale on top of Azure + FPGA: Build 2018 (video)
- Inside the Microsoft FPGA-based configurable cloud (video)
- Project Brainwave for real-time AI: project home page

Deploy a model using a custom Docker base image

4/17/2020 • 11 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to use a custom Docker base image when deploying trained models with Azure Machine Learning.

When you deploy a trained model to a web service or IoT Edge device, a package is created which contains a web server to handle incoming requests.

Azure Machine Learning provides a default Docker base image so you don't have to worry about creating one. You can also use Azure Machine Learning **environments** to select a specific base image, or use a custom one that you provide.

A base image is used as the starting point when an image is created for a deployment. It provides the underlying operating system and components. The deployment process then adds additional components, such as your model, conda environment, and other assets, to the image before deploying it.

Typically, you create a custom base image when you want to use Docker to manage your dependencies, maintain tighter control over component versions or save time during deployment. For example, you might want to standardize on a specific version of Python, Conda, or other component. You might also want to install software required by your model, where the installation process takes a long time. Installing the software when creating the base image means that you don't have to install it for each deployment.

> **IMPORTANT**
> When you deploy a model, you cannot override core components such as the web server or IoT Edge components. These components provide a known working environment that is tested and supported by Microsoft.

> **WARNING**
> Microsoft may not be able to help troubleshoot problems caused by a custom image. If you encounter problems, you may be asked to use the default image or one of the images Microsoft provides to see if the problem is specific to your image.

This document is broken into two sections:

- Create a custom base image: Provides information to admins and DevOps on creating a custom image and configuring authentication to an Azure Container Registry using the Azure CLI and Machine Learning CLI.
- Deploy a model using a custom base image: Provides information to Data Scientists and DevOps / ML Engineers on using custom images when deploying a trained model from the Python SDK or ML CLI.

Prerequisites

- An Azure Machine Learning workgroup. For more information, see the Create a workspace article.
- The Azure Machine Learning SDK.
- The Azure CLI.
- The CLI extension for Azure Machine Learning.
- An Azure Container Registry or other Docker registry that is accessible on the internet.
- The steps in this document assume that you are familiar with creating and using an **inference configuration** object as part of model deployment. For more information, see the "prepare to deploy" section of Where to

deploy and how.

Create a custom base image

The information in this section assumes that you are using an Azure Container Registry to store Docker images. Use the following checklist when planning to create custom images for Azure Machine Learning:

- Will you use the Azure Container Registry created for the Azure Machine Learning workspace, or a standalone Azure Container Registry?

  When using images stored in the **container registry for the workspace**, you do not need to authenticate to the registry. Authentication is handled by the workspace.

  > **WARNING**
  > The Azure Container Registry for your workspace is **created the first time you train or deploy a model** using the workspace. If you've created a new workspace, but not trained or created a model, no Azure Container Registry will exist for the workspace.

  For information on retrieving the name of the Azure Container Registry for your workspace, see the Get container registry name section of this article.

  When using images stored in a **standalone container registry**, you will need to configure a service principal that has at least read access. You then provide the service principal ID (username) and password to anyone that uses images from the registry. The exception is if you make the container registry publicly accessible.

  For information on creating a private Azure Container Registry, see Create a private container registry.

  For information on using service principals with Azure Container Registry, see Azure Container Registry authentication with service principals.

- Azure Container Registry and image information: Provide the image name to anyone that needs to use it. For example, an image named `myimage`, stored in a registry named `myregistry`, is referenced as `myregistry.azurecr.io/myimage` when using the image for model deployment

- Image requirements: Azure Machine Learning only supports Docker images that provide the following software:

  - Ubuntu 16.04 or greater.
  - Conda 4.5.# or greater.
  - Python 3.5.# or 3.6.#.

Get container registry information

In this section, learn how to get the name of the Azure Container Registry for your Azure Machine Learning workspace.

> **WARNING**
> The Azure Container Registry for your workspace is **created the first time you train or deploy a model** using the workspace. If you've created a new workspace, but not trained or created a model, no Azure Container Registry will exist for the workspace.

If you've already trained or deployed models using Azure Machine Learning, a container registry was created for your workspace. To find the name of this container registry, use the following steps:

1. Open a new shell or command-prompt and use the following command to authenticate to your Azure subscription:

   ```
   az login
   ```

   Follow the prompts to authenticate to the subscription.

   > **TIP**
   > After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.
   >
   > To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

2. Use the following command to list the container registry for the workspace. Replace `<myworkspace>` with your Azure Machine Learning workspace name. Replace `<resourcegroup>` with the Azure resource group that contains your workspace:

   ```
   az ml workspace show -w <myworkspace> -g <resourcegroup> --query containerRegistry
   ```

   > **TIP**
   > If you get an error message stating that the ml extension isn't installed, use the following command to install it:
   >
   > ```
   > az extension add -n azure-cli-ml
   > ```

   The information returned is similar to the following text:

   ```
   /subscriptions/<subscription_id>/resourceGroups/<resource_group>/providers/Microsoft.ContainerRegistry
   /registries/<registry_name>
   ```

   The `<registry_name>` value is the name of the Azure Container Registry for your workspace.

Build a custom base image

The steps in this section walk-through creating a custom Docker image in your Azure Container Registry.

1. Create a new text file named `Dockerfile`, and use the following text as the contents:

```
FROM ubuntu:16.04

ARG CONDA_VERSION=4.5.12
ARG PYTHON_VERSION=3.6

ENV LANG=C.UTF-8 LC_ALL=C.UTF-8
ENV PATH /opt/miniconda/bin:$PATH

RUN apt-get update --fix-missing && \
    apt-get install -y wget bzip2 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

RUN wget --quiet https://repo.anaconda.com/miniconda/Miniconda3-${CONDA_VERSION}-Linux-x86_64.sh -O
~/miniconda.sh && \
    /bin/bash ~/miniconda.sh -b -p /opt/miniconda && \
    rm ~/miniconda.sh && \
    /opt/miniconda/bin/conda clean -tipsy

RUN conda install -y conda=${CONDA_VERSION} python=${PYTHON_VERSION} && \
    conda clean -aqy && \
    rm -rf /opt/miniconda/pkgs && \
    find / -type d -name __pycache__ -prune -exec rm -rf {} \;
```

2. From a shell or command-prompt, use the following to authenticate to the Azure Container Registry. Replace the `<registry_name>` with the name of the container registry you want to store the image in:

```
az acr login --name <registry_name>
```

3. To upload the Dockerfile, and build it, use the following command. Replace `<registry_name>` with the name of the container registry you want to store the image in:

```
az acr build --image myimage:v1 --registry <registry_name> --file Dockerfile .
```

> **TIP**
> In this example, a tag of `:v1` is applied to the image. If no tag is provided, a tag of `:latest` is applied.

During the build process, information is streamed to back to the command line. If the build is successful, you receive a message similar to the following text:

```
Run ID: cda was successful after 2m56s
```

For more information on building images with an Azure Container Registry, see Build and run a container image using Azure Container Registry Tasks

For more information on uploading existing images to an Azure Container Registry, see Push your first image to a private Docker container registry.

Use a custom base image

To use a custom image, you need the following information:

- The **image name**. For example, `mcr.microsoft.com/azureml/o16n-sample-user-base/ubuntu-miniconda` is the path to a basic Docker Image provided by Microsoft.

> **IMPORTANT**
>
> For custom images that you've created, be sure to include any tags that were used with the image. For example, if your image was created with a specific tag, such as `:v1`. If you did not use a specific tag when creating the image, a tag of `:latest` was applied.

- If the image is in a **private repository**, you need the following information:
  - The registry **address**. For example, `myregistry.azureecr.io`.
  - A service principal **username** and **password** that has read access to the registry.

  If you do not have this information, speak to the administrator for the Azure Container Registry that contains your image.

Publicly available base images

Microsoft provides several docker images on a publicly accessible repository, which can be used with the steps in this section:

IMAGE	DESCRIPTION
`mcr.microsoft.com/azureml/o16n-sample-user-base/ubuntu-miniconda`	Basic image for Azure Machine Learning
`mcr.microsoft.com/azureml/onnxruntime:latest`	Contains ONNX Runtime for CPU inferencing
`mcr.microsoft.com/azureml/onnxruntime:latest-cuda`	Contains the ONNX Runtime and CUDA for GPU
`mcr.microsoft.com/azureml/onnxruntime:latest-tensorrt`	Contains ONNX Runtime and TensorRT for GPU
`mcr.microsoft.com/azureml/onnxruntime:latest-openvino-vadm`	Contains ONNX Runtime and OpenVINO for Intel Vision Accelerator Design based on Movidius™ MyriadX VPUs
`mcr.microsoft.com/azureml/onnxruntime:latest-openvino-myriad`	Contains ONNX Runtime and OpenVINO for Intel Movidius™ USB sticks

For more information about the ONNX Runtime base images see the ONNX Runtime dockerfile section in the GitHub repo.

> **TIP**
>
> Since these images are publicly available, you do not need to provide an address, username or password when using them.

For more information, see Azure Machine Learning containers.

> **TIP**
>
> **If your model is trained on Azure Machine Learning Compute**, using **version 1.0.22 or greater** of the Azure Machine Learning SDK, an image is created during training. To discover the name of this image, use `run.properties["AzureML.DerivedImageName"]`. The following example demonstrates how to use this image:
>
> ```
> # Use an image built during training with SDK 1.0.22 or greater
> image_config.base_image = run.properties["AzureML.DerivedImageName"]
> ```

Use an image with the Azure Machine Learning SDK

To use an image stored in the **Azure Container Registry for your workspace**, or a **container registry that is publicly accessible**, set the following Environment attributes:

- `docker.enabled=True`
- `docker.base_image`: Set to the registry and path to the image.

```
from azureml.core.environment import Environment
# Create the environment
myenv = Environment(name="myenv")
# Enable Docker and reference an image
myenv.docker.enabled = True
myenv.docker.base_image = "mcr.microsoft.com/azureml/o16n-sample-user-base/ubuntu-miniconda"
```

To use an image from a **private container registry** that is not in your workspace, you must use `docker.base_image_registry` to specify the address of the repository and a user name and password:

```
# Set the container registry information
myenv.docker.base_image_registry.address = "myregistry.azurecr.io"
myenv.docker.base_image_registry.username = "username"
myenv.docker.base_image_registry.password = "password"

myenv.inferencing_stack_version = "latest"  # This will install the inference specific apt packages.

# Define the packages needed by the model and scripts
from azureml.core.conda_dependencies import CondaDependencies
conda_dep = CondaDependencies()
# you must list azureml-defaults as a pip dependency
conda_dep.add_pip_package("azureml-defaults")
myenv.python.conda_dependencies=conda_dep
```

You must add azureml-defaults with version >= 1.0.45 as a pip dependency. This package contains the functionality needed to host the model as a web service. You must also set inferencing_stack_version property on the environment to "latest", this will install specific apt packages needed by web service.

After defining the environment, use it with an InferenceConfig object to define the inference environment in which the model and web service will run.

```
from azureml.core.model import InferenceConfig
# Use environment in InferenceConfig
inference_config = InferenceConfig(entry_script="score.py",
                                   environment=myenv)
```

At this point, you can continue with deployment. For example, the following code snippet would deploy a web service locally using the inference configuration and custom image:

```
from azureml.core.webservice import LocalWebservice, Webservice

deployment_config = LocalWebservice.deploy_configuration(port=8890)
service = Model.deploy(ws, "myservice", [model], inference_config, deployment_config)
service.wait_for_deployment(show_output = True)
print(service.state)
```

For more information on deployment, see Deploy models with Azure Machine Learning.

For more information on customizing your Python environment, see Create and manage environments for training and deployment.

Use an image with the Machine Learning CLI

> **IMPORTANT**
> Currently the Machine Learning CLI can use images from the Azure Container Registry for your workspace or publicly accessible repositories. It cannot use images from standalone private registries.

Before deploying a model using the Machine Learning CLI, create an environment that uses the custom image. Then create an inference configuration file that references the environment. You can also define the environment directly in the inference configuration file. The following JSON document demonstrates how to reference an image in a public container registry. In this example, the environment is defined inline:

```
{
    "entryScript": "score.py",
    "environment": {
        "docker": {
            "arguments": [],
            "baseDockerfile": null,
            "baseImage": "mcr.microsoft.com/azureml/o16n-sample-user-base/ubuntu-miniconda",
            "enabled": false,
            "sharedVolumes": true,
            "shmSize": null
        },
        "environmentVariables": {
            "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
        },
        "name": "my-deploy-env",
        "python": {
            "baseCondaEnvironment": null,
            "condaDependencies": {
                "channels": [
                    "conda-forge"
                ],
                "dependencies": [
                    "python=3.6.2",
                    {
                        "pip": [
                            "azureml-defaults",
                            "azureml-telemetry",
                            "scikit-learn",
                            "inference-schema[numpy-support]"
                        ]
                    }
                ],
                "name": "project_environment"
            },
            "condaDependenciesFile": null,
            "interpreterPath": "python",
            "userManagedDependencies": false
        },
        "version": "1"
    }
}
```

This file is used with the `az ml model deploy` command. The `--ic` parameter is used to specify the inference configuration file.

```
az ml model deploy -n myservice -m mymodel:1 --ic inferenceconfig.json --dc deploymentconfig.json --ct
akscomputetarget
```

For more information on deploying a model using the ML CLI, see the "model registration, profiling, and

deployment" section of the CLI extension for Azure Machine Learning article.

Next steps

- Learn more about Where to deploy and how.
- Learn how to Train and deploy machine learning models using Azure Pipelines.

Use an existing model with Azure Machine Learning

3/17/2020 • 7 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to use an existing machine learning model with Azure Machine Learning.

If you have a machine learning model that was trained outside Azure Machine Learning, you can still use the service to deploy the model as a web service or to an IoT Edge device.

> **TIP**
> This article provides basic information on registering and deploying an existing model. Once deployed, Azure Machine Learning provides monitoring for your model. It also allows you to store input data sent to the deployment, which can be used for data drift analysis or training new versions of the model.
>
> For more information on the concepts and terms used here, see Manage, deploy, and monitor machine learning models.
>
> For general information on the deployment process, see Deploy models with Azure Machine Learning.

Prerequisites

- An Azure Machine Learning workspace. For more information, see Create a workspace.

  > **TIP**
  > The Python examples in this article assume that the `ws` variable is set to your Azure Machine Learning workspace.
  >
  > The CLI examples use a placeholder of `myworkspace` and `myresourcegroup`. Replace these with the name of your workspace and the resource group that contains it.

- The Azure Machine Learning SDK.
- The Azure CLI and Machine Learning CLI extension.
- A trained model. The model must be persisted to one or more files on your development environment.

  > **NOTE**
  > To demonstrate registering a model trained outside Azure Machine Learning, the example code snippets in this article use the models created by Paolo Ripamonti's Twitter sentiment analysis project: https://www.kaggle.com/paoloripamonti/twitter-sentiment-analysis.

Register the model(s)

Registering a model allows you to store, version, and track metadata about models in your workspace. In the following Python and CLI examples, the `models` directory contains the `model.h5`, `model.w2v`, `encoder.pkl`, and `tokenizer.pkl` files. This example uploads the files contained in the `models` directory as a new model registration named `sentiment`:

```
from azureml.core.model import Model
# Tip: When model_path is set to a directory, you can use the child_paths parameter to include
#      only some of the files from the directory
model = Model.register(model_path = "./models",
                       model_name = "sentiment",
                       description = "Sentiment analysis model trained outside Azure Machine Learning",
                       workspace = ws)
```

For more information, see the Model.register() reference.

```
az ml model register -p ./models -n sentiment -w myworkspace -g myresourcegroup
```

TIP

You can also set add `tags` and `properties` dictionary objects to the registered model. These values can be used later to help identify a specific model. For example, the framework used, training parameters, etc.

For more information, see the az ml model register reference.

For more information on model registration in general, see Manage, deploy, and monitor machine learning models.

Define inference configuration

The inference configuration defines the environment used to run the deployed model. The inference configuration references the following entities, which are used to run the model when it's deployed:

- An entry script. This file (named `score.py`) loads the model when the deployed service starts. It is also responsible for receiving data, passing it to the model, and then returning a response.
- An Azure Machine Learning environment. An environment defines the software dependencies needed to run the model and entry script.

The following example shows how to use the SDK to create an environment and then use it with an inference configuration:

```
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies

# Create the environment
myenv = Environment(name="myenv")
conda_dep = CondaDependencies()

# Define the packages needed by the model and scripts
conda_dep.add_conda_package("tensorflow")
conda_dep.add_conda_package("numpy")
conda_dep.add_conda_package("scikit-learn")
# You must list azureml-defaults as a pip dependency
conda_dep.add_pip_package("azureml-defaults")
conda_dep.add_pip_package("keras")
conda_dep.add_pip_package("gensim")

# Adds dependencies to PythonSection of myenv
myenv.python.conda_dependencies=conda_dep

inference_config = InferenceConfig(entry_script="score.py",
                                   environment=myenv)
```

For more information, see the following articles:

- How to use environments.
- InferenceConfig reference.

The CLI loads the inference configuration from a YAML file:

```
{
   "entryScript": "score.py",
   "runtime": "python",
   "condaFile": "myenv.yml"
}
```

With the CLI, the conda environment is defined in the `myenv.yml` file referenced by the inference configuration. The following YAML is the contents of this file:

```
name: inference_environment
dependencies:
- python=3.6.2
- tensorflow
- numpy
- scikit-learn
- pip:
    - azureml-defaults
    - keras
    - gensim
```

For more information on inference configuration, see Deploy models with Azure Machine Learning.

Entry script

The entry script has only two required functions, `init()` and `run(data)`. These functions are used to initialize the service at startup and run the model using request data passed in by a client. The rest of the script handles loading and running the model(s).

> **IMPORTANT**
> There isn't a generic entry script that works for all models. It is always specific to the model that is used. It must understand how to load the model, the data format that the model expects, and how to score data using the model.

The following Python code is an example entry script (`score.py`):

```
import os
import pickle
import json
import time
from keras.models import load_model
from keras.preprocessing.sequence import pad_sequences
from gensim.models.word2vec import Word2Vec

# SENTIMENT
POSITIVE = "POSITIVE"
NEGATIVE = "NEGATIVE"
NEUTRAL = "NEUTRAL"
SENTIMENT_THRESHOLDS = (0.4, 0.7)
SEQUENCE_LENGTH = 300

# Called when the deployed service starts
def init():
    global model
```

```
    global tokenizer
    global encoder
    global w2v_model

    # Get the path where the deployed model can be found.
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), './models')
    # load models
    model = load_model(model_path + '/model.h5')
    w2v_model = Word2Vec.load(model_path + '/model.w2v')

    with open(model_path + '/tokenizer.pkl','rb') as handle:
        tokenizer = pickle.load(handle)

    with open(model_path + '/encoder.pkl','rb') as handle:
        encoder = pickle.load(handle)

# Handle requests to the service
def run(data):
    try:
        # Pick out the text property of the JSON request.
        # This expects a request in the form of {"text": "some text to score for sentiment"}
        data = json.loads(data)
        prediction = predict(data['text'])
        #Return prediction
        return prediction
    except Exception as e:
        error = str(e)
        return error

# Determine sentiment from score
def decode_sentiment(score, include_neutral=True):
    if include_neutral:
        label = NEUTRAL
        if score <= SENTIMENT_THRESHOLDS[0]:
            label = NEGATIVE
        elif score >= SENTIMENT_THRESHOLDS[1]:
            label = POSITIVE
        return label
    else:
        return NEGATIVE if score < 0.5 else POSITIVE

# Predict sentiment using the model
def predict(text, include_neutral=True):
    start_at = time.time()
    # Tokenize text
    x_test = pad_sequences(tokenizer.texts_to_sequences([text]), maxlen=SEQUENCE_LENGTH)
    # Predict
    score = model.predict([x_test])[0]
    # Decode sentiment
    label = decode_sentiment(score, include_neutral=include_neutral)

    return {"label": label, "score": float(score),
       "elapsed_time": time.time()-start_at}
```

For more information on entry scripts, see Deploy models with Azure Machine Learning.

Define deployment

The Webservice package contains the classes used for deployment. The class you use determines where the model is deployed. For example, to deploy as a web service on Azure Kubernetes Service, use AksWebService.deploy_configuration() to create the deployment configuration.

The following Python code defines a deployment configuration for a local deployment. This configuration deploys the model as a web service to your local computer.

IMPORTANT

A local deployment requires a working installation of Docker on your local computer:

```
from azureml.core.webservice import LocalWebservice

deployment_config = LocalWebservice.deploy_configuration()
```

For more information, see the LocalWebservice.deploy_configuration() reference.

The CLI loads the deployment configuration from a YAML file:

```
{
    "computeType": "LOCAL"
}
```

Deploying to a different compute target, such as Azure Kubernetes Service in the Azure cloud, is as easy as changing the deployment configuration. For more information, see How and where to deploy models.

Deploy the model

The following example loads information on the registered model named `sentiment`, and then deploys it as a service named `sentiment`. During deployment, the inference configuration and deployment configuration are used to create and configure the service environment:

```
from azureml.core.model import Model

model = Model(ws, name='sentiment')
service = Model.deploy(ws, 'myservice', [model], inference_config, deployment_config)

service.wait_for_deployment(True)
print(service.state)
print("scoring URI: " + service.scoring_uri)
```

For more information, see the Model.deploy() reference.

To deploy the model from the CLI, use the following command. This command deploys version 1 of the registered model (`sentiment:1`) using the inference and deployment configuration stored in the `inferenceConfig.json` and `deploymentConfig.json` files:

```
az ml model deploy -n myservice -m sentiment:1 --ic inferenceConfig.json --dc deploymentConfig.json
```

For more information, see the az ml model deploy reference.

For more information on deployment, see How and where to deploy models.

Request-response consumption

After deployment, the scoring URI is displayed. This URI can be used by clients to submit requests to the service. The following example is a basic Python client that submits data to the service and displays the response:

```
import requests
import json

scoring_uri = 'scoring uri for your service'
headers = {'Content-Type':'application/json'}

test_data = json.dumps({'text': 'Today is a great day!'})

response = requests.post(scoring_uri, data=test_data, headers=headers)
print(response.status_code)
print(response.elapsed)
print(response.json())
```

For more information on how to consume the deployed service, see Create a client.

Next steps

- Monitor your Azure Machine Learning models with Application Insights
- Collect data for models in production
- How and where to deploy models
- How to create a client for a deployed model

Troubleshooting Azure Machine Learning Azure Kubernetes Service and Azure Container Instances deployment

3/5/2020 • 13 minutes to read • Edit Online

Learn how to work around or solve common Docker deployment errors with Azure Container Instances (ACI) and Azure Kubernetes Service (AKS) using Azure Machine Learning.

When deploying a model in Azure Machine Learning, the system performs a number of tasks.

The recommended and the most up to date approach for model deployment is via the Model.deploy() API using an Environment object as an input parameter. In this case our service will create a base docker image for you during deployment stage and mount the required models all in one call. The basic deployment tasks are:

1. Register the model in the workspace model registry.
2. Define Inference Configuration:
    a. Create an Environment object based on the dependencies you specify in the environment yaml file or use one of our procured environments.
    b. Create an inference configuration (InferenceConfig object) based on the environment and the scoring script.
3. Deploy the model to Azure Container Instance (ACI) service or to Azure Kubernetes Service (AKS).

Learn more about this process in the Model Management introduction.

Prerequisites

- An **Azure subscription**. If you do not have one, try the free or paid version of Azure Machine Learning.
- The Azure Machine Learning SDK.
- The Azure CLI.
- The CLI extension for Azure Machine Learning.
- To debug locally, you must have a working Docker installation on your local system.

  To verify your Docker installation, use the command `docker run hello-world` from a terminal or command prompt. For information on installing Docker, or troubleshooting Docker errors, see the Docker Documentation.

Before you begin

If you run into any issue, the first thing to do is to break down the deployment task (previous described) into individual steps to isolate the problem.

Assuming you are using the new/recommended deployment method via Model.deploy() API with an Environment object as an input parameter, your code can be broken down into three major steps:

1. Register the model. Here is some sample code:

```
from azureml.core.model import Model


# register a model out of a run record
model = best_run.register_model(model_name='my_best_model', model_path='outputs/my_model.pkl')

# or, you can register a file or a folder of files as a model
model = Model.register(model_path='my_model.pkl', model_name='my_best_model', workspace=ws)
```

2. Define inference configuration for deployment:

```
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment


# create inference configuration based on the requirements defined in the YAML
myenv = Environment.from_conda_specification(name="myenv", file_path="myenv.yml")
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
```

3. Deploy the model using the inference configuration created in the previous step:

```
from azureml.core.webservice import AciWebservice


# deploy the model
aci_config = AciWebservice.deploy_configuration(cpu_cores=1, memory_gb=1)
aci_service = Model.deploy(workspace=ws,
                       name='my-service',
                       models=[model],
                       inference_config=inference_config,
                       deployment_config=aci_config)
aci_service.wait_for_deployment(show_output=True)
```

Once you have broken down the deployment process into individual tasks, we can look at some of the most common errors.

Debug locally

If you encounter problems deploying a model to ACI or AKS, try deploying it as a local web service. Using a local web service makes it easier to troubleshoot problems. The Docker image containing the model is downloaded and started on your local system.

> **WARNING**
> Local web service deployments are not supported for production scenarios.

To deploy locally, modify your code to use `LocalWebservice.deploy_configuration()` to create a deployment configuration. Then use `Model.deploy()` to deploy the service. The following example deploys a model (contained in the model variable) as a local web service:

```
from azureml.core.environment import Environment
from azureml.core.model import InferenceConfig, Model
from azureml.core.webservice import LocalWebservice


# Create inference configuration based on the environment definition and the entry script
myenv = Environment.from_conda_specification(name="env", file_path="myenv.yml")
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
# Create a local deployment, using port 8890 for the web service endpoint
deployment_config = LocalWebservice.deploy_configuration(port=8890)
# Deploy the service
service = Model.deploy(
    ws, "mymodel", [model], inference_config, deployment_config)
# Wait for the deployment to complete
service.wait_for_deployment(True)
# Display the port that the web service is available on
print(service.port)
```

Please note that if you are defining your own conda specification YAML, you must list azureml-defaults with version >= 1.0.45 as a pip dependency. This package contains the functionality needed to host the model as a web service.

At this point, you can work with the service as normal. For example, the following code demonstrates sending data to the service:

```
import json

test_sample = json.dumps({'data': [
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
]})

test_sample = bytes(test_sample, encoding='utf8')

prediction = service.run(input_data=test_sample)
print(prediction)
```

For more information on customizing your Python environment, see Create and manage environments for training and deployment.

Update the service

During local testing, you may need to update the `score.py` file to add logging or attempt to resolve any problems that you've discovered. To reload changes to the `score.py` file, use `reload()`. For example, the following code reloads the script for the service, and then sends data to it. The data is scored using the updated `score.py` file:

> **IMPORTANT**
>
> The `reload` method is only available for local deployments. For information on updating a deployment to another compute target, see the update section of Deploy models.

```
service.reload()
print(service.run(input_data=test_sample))
```

> **NOTE**
>
> The script is reloaded from the location specified by the `InferenceConfig` object used by the service.

To change the model, Conda dependencies, or deployment configuration, use update(). The following example updates the model used by the service:

```
service.update([different_model], inference_config, deployment_config)
```

Delete the service

To delete the service, use delete().

Inspect the Docker log

You can print out detailed Docker engine log messages from the service object. You can view the log for ACI, AKS, and Local deployments. The following example demonstrates how to print the logs.

```
# if you already have the service object handy
print(service.get_logs())

# if you only know the name of the service (note there might be multiple services with the same name but
different version number)
print(ws.webservices['mysvc'].get_logs())
```

Service launch fails

After the image is successfully built, the system attempts to start a container using your deployment configuration. As part of container starting-up process, the `init()` function in your scoring script is invoked by the system. If there are uncaught exceptions in the `init()` function, you might see **CrashLoopBackOff** error in the error message.

Use the info in the Inspect the Docker log section to check the logs.

Function fails: get_model_path()

Often, in the `init()` function in the scoring script, Model.get_model_path() function is called to locate a model file or a folder of model files in the container. If the model file or folder cannot be found, the function fails. The easiest way to debug this error is to run the below Python code in the Container shell:

```
from azureml.core.model import Model
import logging
logging.basicConfig(level=logging.DEBUG)
print(Model.get_model_path(model_name='my-best-model'))
```

This example prints out the local path (relative to `/var/azureml-app`) in the container where your scoring script is expecting to find the model file or folder. Then you can verify if the file or folder is indeed where it is expected to be.

Setting the logging level to DEBUG may cause additional information to be logged, which may be useful in identifying the failure.

Function fails: run(input_data)

If the service is successfully deployed, but it crashes when you post data to the scoring endpoint, you can add error catching statement in your `run(input_data)` function so that it returns detailed error message instead. For example:

```
def run(input_data):
    try:
        data = json.loads(input_data)['data']
        data = np.array(data)
        result = model.predict(data)
        return json.dumps({"result": result.tolist()})
    except Exception as e:
        result = str(e)
        # return error message back to the client
        return json.dumps({"error": result})
```

Note: Returning error messages from the `run(input_data)` call should be done for debugging purpose only. For security reasons, you should not return error messages this way in a production environment.

HTTP status code 502

A 502 status code indicates that the service has thrown an exception or crashed in the `run()` method of the score.py file. Use the information in this article to debug the file.

HTTP status code 503

Azure Kubernetes Service deployments support autoscaling, which allows replicas to be added to support additional load. However, the autoscaler is designed to handle **gradual** changes in load. If you receive large spikes in requests per second, clients may receive an HTTP status code 503.

There are two things that can help prevent 503 status codes:

- Change the utilization level at which autoscaling creates new replicas.

  By default, autoscaling target utilization is set to 70%, which means that the service can handle spikes in requests per second (RPS) of up to 30%. You can adjust the utilization target by setting the `autoscale_target_utilization` to a lower value.

  > **IMPORTANT**
  > This change does not cause replicas to be created *faster*. Instead, they are created at a lower utilization threshold. Instead of waiting until the service is 70% utilized, changing the value to 30% causes replicas to be created when 30% utilization occurs.

  If the web service is already using the current max replicas and you are still seeing 503 status codes, increase the `autoscale_max_replicas` value to increase the maximum number of replicas.

- Change the minimum number of replicas. Increasing the minimum replicas provides a larger pool to handle the incoming spikes.

  To increase the minimum number of replicas, set `autoscale_min_replicas` to a higher value. You can calculate the required replicas by using the following code, replacing values with values specific to your project:

```
from math import ceil
# target requests per second
targetRps = 20
# time to process the request (in seconds)
reqTime = 10
# Maximum requests per container
maxReqPerContainer = 1
# target_utilization. 70% in this example
targetUtilization = .7

concurrentRequests = targetRps * reqTime / targetUtilization

# Number of container replicas
replicas = ceil(concurrentRequests / maxReqPerContainer)
```

> **NOTE**
> If you receive request spikes larger than the new minimum replicas can handle, you may receive 503s again. For example, as traffic to your service increases, you may need to increase the minimum replicas.

For more information on setting `autoscale_target_utilization`, `autoscale_max_replicas`, and `autoscale_min_replicas` for, see the AksWebservice module reference.

HTTP status code 504

A 504 status code indicates that the request has timed out. The default timeout is 1 minute.

You can increase the timeout or try to speed up the service by modifying the score.py to remove unnecessary calls. If these actions do not correct the problem, use the information in this article to debug the score.py file. The code may be in a hung state or an infinite loop.

Advanced debugging

In some cases, you may need to interactively debug the Python code contained in your model deployment. For example, if the entry script is failing and the reason cannot be determined by additional logging. By using Visual Studio Code and the Python Tools for Visual Studio (PTVSD), you can attach to the code running inside the Docker container.

> **IMPORTANT**
> This method of debugging does not work when using `Model.deploy()` and `LocalWebservice.deploy_configuration` to deploy a model locally. Instead, you must create an image using the Model.package() method.

Local web service deployments require a working Docker installation on your local system. For more information on using Docker, see the Docker Documentation.

Configure development environment

1. To install the Python Tools for Visual Studio (PTVSD) on your local VS Code development environment, use the following command:

   ```
   python -m pip install --upgrade ptvsd
   ```

   For more information on using PTVSD with VS Code, see Remote Debugging.

2. To configure VS Code to communicate with the Docker image, create a new debug configuration:

a. From VS Code, select the **Debug** menu and then select **Open configurations**. A file named **launch.json** opens.

b. In the **launch.json** file, find the line that contains `"configurations": [`, and insert the following text after it:

```
{
    "name": "Azure Machine Learning: Docker Debug",
    "type": "python",
    "request": "attach",
    "port": 5678,
    "host": "localhost",
    "pathMappings": [
        {
            "localRoot": "${workspaceFolder}",
            "remoteRoot": "/var/azureml-app"
        }
    ]
}
```

> **IMPORTANT**
> If there are already other entries in the configurations section, add a comma (,) after the code that you inserted.

This section attaches to the Docker container using port 5678.

c. Save the **launch.json** file.

Create an image that includes PTVSD

1. Modify the conda environment for your deployment so that it includes PTVSD. The following example demonstrates adding it using the `pip_packages` parameter:

```
from azureml.core.conda_dependencies import CondaDependencies


# Usually a good idea to choose specific version numbers
# so training is made on same packages as scoring
myenv = CondaDependencies.create(conda_packages=['numpy==1.15.4',
                            'scikit-learn==0.19.1', 'pandas==0.23.4'],
                             pip_packages = ['azureml-defaults==1.0.45', 'ptvsd'])

with open("myenv.yml","w") as f:
    f.write(myenv.serialize_to_string())
```

2. To start PTVSD and wait for a connection when the service starts, add the following to the top of your `score.py` file:

```
import ptvsd
# Allows other computers to attach to ptvsd on this IP address and port.
ptvsd.enable_attach(address=('0.0.0.0', 5678), redirect_output = True)
# Wait 30 seconds for a debugger to attach. If none attaches, the script continues as normal.
ptvsd.wait_for_attach(timeout = 30)
print("Debugger attached...")
```

3. Create an image based on the environment definition and pull the image to the local registry. During debugging, you may want to make changes to the files in the image without having to recreate it. To install a text editor (vim) in the Docker image, use the `Environment.docker.base_image` and

`Environment.docker.base_dockerfile` properties:

> **NOTE**
>
> This example assumes that `ws` points to your Azure Machine Learning workspace, and that `model` is the model being deployed. The `myenv.yml` file contains the conda dependencies created in step 1.

```
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment


myenv = Environment.from_conda_specification(name="env", file_path="myenv.yml")
myenv.docker.base_image = None
myenv.docker.base_dockerfile = "FROM mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04\nRUN
apt-get update && apt-get install vim -y"
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
package = Model.package(ws, [model], inference_config)
package.wait_for_creation(show_output=True)  # Or show_output=False to hide the Docker build logs.
package.pull()
```

Once the image has been created and downloaded, the image path (includes repository, name, and tag, which in this case is also its digest) is displayed in a message similar to the following:

```
Status: Downloaded newer image for myregistry.azurecr.io/package@sha256:<image-digest>
```

4. To make it easier to work with the image, use the following command to add a tag. Replace `myimagepath` with the location value from the previous step.

   ```
   docker tag myimagepath debug:1
   ```

   For the rest of the steps, you can refer to the local image as `debug:1` instead of the full image path value.

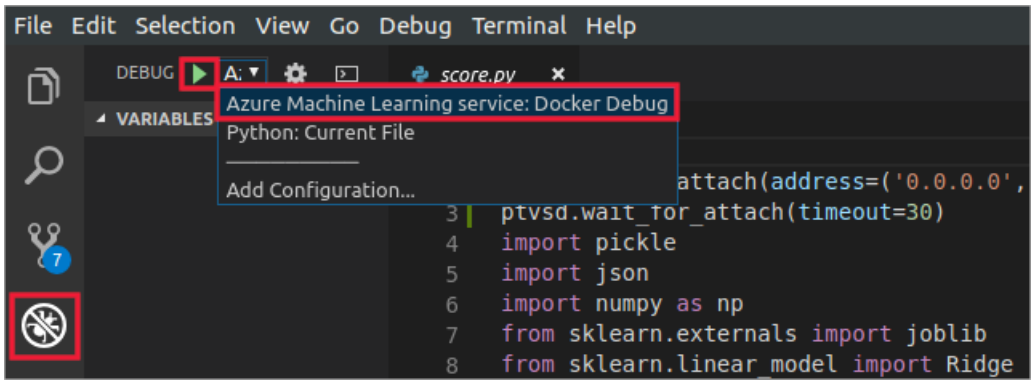
Debug the service

> **TIP**
>
> If you set a timeout for the PTVSD connection in the `score.py` file, you must connect VS Code to the debug session before the timeout expires. Start VS Code, open the local copy of `score.py`, set a breakpoint, and have it ready to go before using the steps in this section.
>
> For more information on debugging and setting breakpoints, see Debugging.

1. To start a Docker container using the image, use the following command:

   ```
   docker run --rm --name debug -p 8000:5001 -p 5678:5678 debug:1
   ```

2. To attach VS Code to PTVSD inside the container, open VS Code and use the F5 key or select **Debug**. When prompted, select the **Azure Machine Learning: Docker Debug** configuration. You can also select the debug icon from the side bar, the **Azure Machine Learning: Docker Debug** entry from the Debug dropdown menu, and then use the green arrow to attach the debugger.



At this point, VS Code connects to PTVSD inside the Docker container and stops at the breakpoint you set previously. You can now step through the code as it runs, view variables, etc.

For more information on using VS Code to debug Python, see Debug your Python code.

Modify the container files

To make changes to files in the image, you can attach to the running container, and execute a bash shell. From there, you can use vim to edit files:

1. To connect to the running container and launch a bash shell in the container, use the following command:

   ```
   docker exec -it debug /bin/bash
   ```

2. To find the files used by the service, use the following command from the bash shell in the container if the default directory is different than `/var/azureml-app`:

   ```
   cd /var/azureml-app
   ```

   From here, you can use vim to edit the `score.py` file. For more information on using vim, see Using the Vim editor.

3. Changes to a container are not normally persisted. To save any changes you make, use the following command, before you exit the shell started in the step above (that is, in another shell):

   ```
   docker commit debug debug:2
   ```

   This command creates a new image named `debug:2` that contains your edits.

   > **TIP**
   > You will need to stop the current container and start using the new version before changes take effect.

4. Make sure to keep the changes you make to files in the container in sync with the local files that VS Code uses. Otherwise, the debugger experience will not work as expected.

Stop the container

To stop the container, use the following command:

```
docker stop debug
```

Next steps

Learn more about deployment:

- How to deploy and where
- Tutorial: Train & deploy models

Consume an Azure Machine Learning model deployed as a web service

4/14/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Deploying an Azure Machine Learning model as a web service creates a REST API endpoint. You can send data to this endpoint and receive the prediction returned by the model. In this document, learn how to create clients for the web service by using C#, Go, Java, and Python.

You create a web service when you deploy a model to your local environment, Azure Container Instances, Azure Kubernetes Service, or field-programmable gate arrays (FPGA). You retrieve the URI used to access the web service by using the Azure Machine Learning SDK. If authentication is enabled, you can also use the SDK to get the authentication keys or tokens.

The general workflow for creating a client that uses a machine learning web service is:

1. Use the SDK to get the connection information.
2. Determine the type of request data used by the model.
3. Create an application that calls the web service.

> **TIP**
> The examples in this document are manually created without the use of OpenAPI (Swagger) specifications. If you've enabled an OpenAPI specification for your deployment, you can use tools such as swagger-codegen to create client libraries for your service.

Connection information

> **NOTE**
> Use the Azure Machine Learning SDK to get the web service information. This is a Python SDK. You can use any language to create a client for the service.

The azureml.core.Webservice class provides the information you need to create a client. The following `Webservice` properties are useful for creating a client application:

- `auth_enabled` - If key authentication is enabled, `True`; otherwise, `False`.
- `token_auth_enabled` - If token authentication is enabled, `True`; otherwise, `False`.
- `scoring_uri` - The REST API address.
- `swagger_uri` - The address of the OpenAPI specification. This URI is available if you enabled automatic schema generation. For more information, see Deploy models with Azure Machine Learning.

There are a three ways to retrieve this information for deployed web services:

- When you deploy a model, a `Webservice` object is returned with information about the service:

```
service = Model.deploy(ws, "myservice", [model], inference_config, deployment_config)
service.wait_for_deployment(show_output = True)
print(service.scoring_uri)
print(service.swagger_uri)
```

- You can use `Webservice.list` to retrieve a list of deployed web services for models in your workspace. You can add filters to narrow the list of information returned. For more information about what can be filtered on, see the Webservice.list reference documentation.

```
services = Webservice.list(ws)
print(services[0].scoring_uri)
print(services[0].swagger_uri)
```

- If you know the name of the deployed service, you can create a new instance of `Webservice`, and provide the workspace and service name as parameters. The new object contains information about the deployed service.

```
service = Webservice(workspace=ws, name='myservice')
print(service.scoring_uri)
print(service.swagger_uri)
```

Secured web service

If you secured the deployed web service using a TLS/SSL certificate, you can use HTTPS to connect to the service using the scoring or swagger URI. HTTPS helps secure communications between a client and a web service by encrypting communications between the two. Encryption uses Transport Layer Security (TLS). TLS is sometimes still referred to as *Secure Sockets Layer* (SSL), which was the predecessor of TLS.

> **IMPORTANT**
> Web services deployed by Azure Machine Learning only support TLS version 1.2. When creating a client application, make sure that it supports this version.

For more information, see Use TLS to secure a web service through Azure Machine Learning.

Authentication for services

Azure Machine Learning provides two ways to control access to your web services.

AUTHENTICATION METHOD	ACI	AKS
Key	Disabled by default	Enabled by default
Token	Not Available	Disabled by default

When sending a request to a service that is secured with a key or token, use the **Authorization** header to pass the key or token. The key or token must be formatted as `Bearer <key-or-token>`, where `<key-or-token>` is your key or token value.

Authentication with keys

When you enable authentication for a deployment, you automatically create authentication keys.

- Authentication is enabled by default when you are deploying to Azure Kubernetes Service.
- Authentication is disabled by default when you are deploying to Azure Container Instances.

To control authentication, use the `auth_enabled` parameter when you are creating or updating a deployment.

If authentication is enabled, you can use the `get_keys` method to retrieve a primary and secondary authentication key:

```
primary, secondary = service.get_keys()
print(primary)
```

> **IMPORTANT**
> If you need to regenerate a key, use `service.regen_key`.

Authentication with tokens

When you enable token authentication for a web service, a user must provide an Azure Machine Learning JWT token to the web service to access it.

- Token authentication is disabled by default when you are deploying to Azure Kubernetes Service.
- Token authentication is not supported when you are deploying to Azure Container Instances.

To control token authentication, use the `token_auth_enabled` parameter when you are creating or updating a deployment.

If token authentication is enabled, you can use the `get_token` method to retrieve a bearer token and that tokens expiration time:

```
token, refresh_by = service.get_token()
print(token)
```

> **IMPORTANT**
> You will need to request a new token after the token's `refresh_by` time.

Request data

The REST API expects the body of the request to be a JSON document with the following structure:

```
{
    "data":
        [
            <model-specific-data-structure>
        ]
}
```

> **IMPORTANT**
> The structure of the data needs to match what the scoring script and model in the service expect. The scoring script might modify the data before passing it to the model.

For example, the model in the Train within notebook example expects an array of 10 numbers. The scoring script for this example creates a Numpy array from the request, and passes it to the model. The following example shows the data this service expects:

```
{
    "data":
        [
            [
                0.0199132141783263,
                0.0506801187398187,
                0.104808689473925,
                0.0700725447072635,
                -0.0359677812752396,
                -0.0266789028311707,
                -0.0249926566315915,
                -0.00259226199818282,
                0.00371173823343597,
                0.0403433716478807
            ]
        ]
}
```

The web service can accept multiple sets of data in one request. It returns a JSON document containing an array of responses.

Binary data

For information on how to enable support for binary data in your service, see Binary data.

> **TIP**
>
> Enabling support for binary data happens in the score.py file used by the deployed model. From the client, use the HTTP functionality of your programming language. For example, the following snippet sends the contents of a JPG file to a web service:
>
> ```
> import requests
> # Load image data
> data = open('example.jpg', 'rb').read()
> # Post raw data to scoring URI
> res = request.post(url='<scoring-uri>', data=data, headers={'Content-Type': 'application/> octet-stream'})
> ```

Cross-origin resource sharing (CORS)

For information on enabling CORS support in your service, see Cross-origin resource sharing.

Call the service (C#)

This example demonstrates how to use C# to call the web service created from the Train within notebook example:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json;

namespace MLWebServiceClient
{
    // The data structure expected by the service
    internal class InputData
    {
        [JsonProperty("data")]
        // The service used by this example expects an array containing
        //   one or more arrays of doubles
```

```
        internal double[,] data;
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Set the scoring URI and authentication key or token
            string scoringUri = "<your web service URI>";
            string authKey = "<your key or token>";

            // Set the data to be sent to the service.
            // In this case, we are sending two sets of data to be scored.
            InputData payload = new InputData();
            payload.data = new double[,] {
                {
                    0.0199132141783263,
                    0.0506801187398187,
                    0.104808689473925,
                    0.0700725447072635,
                    -0.0359677812752396,
                    -0.0266789028311707,
                    -0.0249926566315915,
                    -0.00259226199818282,
                    0.00371173823343597,
                    0.0403433716478807
                },
                {
                    -0.0127796318808497,
                    -0.044641636506989,
                    0.0606183944448076,
                    0.0528581912385822,
                    0.0479653430750293,
                    0.0293746718291555,
                    -0.0176293810234174,
                    0.0343088588777263,
                    0.0702112981933102,
                    0.00720651632920303
                }
            };

            // Create the HTTP client
            HttpClient client = new HttpClient();
            // Set the auth header. Only needed if the web service requires authentication.
            client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer",
authKey);

            // Make the request
            try {
                var request = new HttpRequestMessage(HttpMethod.Post, new Uri(scoringUri));
                request.Content = new StringContent(JsonConvert.SerializeObject(payload));
                request.Content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
                var response = client.SendAsync(request).Result;
                // Display the response from the web service
                Console.WriteLine(response.Content.ReadAsStringAsync().Result);
            }
            catch (Exception e)
            {
                Console.Out.WriteLine(e.Message);
            }
        }
    }
}
```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Call the service (Go)

This example demonstrates how to use Go to call the web service created from the Train within notebook example:

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
)

// Features for this model are an array of decimal values
type Features []float64

// The web service input can accept multiple sets of values for scoring
type InputData struct {
    Data []Features `json:"data",omitempty`
}

// Define some example data
var exampleData = []Features{
    []float64{
        0.0199132141783263,
        0.0506801187398187,
        0.104808689473925,
        0.0700725447072635,
        -0.0359677812752396,
        -0.0266789028311707,
        -0.0249926566315915,
        -0.00259226199818282,
        0.00371173823343597,
        0.0403433716478807,
    },
    []float64{
        -0.0127796318808497,
        -0.044641636506989,
        0.0606183944448076,
        0.0528581912385822,
        0.0479653430750293,
        0.0293746718291555,
        -0.0176293810234174,
        0.0343088588777263,
        0.0702112981933102,
        0.00720651632920303,
    },
}

// Set to the URI for your service
var serviceUri string = "<your web service URI>"
// Set to the authentication key or token (if any) for your service
var authKey string = "<your key or token>"

func main() {
    // Create the input data from example data
    jsonData := InputData{
        Data: exampleData,
    }
    // Create JSON from it and create the body for the HTTP request
    jsonValue, _ := json.Marshal(jsonData)
```

```
    body := bytes.NewBuffer(jsonValue)

    // Create the HTTP request
    client := &http.Client{}
    request, err := http.NewRequest("POST", serviceUri, body)
    request.Header.Add("Content-Type", "application/json")

    // These next two are only needed if using an authentication key
    bearer := fmt.Sprintf("Bearer %v", authKey)
    request.Header.Add("Authorization", bearer)

    // Send the request to the web service
    resp, err := client.Do(request)
    if err != nil {
        fmt.Println("Failure: ", err)
    }

    // Display the response received
    respBody, _ := ioutil.ReadAll(resp.Body)
    fmt.Println(string(respBody))
}
```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Call the service (Java)

This example demonstrates how to use Java to call the web service created from the Train within notebook example:

```
import java.io.IOException;
import org.apache.http.client.fluent.*;
import org.apache.http.entity.ContentType;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;

public class App {
    // Handle making the request
    public static void sendRequest(String data) {
        // Replace with the scoring_uri of your service
        String uri = "<your web service URI>";
        // If using authentication, replace with the auth key or token
        String key = "<your key or token>";
        try {
            // Create the request
            Content content = Request.Post(uri)
            .addHeader("Content-Type", "application/json")
            // Only needed if using authentication
            .addHeader("Authorization", "Bearer " + key)
            // Set the JSON data as the body
            .bodyString(data, ContentType.APPLICATION_JSON)
            // Make the request and display the response.
            .execute().returnContent();
            System.out.println(content);
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
    public static void main(String[] args) {
        // Create the data to send to the service
        JSONObject obj = new JSONObject();
        // In this case, it's an array of arrays
```

```
        // In this case, it's an array of arrays
        JSONArray dataItems = new JSONArray();
        // Inner array has 10 elements
        JSONArray item1 = new JSONArray();
        item1.add(0.0199132141783263);
        item1.add(0.0506801187398187);
        item1.add(0.104808689473925);
        item1.add(0.0700725447072635);
        item1.add(-0.0359677812752396);
        item1.add(-0.0266789028311707);
        item1.add(-0.0249926566315915);
        item1.add(-0.00259226199818282);
        item1.add(0.00371173823343597);
        item1.add(0.0403433716478807);
        // Add the first set of data to be scored
        dataItems.add(item1);
        // Create and add the second set
        JSONArray item2 = new JSONArray();
        item2.add(-0.0127796318808497);
        item2.add(-0.044641636506989);
        item2.add(0.0606183944448076);
        item2.add(0.0528581912385822);
        item2.add(0.0479653430750293);
        item2.add(0.0293746718291555);
        item2.add(-0.0176293810234174);
        item2.add(0.0343088588777263);
        item2.add(0.0702112981933102);
        item2.add(0.00720651632920303);
        dataItems.add(item2);
        obj.put("data", dataItems);

        // Make the request using the JSON document string
        sendRequest(obj.toJSONString());
    }
}
```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Call the service (Python)

This example demonstrates how to use Python to call the web service created from the Train within notebook example:

```
import requests
import json

# URL for the web service
scoring_uri = '<your web service URI>'
# If the service is authenticated, set the key or token
key = '<your key or token>'

# Two sets of data to score, so we get two results back
data = {"data":
        [
            [
                0.0199132141783263,
                0.0506801187398187,
                0.104808689473925,
                0.0700725447072635,
                -0.0359677812752396,
                -0.0266789028311707,
                -0.0249926566315915,
                -0.00259226199818282,
                0.00371173823343597,
                0.0403433716478807
            ],
            [
                -0.0127796318808497,
                -0.044641636506989,
                0.0606183944448076,
                0.0528581912385822,
                0.0479653430750293,
                0.0293746718291555,
                -0.0176293810234174,
                0.0343088588777263,
                0.0702112981933102,
                0.00720651632920303]
        ]
        }
# Convert to JSON string
input_data = json.dumps(data)

# Set the content type
headers = {'Content-Type': 'application/json'}
# If authentication is enabled, set the authorization header
headers['Authorization'] = f'Bearer {key}'

# Make the request and display the response
resp = requests.post(scoring_uri, input_data, headers=headers)
print(resp.text)
```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Consume the service from Power BI

Power BI supports consumption of Azure Machine Learning web services to enrich the data in Power BI with predictions.

To generate a web service that's supported for consumption in Power BI, the schema must support the format that's required by Power BI. Learn how to create a Power BI-supported schema.

Once the web service is deployed, it's consumable from Power BI dataflows. Learn how to consume an Azure Machine Learning web service from Power BI.

Next steps

To view a reference architecture for real-time scoring of Python and deep learning models, go to the Azure architecture center.

Collect data for models in production

3/31/2020 • 5 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

IMPORTANT

The Azure Machine Learning Monitoring SDK will be retired soon. The SDK is still appropriate for developers who currently use the SDK to monitor data drift in models. But for new customers, we recommend using the simplified data monitoring with Application Insights.

This article shows how to collect input model data from Azure Machine Learning. It also shows how to deploy the input data into an Azure Kubernetes Service (AKS) cluster and store the output data in Azure Blob storage.

Once collection is enabled, the data you collect helps you:

- Monitor data drifts as production data enters your model.
- Make better decisions about when to retrain or optimize your model.
- Retrain your model with the collected data.

What is collected and where it goes

The following data can be collected:

- Model input data from web services deployed in an AKS cluster. Voice audio, images, and video are *not* collected.
- Model predictions using production input data.

NOTE

Preaggregation and precalculations on this data are not currently part of the collection service.

The output is saved in Blob storage. Because the data is added to Blob storage, you can choose your favorite tool to run the analysis.

The path to the output data in the blob follows this syntax:

```
/modeldata/<subscriptionid>/<resourcegroup>/<workspace>/<webservice>/<model>/<version>/<designation>/<year>/
<month>/<day>/data.csv
# example: /modeldata/1a2b3c4d-5e6f-7g8h-9i10-j11k12l13m14/myresourcegrp/myWorkspace/aks-w-
collv9/best_model/10/inputs/2018/12/31/data.csv
```

NOTE

In versions of the Azure Machine Learning SDK for Python earlier than version 0.1.0a16, the `designation` argument is named `identifier`. If you developed your code with an earlier version, you need to update it accordingly.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin.
- A AzureMachine Learning workspace, a local directory containing your scripts, and the Azure Machine Learning SDK for Python must be installed. To learn how to install them, see How to configure a development environment.
- You need a trained machine-learning model to be deployed to AKS. If you don't have a model, see the Train image classification model tutorial.
- You need an AKS cluster. For information on how to create one and deploy to it, see How to deploy and where.
- Set up your environment and install the Azure Machine Learning Monitoring SDK.

Enable data collection

You can enable data collection regardless of the model you deploy through Azure Machine Learning or other tools.

To enable data collection, you need to:

1. Open the scoring file.
2. Add the following code at the top of the file:

```
from azureml.monitoring import ModelDataCollector
```

3. Declare your data collection variables in your `init` function:

```
global inputs_dc, prediction_dc
inputs_dc = ModelDataCollector("best_model", designation="inputs", feature_names=["feat1", "feat2",
"feat3". "feat4", "feat5", "feat6"])
prediction_dc = ModelDataCollector("best_model", designation="predictions", feature_names=
["prediction1", "prediction2"])
```

   CorrelationId is an optional parameter. You don't need to use it if your model doesn't require it. Use of *CorrelationId* does help you more easily map with other data, such as *LoanNumber* or *CustomerId*.

   The *Identifier* parameter is later used for building the folder structure in your blob. You can use it to differentiate raw data from processed data.

4. Add the following lines of code to the `run(input_df)` function:

```
data = np.array(data)
result = model.predict(data)
inputs_dc.collect(data) #this call is saving our input data into Azure Blob
prediction_dc.collect(result) #this call is saving our input data into Azure Blob
```

5. Data collection is *not* automatically set to **true** when you deploy a service in AKS. Update your configuration file, as in the following example:

```
aks_config = AksWebservice.deploy_configuration(collect_model_data=True)
```

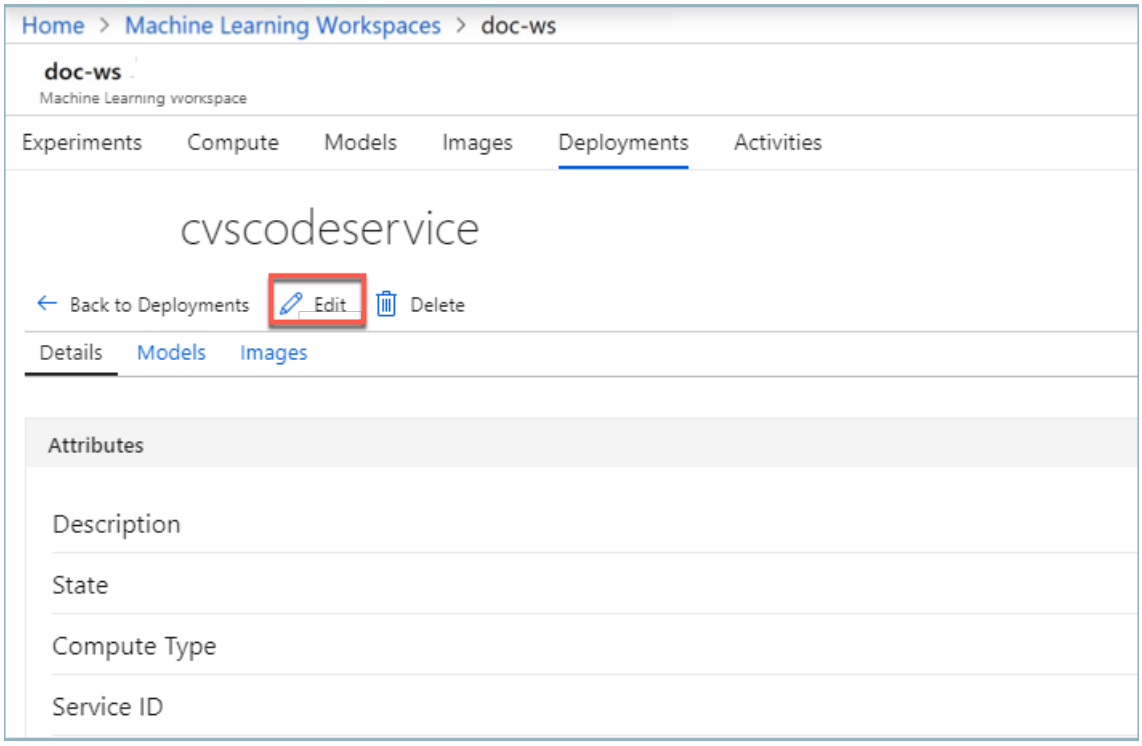
   You can also enable Application Insights for service monitoring by changing this configuration:

```
aks_config = AksWebservice.deploy_configuration(collect_model_data=True, enable_app_insights=True)
```

6. To create a new image and deploy the machine learning model, see How to deploy and where.

If you already have a service with the dependencies installed in your environment file and scoring file, enable data collection by following these steps:

1. Go to Azure Machine Learning.
2. Open your workspace.
3. Select **Deployments** > **Select service** > **Edit**.



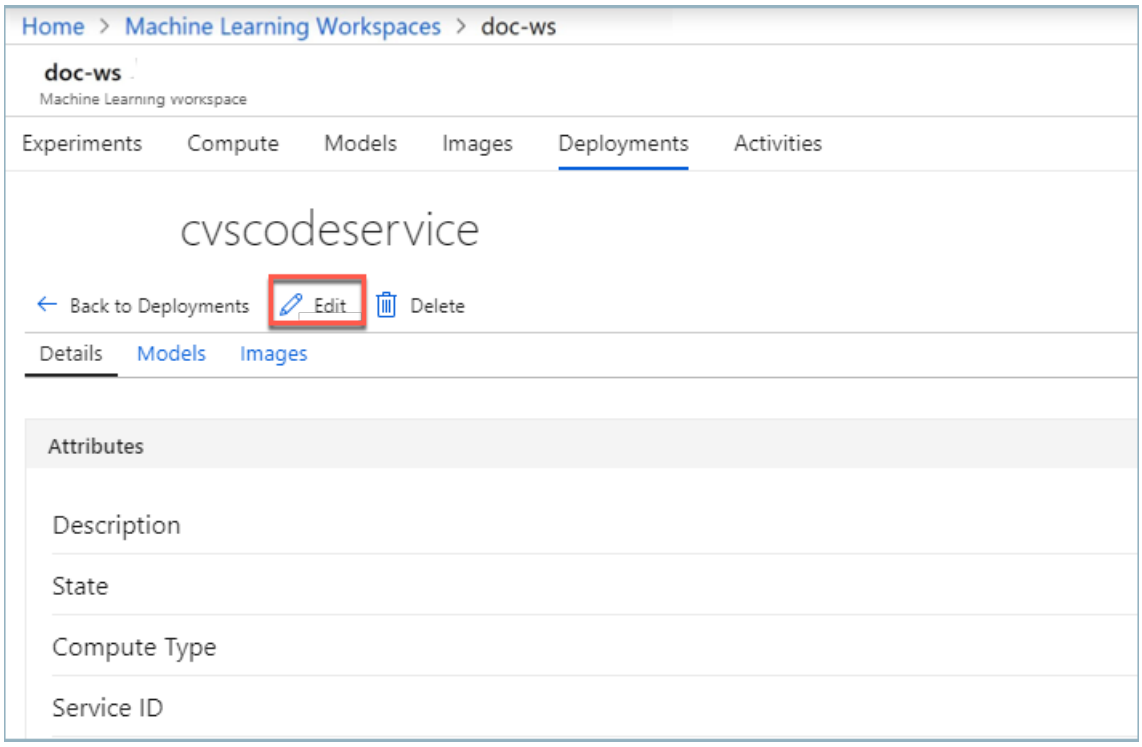
4. In **Advanced Settings**, select **Enable Application Insights diagnostics and data collection**.
5. Select **Update** to apply the changes.

Disable data collection

You can stop collecting data at any time. Use Python code or Azure Machine Learning to disable data collection.

Option 1 - Disable data collection in Azure Machine Learning

1. Sign in to Azure Machine Learning.
2. Open your workspace.
3. Select **Deployments** > **Select service** > **Edit**.



4. In **Advanced Settings**, clear **Enable Application Insights diagnostics and data collection**.

5. Select **Update** to apply the change.

You can also access these settings in your workspace in Azure Machine Learning.

Option 2 - Use Python to disable data collection

```
## replace <service_name> with the name of the web service
<service_name>.update(collect_model_data=False)
```

Validate and analyze your data

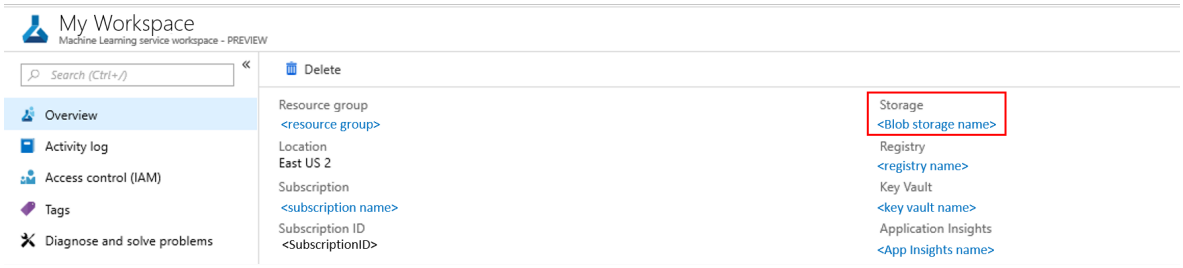
You can choose a tool of your preference to analyze the data collected in your Blob storage.

Quickly access your blob data

1. Sign in to Azure Machine Learning.

2. Open your workspace.

3. Select **Storage**.

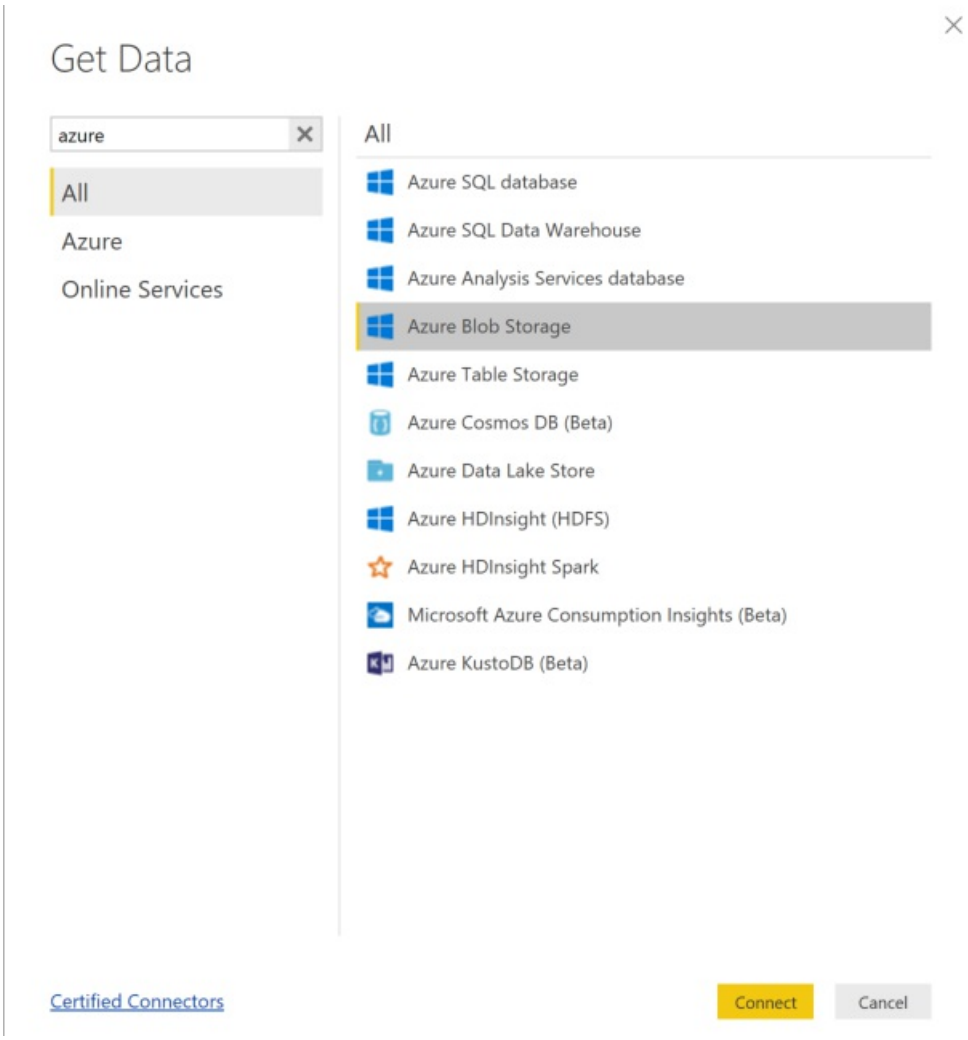


4. Follow the path to the blob's output data with this syntax:

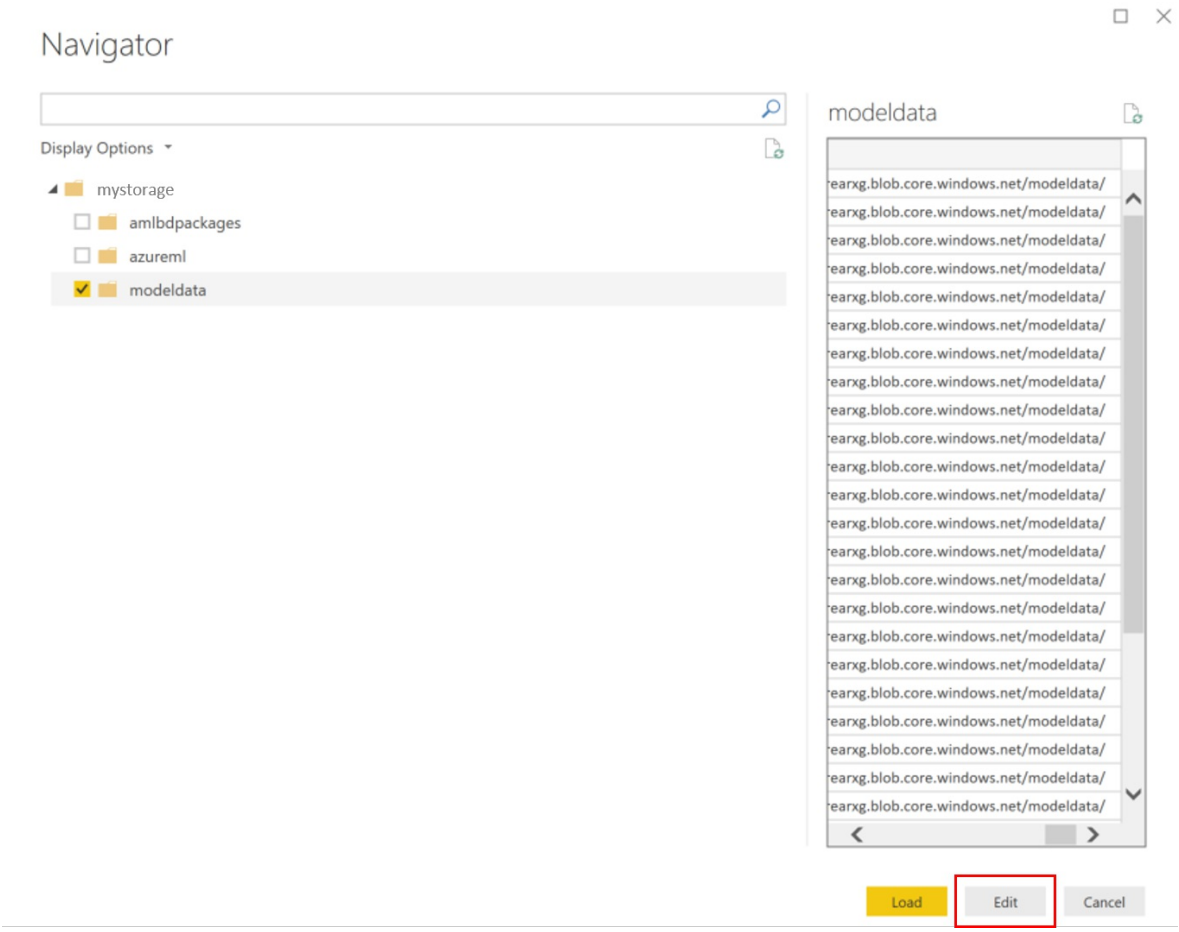
```
/modeldata/<subscriptionid>/<resourcegroup>/<workspace>/<webservice>/<model>/<version>/<designation>/<
year>/<month>/<day>/data.csv
# example: /modeldata/1a2b3c4d-5e6f-7g8h-9i10-j11k12l13m14/myresourcegrp/myWorkspace/aks-w-
collv9/best_model/10/inputs/2018/12/31/data.csv
```

Analyze model data using Power BI

1. Download and open Power BI Desktop.
2. Select **Get Data** and select **Azure Blob Storage**.

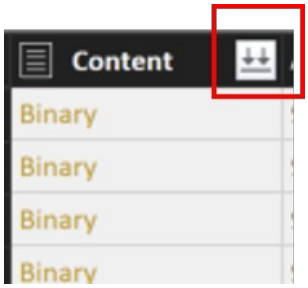


3. Add your storage account name and enter your storage key. You can find this information by selecting **Settings** > **Access keys** in your blob.
4. Select the **model data** container and select **Edit**.

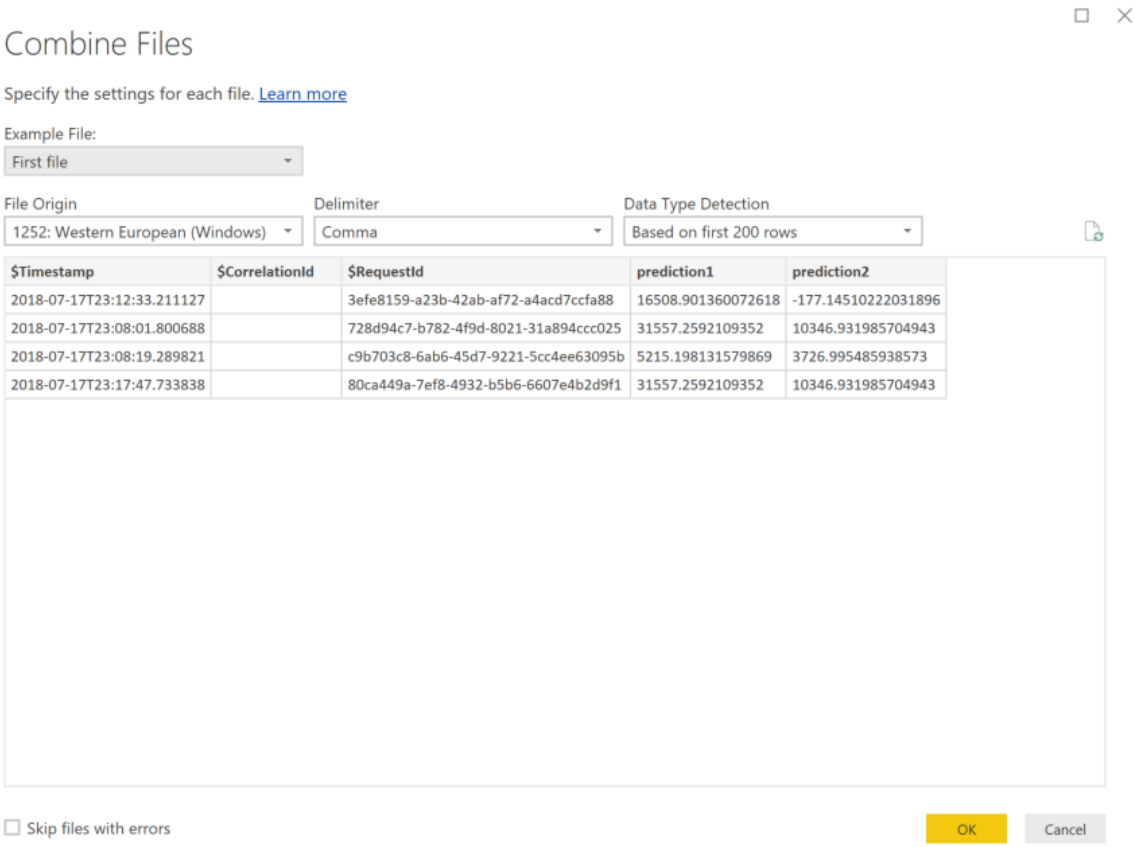


5. In the query editor, click under the **Name** column and add your storage account.
6. Enter your model path into the filter. If you want to look only into files from a specific year or month, just expand the filter path. For example, to look only into March data, use this filter path:

   /modeldata/<subscriptionid>/<resourcegroupname>/<workspacename>/<webservicename>/<modelname>/<modelversion>/<designation>/<year>/3
7. Filter the data that is relevant to you based on **Name** values. If you stored predictions and inputs, you need to create a query for each.
8. Select the downward double arrows next to the **Content** column heading to combine the files.



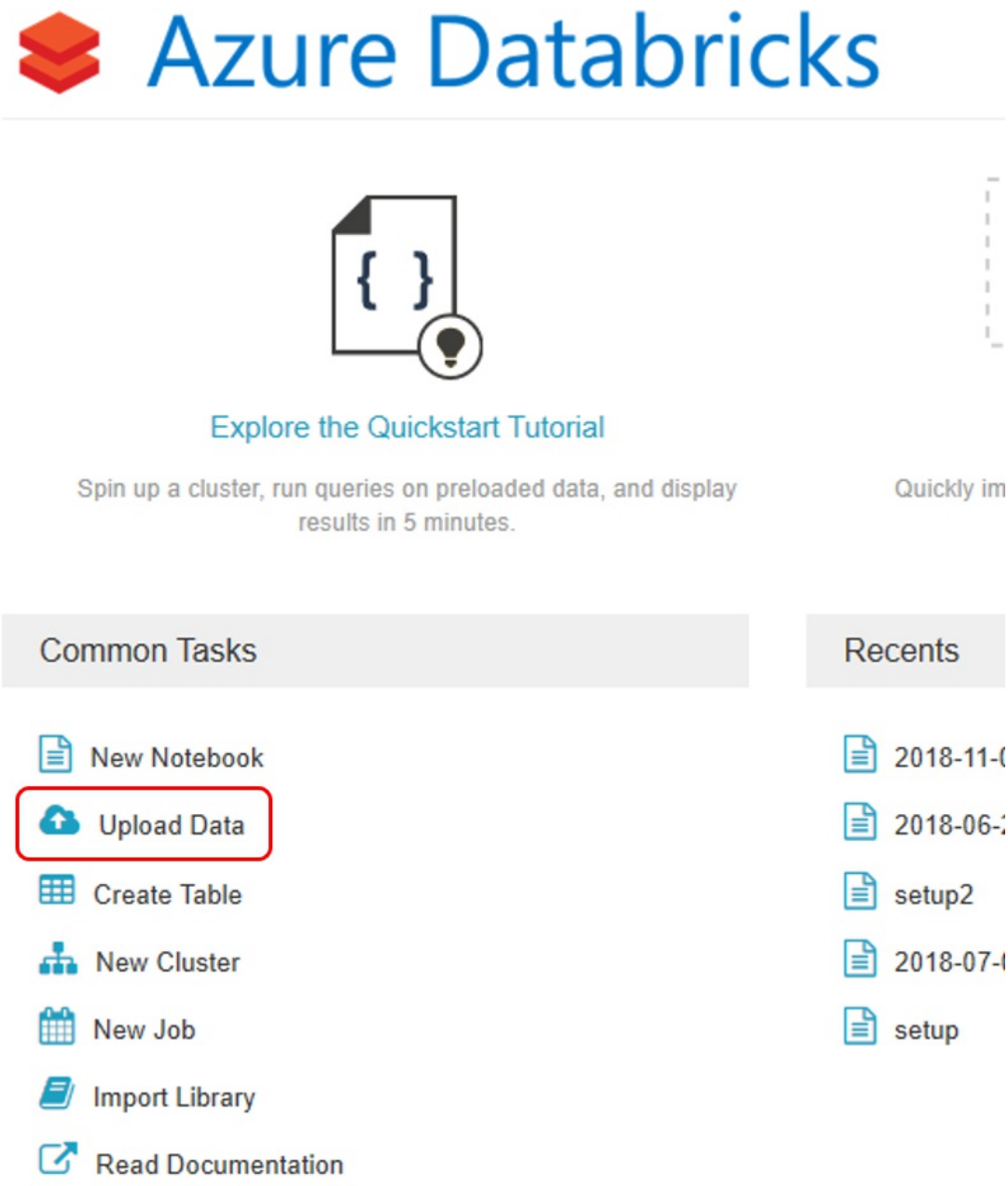
9. Select **OK**. The data preloads.



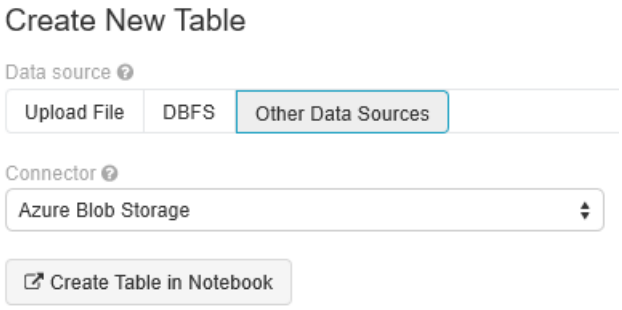
10. Select **Close and Apply**.
11. If you added inputs and predictions, your tables are automatically ordered by **RequestId** values.
12. Start building your custom reports on your model data.

Analyze model data using Azure Databricks

1. Create an Azure Databricks workspace.
2. Go to your Databricks workspace.
3. In your Databricks workspace, select **Upload Data**.

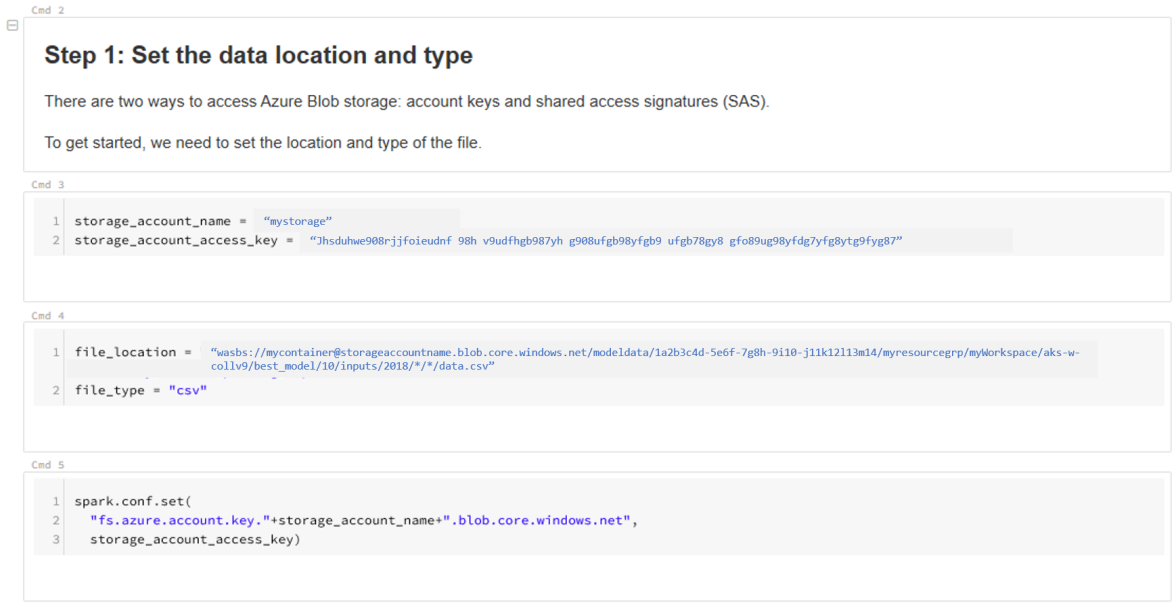


4. Select **Create New Table** and select **Other Data Sources** > **Azure Blob Storage** > **Create Table in Notebook**.



5. Update the location of your data. Here is an example:

```
file_location = "wasbs://mycontainer@storageaccountname.blob.core.windows.net/modeldata/1a2b3c4d-5e6f-
7g8h-9i10-j11k12l13m14/myresourcegrp/myWorkspace/aks-w-collv9/best_model/10/inputs/2018/*/*/data.csv"
file_type = "csv"
```



6. Follow the steps on the template to view and analyze your data.

Detect data drift (preview) on models deployed to Azure Kubernetes Service (AKS)

12/27/2019 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this article, you learn how to monitor for data drift between the training dataset and inference data of a deployed model. In the context of machine learning, trained machine learning models may experience degraded prediction performance because of drift. With Azure Machine Learning, you can monitor data drift and the service can send an email alert to you when drift is detected.

What is data drift?

In the context of machine learning, data drift is the change in model input data that leads to model performance degradation. It is one of the top reasons where model accuracy degrades over time, thus monitoring data drift helps detect model performance issues.

What can I monitor?

With Azure Machine Learning, you can monitor the inputs to a model deployed on AKS and compare this data to the training dataset for the model. At regular intervals, the inference data is snapshot and profiled, then computed against the baseline dataset to produce a data drift analysis that:

- Measures the magnitude of data drift, called the drift coefficient.
- Measures the data drift contribution by feature, indicating which features caused data drift.
- Measures distance metrics. Currently Wasserstein and Energy Distance are computed.
- Measures distributions of features. Currently kernel density estimation and histograms.
- Send alerts to data drift by email.

> **NOTE**
> This service is in (preview) and limited in configuration options. Please see our API Documentation and Release Notes for details and updates.

How data drift is monitored in Azure Machine Learning

Using Azure Machine Learning, data drift is monitored through datasets or deployments. To monitor for data drift, a baseline dataset - usually the training dataset for a model - is specified. A second dataset - usually model input data gathered from a deployment - is tested against the baseline dataset. Both datasets are profiled and input to the data drift monitoring service. A machine learning model is trained to detect differences between the two datasets. The model's performance is converted to the drift coefficient, which measures the magnitude of drift between the two datasets. Using model interpretability, the features that contribute to the drift coefficient are computed. From the dataset profile, statistical information about each feature is tracked.

Prerequisites

- An Azure subscription. If you don't have one, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- The Azure Machine Learning SDK for Python installed. Use the instructions at Azure Machine Learning SDK

to do the following:

  - Create a Miniconda environment
  - Install the Azure Machine Learning SDK for Python
- An Azure Machine Learning workspace.
- A workspace configuration file.
- Install the data drift SDK using the following command:

```
pip install azureml-datadrift
```

- Create a dataset from your model's training data.
- Specify the training dataset when registering the model. The following example demonstrates using the `datasets` parameter to specify the training dataset:

```
model = Model.register(model_path=model_file,
                       model_name=model_name,
                       workspace=ws,
                       datasets=datasets)

print(model_name, image_name, service_name, model)
```

- Enable model data collection to collect data from the AKS deployment of the model and confirm data is being collected in the `modeldata` blob container.

Configure data drift

To configure data drift for your experiment, import dependencies as seen in the following Python example.

This example demonstrates configuring the `DataDriftDetector` object:

```
# Import Azure ML packages
from azureml.core import Experiment, Run, RunDetails
from azureml.datadrift import DataDriftDetector, AlertConfiguration

# if email address is specified, setup AlertConfiguration
alert_config = AlertConfiguration('your_email@contoso.com')

# create a new DataDriftDetector object
datadrift = DataDriftDetector.create(ws, model.name, model.version, services, frequency="Day",
alert_config=alert_config)

print('Details of Datadrift Object:\n{}'.format(datadrift))
```

Submit a DataDriftDetector run

With the `DataDriftDetector` object configured, you can submit a data drift run on a given date for the model. As part of the run, enable DataDriftDetector alerts by setting the `drift_threshold` parameter. If the datadrift_coefficient is above the given `drift_threshold`, an email is sent.

```
# adhoc run today
target_date = datetime.today()

# create a new compute - creates datadrift-server
run = datadrift.run(target_date, services, feature_list=feature_list, create_compute_target=True)

# or specify existing compute cluster
run = datadrift.run(target_date, services, feature_list=feature_list, compute_target='cpu-cluster')

# show details of the data drift run
exp = Experiment(ws, datadrift._id)
dd_run = Run(experiment=exp, run_id=run.id)
RunDetails(dd_run).show()
```

Visualize drift metrics

After you submit your DataDriftDetector run, you are able to see the drift metrics that are saved in each run iteration for a data drift task:

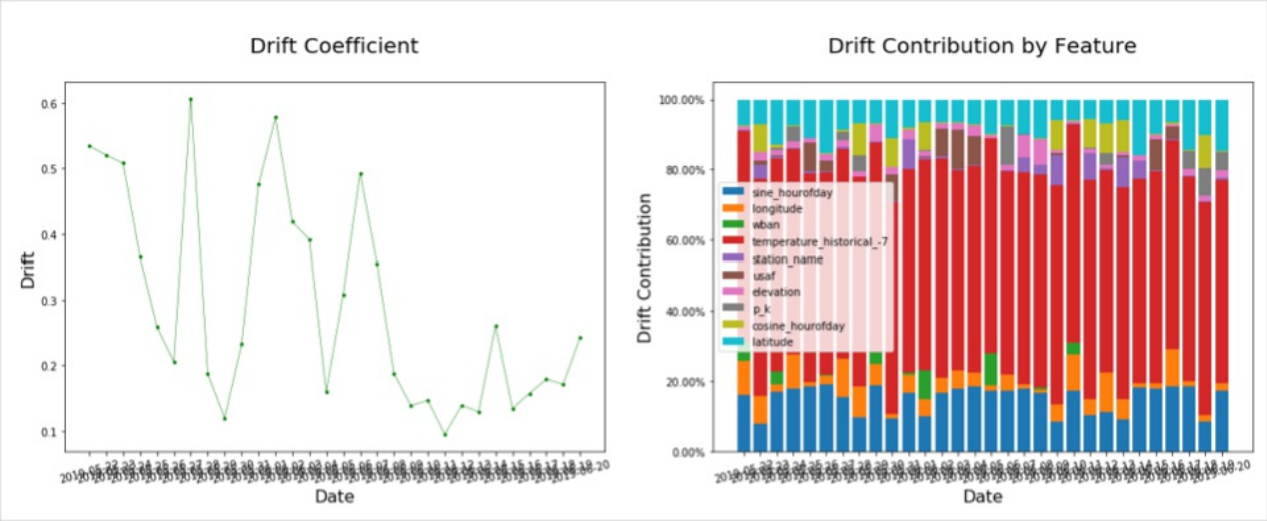
METRIC	DESCRIPTION
wasserstein_distance	Statistical distance defined for one-dimensional numerical distribution.
energy_distance	Statistical distance defined for one-dimensional numerical distribution.
datadrift_coefficient	Calculated similarly as Matthew's correlation coefficient, but this output is a real number ranging from 0 to 1. In the context of drift, 0 indicates no drift and 1 indicates maximum drift.
datadrift_contribution	Feature importance of features contributing to drift.

There are multiple ways to view drift metrics:

- Use the `RunDetails` Jupyter widget.
- Use the `get_metrics()` function on any `datadrift` run object.
- View the metrics from the **Models** section of your workspace in Azure Machine Learning studio.

The following Python example demonstrates how to plot relevant data drift metrics. You can use the returned metrics to build custom visualizations:

```
# start and end are datetime objects
drift_metrics = datadrift.get_output(start_time=start, end_time=end)

# Show all data drift result figures, one per service.
# If setting with_details is False (by default), only the data drift magnitude will be shown; if it's True,
all details will be shown.
drift_figures = datadrift.show(with_details=True)
```

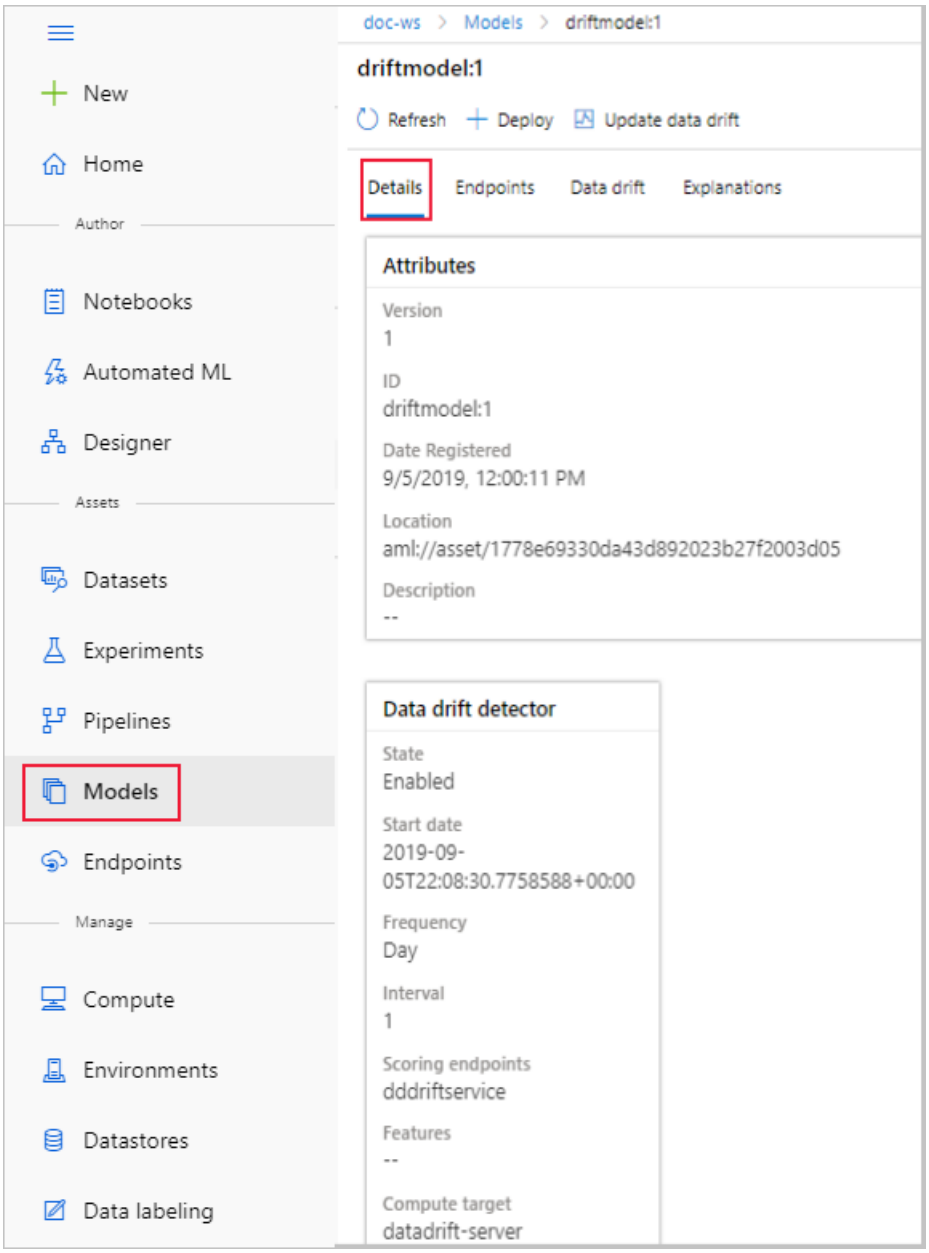


Schedule data drift scans

When you enable data drift detection, a DataDriftDetector is run at the specified, scheduled frequency. If the datadrift_coefficient reaches the given `drift_threshold`, an email is sent with each scheduled run.

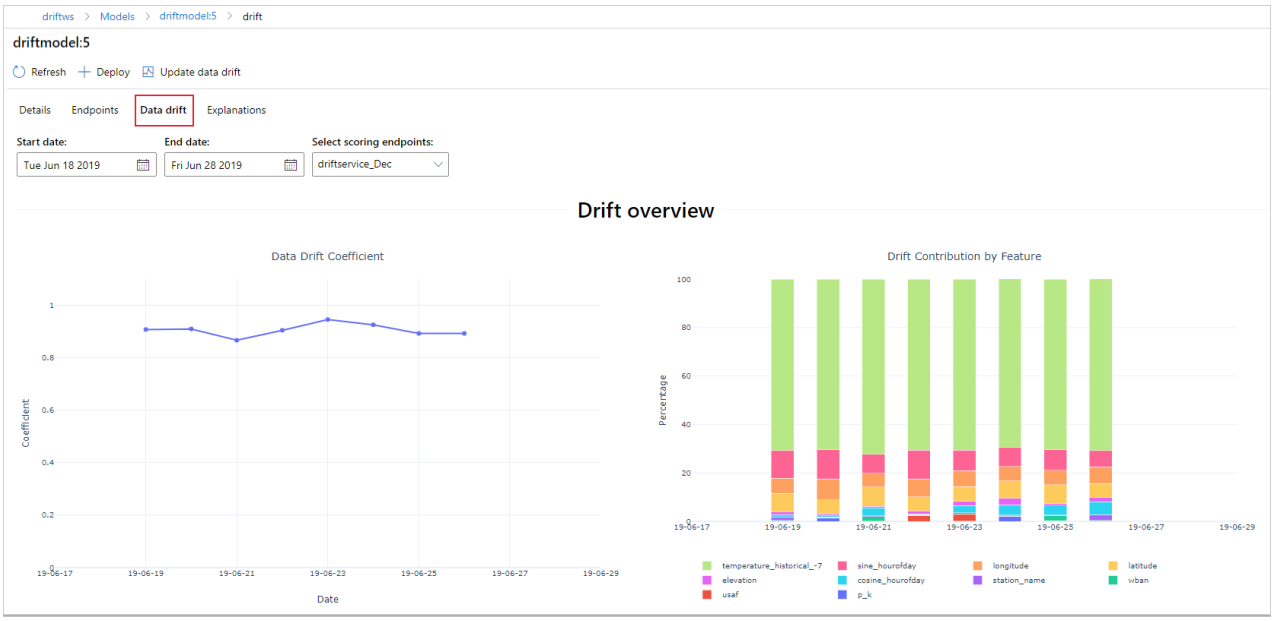
```
datadrift.enable_schedule()
datadrift.disable_schedule()
```

The configuration of the data drift detector can be seen under **Models** in the **Details** tab in your workspace on the Azure Machine Learning studio.



View results in your Azure Machine Learning studio

To view results in your workspace in Azure Machine Learning studio, navigate to the model page. On the details tab of the model, the data drift configuration is shown. A **Data drift** tab is now available visualizing the data drift metrics.



Receiving drift alerts

By setting the drift coefficient alerting threshold and providing an email address, an Azure Monitor email alert is automatically sent whenever the drift coefficient is above the threshold.

In order for you to set up custom alerts and actions, all data drift metrics are stored in the Application Insights resource that was created along with the Azure Machine Learning workspace. You can follow the link in the email alert to the Application Insights query.

Microsoft Azure

Your Azure Monitor alert was triggered

We are notifying you because there are 1 counts of "DataDrift-Alert-Rule-3fb3d9bc-24f9-4675-b1b7-3e94c11352ac".

Essentials

Name	DataDrift-Alert-Rule-3fb3d9bc-24f9-4675-b1b7-3e94c11352ac
Severity	3
Resource	publicdeinsights6e499d8d
Search interval start time	June 11, 2019 16:59:39 UTC
Search interval duration	5 min
Search query	`customMetrics | where (name=='datadrift_coefficient' or name=='ds_mcc_test') and value > 0.2 and customDimensions.datadrift_id == '3fb3d9bc-24f9-4675-b1b7-3e94c11352ac' | summarize AggregatedValue=count() by bin_at(timestamp, 1m, datetime(2019-06-11T17:04:39.0000000))`
Search results	1 result(s)
Description	Alert Rule is triggered every time data drift is detected.

Retrain your model after drift

When data drift negatively impacts the performance of your deployed model, it is time to retrain the model. To do so, proceed with the following steps.

- Investigate the collected data and prepare data to train the new model.
- Split it into train/test data.
- Train the model again using the new data.
- Evaluate performance of the newly generated model.
- Deploy new model if performance is better than the production model.

Next steps

- For a full example of using data drift, see the Azure ML data drift notebook. This Jupyter Notebook demonstrates using an Azure Open Dataset to train a model to predict the weather, deploy it to AKS, and monitor for data drift.
- Detect data drift with dataset monitors.
- We would greatly appreciate your questions, comments, or suggestions as data drift moves toward general availability. Use the product feedback button below!

Monitor and collect data from ML web service endpoints

3/12/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to collect data from and monitor models deployed to web service endpoints in Azure Kubernetes Service (AKS) or Azure Container Instances (ACI) by enabling Azure Application Insights via

- Azure Machine Learning Python SDK
- Azure Machine Learning studio at https://ml.azure.com

In addition to collecting an endpoint's output data and response, you can monitor:

- Request rates, response times, and failure rates
- Dependency rates, response times, and failure rates
- Exceptions

Learn more about Azure Application Insights.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today
- An Azure Machine Learning workspace, a local directory that contains your scripts, and the Azure Machine Learning SDK for Python installed. To learn how to get these prerequisites, see How to configure a development environment
- A trained machine learning model to be deployed to Azure Kubernetes Service (AKS) or Azure Container Instance (ACI). If you don't have one, see the Train image classification model tutorial

Web service metadata and response data

> **IMPORTANT**
> Azure Application Insights only logs payloads of up to 64kb. If this limit is reached then only the most recent outputs of the model are logged.

The metadata and response to the service - corresponding to the web service metadata and the model's predictions - are logged to the Azure Application Insights traces under the message `"model_data_collection"`. You can query Azure Application Insights directly to access this data, or set up a continuous export to a storage account for longer retention or further processing. Model data can then be used in the Azure Machine Learning to set up labeling, retraining, explainability, data analysis, or other use.

Use Python SDK to configure

Update a deployed service

1. Identify the service in your workspace. The value for `ws` is the name of your workspace

```
from azureml.core.webservice import Webservice
aks_service= Webservice(ws, "my-service-name")
```

2. Update your service and enable Azure Application Insights

```
aks_service.update(enable_app_insights=True)
```

Log custom traces in your service

If you want to log custom traces, follow the standard deployment process for AKS or ACI in the How to deploy and where document. Then use the following steps:

1. Update the scoring file by adding print statements

```
print ("model initialized" + time.strftime("%H:%M:%S"))
```

2. Update the service configuration

```
config = Webservice.deploy_configuration(enable_app_insights=True)
```

3. Build an image and deploy it on AKS or ACI.

Disable tracking in Python

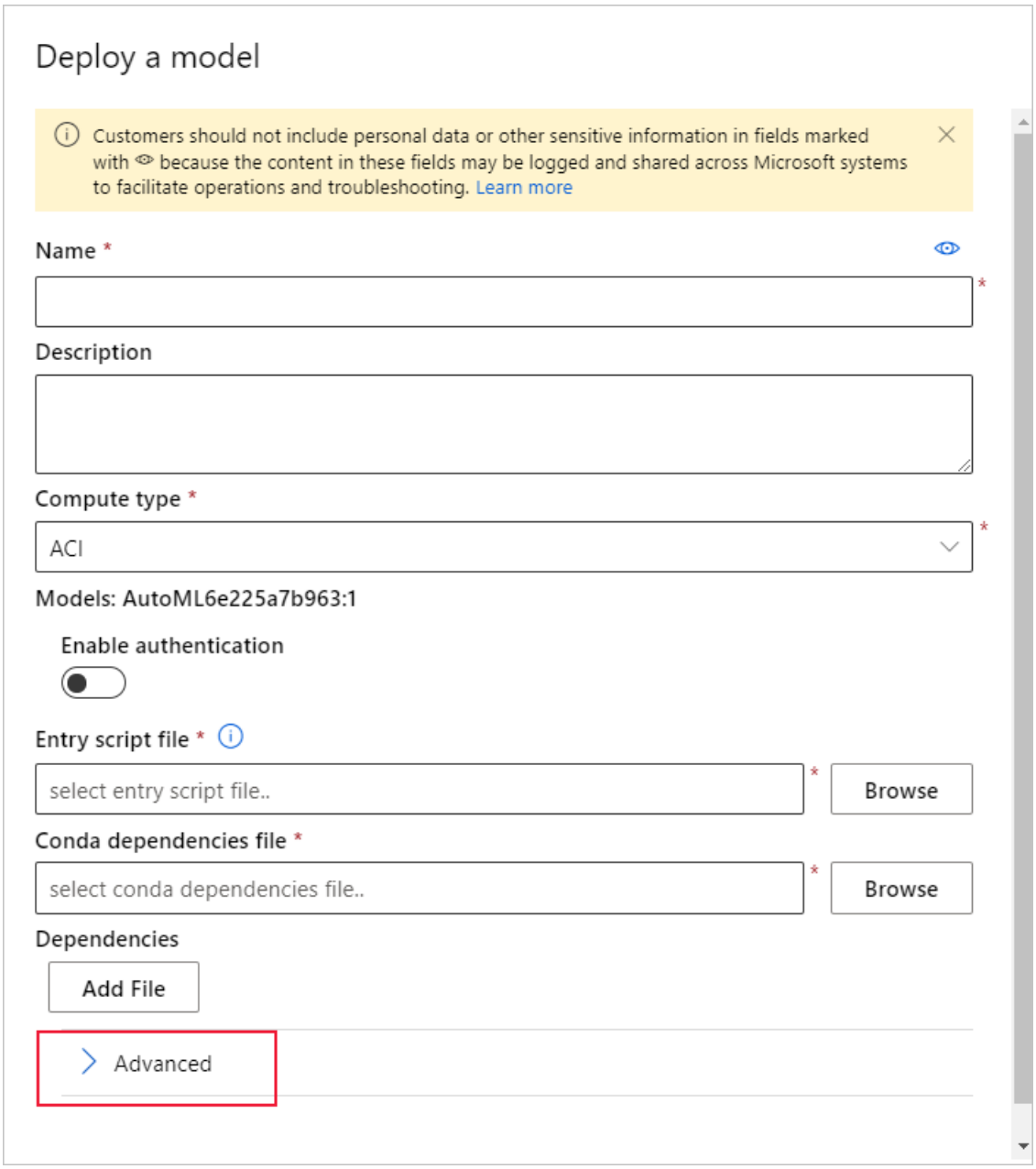
To disable Azure Application Insights, use the following code:

```
## replace <service_name> with the name of the web service
<service_name>.update(enable_app_insights=False)
```

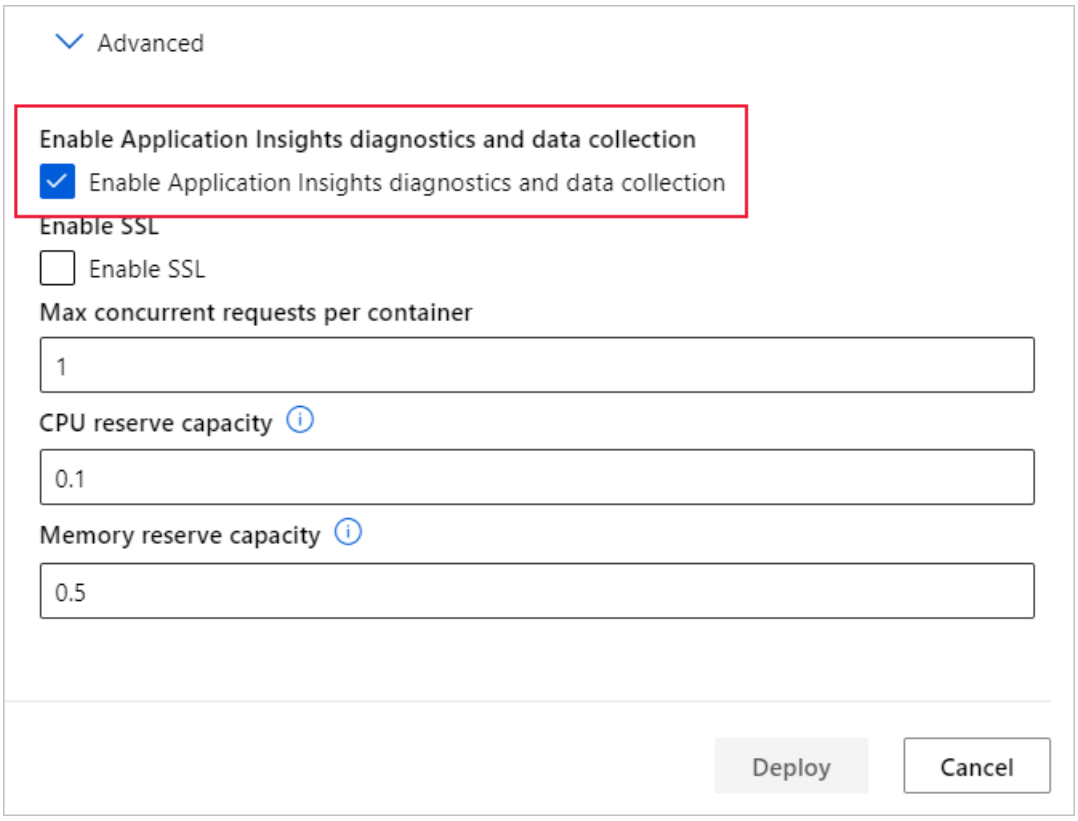
Use Azure Machine Learning studio to configure

You can also enable Azure Application Insights from Azure Machine Learning studio when you're ready to deploy your model with these steps.

1. Sign in to your workspace at https://ml.azure.com/
2. Go to **Models** and select which model you want to deploy
3. Select **+Deploy**
4. Populate the **Deploy model** form
5. Expand the **Advanced** menu



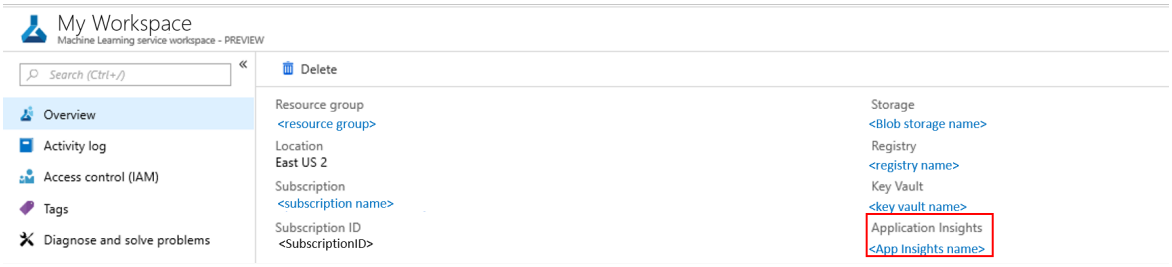
6. Select **Enable Application Insights diagnostics and data collection**



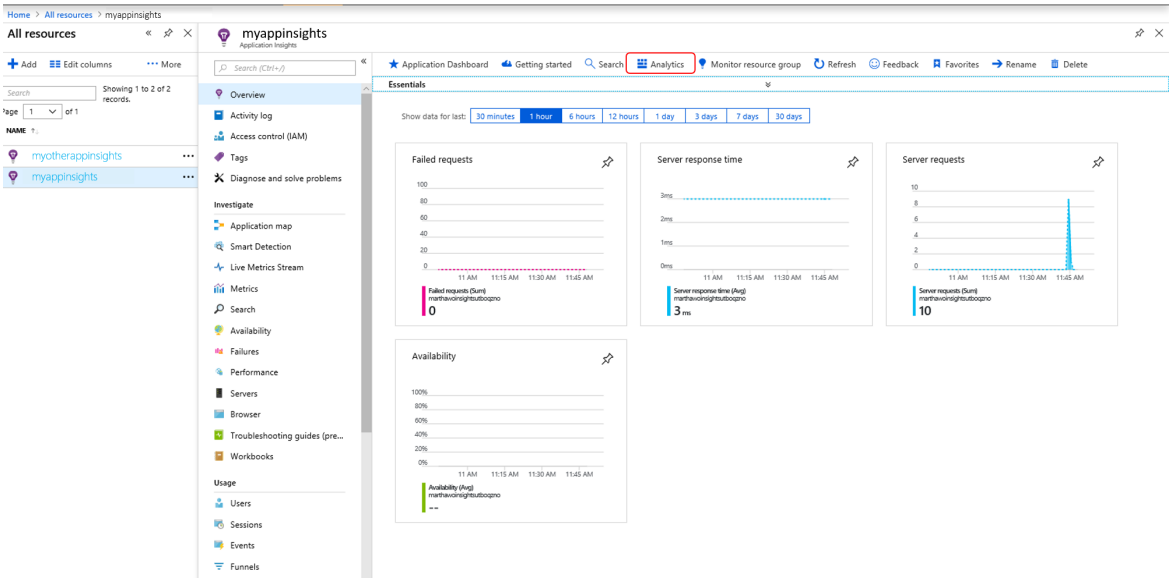
Evaluate data

Your service's data is stored in your Azure Application Insights account, within the same resource group as Azure Machine Learning. To view it:

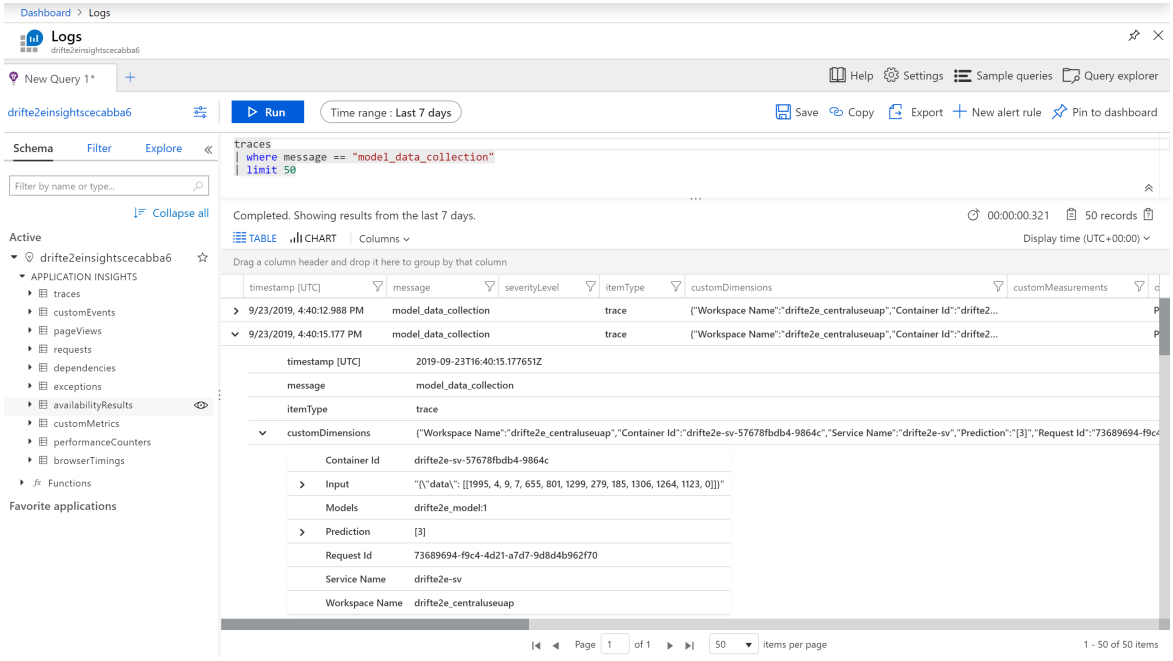
1. Go to your Azure Machine Learning workspace in the Azure portal and click on the Application Insights link



2. Select the **Overview** tab to see a basic set of metrics for your service

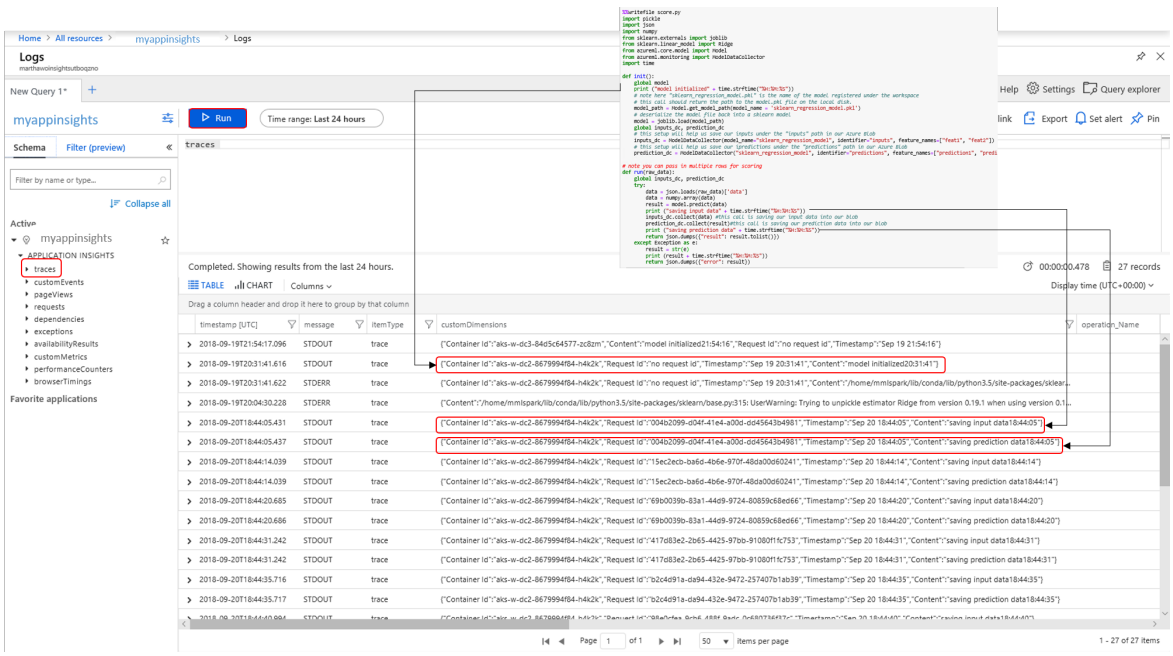


3. To look into your web service request metadata and response, select the **requests** table in the **Logs (Analytics)** section and select **Run** to view requests



4. To look into your custom traces, select **Analytics**

5. In the schema section, select **Traces**. Then select **Run** to run your query. Data should appear in a table format and should map to your custom calls in your scoring file



To learn more about how to use Azure Application Insights, see What is Application Insights?.

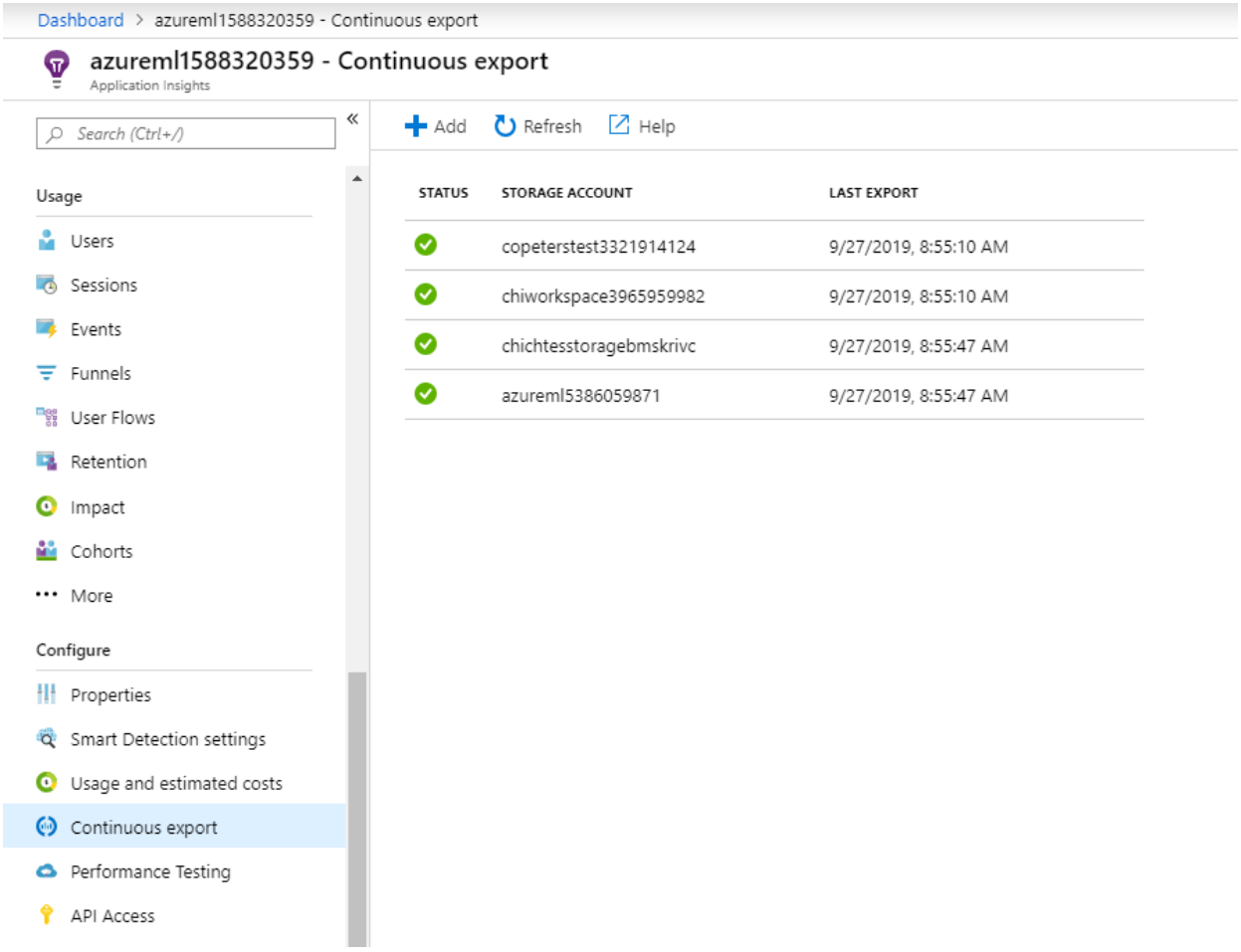
Export data for further processing and longer retention

> **IMPORTANT**
>
> Azure Application Insights only supports exports to blob storage. Additional limits of this export capability are listed in Export telemetry from App Insights.

You can use Azure Application Insights' continuous export to send messages to a supported storage account, where a longer retention can be set. The `"model_data_collection"` messages are stored in JSON format and can

be easily parsed to extract model data.

Azure Data Factory, Azure ML Pipelines, or other data processing tools can be used to transform the data as needed. When you have transformed the data, you can then register it with the Azure Machine Learning workspace as a dataset. To do so, see How to create and register datasets.



Example notebook

The enable-app-insights-in-production-service.ipynb notebook demonstrates concepts in this article.

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Next steps

- See how to deploy a model to an Azure Kubernetes Service cluster or how to deploy a model to Azure Container Instances to deploy your models to web service endpoints, and enable Azure Application Insights to leverage data collection and endpoint monitoring
- See MLOps: Manage, deploy, and monitor models with Azure Machine Learning to learn more about leveraging data collected from models in production. Such data can help to continually improve your machine learning process

Create and run machine learning pipelines with Azure Machine Learning SDK

4/24/2020 • 14 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to create, publish, run, and track a machine learning pipeline by using the Azure Machine Learning SDK. Use **ML pipelines** to create a workflow that stitches together various ML phases, and then publish that pipeline into your Azure Machine Learning workspace to access later or share with others. ML pipelines are ideal for batch scoring scenarios, using various computes, reusing steps instead of rerunning them, as well as sharing ML workflows with others.

While you can use a different kind of pipeline called an Azure Pipeline for CI/CD automation of ML tasks, that type of pipeline is never stored inside your workspace. Compare these different pipelines.

Each phase of an ML pipeline, such as data preparation and model training, can include one or more steps.

The ML pipelines you create are visible to the members of your Azure Machine Learning workspace.

ML pipelines use remote compute targets for computation and the storage of the intermediate and final data associated with that pipeline. They can read and write data to and from supported Azure Storage locations.

If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning.

Prerequisites

- Create an Azure Machine Learning workspace to hold all your pipeline resources.
- Configure your development environment to install the Azure Machine Learning SDK, or use an Azure Machine Learning compute instance (preview) with the SDK already installed.

Start by attaching your workspace:

```
import azureml.core
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()
```

Set up machine learning resources

Create the resources required to run an ML pipeline:

- Set up a datastore used to access the data needed in the pipeline steps.
- Configure a `Dataset` object to point to persistent data that lives in, or is accessible in, a datastore. Configure a `PipelineData` object for temporary data passed between pipeline steps.
- Set up the compute targets on which your pipeline steps will run.

Set up a datastore

A datastore stores the data for the pipeline to access. Each workspace has a default datastore. You can register additional datastores.

When you create your workspace, Azure Files and Azure Blob storage are attached to the workspace. A default datastore is registered to connect to the Azure Blob storage. To learn more, see Deciding when to use Azure Files, Azure Blobs, or Azure Disks.

```
# Default datastore
def_data_store = ws.get_default_datastore()

# Get the blob storage associated with the workspace
def_blob_store = Datastore(ws, "workspaceblobstore")

# Get file storage associated with the workspace
def_file_store = Datastore(ws, "workspacefilestore")
```

Upload data files or directories to the datastore for them to be accessible from your pipelines. This example uses the Blob storage as the datastore:

```
def_blob_store.upload_files(
    ["iris.csv"],
    target_path="train-dataset",
    overwrite=True)
```

A pipeline consists of one or more steps. A step is a unit run on a compute target. Steps might consume data sources and produce "intermediate" data. A step can create data such as a model, a directory with model and dependent files, or temporary data. This data is then available for other steps later in the pipeline.

To learn more about connecting your pipeline to your data, see the articles How to Access Data and How to Register Datasets.

Configure data using `Dataset` and `PipelineData` objects

You just created a data source that can be referenced in a pipeline as an input to a step. The preferred way to provide data to a pipeline is a Dataset object. The `Dataset` object points to data that lives in or is accessible from a datastore or at a Web URL. The `Dataset` class is abstract, so you will create an instance of either a `FileDataset` (referring to one or more files) or a `TabularDataset` that's created by from one or more files with delimited columns of data.

`Dataset` objects support versioning, diffs, and summary statistics. `Dataset`s are lazily evaluated (like Python generators) and it's efficient to subset them by splitting or filtering.

You create a `Dataset` using methods like from_file or from_delimited_files.

```
from azureml.core import Dataset

iris_tabular_dataset = Dataset.Tabular.from_delimited_files([(def_blob_store, 'train-dataset/iris.csv')])
```

Intermediate data (or output of a step) is represented by a PipelineData object. `output_data1` is produced as the output of a step, and used as the input of one or more future steps. `PipelineData` introduces a data dependency between steps, and creates an implicit execution order in the pipeline. This object will be used later when creating pipeline steps.

```
from azureml.pipeline.core import PipelineData

output_data1 = PipelineData(
    "output_data1",
    datastore=def_blob_store,
    output_name="output_data1")
```

More details and sample code for working with datasets and pipeline data can be found in Moving data into and between ML pipeline steps (Python).

Set up a compute target

In Azure Machine Learning, the term **compute** (or **compute target**) refers to the machines or clusters that perform the computational steps in your machine learning pipeline. See compute targets for model training for a full list of compute targets and how to create and attach them to your workspace. The process for creating and or attaching a compute target is the same regardless of whether you are training a model or running a pipeline step. After you create and attach your compute target, use the `ComputeTarget` object in your pipeline step.

> **IMPORTANT**
> Performing management operations on compute targets is not supported from inside remote jobs. Since machine learning pipelines are submitted as a remote job, do not use management operations on compute targets from inside the pipeline.

Below are examples of creating and attaching compute targets for:

- Azure Machine Learning Compute
- Azure Databricks
- Azure Data Lake Analytics

Azure Machine Learning compute

You can create an Azure Machine Learning compute for running your steps.

```
from azureml.core.compute import ComputeTarget, AmlCompute

compute_name = "aml-compute"
vm_size = "STANDARD_NC6"
if compute_name in ws.compute_targets:
    compute_target = ws.compute_targets[compute_name]
    if compute_target and type(compute_target) is AmlCompute:
        print('Found compute target: ' + compute_name)
else:
    print('Creating a new compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size=vm_size,  # STANDARD_NC6 is GPU-
enabled
                                                                 min_nodes=0,
                                                                 max_nodes=4)
    # create the compute target
    compute_target = ComputeTarget.create(
        ws, compute_name, provisioning_config)

    # Can poll for a minimum number of nodes and for a specific timeout.
    # If no min node count is provided it will use the scale settings for the cluster
    compute_target.wait_for_completion(
        show_output=True, min_node_count=None, timeout_in_minutes=20)

    # For a more detailed view of current cluster status, use the 'status' property
    print(compute_target.status.serialize())
```

Azure Databricks

Azure Databricks is an Apache Spark-based environment in the Azure cloud. It can be used as a compute target with an Azure Machine Learning pipeline.

Create an Azure Databricks workspace before using it. To create a workspace resource, see the Run a Spark job on Azure Databricks document.

To attach Azure Databricks as a compute target, provide the following information:

- **Databricks compute name**: The name you want to assign to this compute resource.
- **Databricks workspace name**: The name of the Azure Databricks workspace.
- **Databricks access token**: The access token used to authenticate to Azure Databricks. To generate an access token, see the Authentication document.

The following code demonstrates how to attach Azure Databricks as a compute target with the Azure Machine Learning SDK (**The Databricks workspace need to be present in the same subscription as your AML workspace**):

```
import os
from azureml.core.compute import ComputeTarget, DatabricksCompute
from azureml.exceptions import ComputeTargetException

databricks_compute_name = os.environ.get(
    "AML_DATABRICKS_COMPUTE_NAME", "<databricks_compute_name>")
databricks_workspace_name = os.environ.get(
    "AML_DATABRICKS_WORKSPACE", "<databricks_workspace_name>")
databricks_resource_group = os.environ.get(
    "AML_DATABRICKS_RESOURCE_GROUP", "<databricks_resource_group>")
databricks_access_token = os.environ.get(
    "AML_DATABRICKS_ACCESS_TOKEN", "<databricks_access_token>")

try:
    databricks_compute = ComputeTarget(
        workspace=ws, name=databricks_compute_name)
    print('Compute target already exists')
except ComputeTargetException:
    print('compute not found')
    print('databricks_compute_name {}'.format(databricks_compute_name))
    print('databricks_workspace_name {}'.format(databricks_workspace_name))
    print('databricks_access_token {}'.format(databricks_access_token))

    # Create attach config
    attach_config = DatabricksCompute.attach_configuration(resource_group=databricks_resource_group,
                                                           workspace_name=databricks_workspace_name,
                                                           access_token=databricks_access_token)
    databricks_compute = ComputeTarget.attach(
        ws,
        databricks_compute_name,
        attach_config
    )

    databricks_compute.wait_for_completion(True)
```

For a more detailed example, see an example notebook on GitHub.

Azure Data Lake Analytics

Azure Data Lake Analytics is a big data analytics platform in the Azure cloud. It can be used as a compute target with an Azure Machine Learning pipeline.

Create an Azure Data Lake Analytics account before using it. To create this resource, see the Get started with Azure Data Lake Analytics document.

To attach Data Lake Analytics as a compute target, you must use the Azure Machine Learning SDK and provide the following information:

- **Compute name**: The name you want to assign to this compute resource.
- **Resource Group**: The resource group that contains the Data Lake Analytics account.
- **Account name**: The Data Lake Analytics account name.

The following code demonstrates how to attach Data Lake Analytics as a compute target:

```
import os
from azureml.core.compute import ComputeTarget, AdlaCompute
from azureml.exceptions import ComputeTargetException


adla_compute_name = os.environ.get(
    "AML_ADLA_COMPUTE_NAME", "<adla_compute_name>")
adla_resource_group = os.environ.get(
    "AML_ADLA_RESOURCE_GROUP", "<adla_resource_group>")
adla_account_name = os.environ.get(
    "AML_ADLA_ACCOUNT_NAME", "<adla_account_name>")

try:
    adla_compute = ComputeTarget(workspace=ws, name=adla_compute_name)
    print('Compute target already exists')
except ComputeTargetException:
    print('compute not found')
    print('adla_compute_name {}'.format(adla_compute_name))
    print('adla_resource_id {}'.format(adla_resource_group))
    print('adla_account_name {}'.format(adla_account_name))
    # create attach config
    attach_config = AdlaCompute.attach_configuration(resource_group=adla_resource_group,
                                                     account_name=adla_account_name)
    # Attach ADLA
    adla_compute = ComputeTarget.attach(
        ws,
        adla_compute_name,
        attach_config
    )

    adla_compute.wait_for_completion(True)
```

For a more detailed example, see an example notebook on GitHub.

> **TIP**
> Azure Machine Learning pipelines can only work with data stored in the default data store of the Data Lake Analytics account. If the data you need to work with is in a non-default store, you can use a `DataTransferStep` to copy the data before training.

Construct your pipeline steps

Once you create and attach a compute target to your workspace, you are ready to define a pipeline step. There are many built-in steps available via the Azure Machine Learning SDK. The most basic of these steps is a PythonScriptStep, which runs a Python script in a specified compute target:

```
from azureml.pipeline.steps import PythonScriptStep

ds_input = my_dataset.as_named_input('input1')

trainStep = PythonScriptStep(
    script_name="train.py",
    arguments=["--input", ds_input.as_download(), "--output", output_data1],
    inputs=[ds_input],
    outputs=[output_data1],
    compute_target=compute_target,
    source_directory=project_folder,
    allow_reuse=True
)
```

Reuse of previous results (`allow_reuse`) is key when using pipelines in a collaborative environment since eliminating unnecessary reruns offers agility. Reuse is the default behavior when the script_name, inputs, and the parameters of a step remain the same. When the output of the step is reused, the job is not submitted to the compute, instead, the results from the previous run are immediately available to the next step's run. If `allow_reuse` is set to false, a new run will always be generated for this step during pipeline execution.

After you define your steps, you build the pipeline by using some or all of those steps.

> **NOTE**
> No file or data is uploaded to Azure Machine Learning when you define the steps or build the pipeline.

```
# list of steps to run
compareModels = [trainStep, extractStep, compareStep]

from azureml.pipeline.core import Pipeline

# Build the pipeline
pipeline1 = Pipeline(workspace=ws, steps=[compareModels])
```

The following example uses the Azure Databricks compute target created earlier:

```
from azureml.pipeline.steps import DatabricksStep

dbStep = DatabricksStep(
    name="databricksmodule",
    inputs=[step_1_input],
    outputs=[step_1_output],
    num_workers=1,
    notebook_path=notebook_path,
    notebook_params={'myparam': 'testparam'},
    run_name='demo run name',
    compute_target=databricks_compute,
    allow_reuse=False
)
# List of steps to run
steps = [dbStep]

# Build the pipeline
pipeline1 = Pipeline(workspace=ws, steps=steps)
```

Use a dataset

Datasets created from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL can be used as input to any pipeline step. With

the exception of writing output to a DataTransferStep or DatabricksStep, output data (PipelineData) can only be written to Azure Blob and Azure File share datastores.

```
dataset_consuming_step = PythonScriptStep(
    script_name="iris_train.py",
    inputs=[iris_tabular_dataset.as_named_input("iris_data")],
    compute_target=compute_target,
    source_directory=project_folder
)
```

You then retrieve the dataset in your pipeline by using the Run.input_datasets dictionary.

```
# iris_train.py
from azureml.core import Run, Dataset

run_context = Run.get_context()
iris_dataset = run_context.input_datasets['iris_data']
dataframe = iris_dataset.to_pandas_dataframe()
```

The line `Run.get_context()` is worth highlighting. This function retrieves a `Run` representing the current experimental run. In the above sample, we use it to retrieve a registered dataset. Another common use of the `Run` object is to retrieve both the experiment itself and the workspace in which the experiment resides:

```
# Within a PythonScriptStep

ws = Run.get_context().experiment.workspace
```

For more detail, including alternate ways to pass and access data, see Moving data into and between ML pipeline steps (Python).

Submit the pipeline

When you submit the pipeline, Azure Machine Learning checks the dependencies for each step and uploads a snapshot of the source directory you specified. If no source directory is specified, the current local directory is uploaded. The snapshot is also stored as part of the experiment in your workspace.

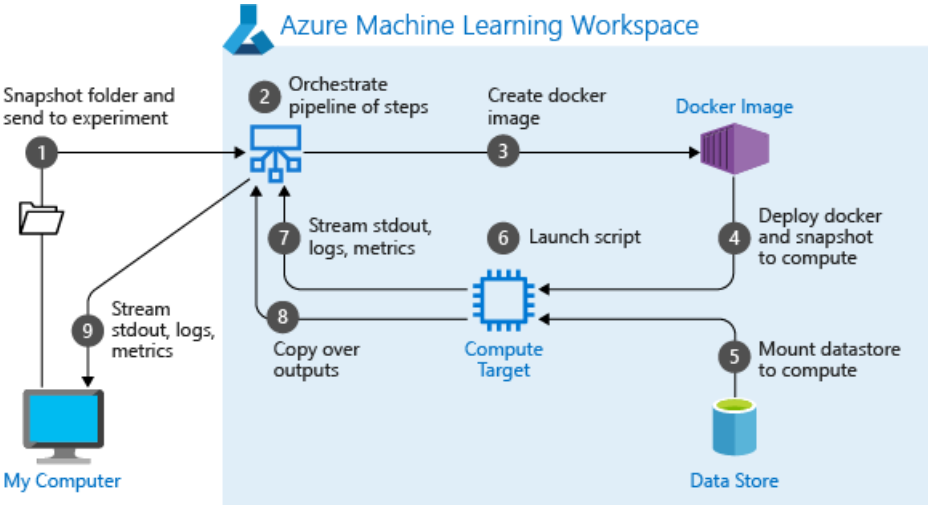
> **IMPORTANT**
>
> To prevent files from being included in the snapshot, create a .gitignore or `.amlignore` file in the directory and add the files to it. The `.amlignore` file uses the same syntax and patterns as the .gitignore file. If both files exist, the `.amlignore` file takes precedence.
>
> For more information, see Snapshots.

```
from azureml.core import Experiment

# Submit the pipeline to be run
pipeline_run1 = Experiment(ws, 'Compare_Models_Exp').submit(pipeline1)
pipeline_run1.wait_for_completion()
```

When you first run a pipeline, Azure Machine Learning:

- Downloads the project snapshot to the compute target from the Blob storage associated with the workspace.
- Builds a Docker image corresponding to each step in the pipeline.
- Downloads the Docker image for each step to the compute target from the container registry.

- Configures access to `Dataset` and `PipelineData` objects. For as `as_mount()` access mode, FUSE is used to provide virtual access. If mount is not supported or if the user specified access as `as_download()`, the data is instead copied to the compute target.
- Runs the step in the compute target specified in the step definition.
- Creates artifacts, such as logs, stdout and stderr, metrics, and output specified by the step. These artifacts are then uploaded and kept in the user's default datastore.

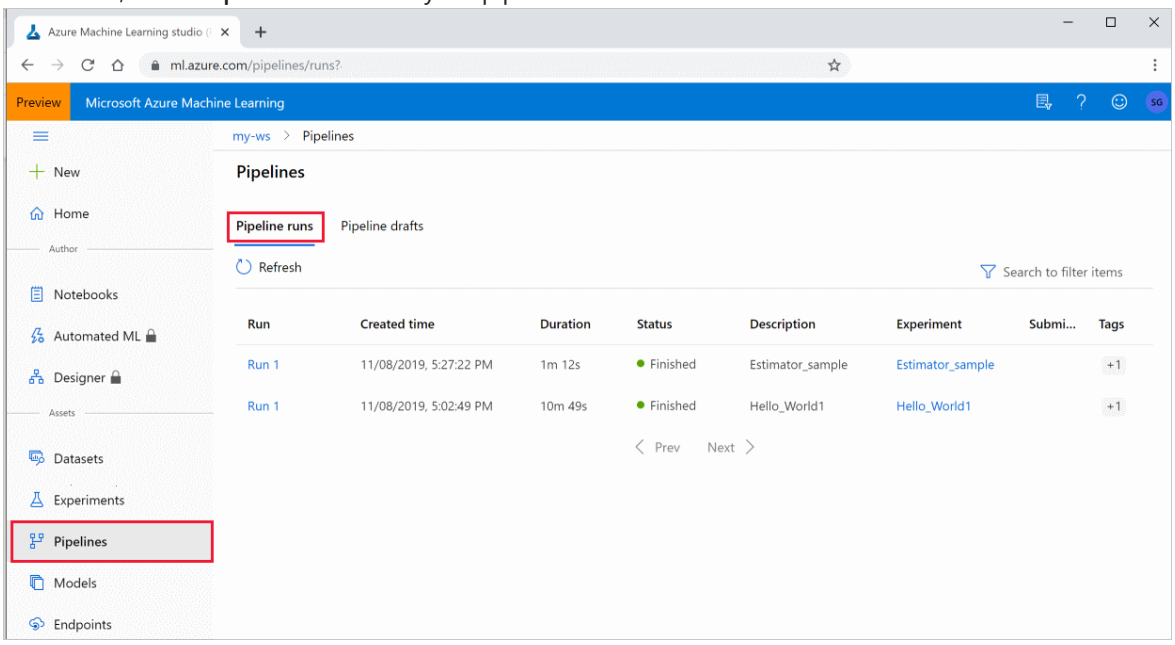


For more information, see the Experiment class reference.

View results of a pipeline

See the list of all your pipelines and their run details in the studio:

1. Sign in to Azure Machine Learning studio.
2. View your workspace.
3. On the left, select **Pipelines** to see all your pipeline runs.



4. Select a specific pipeline to see the run results.

Git tracking and integration

When you start a training run where the source directory is a local Git repository, information about the repository is stored in the run history. For more information, see Git integration for Azure Machine Learning.

Publish a pipeline

You can publish a pipeline to run it with different inputs later. For the REST endpoint of an already published pipeline to accept parameters, you must parameterize the pipeline before publishing.

1. To create a pipeline parameter, use a PipelineParameter object with a default value.

```
from azureml.pipeline.core.graph import PipelineParameter

pipeline_param = PipelineParameter(
  name="pipeline_arg",
  default_value=10)
```

2. Add this `PipelineParameter` object as a parameter to any of the steps in the pipeline as follows:

```
compareStep = PythonScriptStep(
  script_name="compare.py",
  arguments=["--comp_data1", comp_data1, "--comp_data2", comp_data2, "--output_data", out_data3, "--
param1", pipeline_param],
  inputs=[ comp_data1, comp_data2],
  outputs=[out_data3],
  compute_target=compute_target,
  source_directory=project_folder)
```

3. Publish this pipeline that will accept a parameter when invoked.

```
published_pipeline1 = pipeline_run1.publish_pipeline(
     name="My_Published_Pipeline",
     description="My Published Pipeline Description",
     version="1.0")
```

Run a published pipeline

All published pipelines have a REST endpoint. This endpoint invokes the run of the pipeline from external systems, such as non-Python clients. This endpoint enables "managed repeatability" in batch scoring and retraining scenarios.

To invoke the run of the preceding pipeline, you need an Azure Active Directory authentication header token, as described in AzureCliAuthentication class reference or get more details in the Authentication in Azure Machine Learning notebook.

```
from azureml.pipeline.core import PublishedPipeline
import requests

response = requests.post(published_pipeline1.endpoint,
                         headers=aad_token,
                         json={"ExperimentName": "My_Pipeline",
                               "ParameterAssignments": {"pipeline_arg": 20}})
```

Create a versioned pipeline endpoint

You can create a Pipeline Endpoint with multiple published pipelines behind it. This can be used like a published pipeline but gives you a fixed REST endpoint as you iterate on and update your ML pipelines.

```
from azureml.pipeline.core import PipelineEndpoint

published_pipeline = PublishedPipeline.get(workspace="ws", name="My_Published_Pipeline")
pipeline_endpoint = PipelineEndpoint.publish(workspace=ws, name="PipelineEndpointTest",
                                             pipeline=published_pipeline, description="Test description
Notebook")
```

Submit a job to a pipeline endpoint

You can submit a job to the default version of a pipeline endpoint:

```
pipeline_endpoint_by_name = PipelineEndpoint.get(workspace=ws, name="PipelineEndpointTest")
run_id = pipeline_endpoint_by_name.submit("PipelineEndpointExperiment")
print(run_id)
```

You can also submit a job to a specific version:

```
run_id = pipeline_endpoint_by_name.submit("PipelineEndpointExperiment", pipeline_version="0")
print(run_id)
```

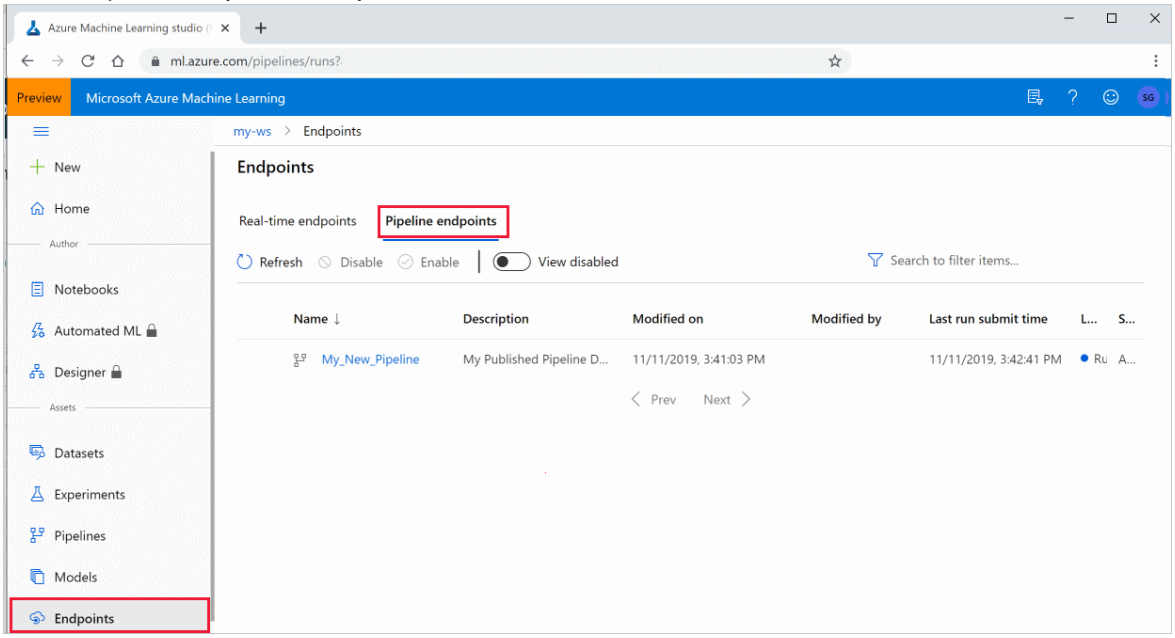
The same can be accomplished using the REST API:

```
rest_endpoint = pipeline_endpoint_by_name.endpoint
response = requests.post(rest_endpoint,
                         headers=aad_token,
                         json={"ExperimentName": "PipelineEndpointExperiment",
                               "RunSource": "API",
                               "ParameterAssignments": {"1": "united", "2":"city"}})
```

Use published pipelines in the studio

You can also run a published pipeline from the studio:

1. Sign in to Azure Machine Learning studio.
2. View your workspace.
3. On the left, select **Endpoints**.
4. On the top, select **Pipeline endpoints**.



5. Select a specific pipeline to run, consume, or review results of previous runs of the pipeline endpoint.

Disable a published pipeline

To hide a pipeline from your list of published pipelines, you disable it, either in the studio or from the SDK:

```
# Get the pipeline by using its ID from Azure Machine Learning studio
p = PublishedPipeline.get(ws, id="068f4885-7088-424b-8ce2-eeb9ba5381a6")
p.disable()
```

You can enable it again with `p.enable()`. For more information, see PublishedPipeline class reference.

Caching & reuse

In order to optimize and customize the behavior of your pipelines, you can do a few things around caching and reuse. For example, you can choose to:

- **Turn off the default reuse of the step run output** by setting `allow_reuse=False` during step definition. Reuse is key when using pipelines in a collaborative environment since eliminating unnecessary runs offers agility. However, you can opt out of reuse.
- **Force output regeneration for all steps in a run** with `pipeline_run = exp.submit(pipeline, regenerate_outputs=False)`

By default, `allow_reuse` for steps is enabled and the `source_directory` specified in the step definition is hashed. So, if the script for a given step remains the same (`script_name`, inputs, and the parameters), and nothing else in the `source_directory` has changed, the output of a previous step run is reused, the job is not submitted to the compute, and the results from the previous run are immediately available to the next step instead.

```
step = PythonScriptStep(name="Hello World",
                        script_name="hello_world.py",
                        compute_target=aml_compute,
                        source_directory=source_directory,
                        allow_reuse=False,
                        hash_paths=['hello_world.ipynb'])
```

Next steps

- Use these Jupyter notebooks on GitHub to explore machine learning pipelines further.
- See the SDK reference help for the azureml-pipelines-core package and the azureml-pipelines-steps package.
- See the how-to for tips on debugging and troubleshooting pipelines.

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Moving data into and between ML pipeline steps (Python)

4/7/2020 • 6 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Data is central to machine learning pipelines. This article provides code for importing, transforming, and moving data between steps in an Azure Machine Learning pipeline. For an overview of how data works in Azure Machine Learning, see Access data in Azure storage services. For the benefits and structure of Azure Machine Learning pipelines, see What are Azure Machine Learning pipelines?.

This article will show you how to:

- Use `Dataset` objects for pre-existing data
- Access data within your steps
- Split `Dataset` data into subsets, such as training and validation subsets
- Create `PipelineData` objects to transfer data to the next pipeline step
- Use `PipelineData` objects as input to pipeline steps
- Create new `Dataset` objects from `PipelineData` you wish to persist

Prerequisites

You'll need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning.
- The Azure Machine Learning SDK for Python, or access to Azure Machine Learning studio.
- An Azure Machine Learning workspace.

  Either create an Azure Machine Learning workspace or use an existing one via the Python SDK. Import the `Workspace` and `Datastore` class, and load your subscription information from the file `config.json` using the function `from_config()`. This function looks for the JSON file in the current directory by default, but you can also specify a path parameter to point to the file using `from_config(path="your/file/path")`.

  ```
  import azureml.core
  from azureml.core import Workspace, Datastore

  ws = Workspace.from_config()
  ```

- Some pre-existing data. This article briefly shows the use of an Azure blob container.
- Optional: An existing machine learning pipeline, such as the one described in Create and run machine learning pipelines with Azure Machine Learning SDK.

Use `Dataset` objects for pre-existing data

The preferred way to ingest data into a pipeline is to use a Dataset object. `Dataset` objects represent persistent data available throughout a workspace.

There are many ways to create and register `Dataset` objects. Tabular datasets are for delimited data available in one or more files. File datasets are for binary data (such as images) or for data that you'll parse. The simplest programmatic ways to create `Dataset` objects are to use existing blobs in workspace storage or public URLs:

```
datastore = Datastore.get(workspace, 'training_data')
iris_dataset = Dataset.Tabular.from_delimited_files(DataPath(datastore, 'iris.csv'))

cats_dogs_dataset = Dataset.File.from_files(
    paths='https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-
6DEBA77B919F/kagglecatsanddogs_3367a.zip',
    archive_options=ArchiveOptions(archive_type=ArchiveType.ZIP, entry_glob='**/*.jpg')
)
```

For more options on creating datasets with different options and from different sources, registering them and reviewing them in the Azure Machine Learning UI, understanding how data size interacts with compute capacity, and versioning them, see Create Azure Machine Learning datasets.

Pass datasets to your script

To pass the dataset's path to your script, use the `Dataset` object's `as_named_input()` method. You can either pass the resulting `DatasetConsumptionConfig` object to your script as an argument or, by using the `inputs` argument to your pipeline script, you can retrieve the dataset using `Run.get_context().input_datasets[]`.

Once you've created a named input, you can choose its access mode: `as_mount()` or `as_download()`. If your script processes all the files in your dataset and the disk on your compute resource is large enough for the dataset, the download access mode is the better choice. The download access mode will avoid the overhead of streaming the data at runtime. If your script accesses a subset of the dataset or it's too large for your compute, use the mount access mode. For more information, read Mount vs. Download

To pass a dataset to your pipeline step:

1. Use `TabularDataset.as_named_inputs()` or `FileDataset.as_named_input()` (no 's' at end) to create a `DatasetConsumptionConfig` object
2. Use `as_mount()` or `as_download()` to set the access mode
3. Pass the datasets to your pipeline steps using either the `arguments` or the `inputs` argument

The following snippet shows the common pattern of combining these steps within the `PythonScriptStep` constructor:

```
train_step = PythonScriptStep(
    name="train_data",
    script_name="train.py",
    compute_target=cluster,
    inputs=[iris_dataset.as_named_inputs('iris').as_mount()]
)
```

You can also use methods such as `random_split()` and `take_sample()` to create multiple inputs or reduce the amount of data passed to your pipeline step:

```
seed = 42 # PRNG seed
smaller_dataset = iris_dataset.take_sample(0.1, seed=seed) # 10%
train, test = smaller_dataset.random_split(percentage=0.8, seed=seed)

train_step = PythonScriptStep(
    name="train_data",
    script_name="train.py",
    compute_target=cluster,
    inputs=[train.as_named_inputs('train').as_download(), test.as_named_inputs('test').as_download()]
)
```

Access datasets within your script

Named inputs to your pipeline step script are available as a dictionary within the `Run` object. Retrieve the active `Run` object using `Run.get_context()` and then retrieve the dictionary of named inputs using `input_datasets`. If you passed the `DatasetConsumptionConfig` object using the `arguments` argument rather than the `inputs` argument, access the data using `ArgParser` code. Both techniques are demonstrated in the following snippet.

```
# In pipeline definition script:
# Code for demonstration only: It would be very confusing to split datasets between `arguments` and `inputs`
train_step = PythonScriptStep(
    name="train_data",
    script_name="train.py",
    compute_target=cluster,
    arguments=['--training-folder', train.as_named_inputs('train').as_download()]
    inputs=[test.as_named_inputs('test').as_download()]
)

# In pipeline script
parser = argparse.ArgumentParser()
parser.add_argument('--training-folder', type=str, dest='train_folder', help='training data folder mounting
point')
args = parser.parse_args()
training_data_folder = args.train_folder

testing_data_folder = Run.get_context().input_datasets['test']
```

The passed value will be the path to the dataset file(s).

It's also possible to access a registered `Dataset` directly. Since registered datasets are persistent and shared across a workspace, you can retrieve them directly:

```
run = Run.get_context()
ws = run.experiment.workspace
ds = Dataset.get_by_name(workspace=ws, name='mnist_opendataset')
```

Use `PipelineData` for intermediate data

While `Dataset` objects represent persistent data, PipelineData objects are used for temporary data that is output from pipeline steps. Because the lifespan of a `PipelineData` object is longer than a single pipeline step, you define them in the pipeline definition script. When you create a `PipelineData` object, you must provide a name and a datastore at which the data will reside. Pass your `PipelineData` object(s) to your `PythonScriptStep` using *both* the `arguments` and the `outputs` arguments:

```
default_datastore = workspace.get_default_datastore()
dataprep_output = PipelineData("clean_data", datastore=default_datastore)

dataprep_step = PythonScriptStep(
    name="prep_data",
    script_name="dataprep.py",
    compute_target=cluster,
    arguments=["--output-path", dataprep_output]
    inputs=[Dataset.get_by_name(workspace, 'raw_data')],
    outputs=[dataprep_output]
)
```

You may choose to create your `PipelineData` object using an access mode that provides an immediate upload. In that case, when you create your `PipelineData`, set the `upload_mode` to `"upload"` and use the `output_path_on_compute` argument to specify the path to which you'll be writing the data:

```
PipelineData("clean_data", datastore=def_blob_store, output_mode="upload",
output_path_on_compute="clean_data_output/")
```

Use `PipelineData` as outputs of a training step

Within your pipeline's `PythonScriptStep`, you can retrieve the available output paths using the program's arguments. If this step is the first and will initialize the output data, you must create the directory at the specified path. You can then write whatever files you wish to be contained in the `PipelineData`.

```
parser = argparse.ArgumentParser()
parser.add_argument('--output_path', dest='output_path', required=True)
args = parser.parse_args()

# Make directory for file
os.makedirs(os.path.dirname(args.output_path), exist_ok=True)
with open(args.output_path, 'w') as f:
    f.write("Step 1's output")
```

If you created your `PipelineData` with the `is_directory` argument set to `True`, it would be enough to just perform the `os.makedirs()` call and then you would be free to write whatever files you wished to the path. For more details, see the PipelineData reference documentation.

Read `PipelineData` as inputs to non-initial steps

After the initial pipeline step writes some data to the `PipelineData` path and it becomes an output of that initial step, it can be used as an input to a later step:

```
step1_output_data = PipelineData("processed_data", datastore=def_blob_store, output_mode="upload")

step1 = PythonScriptStep(
    name="generate_data",
    script_name="step1.py",
    runconfig = aml_run_config,
    arguments = ["--output_path", step1_output_data],
    inputs=[],
    outputs=[step1_output_data]
)

step2 = PythonScriptStep(
    name="read_pipeline_data",
    script_name="step2.py",
    compute_target=compute,
    runconfig = aml_run_config,
    arguments = ["--pd", step1_output_data],
    inputs=[step1_output_data]
)

pipeline = Pipeline(workspace=ws, steps=[step1, step2])
```

The value of a `PipelineData` input is the path to the previous output. If, as shown previously, the first step wrote a single file, consuming it might look like:

```
parser = argparse.ArgumentParser()
parser.add_argument('--pd', dest='pd', required=True)
args = parser.parse_args()

with open(args.pd) as f:
    print(f.read())
```

Convert `PipelineData` objects to `Dataset`s

If you'd like to make your `PipelineData` available for longer than the duration of a run, use its `as_dataset()` function to convert it to a `Dataset`. You may then register the `Dataset`, making it a first-class citizen in your workspace. Since your `PipelineData` object will have a different path every time the pipeline runs, it's highly recommended that you set `create_new_version` to `True` when registering a `Dataset` created from a `PipelineData` object.

```
step1_output_ds = step1_output_data.as_dataset()
step1_output_ds.register(name="processed_data", create_new_version=True)
```

Next steps

- Create an Azure machine learning dataset
- Create and run machine learning pipelines with Azure Machine Learning SDK

Schedule machine learning pipelines with Azure Machine Learning SDK for Python

4/22/2020 • 4 minutes to read • Edit Online

In this article, you'll learn how to programmatically schedule a pipeline to run on Azure. You can choose to create a schedule based on elapsed time or on file-system changes. Time-based schedules can be used to take care of routine tasks, such as monitoring for data drift. Change-based schedules can be used to react to irregular or unpredictable changes, such as new data being uploaded or old data being edited. After learning how to create schedules, you'll learn how to retrieve and deactivate them.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account.
- A Python environment in which the Azure Machine Learning SDK for Python is installed. For more information, see Create and manage reusable environments for training and deployment with Azure Machine Learning.
- A Machine Learning workspace with a published pipeline. You can use the one built in Create and run machine learning pipelines with Azure Machine Learning SDK.

Initialize the workspace & get data

To schedule a pipeline, you'll need a reference to your workspace, the identifier of your published pipeline, and the name of the experiment in which you wish to create the schedule. You can get these values with the following code:

```
import azureml.core
from azureml.core import Workspace
from azureml.pipeline.core import Pipeline, PublishedPipeline
from azureml.core.experiment import Experiment

ws = Workspace.from_config()

experiments = Experiment.list(ws)
for experiment in experiments:
    print(experiment.name)

published_pipelines = PublishedPipeline.list(ws)
for published_pipeline in  published_pipelines:
    print(f"{published_pipeline.name},'{published_pipeline.id}'")

experiment_name = "MyExperiment"
pipeline_id = "aaaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
```

Create a schedule

To run a pipeline on a recurring basis, you'll create a schedule. A `Schedule` associates a pipeline, an experiment, and a trigger. The trigger can either be a `ScheduleRecurrence` that describes the wait between runs or a Datastore path that specifies a directory to watch for changes. In either case, you'll need the pipeline identifier and the name of the experiment in which to create the schedule.

At the top of your python file, import the `Schedule` and `ScheduleRecurrence` classes:

```
from azureml.pipeline.core.schedule import ScheduleRecurrence, Schedule
```

Create a time-based schedule

The `ScheduleRecurrence` constructor has a required `frequency` argument that must be one of the following strings: "Minute", "Hour", "Day", "Week", or "Month". It also requires an integer `interval` argument specifying how many of the `frequency` units should elapse between schedule starts. Optional arguments allow you to be more specific about starting times, as detailed in the ScheduleRecurrence SDK docs.

Create a `Schedule` that begins a run every 15 minutes:

```
recurrence = ScheduleRecurrence(frequency="Minute", interval=15)
recurring_schedule = Schedule.create(ws, name="MyRecurringSchedule",
                            description="Based on time",
                            pipeline_id=pipeline_id,
                            experiment_name=experiment_name,
                            recurrence=recurrence)
```

Create a change-based schedule

Pipelines that are triggered by file changes may be more efficient than time-based schedules. For instance, you may want to perform a preprocessing step when a file is changed, or when a new file is added to a data directory. You can monitor any changes to a datastore or changes within a specific directory within the datastore. If you monitor a specific directory, changes within subdirectories of that directory will *not* trigger a run.

To create a file-reactive `Schedule`, you must set the `datastore` parameter in the call to Schedule.create. To monitor a folder, set the `path_on_datastore` argument.

The `polling_interval` argument allows you to specify, in minutes, the frequency at which the datastore is checked for changes.

If the pipeline was constructed with a DataPath PipelineParameter, you can set that variable to the name of the changed file by setting the `data_path_parameter_name` argument.

```
datastore = Datastore(workspace=ws, name="workspaceblobstore")

reactive_schedule = Schedule.create(ws, name="MyReactiveSchedule", description="Based on input file change.",
                            pipeline_id=pipeline_id, experiment_name=experiment_name, datastore=datastore,
data_path_parameter_name="input_data")
```

Optional arguments when creating a schedule

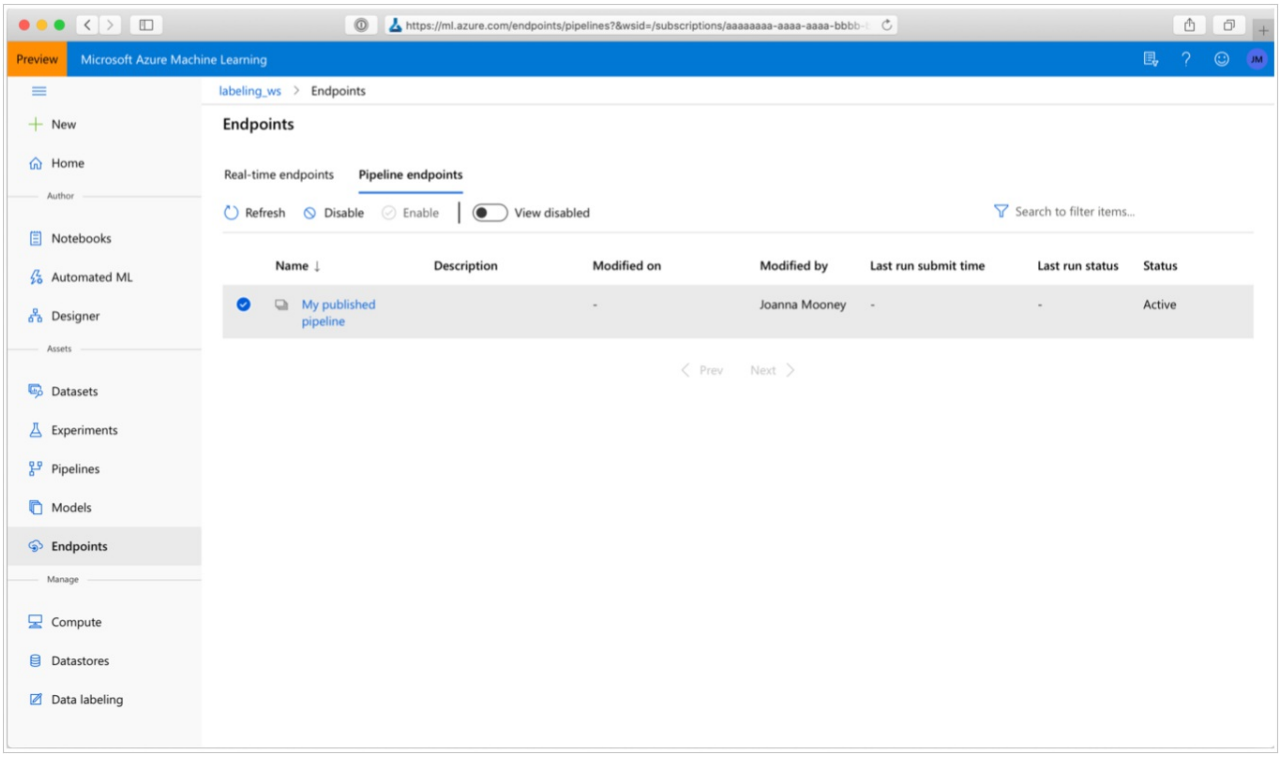
In addition to the arguments discussed previously, you may set the `status` argument to `"Disabled"` to create an inactive schedule. Finally, the `continue_on_step_failure` allows you to pass a Boolean that will override the pipeline's default failure behavior.

Use Azure Logic Apps for more complex workflows

Azure Logic Apps supports more complex workflows and is far more broadly integrated than Azure Machine Learning pipelines. See Trigger a run of a Machine Learning pipeline from a Logic App for more information.

View your scheduled pipelines

In your Web browser, navigate to Azure Machine Learning. From the **Endpoints** section of the navigation panel, choose **Pipeline endpoints**. This takes you to a list of the pipelines published in the Workspace.



In this page you can see summary information about all the pipelines in the Workspace: names, descriptions, status, and so forth. Drill in by clicking in your pipeline. On the resulting page, there are more details about your pipeline and you may drill down into individual runs.

Deactivate the pipeline

If you have a `Pipeline` that is published, but not scheduled, you can disable it with:

```
pipeline = PublishedPipeline.get(ws, id=pipeline_id)
pipeline.disable()
```

If the pipeline is scheduled, you must cancel the schedule first. Retrieve the schedule's identifier from the portal or by running:

```
ss = Schedule.list(ws)
for s in ss:
    print(s)
```

Once you have the `schedule_id` you wish to disable, run:

```
def stop_by_schedule_id(ws, schedule_id):
    s = next(s for s in Schedule.list(ws) if s.id == schedule_id)
    s.disable()
    return s

stop_by_schedule_id(ws, schedule_id)
```

If you then run `Schedule.list(ws)` again, you should get an empty list.

Next steps

In this article, you used the Azure Machine Learning SDK for Python to schedule a pipeline in two different ways. One schedule recurs based on elapsed clock time. The other schedule runs if a file is modified on a specified

`Datastore` or within a directory on that store. You saw how to use the portal to examine the pipeline and individual runs. Finally, you learned how to disable a schedule so that the pipeline stops running.

For more information, see:

Use Azure Machine Learning Pipelines for batch scoring

- Learn more about pipelines
- Learn more about exploring Azure Machine Learning with Jupyter

Trigger a run of a Machine Learning pipeline from a Logic App

2/10/2020 • 2 minutes to read • Edit Online

Trigger the run of your Azure Machine Learning Pipeline when new data appears. For example, you may want to trigger the pipeline to train a new model when new data appears in the blob storage account. Set up the trigger with Azure Logic Apps.

Prerequisites

- An Azure Machine Learning workspace. For more information, see Create an Azure Machine Learning workspace.
- The REST endpoint for a published Machine Learning pipeline. Create and publish your pipeline. Then find the REST endpoint of your PublishedPipeline by using the pipeline ID:

  ```
  # You can find the pipeline ID in Azure Machine Learning studio

  published_pipeline = PublishedPipeline.get(ws, id="<pipeline-id-here>")
  published_pipeline.endpoint
  ```

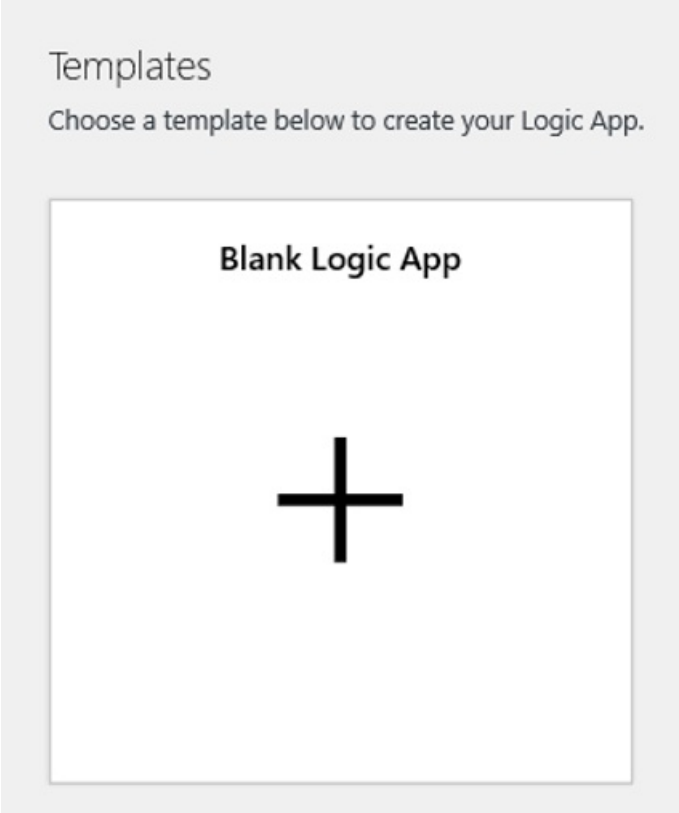
- Azure blob storage to store your data.
- A datastore in your workspace that contains the details of your blob storage account.

Create a Logic App

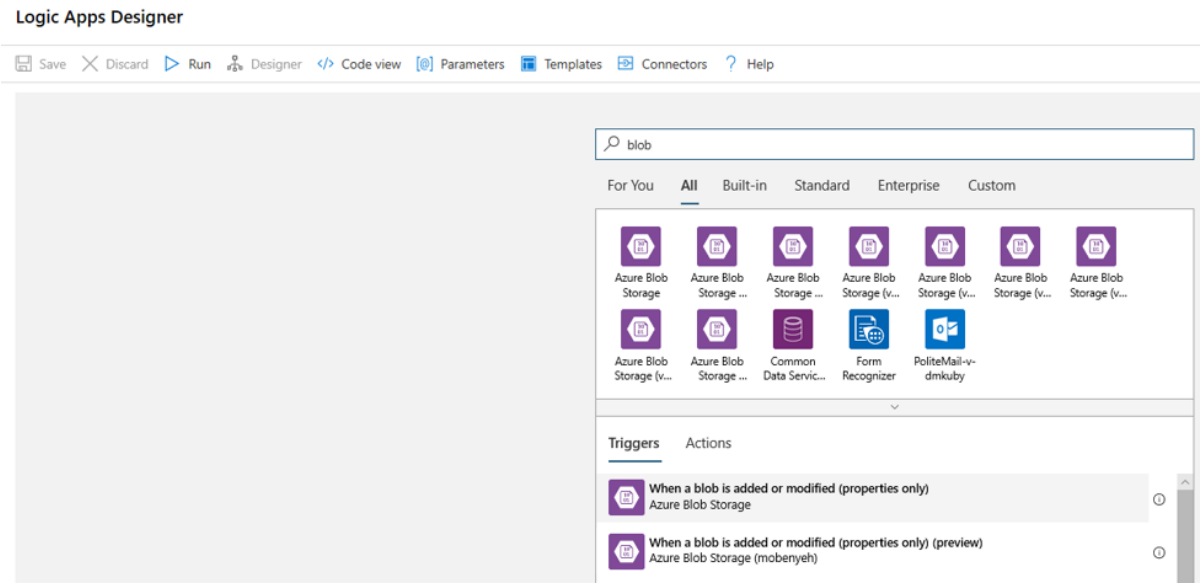
Now create an Azure Logic App instance. If you wish, use an integration service environment (ISE) and set up a customer-managed key for use by your Logic App.

Once your Logic App has been provisioned, use these steps to configure a trigger for your pipeline:

1. Create a system-assigned managed identity to give the app access to your Azure Machine Learning Workspace.
2. Navigate to the Logic App Designer view and select the Blank Logic App template.



3. In the Designer, search for **blob**. Select the **When a blob is added or modified (properties only)** trigger and add this trigger to your Logic App.

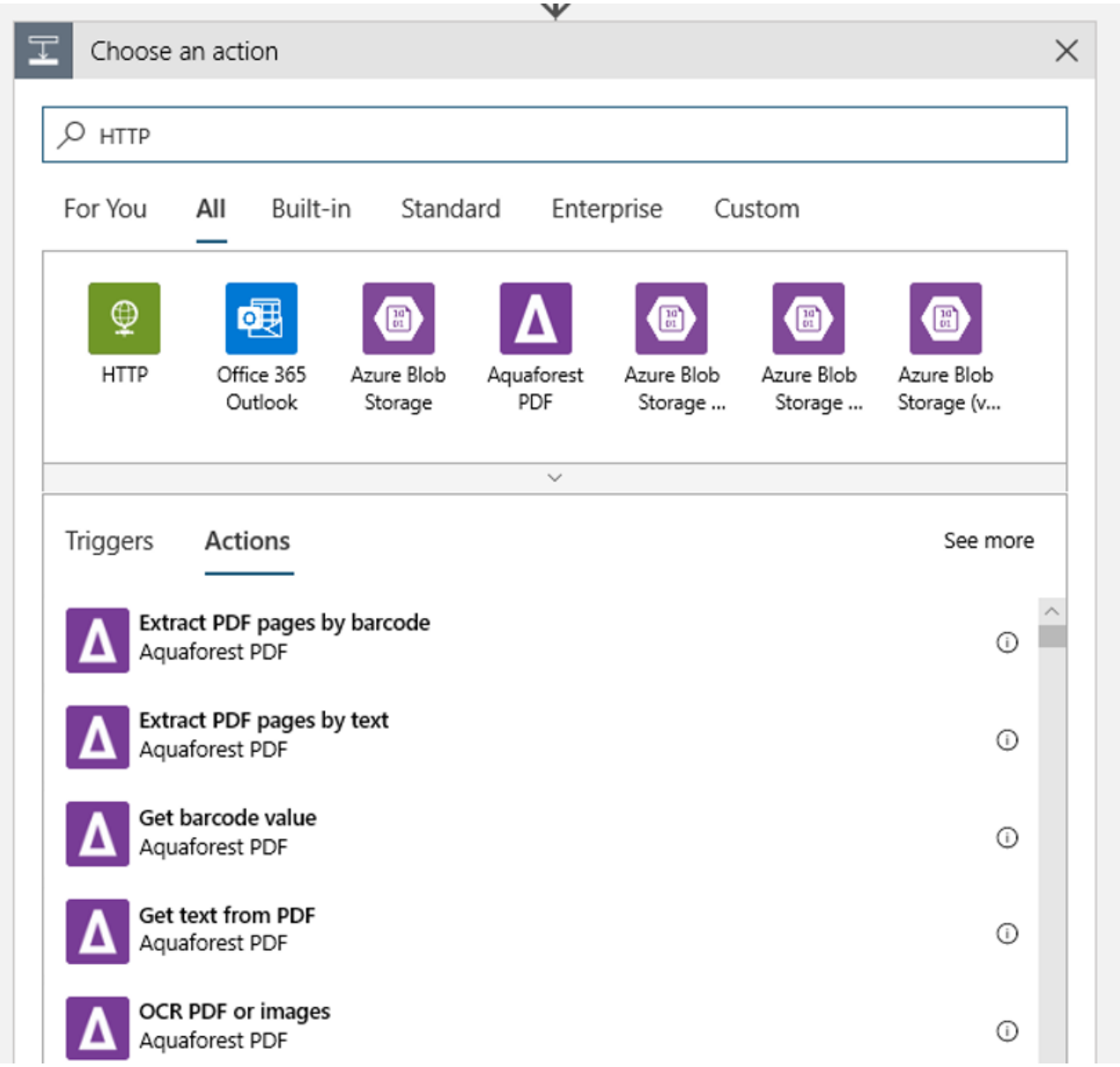


4. Fill in the connection info for the Blob storage account you wish to monitor for blob additions or modifications. Select the Container to monitor.

   Choose the **Interval** and **Frequency** to poll for updates that work for you.

   > **NOTE**
   > This trigger will monitor the selected Container but will not monitor subfolders.

5. Add an HTTP action that will run when a new or modified blob is detected. Select **+ New Step**, then search for and select the HTTP action.



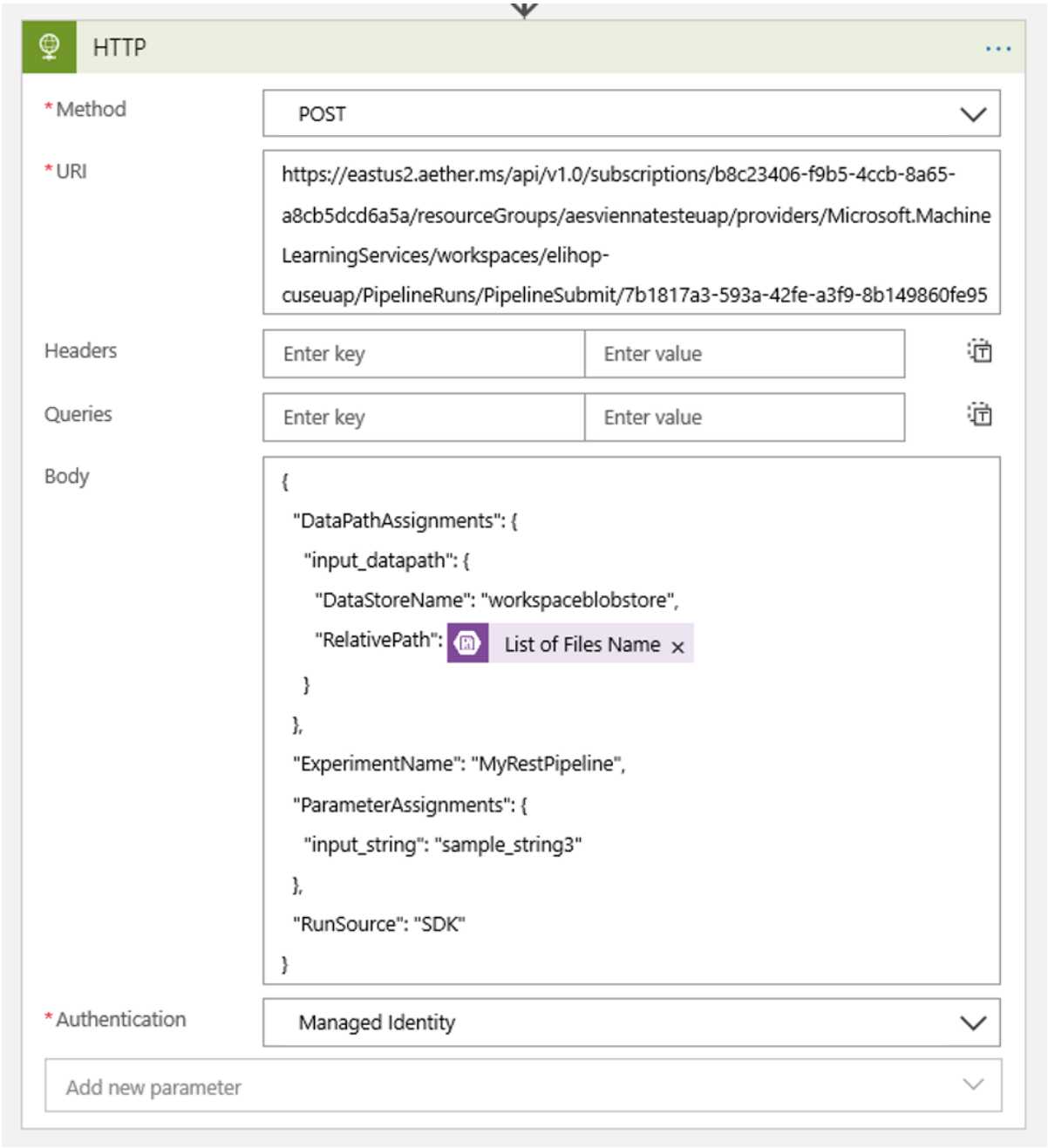
Use the following settings to configure your action:

SETTING	VALUE
HTTP action	POST
URI	the endpoint to the published pipeline that you found as a Prerequisite
Authentication mode	Managed Identity

1. Set up your schedule to set the value of any DataPath PipelineParameters you may have:

```
"DataPathAssignments":{
     "input_datapath":{
          "DataStoreName":"<datastore-name>",
"RelativePath":"@triggerBody()?['Name']"
}
},
"ExperimentName":"MyRestPipeline",
"ParameterAssignments":{
"input_string":"sample_string3"
},
```

Use the `DataStoreName` you added to your workspace as a Prerequisite.



2. Select **Save** and your schedule is now ready.

Debug and troubleshoot machine learning pipelines

4/10/2020 • 12 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to debug and troubleshoot machine learning pipelines in the Azure Machine Learning SDK and Azure Machine Learning designer (preview). Information is provided on how to:

- Debug using the Azure Machine Learning SDK
- Debug using the Azure Machine Learning designer
- Debug using Application Insights
- Debug interactively using Visual Studio Code (VS Code) and the Python Tools for Visual Studio (PTVSD)

Debug and troubleshoot in the Azure Machine Learning SDK

The following sections provide an overview of common pitfalls when building pipelines, and different strategies for debugging your code that's running in a pipeline. Use the following tips when you're having trouble getting a pipeline to run as expected.

Testing scripts locally

One of the most common failures in a pipeline is that an attached script (data cleansing script, scoring script, etc.) is not running as intended, or contains runtime errors in the remote compute context that are difficult to debug in your workspace in the Azure Machine Learning studio.

Pipelines themselves cannot be run locally, but running the scripts in isolation on your local machine allows you to debug faster because you don't have to wait for the compute and environment build process. Some development work is required to do this:

- If your data is in a cloud datastore, you will need to download data and make it available to your script. Using a small sample of your data is a good way to cut down on runtime and quickly get feedback on script behavior
- If you are attempting to simulate an intermediate pipeline step, you may need to manually build the object types that the particular script is expecting from the prior step
- You will also need to define your own environment, and replicate the dependencies defined in your remote compute environment

Once you have a script setup to run on your local environment, it is much easier to do debugging tasks like:

- Attaching a custom debug configuration
- Pausing execution and inspecting object-state
- Catching type or logical errors that won't be exposed until runtime

> **TIP**
> Once you can verify that your script is running as expected, a good next step is running the script in a single-step pipeline before attempting to run it in a pipeline with multiple steps.

Debugging scripts from remote context

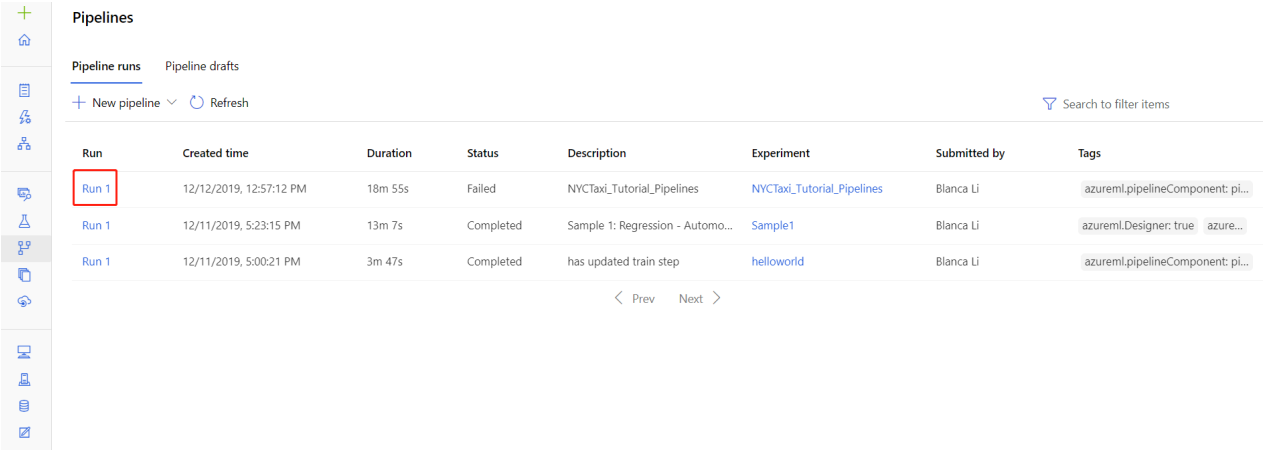
Testing scripts locally is a great way to debug major code fragments and complex logic before you start building a pipeline, but at some point you will likely need to debug scripts during the actual pipeline run itself, especially when diagnosing behavior that occurs during the interaction between pipeline steps. We recommend liberal use of

`print()` statements in your step scripts so that you can see object state and expected values during remote execution, similar to how you would debug JavaScript code.

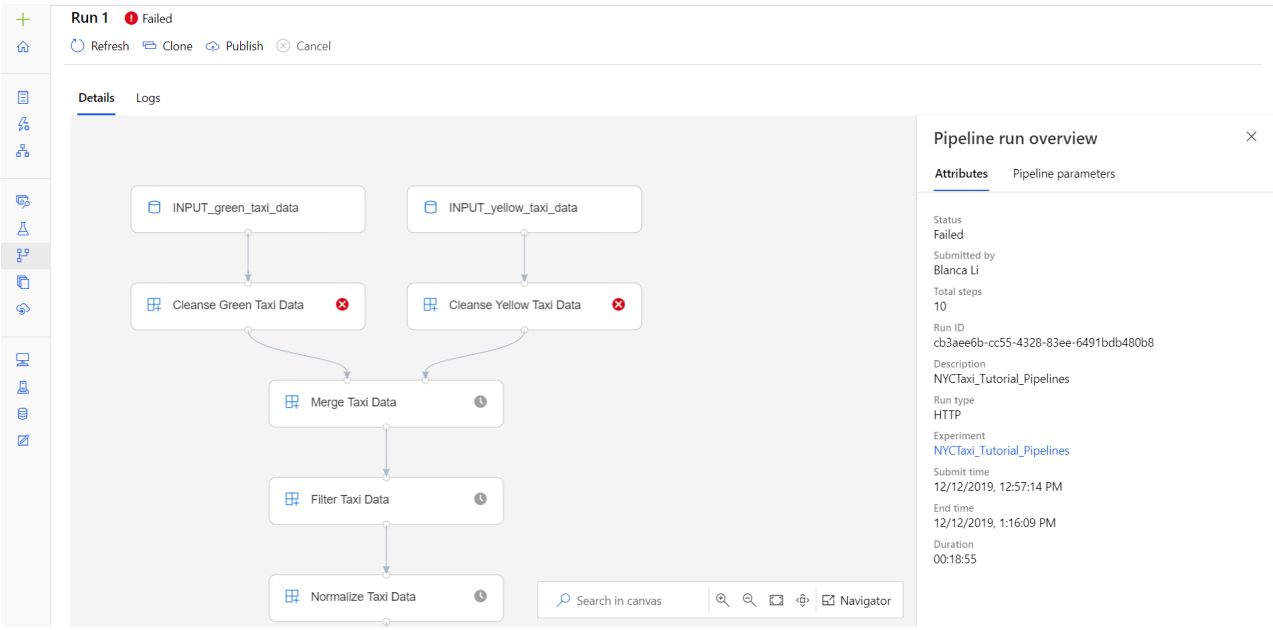
The log file `70_driver_log.txt` contains:

- All printed statements during your script's execution
- The stack trace for the script

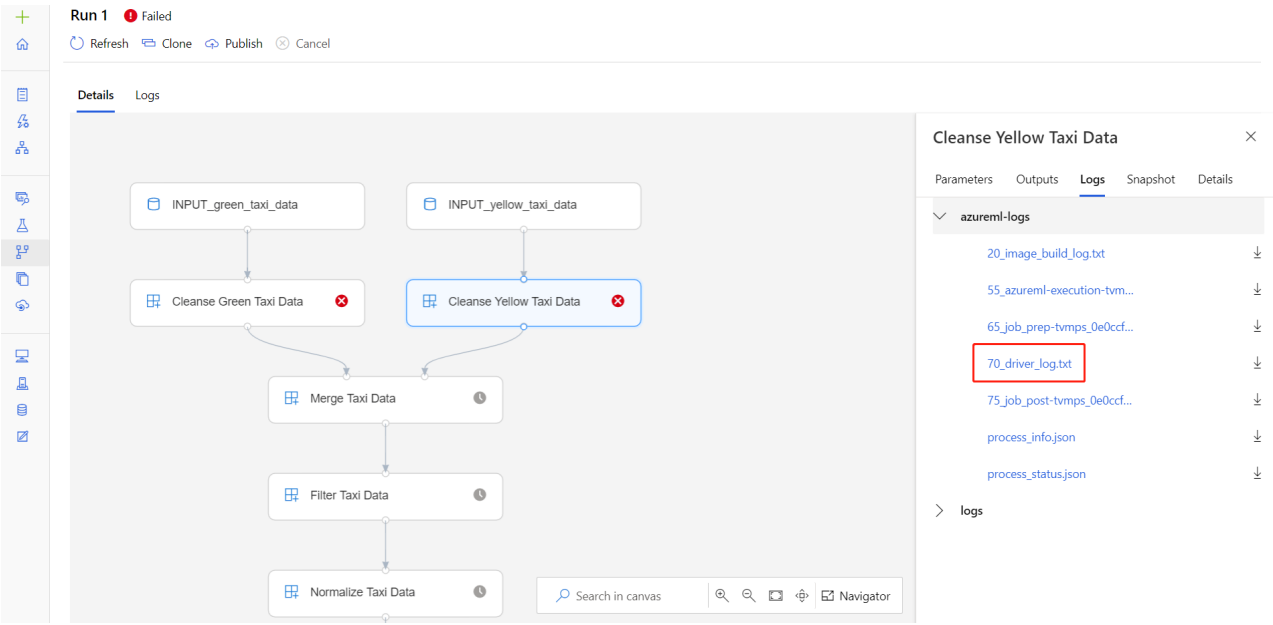
To find this and other log files in the portal, first click on the pipeline run in your workspace.



Navigate to the pipeline run detail page.



Click on the module for the specific step. Navigate to the **Logs** tab. Other logs include information about your environment image build process and step preparation scripts.



TIP

Runs for *published pipelines* can be found in the **Endpoints** tab in your workspace. Runs for *non-published pipelines* can be found in **Experiments** or **Pipelines**.

Troubleshooting tips

The following table contains common problems during pipeline development, with potential solutions.

PROBLEM	POSSIBLE SOLUTION
Unable to pass data to `PipelineData` directory	Ensure you have created a directory in the script that corresponds to where your pipeline expects the step output data. In most cases, an input argument will define the output directory, and then you create the directory explicitly. Use `os.makedirs(args.output_dir, exist_ok=True)` to create the output directory. See the tutorial for a scoring script example that shows this design pattern.
Dependency bugs	If you have developed and tested scripts locally but find dependency issues when running on a remote compute in the pipeline, ensure your compute environment dependencies and versions match your test environment. (See Environment building, caching, and reuse
Ambiguous errors with compute targets	Deleting and re-creating compute targets can solve certain issues with compute targets.
Pipeline not reusing steps	Step reuse is enabled by default, but ensure you haven't disabled it in a pipeline step. If reuse is disabled, the `allow_reuse` parameter in the step will be set to `False`.
Pipeline is rerunning unnecessarily	To ensure that steps only rerun when their underlying data or scripts change, decouple your directories for each step. If you use the same source directory for multiple steps, you may experience unnecessary reruns. Use the `source_directory` parameter on a pipeline step object to point to your isolated directory for that step, and ensure you aren't using the same `source_directory` path for multiple steps.

Logging options and behavior

The table below provides information for different debug options for pipelines. It isn't an exhaustive list, as other options exist besides just the Azure Machine Learning, Python, and OpenCensus ones shown here.

LIBRARY	TYPE	EXAMPLE	DESTINATION	RESOURCES
Azure Machine Learning SDK	Metric	`run.log(name, val)`	Azure Machine Learning Portal UI	How to track experiments azureml.core.Run class
Python printing/logging	Log	`print(val)` `logging.info(message)`	Driver logs, Azure Machine Learning designer	How to track experiments Python logging
OpenCensus Python	Log	`logger.addHandler(AzureLogHandler())` `logging.log(message)`	Application Insights - traces	Debug pipelines in Application Insights OpenCensus Azure Monitor Exporters Python logging cookbook

Logging options example

```
import logging

from azureml.core.run import Run
from opencensus.ext.azure.log_exporter import AzureLogHandler

run = Run.get_context()

# Azure ML Scalar value logging
run.log("scalar_value", 0.95)

# Python print statement
print("I am a python print statement, I will be sent to the driver logs.")

# Initialize python logger
logger = logging.getLogger(__name__)
logger.setLevel(args.log_level)

# Plain python logging statements
logger.debug("I am a plain debug statement, I will be sent to the driver logs.")
logger.info("I am a plain info statement, I will be sent to the driver logs.")

handler = AzureLogHandler(connection_string='<connection string>')
logger.addHandler(handler)

# Python logging with OpenCensus AzureLogHandler
logger.warning("I am an OpenCensus warning statement, find me in Application Insights!")
logger.error("I am an OpenCensus error statement with custom dimensions", {'step_id': run.id})
```

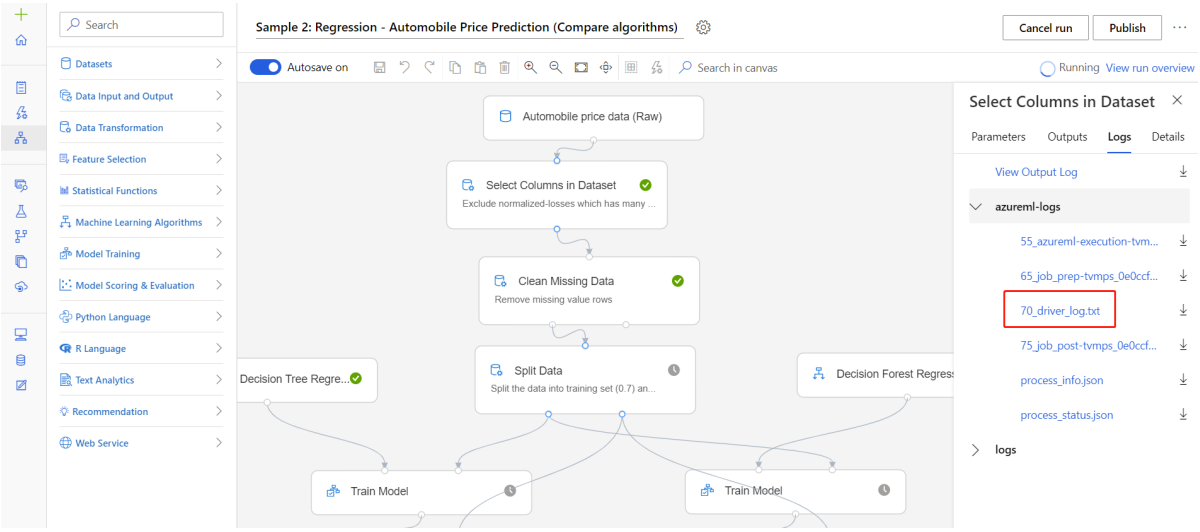
Debug and troubleshoot in Azure Machine Learning designer (preview)

This section provides an overview of how to troubleshoot pipelines in the designer. For pipelines created in the designer, you can find the **log files** on either the authoring page, or in the pipeline run detail page.

Access logs from the authoring page

When you submit a pipeline run and stay in the authoring page, you can find the log files generated for each module.

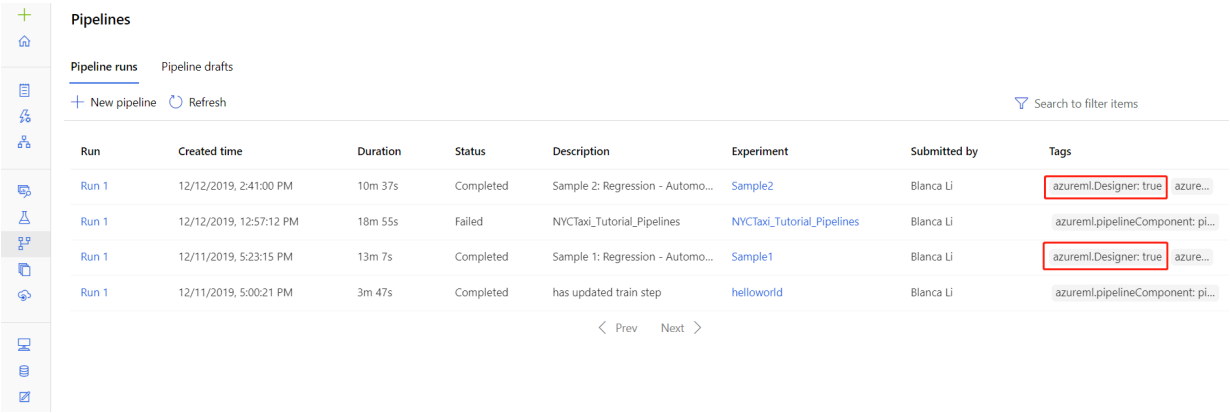
1. Select any module in the authoring canvas.
2. In the right pane of the module, go to the **Outputs + logs** tab.
3. Select the log file `70_driver_log.txt`.



Access logs from pipeline runs

You can also find the log files of specific runs in the pipeline run detail page in either the **Pipelines** or **Experiments** sections.

1. Select a pipeline run created in the designer.



2. Select any module in the preview pane.
3. In the right pane of the module, go to the **Outputs + logs** tab.
4. Select the log file `70_driver_log.txt`.

Debug and troubleshoot in Application Insights

For more information on using the OpenCensus Python library in this manner, see this guide: Debug and troubleshoot machine learning pipelines in Application Insights

Debug and troubleshoot in Visual Studio Code

In some cases, you may need to interactively debug the Python code used in your ML pipeline. By using Visual Studio Code (VS Code) and the Python Tools for Visual Studio (PTVSD), you can attach to the code as it runs in the training environment.

Prerequisites

- An **Azure Machine Learning workspace** that is configured to use an **Azure Virtual Network**.

- An **Azure Machine Learning pipeline** that uses Python scripts as part of the pipeline steps. For example, a PythonScriptStep.
- An Azure Machine Learning Compute cluster, which is **in the virtual network** and is **used by the pipeline for training**.
- A **development environment** that is **in the virtual network**. The development environment might be one of the following:
  - An Azure Virtual Machine in the virtual network
  - A Compute instance of Notebook VM in the virtual network
  - A client machine connected to the virtual network by a virtual private network (VPN).

For more information on using an Azure Virtual Network with Azure Machine Learning, see Secure Azure ML experimentation and inference jobs within an Azure Virtual Network.

How it works

Your ML pipeline steps run Python scripts. These scripts are modified to perform the following actions:

1. Log the IP address of the host that they are running on. You use the IP address to connect the debugger to the script.
2. Start the PTVSD debug component, and wait for a debugger to connect.
3. From your development environment, you monitor the logs created by the training process to find the IP address where the script is running.
4. You tell VS Code the IP address to connect the debugger to by using a `launch.json` file.
5. You attach the debugger and interactively step through the script.

Configure Python scripts

To enable debugging, make the following changes to the Python script(s) used by steps in your ML pipeline:

1. Add the following import statements:

```
import ptvsd
import socket
from azureml.core import Run
```

2. Add the following arguments. These arguments allow you to enable the debugger as needed, and set the timeout for attaching the debugger:

```
parser.add_argument('--remote_debug', action='store_true')
parser.add_argument('--remote_debug_connection_timeout', type=int,
                default=300,
                help=f'Defines how much time the Azure ML compute target '
                f'will await a connection from a debugger client (VSCODE).')
```

3. Add the following statements. These statements load the current run context so that you can log the IP address of the node that the code is running on:

```
global run
run = Run.get_context()
```

4. Add an `if` statement that starts PTVSD and waits for a debugger to attach. If no debugger attaches before the timeout, the script continues as normal.

```
  if args.remote_debug:
      print(f'Timeout for debug connection: {args.remote_debug_connection_timeout}')
      # Log the IP and port
      ip = socket.gethostbyname(socket.gethostname())
      print(f'ip_address: {ip}')
      ptvsd.enable_attach(address=('0.0.0.0', 5678),
                          redirect_output=True)
      # Wait for the timeout for debugger to attach
      ptvsd.wait_for_attach(timeout=args.remote_debug_connection_timeout)
      print(f'Debugger attached = {ptvsd.is_attached()}')
```

The following Python example shows a basic `train.py` file that enables debugging:

```
# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license.

import argparse
import os
import ptvsd
import socket
from azureml.core import Run

print("In train.py")
print("As a data scientist, this is where I use my training code.")

parser = argparse.ArgumentParser("train")

parser.add_argument("--input_data", type=str, help="input data")
parser.add_argument("--output_train", type=str, help="output_train directory")

# Argument check for remote debugging
parser.add_argument('--remote_debug', action='store_true')
parser.add_argument('--remote_debug_connection_timeout', type=int,
                    default=300,
                    help=f'Defines how much time the AML compute target '
                    f'will await a connection from a debugger client (VSCODE).')
# Get run object, so we can find and log the IP of the host instance
global run
run = Run.get_context()

args = parser.parse_args()

# Start debugger if remote_debug is enabled
if args.remote_debug:
    print(f'Timeout for debug connection: {args.remote_debug_connection_timeout}')
    # Log the IP and port
    ip = socket.gethostbyname(socket.gethostname())
    print(f'ip_address: {ip}')
    ptvsd.enable_attach(address=('0.0.0.0', 5678),
                        redirect_output=True)
    # Wait for the timeout for debugger to attach
    ptvsd.wait_for_attach(timeout=args.remote_debug_connection_timeout)
    print(f'Debugger attached = {ptvsd.is_attached()}')

print("Argument 1: %s" % args.input_data)
print("Argument 2: %s" % args.output_train)

if not (args.output_train is None):
    os.makedirs(args.output_train, exist_ok=True)
    print("%s created" % args.output_train)
```

Configure ML pipeline

To provide the Python packages needed to start PTVSD and get the run context, create an environment and set

`pip_packages=['ptvsd', 'azureml-sdk==1.0.83']`. Change the SDK version to match the one you are using. The following code snippet demonstrates how to create an environment:

```
# Use a RunConfiguration to specify some additional requirements for this step.
from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import DEFAULT_CPU_IMAGE

# create a new runconfig object
run_config = RunConfiguration()

# enable Docker
run_config.environment.docker.enabled = True

# set Docker base image to the default CPU-based image
run_config.environment.docker.base_image = DEFAULT_CPU_IMAGE

# use conda_dependencies.yml to create a conda environment in the Docker image for execution
run_config.environment.python.user_managed_dependencies = False

# specify CondaDependencies obj
run_config.environment.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-learn'],
                                                                            pip_packages=['ptvsd', 'azureml-
sdk==1.0.83'])
```

In the Configure Python scripts section, two new arguments were added to the scripts used by your ML pipeline steps. The following code snippet demonstrates how to use these arguments to enable debugging for the component and set a timeout. It also demonstrates how to use the environment created earlier by setting `runconfig=run_config`:

```
# Use RunConfig from a pipeline step
step1 = PythonScriptStep(name="train_step",
                         script_name="train.py",
                         arguments=['--remote_debug', '--remote_debug_connection_timeout', 300],
                         compute_target=aml_compute,
                         source_directory=source_directory,
                         runconfig=run_config,
                         allow_reuse=False)
```

When the pipeline runs, each step creates a child run. If debugging is enabled, the modified script logs information similar to the following text in the `70_driver_log.txt` for the child run:

```
Timeout for debug connection: 300
ip_address: 10.3.0.5
```

Save the `ip_address` value. It is used in the next section.

> **TIP**
> You can also find the IP address from the run logs for the child run for this pipeline step. For more information on viewing this information, see Monitor Azure ML experiment runs and metrics.

Configure development environment

1. To install the Python Tools for Visual Studio (PTVSD) on your VS Code development environment, use the following command:

```
python -m pip install --upgrade ptvsd
```

For more information on using PTVSD with VS Code, see Remote Debugging.

2. To configure VS Code to communicate with the Azure Machine Learning compute that is running the debugger, create a new debug configuration:

   a. From VS Code, select the **Debug** menu and then select **Open configurations**. A file named **launch.json** opens.

   b. In the **launch.json** file, find the line that contains `"configurations": [`, and insert the following text after it. Change the `"host": "10.3.0.5"` entry to the IP address returned in your logs from the previous section. Change the `"localRoot": "${workspaceFolder}/code/step"` entry to a local directory that contains a copy of the script being debugged:

   ```
   {
       "name": "Azure Machine Learning Compute: remote debug",
       "type": "python",
       "request": "attach",
       "port": 5678,
       "host": "10.3.0.5",
       "redirectOutput": true,
       "pathMappings": [
           {
               "localRoot": "${workspaceFolder}/code/step1",
               "remoteRoot": "."
           }
       ]
   }
   ```

   > **IMPORTANT**
   > If there are already other entries in the configurations section, add a comma (,) after the code that you inserted.

   > **TIP**
   > The best practice is to keep the resources for scripts in separate directories, which is why the `localRoot` example value references `/code/step1`.
   >
   > If you are debugging multiple scripts, in different directories, create a separate configuration section for each script.

   c. Save the **launch.json** file.

Connect the debugger

1. Open VS Code and open a local copy of the script.

2. Set breakpoints where you want the script to stop once you've attached.

3. While the child process is running the script, and the `Timeout for debug connection` is displayed in the logs, use the F5 key or select **Debug**. When prompted, select the **Azure Machine Learning Compute: remote debug** configuration. You can also select the debug icon from the side bar, the **Azure Machine Learning: remote debug** entry from the Debug dropdown menu, and then use the green arrow to attach the debugger.

At this point, VS Code connects to PTVSD on the compute node and stops at the breakpoint you set previously. You can now step through the code as it runs, view variables, etc.

> **NOTE**
>
> If the log displays an entry stating `Debugger attached = False`, then the timeout has expired and the script continued without the debugger. Submit the pipeline again and connect the debugger after the `Timeout for debug connection` message, and before the timeout expires.

Next steps

- See the SDK reference for help with the azureml-pipelines-core package and the azureml-pipelines-steps package.
- See the list of designer exceptions and error codes.

Debug and troubleshoot machine learning pipelines in Application Insights

4/8/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

The OpenCensus python library can be used to route logs to Application Insights from your scripts. Aggregating logs from pipeline runs in one place allows you to build queries and diagnose issues. Using Application Insights will allow you to track logs over time and compare pipeline logs across runs.

Having your logs in once place will provide a history of exceptions and error messages. Since Application Insights integrates with Azure Alerts, you can also create alerts based on Application Insights queries.

Prerequisites

- Follow the steps to create an Azure Machine Learning workspace and create your first pipeline
- Configure your development environment to install the Azure Machine Learning SDK.
- Install the OpenCensus Azure Monitor Exporter package locally:

```
pip install opencensus-ext-azure
```

- Create an Application Insights instance (this doc also contains information on getting the connection string for the resource)

Getting Started

This section is an introduction specific to using OpenCensus from an Azure Machine Learning pipeline. For a detailed tutorial, see OpenCensus Azure Monitor Exporters

Add a PythonScriptStep to your Azure ML Pipeline. Configure your RunConfiguration with the dependency on opencensus-ext-azure. Configure the `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable.

```
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import RunConfiguration
from azureml.pipeline.core import Pipeline
from azureml.pipeline.steps import PythonScriptStep

# Connecting to the workspace and compute target not shown

# Add pip dependency on OpenCensus
dependencies = CondaDependencies()
dependencies.add_pip_package("opencensus-ext-azure>=1.0.1")
run_config = RunConfiguration(conda_dependencies=dependencies)

# Add environment variable with Application Insights Connection String
# Replace the value with your own connection string
run_config.environment.environment_variables = {
    "APPLICATIONINSIGHTS_CONNECTION_STRING": 'InstrumentationKey=00000000-0000-0000-0000-000000000000'
}

# Configure step with runconfig
sample_step = PythonScriptStep(
        script_name="sample_step.py",
        compute_target=compute_target,
        runconfig=run_config
)

# Submit new pipeline run
pipeline = Pipeline(workspace=ws, steps=[sample_step])
pipeline.submit(experiment_name="Logging_Experiment")
```

Create a file called `sample_step.py`. Import the AzureLogHandler class to route logs to Application Insights. You'll also need to import the Python Logging library.

```
from opencensus.ext.azure.log_exporter import AzureLogHandler
import logging
```

Next, add the AzureLogHandler to the python logger.

```
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)
logger.addHandler(logging.StreamHandler())

# Assumes the environment variable APPLICATIONINSIGHTS_CONNECTION_STRING is already set
logger.addHandler(AzureLogHandler())
logger.warning("I will be sent to Application Insights")
```

Logging with Custom Dimensions

By default, logs forwarded to Application Insights won't have enough context to trace back to the run or experiment. To make the logs actionable for diagnosing issues, additional fields are needed.

To add these fields, Custom Dimensions can be added to provide context to a log message. One example is when someone wants to view logs across multiple steps in the same pipeline run.

Custom Dimensions make up a dictionary of key-value (stored as string, string) pairs. The dictionary is then sent to Application Insights and displayed as a column in the query results. Its individual dimensions can be used as query parameters.

Helpful Context to include

FIELD	REASONING/EXAMPLE
parent_run_id	Can query logs for ones with the same parent_run_id to see logs over time for all steps, instead of having to dive into each individual step
step_id	Can query logs for ones with the same step_id to see where an issue occurred with a narrow scope to just the individual step
step_name	Can query logs to see step performance over time. Also helps to find a step_id for recent runs without diving into the portal UI
experiment_name	Can query across logs to see experiment performance over time. Also helps find a parent_run_id or step_id for recent runs without diving into the portal UI
run_url	Can provide a link directly back to the run for investigation.

Other helpful fields

These fields may require additional code instrumentation, and aren't provided by the run context.

FIELD	REASONING/EXAMPLE
build_url/build_version	If using CI/CD to deploy, this field can correlate logs to the code version that provided the step and pipeline logic. This link can further help to diagnose issues, or identify models with specific traits (log/metric values)
run_type	Can differentiate between different model types, or training vs. scoring runs

Creating a Custom Dimensions dictionary

```
from azureml.core import Run

run = Run.get_context(allow_offline=False)

custom_dimensions = {
    "parent_run_id": run.parent.id,
    "step_id": run.id,
    "step_name": run.name,
    "experiment_name": run.experiment.name,
    "run_url": run.parent.get_portal_url(),
    "run_type": "training"
}

# Assumes AzureLogHandler was already registered above
logger.info("I will be sent to Application Insights with Custom Dimensions", custom_dimensions)
```

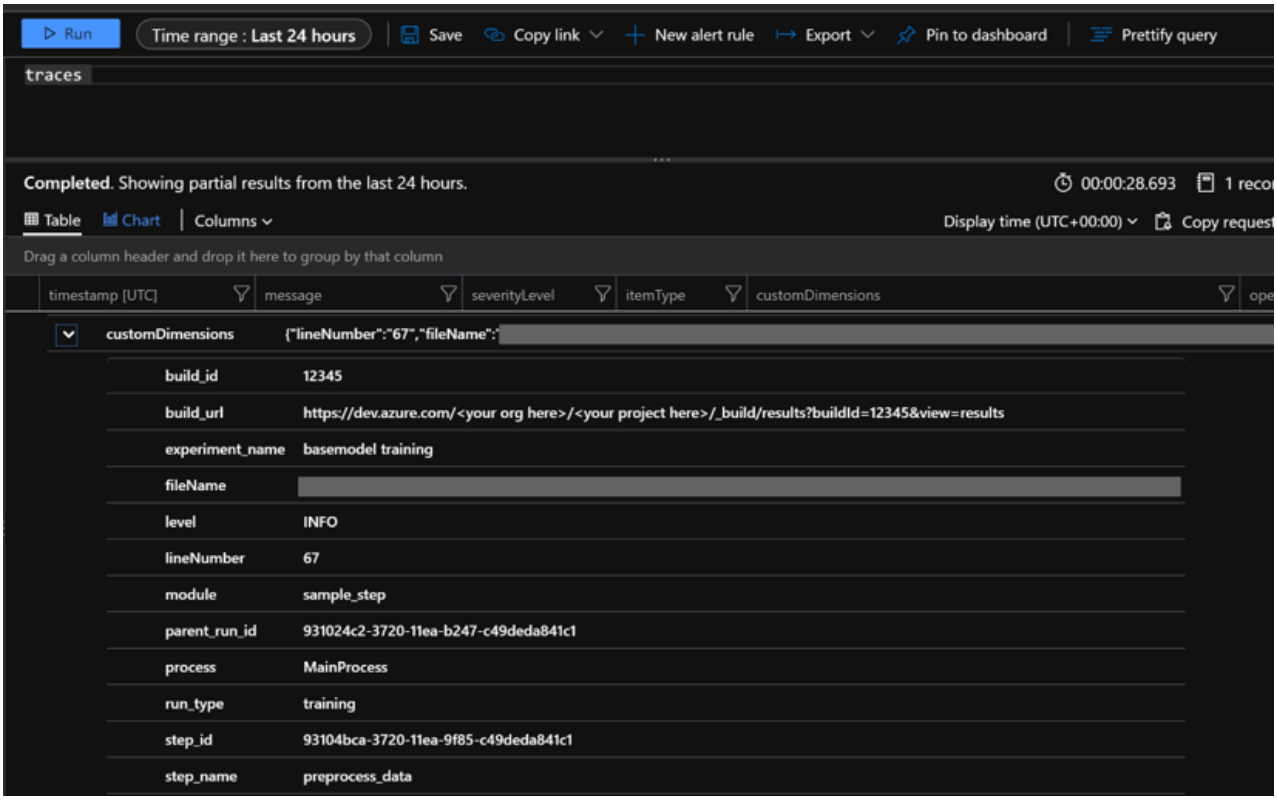
OpenCensus Python logging considerations

The OpenCensus AzureLogHandler is used to route Python logs to Application Insights. As a result, Python logging nuances should be considered. When a logger is created, it has a default log level and will show logs greater than or equal to that level. A good reference for using Python logging features is the Logging Cookbook.

The `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable is needed for the OpenCensus library. We recommend setting this environment variable instead of passing it in as a pipeline parameter to avoid passing around plaintext connection strings.

Querying logs in Application Insights

The logs routed to Application Insights will show up under 'traces' or 'exceptions'. Be sure to adjust your time window to include your pipeline run.



The result in Application Insights will show the log message and level, file path, and code line number. It will also show any custom dimensions included. In this image, the customDimensions dictionary shows the key/value pairs from the previous code sample.

Additional helpful queries

Some of the queries below use 'customDimensions.Level'. These severity levels correspond to the level the Python log was originally sent with. For additional query information, see Azure Monitor Log Queries.

USE CASE	QUERY
Log results for specific custom dimension, for example 'parent_run_id'	`traces | where customDimensions.parent_run_id == '931024c2-3720-11ea-b247-c49deda841c1`
Log results for all training runs over the last 7 days	`traces | where timestamp > ago(7d) and customDimensions.run_type == 'training'`

USE CASE	QUERY
Log results with severityLevel Error from the last 7 days	`traces | where timestamp > ago(7d) and customDimensions.Level == 'ERROR'`
Count of log results with severityLevel Error over the last 7 days	`traces | where timestamp > ago(7d) and customDimensions.Level == 'ERROR' | summarize count()`

Next Steps

Once you have logs in your Application Insights instance, they can be used to set Azure Monitor alerts based on query results.

You can also add results from queries to an Azure Dashboard for additional insights.

Retrain models with Azure Machine Learning designer (preview)

4/7/2020 • 3 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this how-to article, you learn how to use Azure Machine Learning designer to retrain a machine learning model. You will use published pipelines to automate your workflow and set parameters to train your model on new data.

In this article, you learn how to:

- Train a machine learning model.
- Create a pipeline parameter.
- Publish your training pipeline.
- Retrain your model with new parameters.

Prerequisites

- An Azure Machine Learning workspace with the Enterprise SKU.
- A dataset accessible to the designer. This can be one of the following:
  - An Azure Machine Learning registered dataset

    -or-

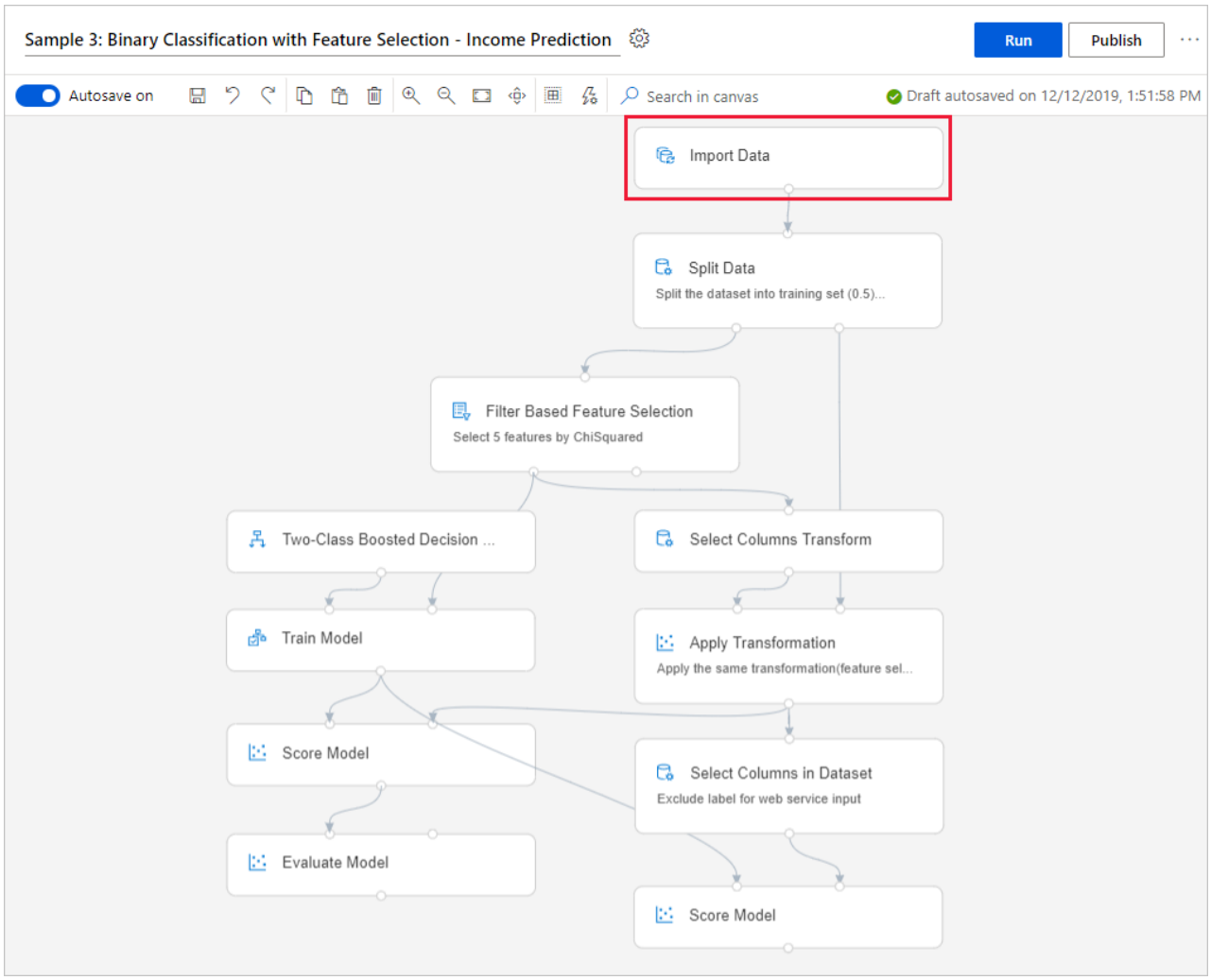
  - A data file stored in an Azure Machine Learning datastore.

For information on data access using the designer see How to import data into the designer.

This article also assumes that you have basic knowledge of building pipelines in the designer. For a guided introduction, complete the tutorial.

Sample pipeline

The pipeline used in this article is an altered version of Sample 3: Income prediction. The pipeline uses the Import Data module instead of the sample dataset to show you how to train models using your own data.



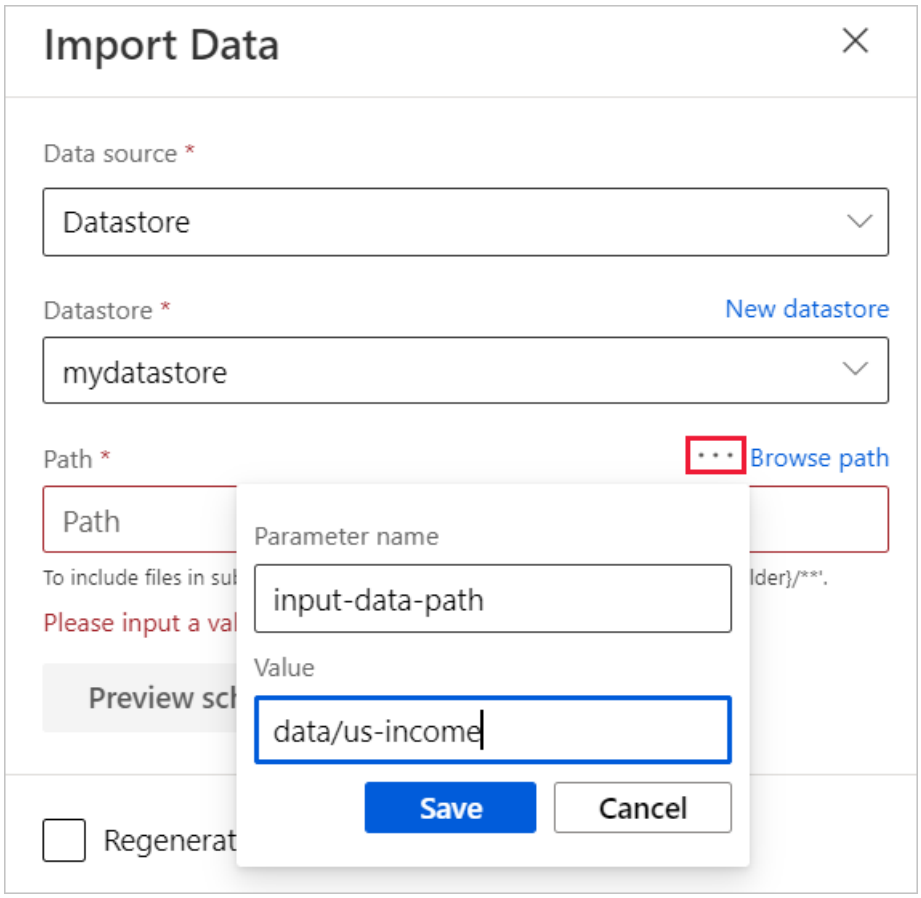
Create a pipeline parameter

Create pipeline parameters to dynamically set variables at runtime. For this example, you will change the training data path from a fixed value to a parameter, so that you can retrain your model on different data.

1. Select the **Import Data** module.

   > **NOTE**
   > This example uses the Import Data module to access data in a registered datastore. However, you can follow similar steps if you use alternative data access patterns.

2. In the module detail pane, to the right of the canvas, select your data source.
3. Enter the path to your data. You can also select **Browse path** to browse your file tree.
4. Mouseover the **Path** field, and select the ellipses above the **Path** field that appear.



5. Select **Add to pipeline parameter**.
6. Provide a parameter name and a default value.

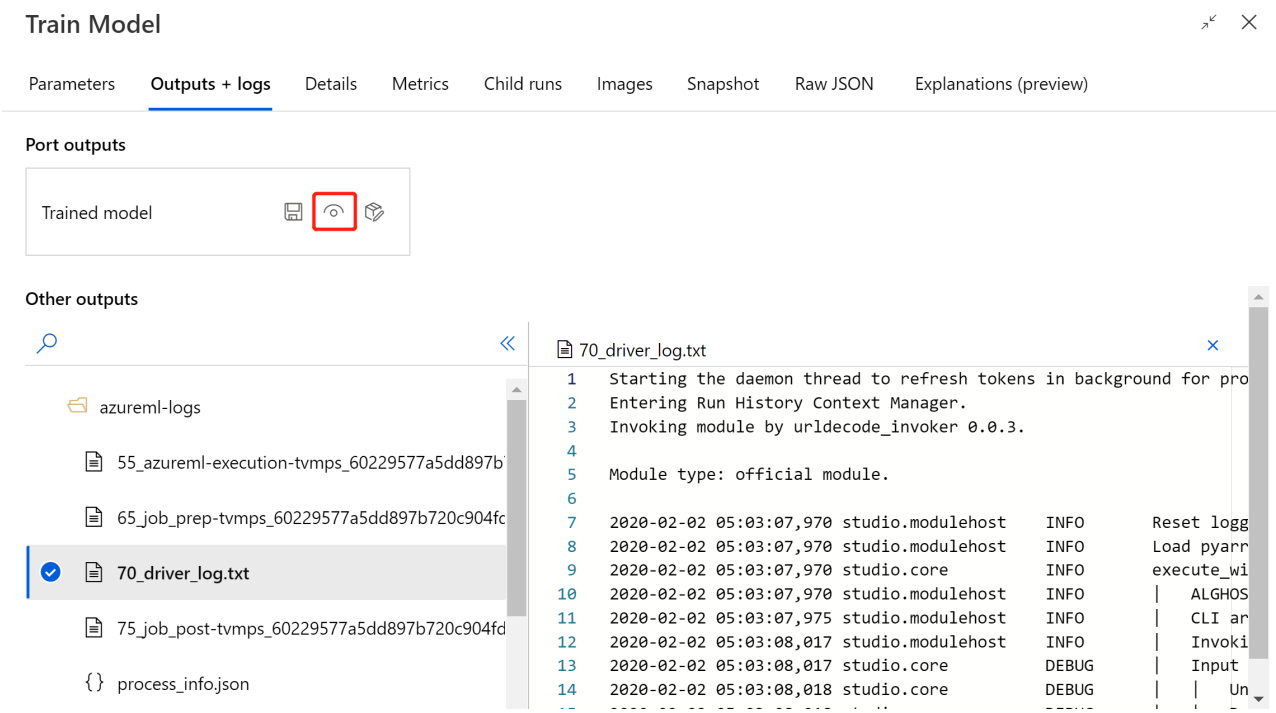
> **NOTE**
> You can inspect and edit your pipeline parameters by selecting the **Settings** gear icon next to the title of your pipeline draft.

7. Select **Save**.
8. Submit the pipeline run.

Find a trained model

The designer saves all pipeline output, including trained models, to the default workspace storage account. You can also access trained models directly in the designer:

1. Wait for the pipeline to finish running.
2. Select the **Train Model** module.
3. In the module details pane, to the right of the canvas, select **Outputs + logs**.
4. You can find your model in **Other outputs** along with run logs.
5. Alternatively, select the **View output** icon. From here, you can follow the instruction in the dialog to navigate directly to your datastore.



Publish a training pipeline

Publish a pipeline to a pipeline endpoint to easily reuse your pipelines in the future. A pipeline endpoint creates a REST endpoint to invoke pipeline in the future. In this example, your pipeline endpoint lets you reuse your pipeline to retrain a model on different data.

1. Select **Publish** above the designer canvas.
2. Select or create a pipeline endpoint.

> **NOTE**
> You can publish multiple pipelines to a single endpoint. Each pipeline in a given endpoint is given a version number, which you can specify when you call the pipeline endpoint.

3. Select **Publish**.

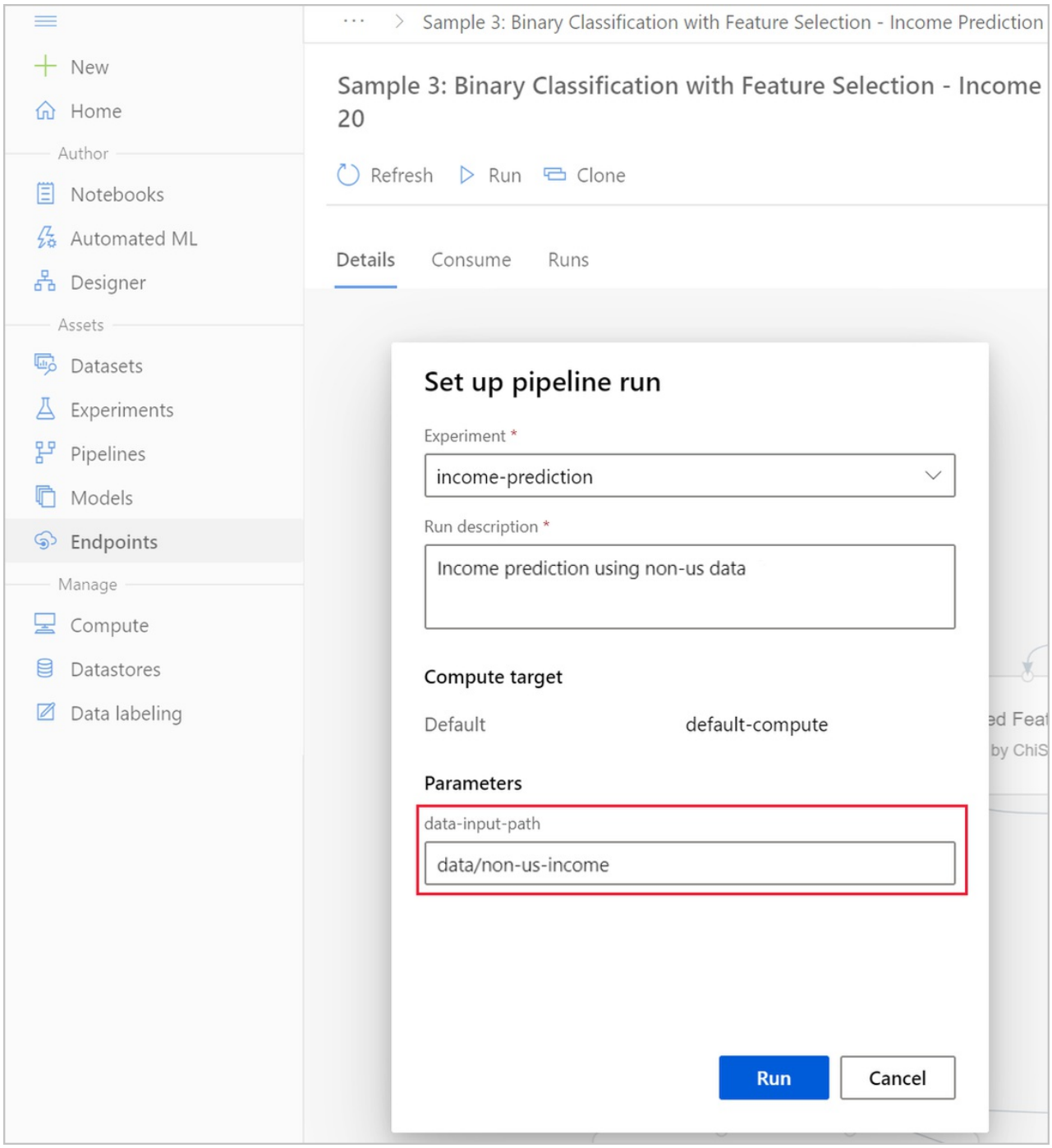
Retrain your model

Now that you have a published training pipeline, you can use it to retrain your model on new data. You can submit runs from a pipeline endpoint from the studio workspace or programmatically.

Submit runs by using the designer

Use the following steps to submit a parameterized pipeline endpoint run from the designer:

1. Go to the **Endpoints** page in your studio workspace.
2. Select the **Pipeline endpoints** tab. Then, select your pipeline endpoint.
3. Select the **Published pipelines** tab. Then, select the pipeline version that you want to run.
4. Select **Submit**.
5. In the setup dialog box, you can specify the parameters values for the run. For this example, update the data path to train your model using a non-US dataset.



Submit runs by using code

You can find the REST endpoint of a published pipeline in the overview panel. By calling the endpoint, you can retrain the published pipeline.

To make a REST call, you need an OAuth 2.0 bearer-type authentication header. For information about setting up authentication to your workspace and making a parameterized REST call, see Build an Azure Machine Learning pipeline for batch scoring.

Next steps

In this article, you learned how to create a parameterized training pipeline endpoint using the designer.

For a complete walkthrough of how you can deploy a model to make predictions, see the designer tutorial to train and deploy a regression model.

Run batch predictions using Azure Machine Learning designer (preview)

3/17/2020 • 3 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise)

In this article, you learn how to use the designer to create a batch prediction pipeline. Batch prediction lets you continuously score large datasets on-demand using a web service that can be triggered from any HTTP library.

In this how-to, you learn to do the following tasks:

- Create and publish a batch inference pipeline
- Consume a pipeline endpoint
- Manage endpoint versions

To learn how to set up batch scoring services using the SDK, see the accompanying how-to.

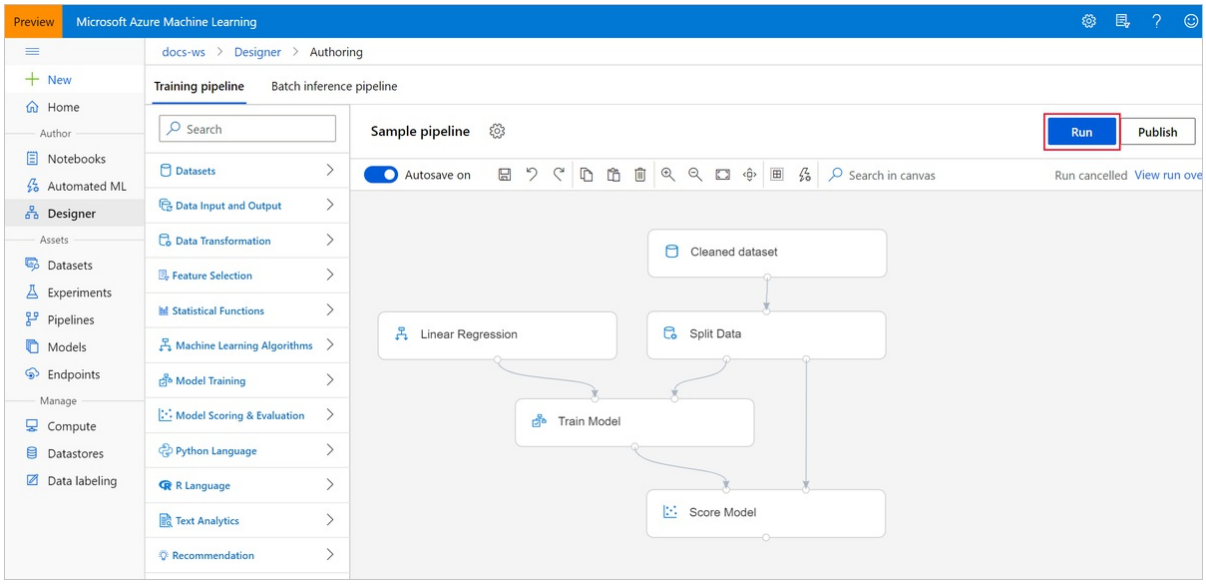
Prerequisites

This how-to assumes you already have a training pipeline. For a guided introduction to the designer, complete part one of the designer tutorial.

Create a batch inference pipeline

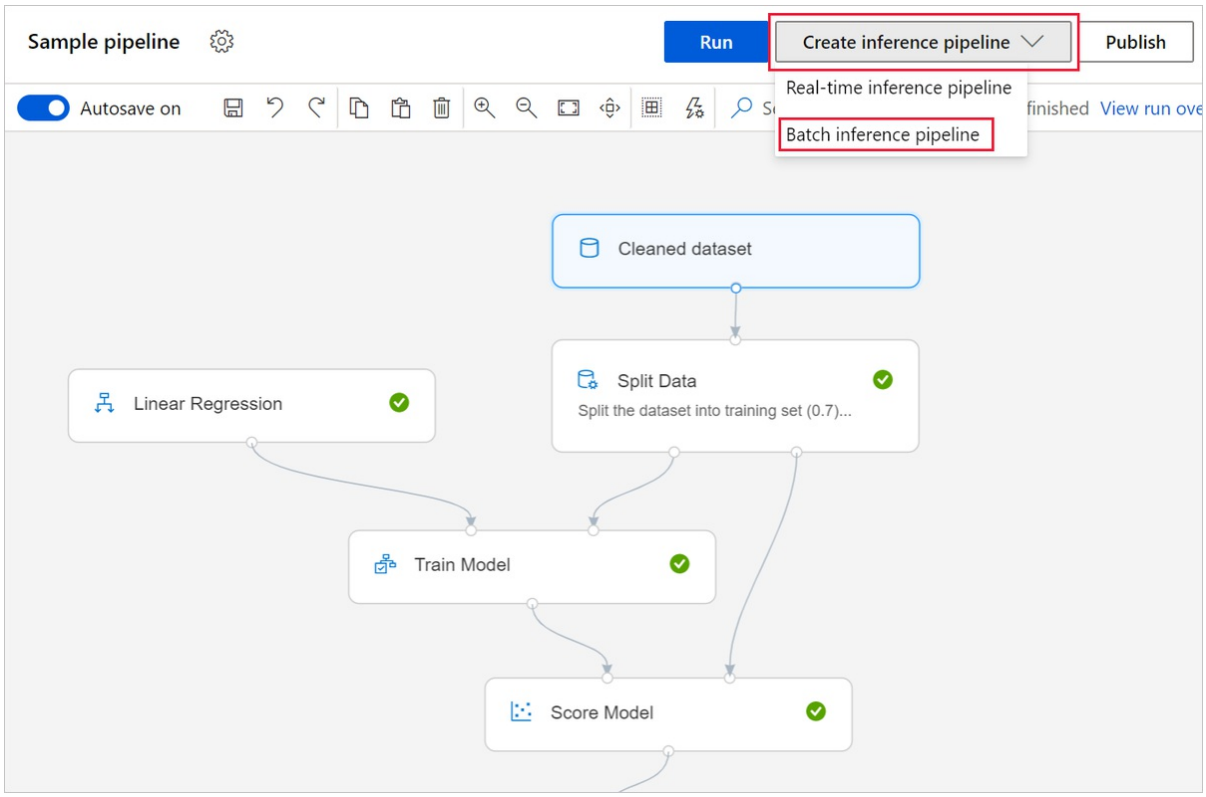
Your training pipeline must be run at least once to be able to create an inferencing pipeline.

1. Go to the **Designer** tab in your workspace.
2. Select the training pipeline that trains the model you want to use to make prediction.
3. **Submit** the pipeline.



Now that the training pipeline has been run, you can create a batch inference pipeline.

1. Next to **Submit**, select the new dropdown **Create inference pipeline**.
2. Select **Batch inference pipeline**.



The result is a default batch inference pipeline.

Add a pipeline parameter

To create predictions on new data, you can either manually connect a different dataset in this pipeline draft view or create a parameter for your dataset. Parameters let you change the behavior of the batch inferencing process at runtime.

In this section, you create a dataset parameter to specify a different dataset to make predictions on.

1. Select the dataset module.
2. A pane will appear to the right of the canvas. At the bottom of the pane, select **Set as pipeline parameter**.

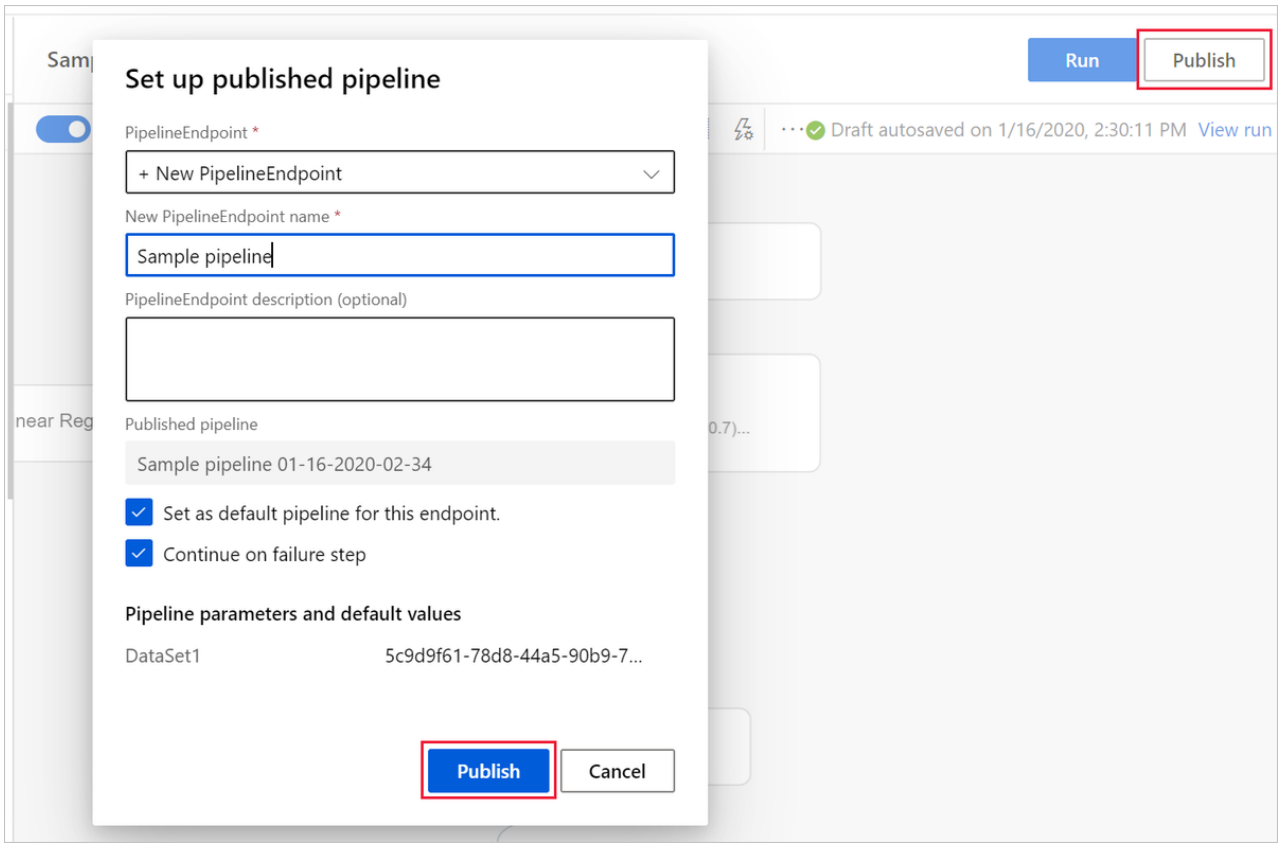
   Enter a name for the parameter, or accept the default value.

Publish your batch inferencing pipeline

Now you're ready to deploy the inferencing pipeline. This will deploy the pipeline and make it available for others to use.

1. Select the **Publish** button.
2. In the dialog that appears, expand the drop-down for **PipelineEndpoint**, and select **New PipelineEndpoint**.
3. Provide an endpoint name and optional description.

   Near the bottom of the dialog, you can see the parameter you configured with a default value of the dataset ID used during training.
4. Select **Publish**.



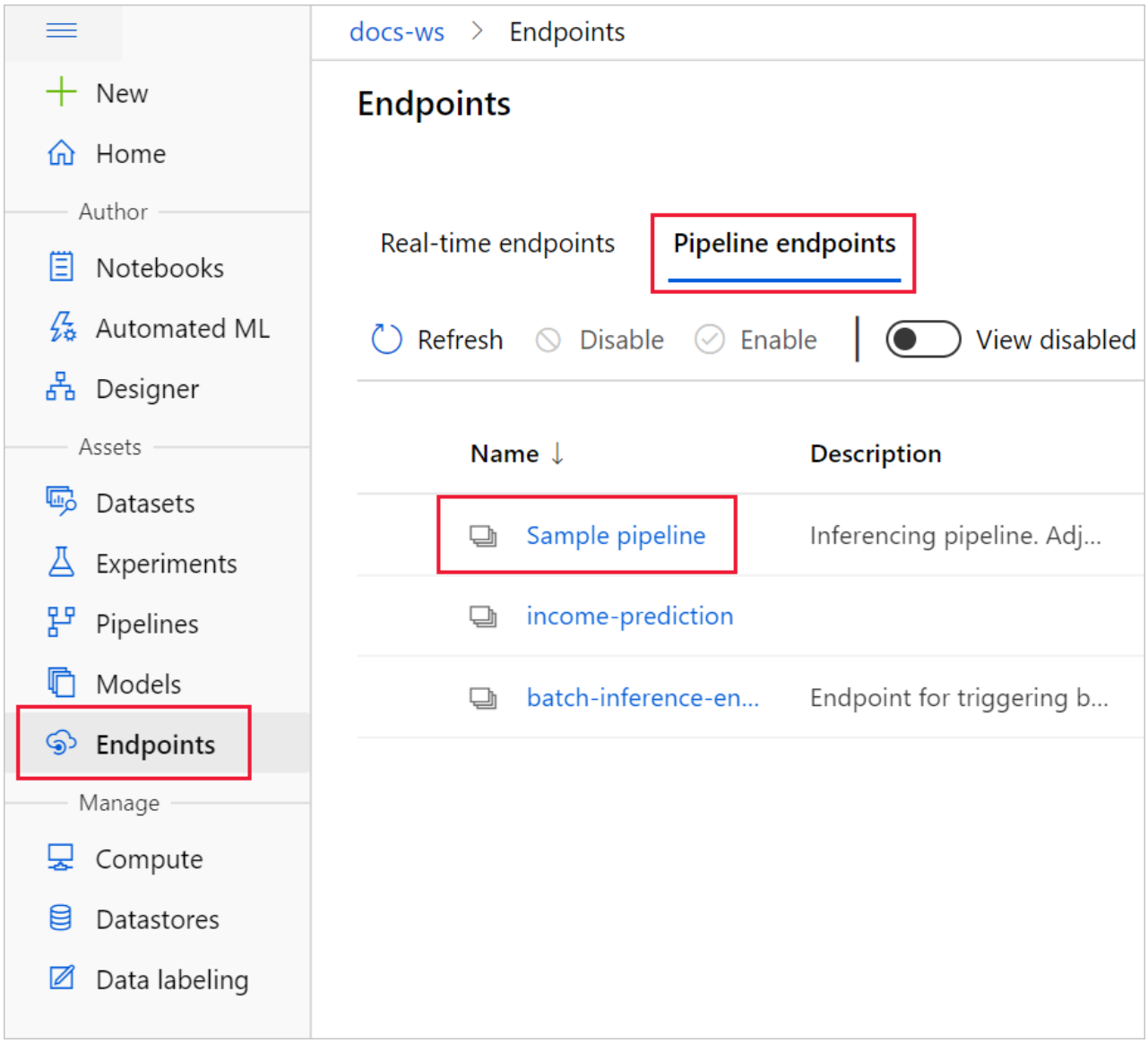
Consume an endpoint

Now, you have a published pipeline with a dataset parameter. The pipeline will use the trained model created in the training pipeline to score the dataset you provide as a parameter.

Submit a pipeline run

In this section, you will set up a manual pipeline run and alter the pipeline parameter to score new data.

1. After the deployment is complete, go to the **Endpoints** section.
2. Select **Pipeline endpoints**.
3. Select the name of the endpoint you created.



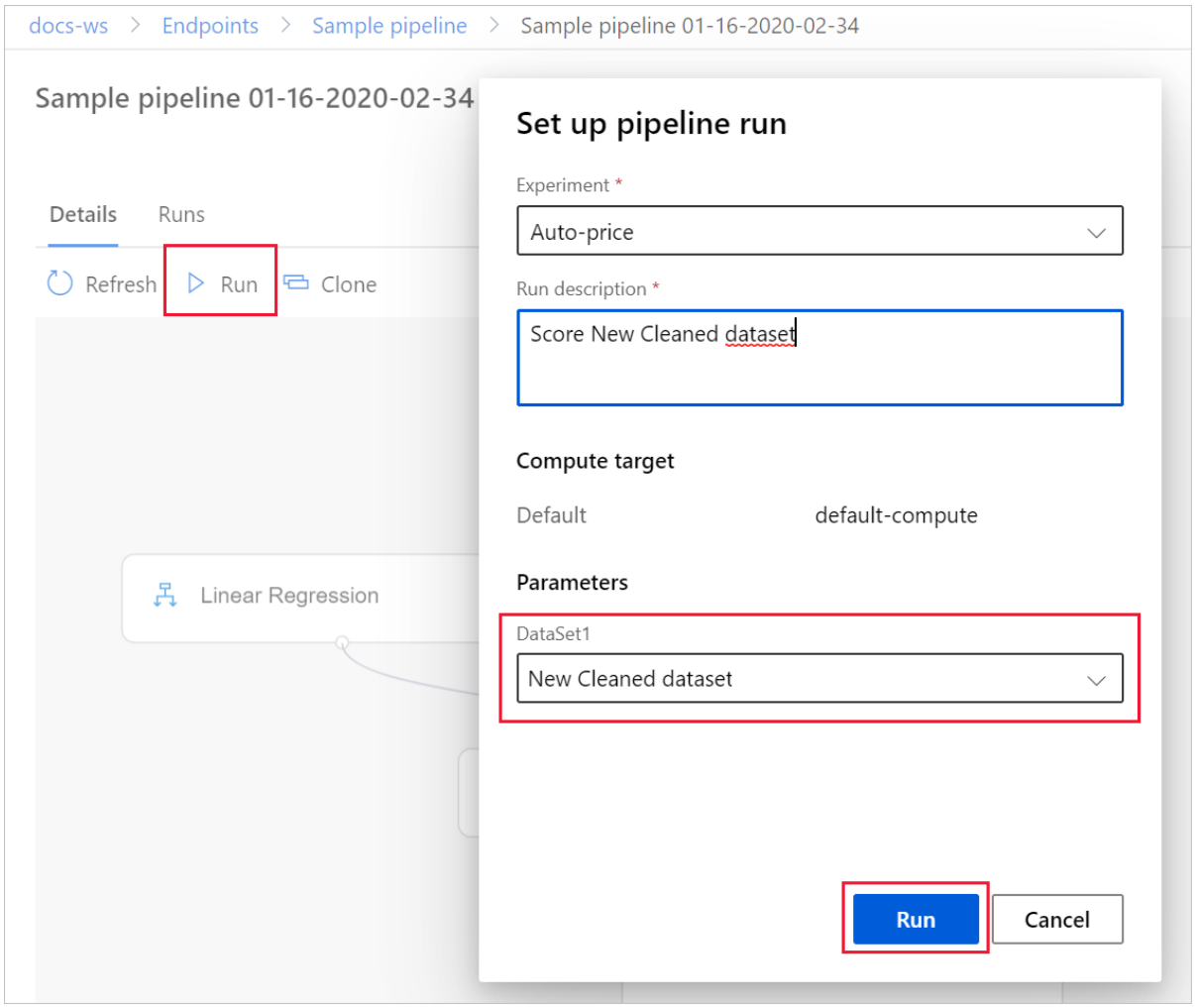
1. Select **Published pipelines**.

   This screen shows all published pipelines published under this endpoint.

2. Select the pipeline you published.

   The pipeline details page shows you a detailed run history and connection string information for your pipeline.

3. Select **Submit** to create a manual run of the pipeline.



4. Change the parameter to use a different dataset.

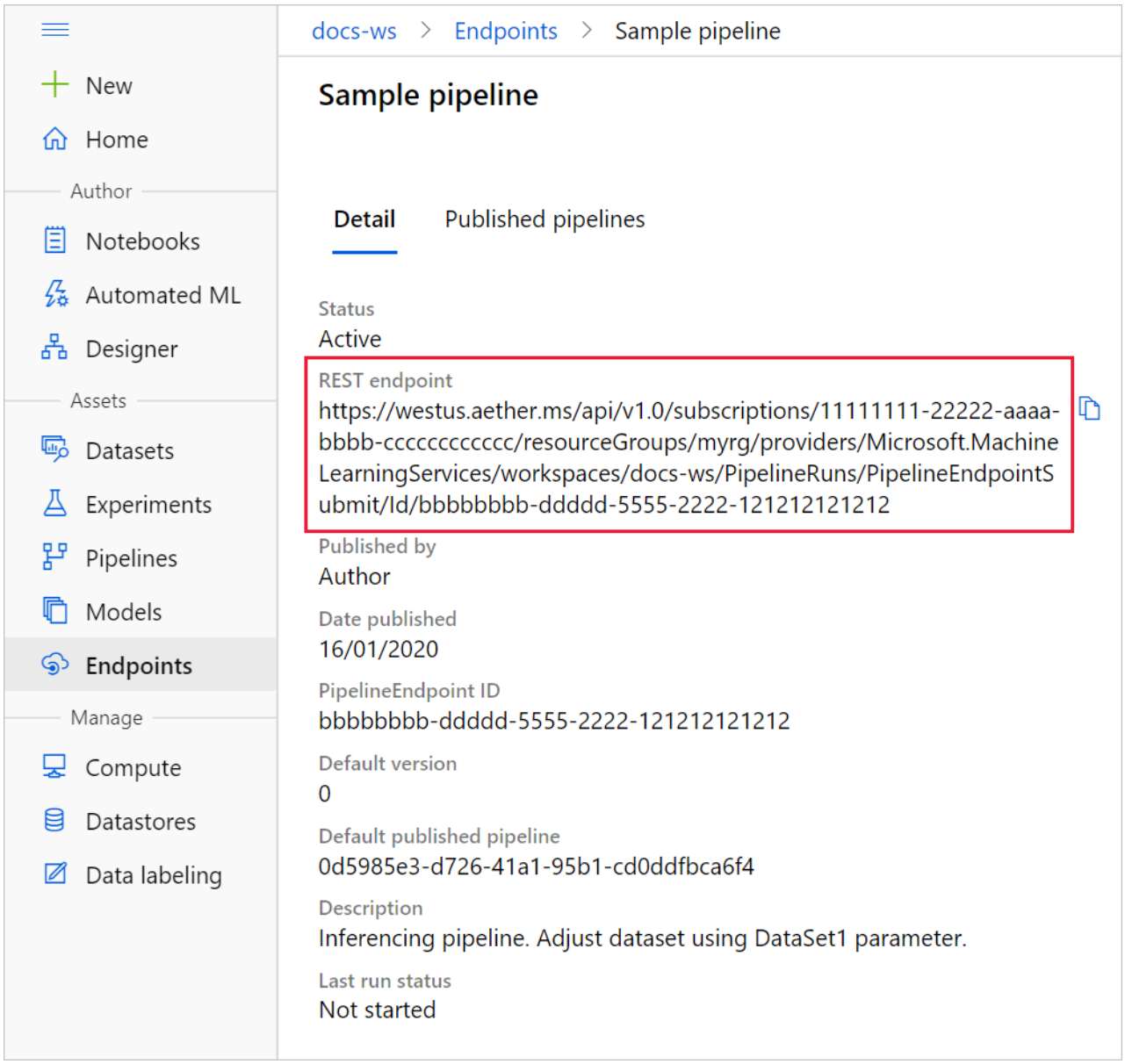
5. Select **Submit** to run the pipeline.

Use the REST endpoint

You can find information on how to consume pipeline endpoints and published pipeline in the **Endpoints** section.

You can find the REST endpoint of a pipeline endpoint in the run overview panel. By calling the endpoint, you are consuming its default published pipeline.

You can also consume a published pipeline in the **Published pipelines** page. Select a published pipeline and find the REST endpoint of it.

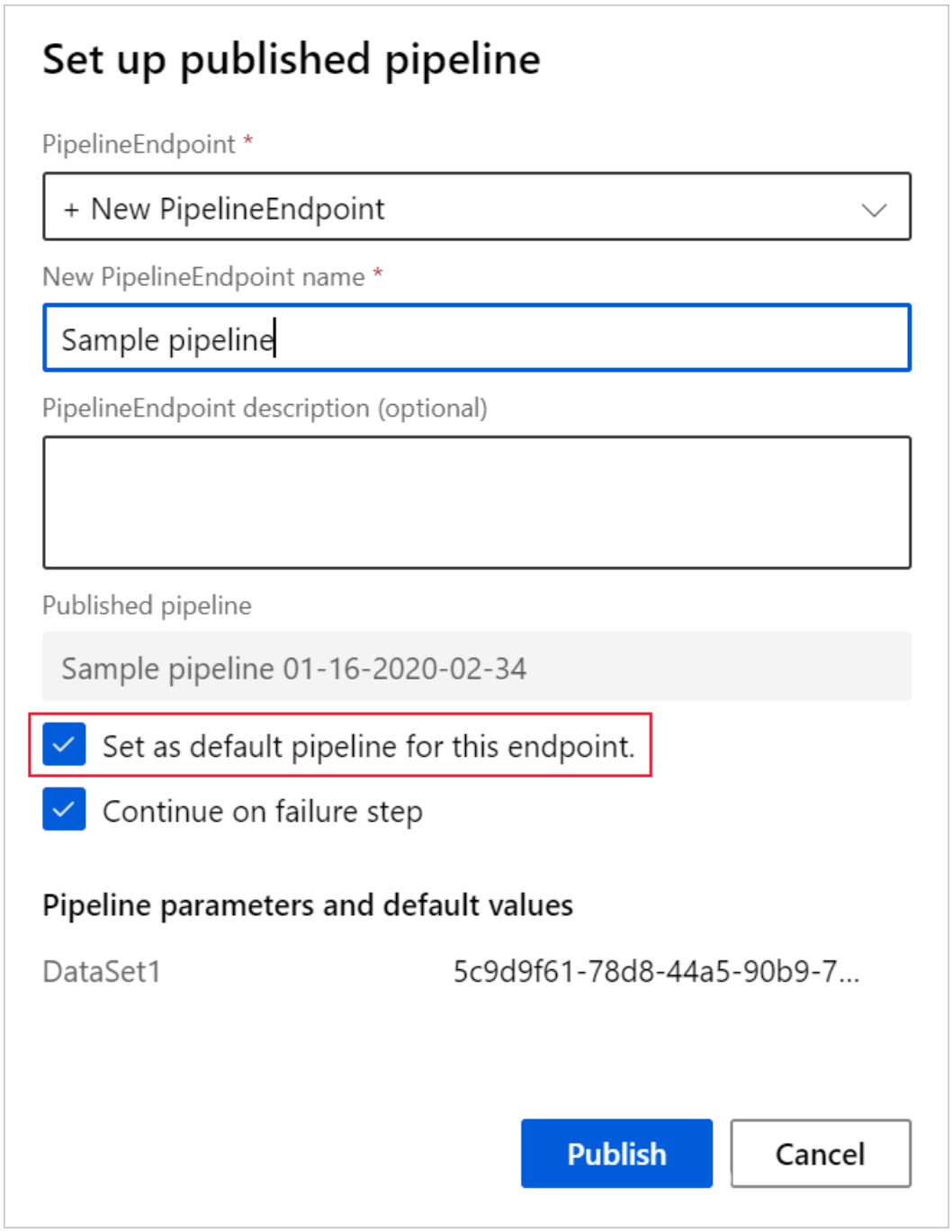


To make a REST call, you will need an OAuth 2.0 bearer-type authentication header. See the following tutorial section for more detail on setting up authentication to your workspace and making a parameterized REST call.

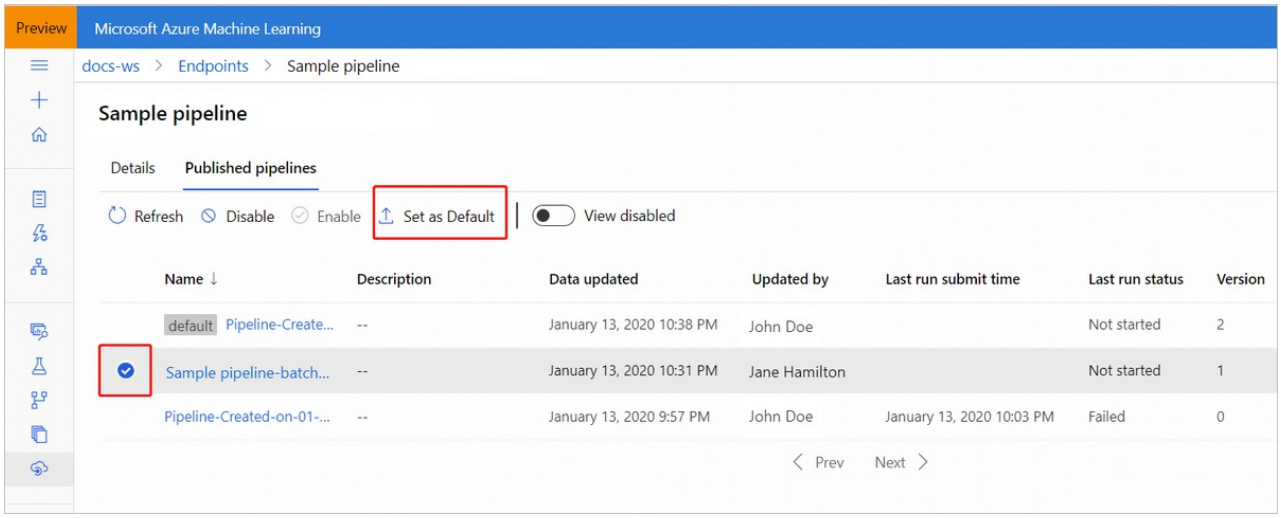
Versioning endpoints

The designer assigns a version to each subsequent pipeline that you publish to an endpoint. You can specify the pipeline version that you want to execute as a parameter in your REST call. If you don't specify a version number, the designer will use the default pipeline.

When you publish a pipeline, you can choose to make it the new default pipeline for that endpoint.



You can also set a new default pipeline in the **Published pipelines** tab of your endpoint.



Next steps

Follow the designer tutorial to train and deploy a regression model. ''

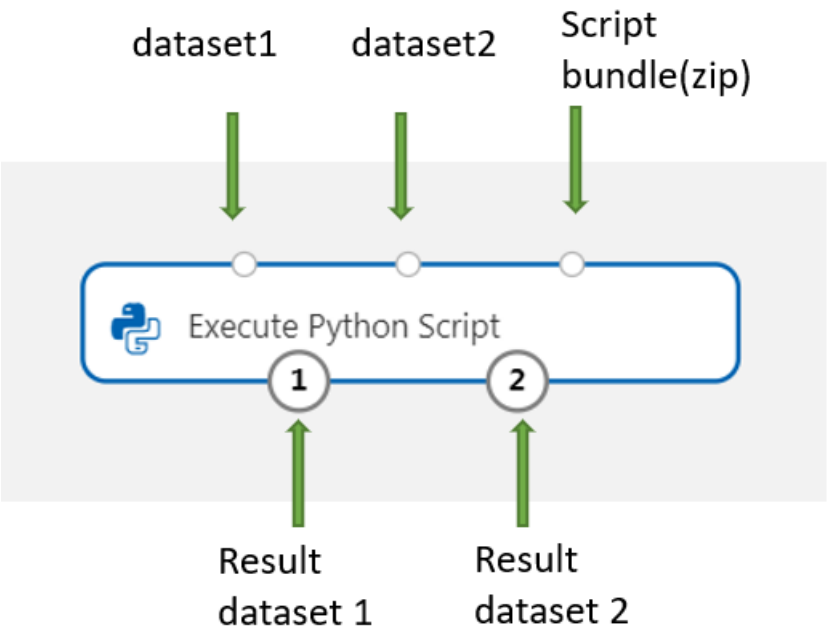
Execute Python code in Azure Machine Learning designer

3/17/2020 • 2 minutes to read • Edit Online

In this article, you learn how to use the Execute Python Script module to add custom logic to Azure Machine Learning designer. In the following how-to, you use the Pandas library to do simple feature engineering.

You can use the in-built code editor to quickly add simple Python logic. If you want to add more complex code or upload additional Python libraries, you should use the zip file method.

The default execution environment uses the Anacondas distribution of Python. For a complete list of pre-installed packages, see the Execute Python Script module reference page.



Execute Python written in the designer

Add the Execute Python Script module

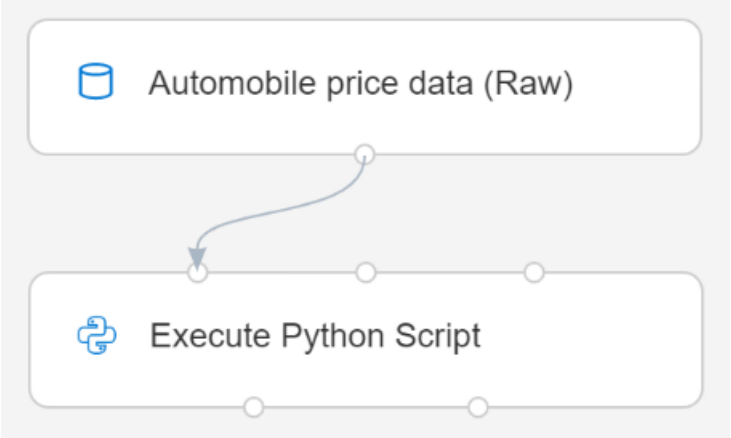
1. Find the **Execute Python Script** module in the designer palette. It can be found in the **Python Language** section.
2. Drag and drop the module onto the pipeline canvas.

Connect input datasets

This article uses the sample dataset, **Automobile price data (Raw)**.

1. Drag and drop your dataset to the pipeline canvas.
2. Connect the output port of the dataset to the top-left input port of the **Execute Python Script** module. The designer exposes the input as a parameter to the entry point script.

   The right input port is reserved for zipped python libraries.



3. Take note of which input port you use. The designer assigns the left input port to the variable `dataset1` and the middle input port to `dataset2`.

Input modules are optional since you can generate or import data directly in the **Execute Python Script** module.

Write your Python code

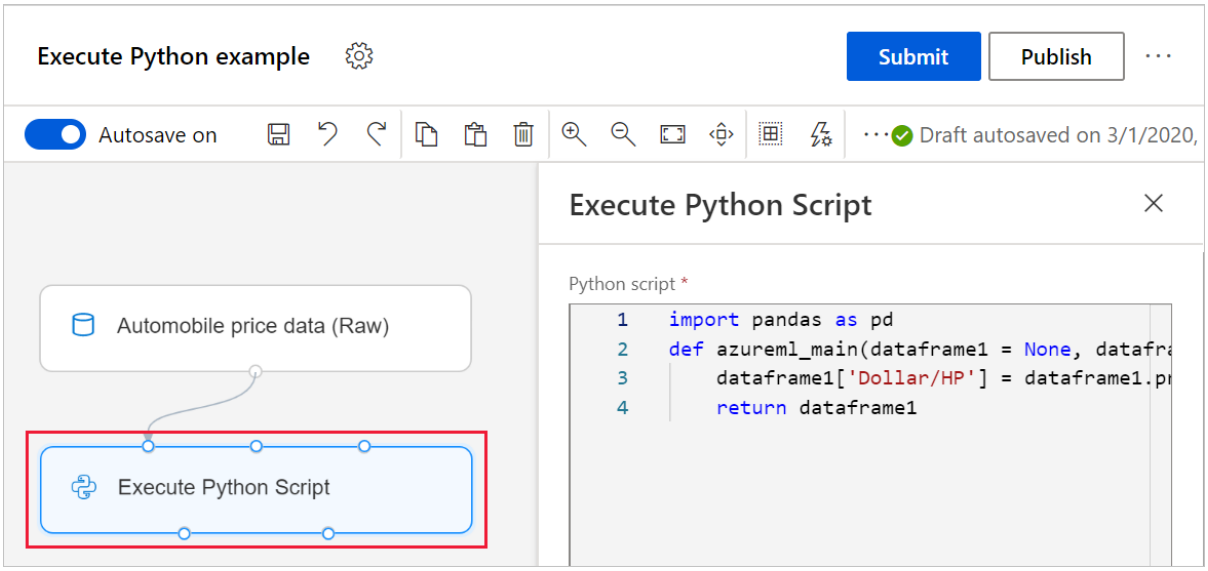
The designer provides an initial entry point script for you to edit and enter your own Python code.

In this example, you use Pandas to combine two columns found in the automobile dataset, **Price** and **Horsepower**, to create a new column, **Dollars per horsepower**. This column represents how much you pay for each horsepower, which could be a useful feature to decide if a car is a good deal for the money.

1. Select the **Execute Python Script** module.
2. In the pane that appears to the right of the canvas, select the **Python script** text box.
3. Copy and paste the following code into the text box.

```
import pandas as pd

def azureml_main(dataframe1 = None, dataframe2 = None):
    dataframe1['Dollar/HP'] = dataframe1.price / dataframe1.horsepower
    return dataframe1
```

Your pipeline should look the following image:



The entry point script must contain the function `azureml_main`. There are two function parameters that map to the two input ports for the **Execute Python Script** module.

The return value must be a Pandas Dataframe. You can return up to two dataframes as module outputs.

4. Submit the pipeline.

Now, you have a dataset with the new feature **Dollars/HP**, which could be useful in training a car recommender. This is an example of feature extraction and dimensionality reduction.

Next steps

Learn how to import your own data in Azure Machine Learning designer.

Run batch inference on large amounts of data by using Azure Machine Learning

3/17/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

Learn how to process large amounts of data asynchronously and in parallel by using Azure Machine Learning. The ParallelRunStep capability described here is in public preview. It's a high-performance and high-throughput way to generate inferences and processing data. It provides asynchronous capabilities out of the box.

With ParallelRunStep, it's straightforward to scale offline inferences to large clusters of machines on terabytes of production data resulting in improved productivity and optimized cost.

In this article, you learn the following tasks:

- Create a remote compute resource.
- Write a custom inference script.
- Create a machine learning pipeline to register a pre-trained image classification model based on the MNIST dataset.
- Use the model to run batch inference on sample images available in your Azure Blob storage account.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of the Azure Machine Learning.
- For a guided quickstart, complete the setup tutorial if you don't already have an Azure Machine Learning workspace or notebook virtual machine.
- To manage your own environment and dependencies, see the how-to guide on configuring your own environment. Run `pip install azureml-sdk[notebooks] azureml-pipeline-core azureml-contrib-pipeline-steps` in your environment to download the necessary dependencies.

Set up machine learning resources

The following actions set up the resources that you need to run a batch inference pipeline:

- Create a datastore that points to a blob container that has images to inference.
- Set up data references as inputs and outputs for the batch inference pipeline step.
- Set up a compute cluster to run the batch inference step.

Create a datastore with sample images

Get the MNIST evaluation set from the public blob container `sampledata` on an account named `pipelinedata`. Create a datastore with the name `mnist_datastore`, which points to this container. In the following call to `register_azure_blob_container`, setting the `overwrite` flag to `True` overwrites any datastore that was created previously with that name.

You can change this step to point to your blob container by providing your own values for `datastore_name`, `container_name`, and `account_name`.

```
from azureml.core import Datastore
from azureml.core import Workspace

# Load workspace authorization details from config.json
ws = Workspace.from_config()

mnist_blob = Datastore.register_azure_blob_container(ws,
                      datastore_name="mnist_datastore",
                      container_name="sampledata",
                      account_name="pipelinedata",
                      overwrite=True)
```

Next, specify the workspace default datastore as the output datastore. You'll use it for inference output.

When you create your workspace, Azure Files and Blob storage are attached to the workspace by default. Azure Files is the default datastore for a workspace, but you can also use Blob storage as a datastore. For more information, see Azure storage options.

```
def_data_store = ws.get_default_datastore()
```

Configure data inputs and outputs

Now you need to configure data inputs and outputs, including:

- The directory that contains the input images.
- The directory where the pre-trained model is stored.
- The directory that contains the labels.
- The directory for output.

`Dataset` is a class for exploring, transforming, and managing data in Azure Machine Learning. This class has two types: `TabularDataset` and `FileDataset`. In this example, you'll use `FileDataset` as the inputs to the batch inference pipeline step.

> **NOTE**
>
> `FileDataset` support in batch inference is restricted to Azure Blob storage for now.

You can also reference other datasets in your custom inference script. For example, you can use it to access labels in your script for labeling images by using `Dataset.register` and `Dataset.get_by_name`.

For more information about Azure Machine Learning datasets, see Create and access datasets (preview).

`PipelineData` objects are used for transferring intermediate data between pipeline steps. In this example, you use it for inference outputs.

```
from azureml.core.dataset import Dataset

mnist_ds_name = 'mnist_sample_data'

path_on_datastore = mnist_blob.path('mnist/')
input_mnist_ds = Dataset.File.from_files(path=path_on_datastore, validate=False)
registered_mnist_ds = input_mnist_ds.register(ws, mnist_ds_name, create_new_version=True)
named_mnist_ds = registered_mnist_ds.as_named_input(mnist_ds_name)

output_dir = PipelineData(name="inferences",
                          datastore=def_data_store,
                          output_path_on_compute="mnist/results")
```

Set up a compute target

In Azure Machine Learning, *compute* (or *compute target*) refers to the machines or clusters that perform the computational steps in your machine learning pipeline. Run the following code to create a CPU based AmlCompute target.

```
from azureml.core.compute import AmlCompute, ComputeTarget
from azureml.core.compute_target import ComputeTargetException

# choose a name for your cluster
compute_name = os.environ.get("AML_COMPUTE_CLUSTER_NAME", "cpu-cluster")
compute_min_nodes = os.environ.get("AML_COMPUTE_CLUSTER_MIN_NODES", 0)
compute_max_nodes = os.environ.get("AML_COMPUTE_CLUSTER_MAX_NODES", 4)

# This example uses CPU VM. For using GPU VM, set SKU to STANDARD_NC6
vm_size = os.environ.get("AML_COMPUTE_CLUSTER_SKU", "STANDARD_D2_V2")


if compute_name in ws.compute_targets:
    compute_target = ws.compute_targets[compute_name]
    if compute_target and type(compute_target) is AmlCompute:
        print('found compute target. just use it. ' + compute_name)
else:
    print('creating a new compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size = vm_size,
                                                                 min_nodes = compute_min_nodes,
                                                                 max_nodes = compute_max_nodes)

    # create the cluster
    compute_target = ComputeTarget.create(ws, compute_name, provisioning_config)

    # can poll for a minimum number of nodes and for a specific timeout.
    # if no min node count is provided it will use the scale settings for the cluster
    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

     # For a more detailed view of current AmlCompute status, use get_status()
    print(compute_target.get_status().serialize())
```

Prepare the model

Download the pre-trained image classification model, and then extract it to the `models` directory.

```
import os
import tarfile
import urllib.request

model_dir = 'models'
if not os.path.isdir(model_dir):
    os.mkdir(model_dir)

url="https://pipelinedata.blob.core.windows.net/mnist-model/mnist-tf.tar.gz"
response = urllib.request.urlretrieve(url, "model.tar.gz")
tar = tarfile.open("model.tar.gz", "r:gz")
tar.extractall(model_dir)
```

Then register the model with your workspace so it's available to your remote compute resource.

```
from azureml.core.model import Model

# Register the downloaded model
model = Model.register(model_path="models/",
                       model_name="mnist",
                       tags={'pretrained': "mnist"},
                       description="Mnist trained tensorflow model",
                       workspace=ws)
```

Write your inference script

> **WARNING**
>
> The following code is only a sample that the sample notebook uses. You'll need to create your own script for your scenario.

The script *must contain* two functions:

- `init()`: Use this function for any costly or common preparation for later inference. For example, use it to load the model into a global object. This function will be called only once at beginning of process.
- `run(mini_batch)`: The function will run for each `mini_batch` instance.
  - `mini_batch`: Parallel run step will invoke run method and pass either a list or Pandas DataFrame as an argument to the method. Each entry in min_batch will be - a file path if input is a FileDataset, a Pandas DataFrame if input is a TabularDataset.
  - `response`: run() method should return a Pandas DataFrame or an array. For append_row output_action, these returned elements are appended into the common output file. For summary_only, the contents of the elements are ignored. For all output actions, each returned output element indicates one successful run of input element in the input mini-batch. You should make sure that enough data is included in run result to map input to run result. Run output will be written in output file and not guaranteed to be in order, you should use some key in the output to map it to input.

```
# Snippets from a sample script.
# Refer to the accompanying digit_identification.py
# (https://aka.ms/batch-inference-notebooks)
# for the implementation script.

import os
import numpy as np
import tensorflow as tf
from PIL import Image
from azureml.core import Model


def init():
    global g_tf_sess

    # Pull down the model from the workspace
    model_path = Model.get_model_path("mnist")

    # Construct a graph to execute
    tf.reset_default_graph()
    saver = tf.train.import_meta_graph(os.path.join(model_path, 'mnist-tf.model.meta'))
    g_tf_sess = tf.Session()
    saver.restore(g_tf_sess, os.path.join(model_path, 'mnist-tf.model'))


def run(mini_batch):
    print(f'run method start: {__file__}, run({mini_batch})')
    resultList = []
    in_tensor = g_tf_sess.graph.get_tensor_by_name("network/X:0")
    output = g_tf_sess.graph.get_tensor_by_name("network/output/MatMul:0")

    for image in mini_batch:
        # Prepare each image
        data = Image.open(image)
        np_im = np.array(data).reshape((1, 784))
        # Perform inference
        inference_result = output.eval(feed_dict={in_tensor: np_im}, session=g_tf_sess)
        # Find the best probability, and add it to the result list
        best_result = np.argmax(inference_result)
        resultList.append("{}: {}".format(os.path.basename(image), best_result))

    return resultList
```

How to access other files in source directory in entry_script

If you have another file or folder in the same directory as your entry script, you can reference it by finding the current working directory.

```
script_dir = os.path.realpath(os.path.join(__file__, '..',))
file_path = os.path.join(script_dir, "<file_name>")
```

Build and run the pipeline containing ParallelRunStep

Now you have everything you need to build the pipeline.

Prepare the run environment

First, specify the dependencies for your script. You use this object later when you create the pipeline step.

```
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import DEFAULT_GPU_IMAGE

batch_conda_deps = CondaDependencies.create(pip_packages=["tensorflow==1.13.1", "pillow"])

batch_env = Environment(name="batch_environment")
batch_env.python.conda_dependencies = batch_conda_deps
batch_env.docker.enabled = True
batch_env.docker.base_image = DEFAULT_GPU_IMAGE
batch_env.spark.precache_packages = False
```

Specify the parameters for your batch inference pipeline step

`ParallelRunConfig` is the major configuration for the newly introduced batch inference `ParallelRunStep` instance within the Azure Machine Learning pipeline. You use it to wrap your script and configure necessary parameters, including all of the following parameters:

- `entry_script`: A user script as a local file path that will be run in parallel on multiple nodes. If `source_directory` is present, use a relative path. Otherwise, use any path that's accessible on the machine.
- `mini_batch_size`: The size of the mini-batch passed to a single `run()` call. (optional; the default value is `10` files for FileDataset and `1MB` for TabularDataset.)
  - For `FileDataset`, it's the number of files with a minimum value of `1`. You can combine multiple files into one mini-batch.
  - For `TabularDataset`, it's the size of data. Example values are `1024`, `1024KB`, `10MB`, and `1GB`. The recommended value is `1MB`. The mini-batch from `TabularDataset` will never cross file boundaries. For example, if you have .csv files with various sizes, the smallest file is 100 KB and the largest is 10 MB. If you set `mini_batch_size = 1MB`, then files with a size smaller than 1 MB will be treated as one mini-batch. Files with a size larger than 1 MB will be split into multiple mini-batches.
- `error_threshold`: The number of record failures for `TabularDataset` and file failures for `FileDataset` that should be ignored during processing. If the error count for the entire input goes above this value, the job will be aborted. The error threshold is for the entire input and not for individual mini-batches sent to the `run()` method. The range is `[-1, int.max]`. The `-1` part indicates ignoring all failures during processing.
- `output_action`: One of the following values indicates how the output will be organized:
  - `summary_only`: The user script will store the output. `ParallelRunStep` will use the output only for the error threshold calculation.
  - `append_row`: For all input files, only one file will be created in the output folder to append all outputs separated by line. The file name will be `parallel_run_step.txt`.
- `source_directory`: Paths to folders that contain all files to execute on the compute target (optional).
- `compute_target`: Only `AmlCompute` is supported.
- `node_count`: The number of compute nodes to be used for running the user script.
- `process_count_per_node`: The number of processes per node.
- `environment`: The Python environment definition. You can configure it to use an existing Python environment or to set up a temporary environment for the experiment. The definition is also responsible for setting the required application dependencies (optional).
- `logging_level`: Log verbosity. Values in increasing verbosity are: `WARNING`, `INFO`, and `DEBUG`. (optional; the default value is `INFO`)
- `run_invocation_timeout`: The `run()` method invocation timeout in seconds. (optional; default value is `60`)

```
from azureml.contrib.pipeline.steps import ParallelRunConfig

parallel_run_config = ParallelRunConfig(
    source_directory=scripts_folder,
    entry_script="digit_identification.py",
    mini_batch_size="5",
    error_threshold=10,
    output_action="append_row",
    environment=batch_env,
    compute_target=compute_target,
    node_count=4)
```

Create the pipeline step

Create the pipeline step by using the script, environment configuration, and parameters. Specify the compute target that you already attached to your workspace as the target of execution for the script. Use `ParallelRunStep` to create the batch inference pipeline step, which takes all the following parameters:

- `name`: The name of the step, with the following naming restrictions: unique, 3-32 characters, and regex ^[a-z]([-a-z0-9]*[a-z0-9])?$.
- `models`: Zero or more model names already registered in the Azure Machine Learning model registry.
- `parallel_run_config`: A `ParallelRunConfig` object, as defined earlier.
- `inputs`: One or more single-typed Azure Machine Learning datasets.
- `output`: A `PipelineData` object that corresponds to the output directory.
- `arguments`: A list of arguments passed to the user script (optional).
- `allow_reuse`: Whether the step should reuse previous results when run with the same settings/inputs. If this parameter is `False`, a new run will always be generated for this step during pipeline execution. (optional; the default value is `True`.)

```
from azureml.contrib.pipeline.steps import ParallelRunStep

parallelrun_step = ParallelRunStep(
    name="batch-mnist",
    models=[model],
    parallel_run_config=parallel_run_config,
    inputs=[named_mnist_ds],
    output=output_dir,
    arguments=[],
    allow_reuse=True
)
```

> **NOTE**
>
> The above step depends on `azureml-contrib-pipeline-steps`, as described in Prerequisites.

Submit the pipeline

Now, run the pipeline. First, create a `Pipeline` object by using your workspace reference and the pipeline step that you created. The `steps` parameter is an array of steps. In this case, there's only one step for batch scoring. To build pipelines that have multiple steps, place the steps in order in this array.

Next, use the `Experiment.submit()` function to submit the pipeline for execution.

```
from azureml.pipeline.core import Pipeline
from azureml.core.experiment import Experiment

pipeline = Pipeline(workspace=ws, steps=[parallelrun_step])
pipeline_run = Experiment(ws, 'digit_identification').submit(pipeline)
```

Monitor the parallel run job

A batch inference job can take a long time to finish. This example monitors progress by using a Jupyter widget. You can also manage the job's progress by using:

- Azure Machine Learning Studio.
- Console output from the `PipelineRun` object.

```
from azureml.widgets import RunDetails
RunDetails(pipeline_run).show()

pipeline_run.wait_for_completion(show_output=True)
```

Next steps

To see this process working end to end, try the batch inference notebook.

For debugging and troubleshooting guidance for ParallelRunStep, see the how-to guide.

For debugging and troubleshooting guidance for pipelines, see the how-to guide.

Learn how to run notebooks by following the article Use Jupyter notebooks to explore this service.

Debug and troubleshoot ParallelRunStep

1/16/2020 • 3 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In this article, you learn how to debug and troubleshoot the ParallelRunStep class from the Azure Machine Learning SDK.

Testing scripts locally

See the Testing scripts locally section for machine learning pipelines. Your ParallelRunStep runs as a step in ML pipelines so the same answer applies to both.

Debugging scripts from remote context

The transition from debugging a scoring script locally to debugging a scoring script in an actual pipeline can be a difficult leap. For information on finding your logs in the portal, the machine learning pipelines section on debugging scripts from a remote context. The information in that section also applies to a parallel step run.

For example, the log file `70_driver_log.txt` contains information from the controller that launches parallel run step code.

Because of the distributed nature of parallel run jobs, there are logs from several different sources. However, two consolidated files are created that provide high-level information:

- `~/logs/overview.txt`: This file provides a high-level info about the number of mini-batches (also known as tasks) created so far and number of mini-batches processed so far. At this end, it shows the result of the job. If the job failed, it will show the error message and where to start the troubleshooting.
- `~/logs/sys/master.txt`: This file provides the master node (also known as the orchestrator) view of the running job. Includes task creation, progress monitoring, the run result.

Logs generated from entry script using EntryScript.logger and print statements will be found in following files:

- `~/logs/user/<ip_address>/Process-*.txt`: This file contains logs written from entry_script using EntryScript.logger. It also contains print statement (stdout) from entry_script.

When you need a full understanding of how each node executed the score script, look at the individual process logs for each node. The process logs can be found in the `sys/worker` folder, grouped by worker nodes:

- `~/logs/sys/worker/<ip_address>/Process-*.txt`: This file provides detailed info about each mini-batch as it is picked up or completed by a worker. For each mini-batch, this file includes:
  - The IP address and the PID of the worker process.
  - The total number of items, successfully processed items count, and failed item count.
  - The start time, duration, process time and run method time.

You can also find information on the resource usage of the processes for each worker. This information is in CSV format and is located at `~/logs/sys/perf/<ip_address>/`. For a single node, job files will be available under `~logs/sys/perf`. For example, when checking for resource utilization, look at the following files:

- `Process-*.csv`: Per worker process resource usage.
- `sys.csv`: Per node log.

How do I log from my user script from a remote context?

You can get a logger from EntryScript as shown in below sample code to make the logs show up in **logs/user** folder in the portal.

A sample entry script using the logger:

```
from entry_script import EntryScript

def init():
    """ Initialize the node."""
    entry_script = EntryScript()
    logger = entry_script.logger
    logger.debug("This will show up in files under logs/user on the Azure portal.")


def run(mini_batch):
    """ Accept and return the list back."""
    # This class is in singleton pattern and will return same instance as the one in init()
    entry_script = EntryScript()
    logger = entry_script.logger
    logger.debug(f"{__file__}: {mini_batch}.")
    ...

    return mini_batch
```

How could I pass a side input such as, a file or file(s) containing a lookup table, to all my workers?

Construct a Dataset object containing the side input and register with your workspace. After that you can access it in your inference script (for example, in your init() method) as follows:

```
from azureml.core.run import Run
from azureml.core.dataset import Dataset

ws = Run.get_context().experiment.workspace
lookup_ds = Dataset.get_by_name(ws, "<registered-name>")
lookup_ds.download(target_path='.', overwrite=True)
```

Next steps

- See the SDK reference for help with the azureml-contrib-pipeline-step package and the documentation for ParallelRunStep class.
- Follow the advanced tutorial on using pipelines with parallel run step.

Manage and request quotas for Azure resources

4/6/2020 • 10 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

This article provides details on preconfigured limits on Azure resources for your subscription. Also included are instructions on how to request quota enhancements for each type of resource. These limits are put in place to prevent budget over-runs due to fraud, and to honor Azure capacity constraints.

As with other Azure services, there are limits on certain resources associated with Azure Machine Learning. These limits range from a cap on the number of workspaces to limits on the actual underlying compute that gets used for model training or inference/scoring.

As you design and scale your Azure Machine Learning resources for production workloads, consider these limits. For example, if your cluster doesn't reach the target number of nodes, then you may have reached an Azure Machine Learning Compute cores limit for your subscription. If you want to raise the limit or quota above the Default Limit, open an online customer support request at no charge. The limits can't be raised above the Maximum Limit value shown in the following tables due to Azure Capacity constraints. If there is no Maximum Limit column, then the resource doesn't have adjustable limits.

Special considerations

- A quota is a credit limit, not a capacity guarantee. If you have large-scale capacity needs, contact Azure support.
- Your quota is shared across all the services in your subscriptions including Azure Machine Learning. The only exception is Azure Machine Learning compute which has a separate quota from the core compute quota. Be sure to calculate the quota usage across all services when evaluating your capacity needs.
- Default limits vary by offer Category Type, such as Free Trial, Pay-As-You-Go, and VM series, such as Dv2, F, G, and so on.

Default resource quotas

Here is a breakdown of the quota limits by various resource types within your Azure subscription.

> **IMPORTANT**
> Limits are subject to change. The latest can always be found at the service-level quota document for all of Azure.

Virtual machines

For each Azure subscription, there is a limit on the number of virtual machines you can have across your services or standalone. This limit is at the region level both on the total cores and also on a per family basis.

Virtual machine cores have a regional total limit and a regional per size series (Dv2, F, etc.) limit, both of which are separately enforced. For example, consider a subscription with a US East total VM core limit of 30, an A series core limit of 30, and a D series core limit of 30. This subscription would be allowed to deploy 30 A1 VMs, or 30 D1 VMs, or a combination of the two not to exceed a total of 30 cores (for example, 10 A1 VMs and 20 D1 VMs).

RESOURCE	LIMIT
Subscriptions per Azure Active Directory tenant	Unlimited.
Coadministrators per subscription	Unlimited.
Resource groups per subscription	980
Azure Resource Manager API request size	4,194,304 bytes.
Tags per subscription[1]	50
Unique tag calculations per subscription[1]	10,000
Subscription-level deployments per location	800[2]

[1]You can apply up to 50 tags directly to a subscription. However, the subscription can contain an unlimited number of tags that are applied to resource groups and resources within the subscription. The number of tags per resource or resource group is limited to 50. Resource Manager returns a list of unique tag name and values in the subscription only when the number of tags is 10,000 or less. You still can find a resource by tag when the number exceeds 10,000.

[2]If you reach the limit of 800 deployments, delete deployments from the history that are no longer needed. To delete subscription level deployments, use Remove-AzDeployment or az deployment sub delete.

For a more detailed and up-to-date list of quota limits, check the Azure-wide quota article here.

Azure Machine Learning Compute

For Azure Machine Learning Compute, there is a default quota limit on both the number of cores and number of unique compute resources allowed per region in a subscription. This quota is separate from the VM core quota above and the core limits are not shared between the two resource types since AmlCompute is a managed service that deploys resources in a hosted-on-behalf-of model.

Available resources:

- Dedicated cores per region have a default limit of 24 - 300 depending on your subscription offer type with higher defaults for EA and CSP offer types. The number of dedicated cores per subscription can be increased and is different for each VM family. Certain specialized VM families like NCv2, NCv3, or ND series start with a default of zero cores. Contact Azure support by raising a quota request to discuss increase options.
- Low-priority cores per region have a default limit of 100 - 3000 depending on your subscription offer type with higher defaults for EA and CSP offer types. The number of low-priority cores per subscription can be increased and is a single value across VM families. Contact Azure support to discuss increase options.
- Clusters per region have a default limit of 200. These are shared between a training cluster and a compute instance (which is considered as a single node cluster for quota purposes). Contact Azure support if you want to request an increase beyond this limit.
- There are other strict limits that cannot be exceeded once hit.

RESOURCE	MAXIMUM LIMIT
Maximum workspaces per resource group	800

RESOURCE	MAXIMUM LIMIT
Maximum nodes in a single Azure Machine Learning Compute (AmlCompute) resource	100 nodes
Maximum GPU MPI processes per node	1-4
Maximum GPU workers per node	1-4
Maximum job lifetime	90 days[1]
Maximum job lifetime on a Low-Priority Node	7 days[2]
Maximum parameter servers per node	1

[1] The maximum lifetime refers to the time that a run start and when it finishes. Completed runs persist indefinitely; data for runs not completed within the maximum lifetime is not accessible. [2] Jobs on a Low-Priority node could be preempted anytime there is a capacity constraint. We recommend you implement checkpointing in your job.

Azure Machine Learning Pipelines

For Azure Machine Learning Pipelines, there is a quota limit on the number of steps in a pipeline and on the number of schedule-based runs of published pipelines per region in a subscription.

- Maximum number of steps allowed in a pipeline is 30,000
- Maximum number of the sum of schedule-based runs and blob pulls for blog-triggered schedules of published pipelines per subscription per month is 100,000

> **NOTE**
> If you want to increase this limit, contact Microsoft Support.

Container instances

There is also a limit on the number of container instances that you can spin up in a given time period (scoped hourly) or across your entire subscription.

RESOURCE	LIMIT
Standard sku container groups per region per subscription	100[1]
Dedicated sku container groups per region per subscription	0[1]
Number of containers per container group	60
Number of volumes per container group	20
Ports per IP	5
Container instance log size - running instance	4 MB
Container instance log size - stopped instance	16 KB or 1,000 lines
Container creates per hour	300[1]

RESOURCE	LIMIT
Container creates per 5 minutes	100[1]
Container deletes per hour	300[1]
Container deletes per 5 minutes	100[1]

[1]To request a limit increase, create an Azure Support request.

For a more detailed and up-to-date list of quota limits, check the Azure-wide quota article here.

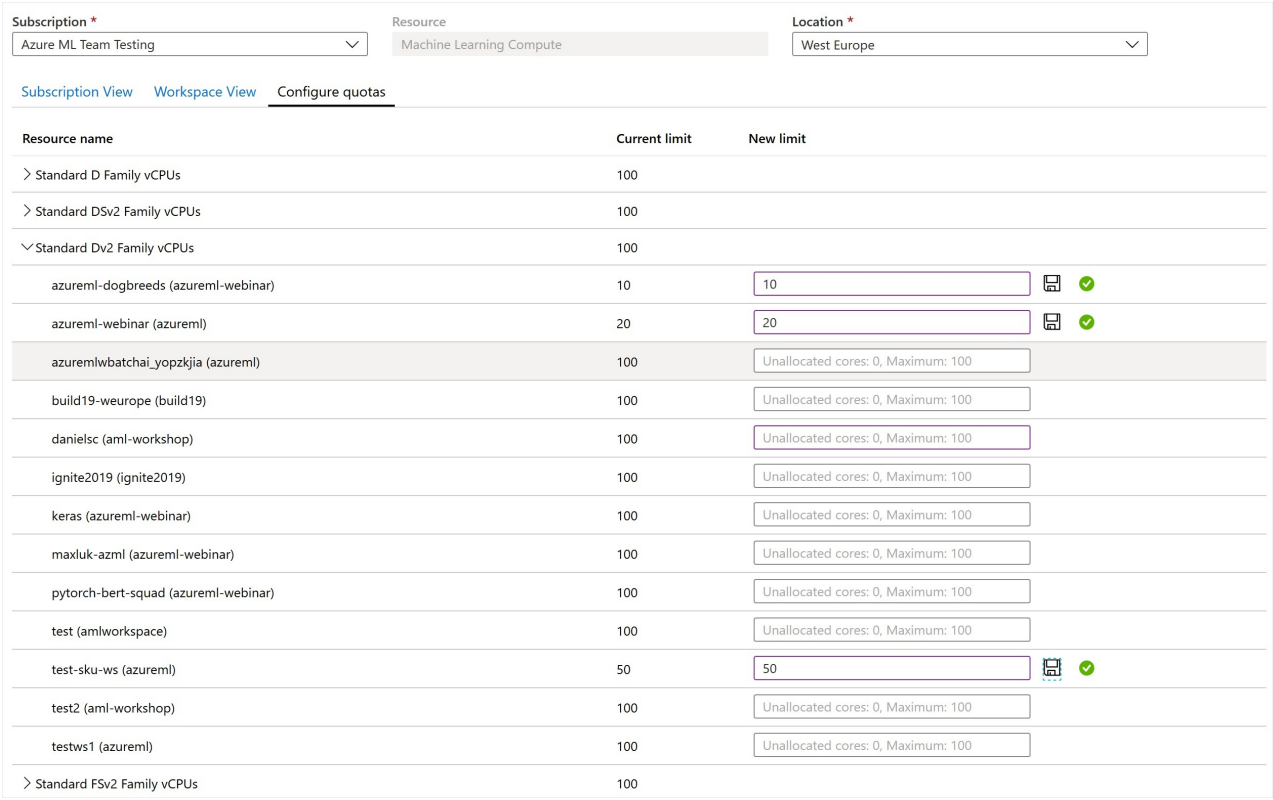
Storage

There is a limit on the number of storage accounts per region as well in a given subscription. The default limit is 250 and includes both Standard and Premium Storage accounts. If you require more than 250 storage accounts in a given region, make a request through Azure Support. The Azure Storage team will review your business case and may approve up to 250 storage accounts for a given region.

Workspace level quota

To better manage resource allocations for Amlcompute between various workspaces, we have introduced a feature that allows you to distribute subscription level quotas (by VM family) and configure them at the workspace level. The default behavior is that all workspaces have the same quota as the subscription level quota for any VM family. However, as the number of workspaces increases, and workloads of varying priority start sharing the same resources, users want a way to better share capacity and avoid resource contention issues. Azure Machine Learning provides a solution with its managed compute offering by allowing users to set a maximum quota for a particular VM family on each workspace. This is analogous to distributing your capacity between workspaces, and the users can choose to also over-allocate to drive maximum utilization.

To set quotas at the workspace level, go to any workspace in your subscription, and click on **Usages + quotas** in the left pane. Then select the **Configure quotas** tab to view the quotas, expand any VM family, and set a quota limit on any workspace listed under that VM family. Remember that you cannot set a negative value or a value higher than the subscription level quota. Also, as you would observe, by default all workspaces are assigned the entire subscription quota to allow for full utilization of the allocated quota.



NOTE

This is an Enterprise edition feature only. If you have both a Basic and an Enterprise edition workspace in your subscription, you can use this to only set quotas on your Enterprise workspaces. Your Basic workspaces will continue to have the subscription level quota which is the default behavior.

You need subscription level permissions to set quota at the workspace level. This is enforced so that individual workspace owners do not edit or increase their quotas and start encroaching onto resources set aside for another workspace. Thus a subscription admin is best suited to allocate and distribute these quotas across workspaces.

View your usage and quotas

Viewing your quota for various resources, such as Virtual Machines, Storage, Network, is easy through the Azure portal.

1. On the left pane, select **All services** and then select **Subscriptions** under the General category.
2. From the list of subscriptions, select the subscription whose quota you are looking for.

   There is a caveat, specifically for viewing the Azure Machine Learning Compute quota. As mentioned above, that quota is separate from the compute quota on your subscription.
3. On the left pane, select **Machine Learning service** and then select any workspace from the list shown
4. On the next blade, under the **Support + troubleshooting section** select **Usage + quotas** to view your current quota limits and usage.
5. Select a subscription to view the quota limits. Remember to filter to the region you are interested in.
6. You can now toggle between a subscription level view and a workspace level view:

   - **Subscription view:** This allows you to view your usage of core quota by VM family, expanding it by workspace, and further expanding it by the actual cluster names. This view is optimal for quickly getting into the details of core usage for a particular VM family to see the break-up by workspaces and further by the underlying clusters for each of those workspaces. The general convention in this

view is (usage/quota), where the usage is the current number of scaled up cores, and quota is the logical maximum number of cores that the resource can scale to. For each **workspace**, the quota would be the workspace level quota (as explained above) which denotes the maximum number of cores that you can scale to for a particular VM family. For a **cluster** similarly, the quota is actually the cores corresponding to the maximum number of nodes that the cluster can scale to defined by the max_nodes property.

- **Workspace view:** This allows you to view your usage of core quota by Workspace, expanding it by VM family, and further expanding it by the actual cluster names. This view is optimal for quickly getting into the details of core usage for a particular workspace to see the break-up by VM families and further by the underlying clusters for each of those families.

Request quota increases

If you want to raise the limit or quota above the default limit, open an online customer support request at no charge.

The limits can't be raised above the maximum limit value shown in the tables. If there is no maximum limit, then the resource doesn't have adjustable limits. This article covers the quota increase process in more detail.

When requesting a quota increase, you need to select the service you are requesting to raise the quota against, which could be services such as Machine Learning service quota, Container instances or Storage quota. In addition for Azure Machine Learning Compute, you can click on the **Request Quota** button while viewing the quota following the steps above.

> **NOTE**
>
> Free Trial subscriptions are not eligible for limit or quota increases. If you have a Free Trial subscription, you can upgrade to a Pay-As-You-Go subscription. For more information, see Upgrade Azure Free Trial to Pay-As-You-Go and Free Trial subscription FAQ.

Export or delete your Machine Learning service workspace data

3/12/2020 • 4 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

In Azure Machine Learning, you can export or delete your workspace data with the authenticated REST API. This article tells you how.

> **NOTE**
> For information about viewing or deleting personal data, see Azure Data Subject Requests for the GDPR. For more information about GDPR, see the GDPR section of the Service Trust portal.

> **NOTE**
> This article provides steps for how to delete personal data from the device or service and can be used to support your obligations under the GDPR. If you're looking for general info about GDPR, see the GDPR section of the Service Trust portal.

Control your workspace data

In-product data stored by Azure Machine Learning is available for export and deletion through Azure Machine Learning studio, CLI, SDK, and authenticated REST APIs. Telemetry data can be accessed through the Azure Privacy portal.

In Azure Machine Learning, personal data consists of user information in run history documents and telemetry records of some user interactions with the service.

Delete workspace data with the REST API

In order to delete data, the following API calls can be made with the HTTP DELETE verb. These are authorized by having an `Authorization: Bearer <arm-token>` header in the request, where `<arm-token>` is the AAD access token for the `https://management.core.windows.net/` endpoint.

To learn how to get this token and call Azure endpoints, see Use REST to manage ML resources and Azure REST API documentation.

In the examples following, replace the text in {} with the instance names that determine the associated resource.

Delete an entire workspace

Use this call to delete an entire workspace.

> **WARNING**
> All information will be deleted and the workspace will no longer be usable.

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}?api-version=2019-11-01
```

Delete models

Use this call to get a list of models and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/models?api-version=2019-11-01
```

Individual models can be deleted with:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/models/{id}?api-version=2019-11-01
```

Delete assets

Use this call to get a list of assets and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/assets?api-version=2019-11-01
```

Individual assets can be deleted with:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/assets/{id}?api-version=2019-11-01
```

Delete images

Use this call to get a list of images and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/images?api-version=2019-11-01
```

Individual images can be deleted with:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/images/{id}?api-version=2019-11-01
```

Delete services

Use this call to get a list of services and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/services?api-version=2019-11-01
```

Individual services can be deleted with:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/services/{id}?api-version=2019-11-01
```

Export service data with the REST API

In order to export data, the following API calls can be made with the HTTP GET verb. These are authorized by having an `Authorization: Bearer <arm-token>` header in the request, where `<arm-token>` is the AAD access token for the endpoint `https://management.core.windows.net/`

To learn how to get this token and call Azure endpoints, see Use REST to manage ML resources and Azure REST API documentation..

In the examples following, replace the text in {} with the instance names that determine the associated resource.

Export Workspace information

Use this call to get a list of all workspaces:

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces?api-version=2019-11-01
```

Information about an individual workspace can be obtained by:

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}?api-version=2019-11-01
```

Export Compute Information

All compute targets attached to a workspace can be obtained by:

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroup/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/computes?api-version=2019-11-01
```

Information about a single compute target can be obtained by:

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroup/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/computes/{computeName}?api-version=2019-11-01
```

Export run history data

Use this call to get a list of all experiments and their information:

```
https://{location}.experiments.azureml.net/history/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/experiments
```

All the runs for a particular experiment can be obtained by:

```
https://{location}.experiments.azureml.net/history/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/experiments/{experimentName}/runs
```

Run history items can be obtained by:

```
https://{location}.experiments.azureml.net/history/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/experiments/{experimentName}/runs/{runId}
```

All run metrics for an experiment can be obtained by:

```
https://{location}.experiments.azureml.net/history/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/experiments/{experimentName}/metrics
```

A single run metric can be obtained by:

```
https://{location}.experiments.azureml.net/history/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resource
GroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/experiments/{experimentName}/
metrics/{metricId}
```

Export artifacts

Use this call to get a list of artifacts and their paths:

```
https://{location}.experiments.azureml.net/artifact/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourc
eGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/artifacts/origins/Experiment
Run/containers/{runId}
```

Export notifications

Use this call to get a list of stored tasks:

```
https://{location}.experiments.azureml.net/notification/v1.0/subscriptions/{subscriptionId}/resourceGroups/{res
ourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/tasks
```

Notifications for a single task can be obtained by:

```
https://{location}.experiments.azureml.net/notification/v1.0/subscriptions/{subscriptionId}/resourceGroups/{res
ourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}tasks/{taskId}
```

Export data stores

Use this call to get a list of data stores:

```
https://{location}.experiments.azureml.net/datastore/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resour
ceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/datastores
```

Individual data stores can be obtained by:

```
https://{location}.experiments.azureml.net/datastore/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resour
ceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/datastores/{name}
```

Export models

Use this call to get a list of models and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/models?api-version=2019-11-01
```

Individual models can be obtained by:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/models/{id}?api-version=2019-11-01
```

Export assets

Use this call to get a list of assets and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/assets?api-version=2019-11-01
```

Individual assets can be obtained by:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/assets/{id}?api-version=2019-11-01
```

Export images

Use this call to get a list of images and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/images?api-version=2019-11-01
```

Individual images can be obtained by:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/images/{id}?api-version=2019-11-01
```

Export services

Use this call to get a list of services and their IDs:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/services?api-version=2019-11-01
```

Individual services can be obtained by:

```
https://{location}.modelmanagement.azureml.net/api/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup
Name}/providers/Microsoft.MachineLearningServices/workspaces/{workspace}/services/{id}?api-version=2019-11-01
```

Export Pipeline Experiments

Individual experiments can be obtained by:

```
https://{location}.aether.ms/api/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/provide
rs/Microsoft.MachineLearningServices/workspaces/{workspaceName}/Experiments/{experimentId}
```

Export Pipeline Graphs

Individual graphs can be obtained by:

```
https://{location}.aether.ms/api/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/provide
rs/Microsoft.MachineLearningServices/workspaces/{workspaceName}/Graphs/{graphId}
```

Export Pipeline Modules

Modules can be obtained by:

```
https://{location}.aether.ms/api/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/provide
rs/Microsoft.MachineLearningServices/workspaces/{workspaceName}/Modules/{id}
```

Export Pipeline Templates

Templates can be obtained by:

```
https://{location}.aether.ms/api/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/Templates/{templateId}
```

Export Pipeline Data Sources

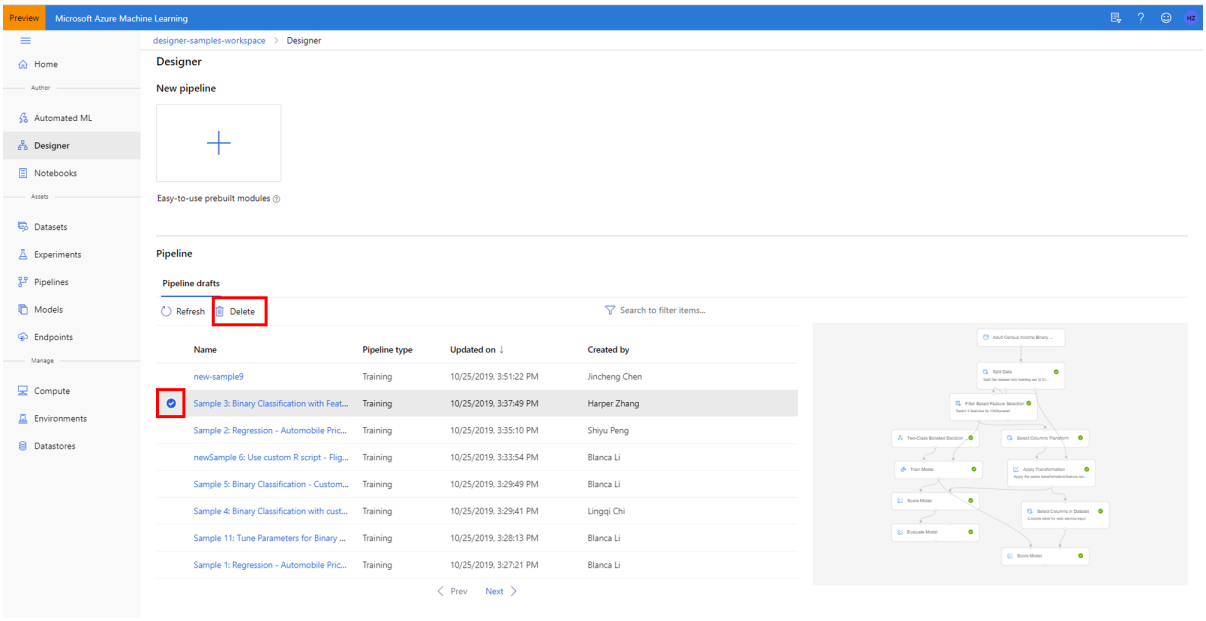
Data Sources can be obtained by:

```
https://{location}.aether.ms/api/v1.0/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.MachineLearningServices/workspaces/{workspaceName}/DataSources/{id}
```

Delete assets in the designer

In the designer where you created your experiment, delete individual assets:

1. Go to designer



2. In the list, select the individual pipeline draft to delete.
3. Select **Delete**.

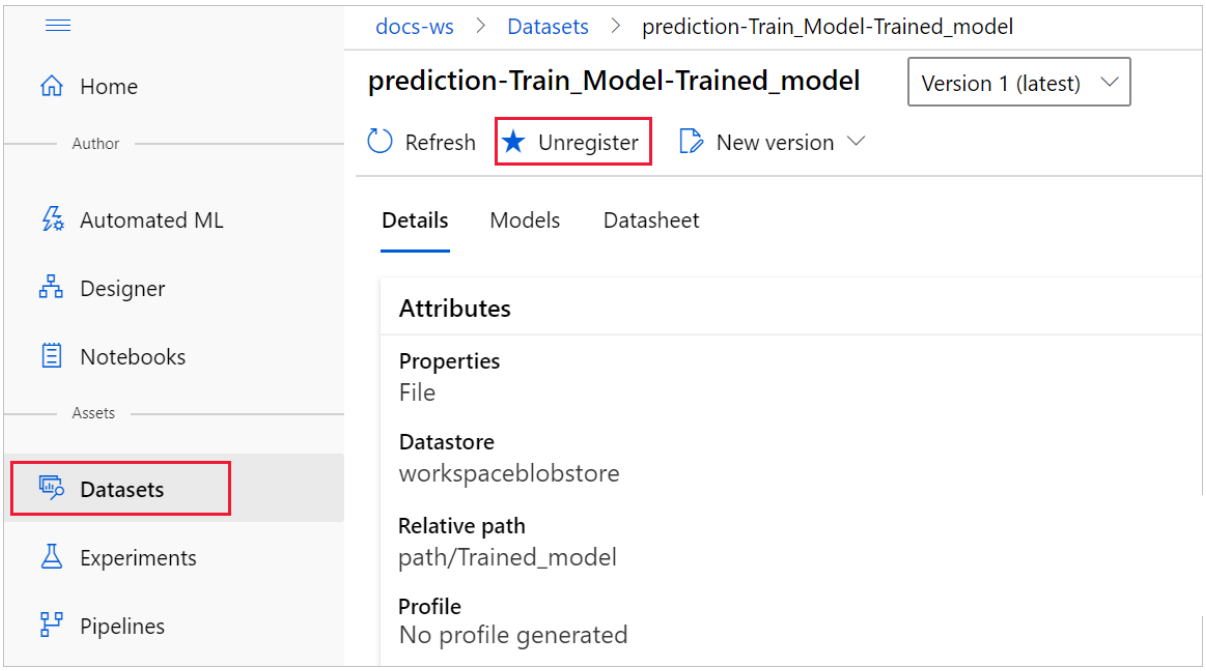
Delete datasets in the designer

To delete datasets in the designer, use the Azure portal or Storage Explorer to navigate to connected storage accounts and delete datasets there. Unregistering datasets in the designer only removes the reference point in storage.

Export data in the designer

In the designer where you created your experiment, export data you have added:

1. On the left, select **Datasets**.
2. In the list, select the dataset to export


docs-ws
Datasets
prediction-Train_Model-Trained_model
Home
Author
Automated ML
Designer
Notebooks
Assets
Datasets
Experiments
Pipelines
prediction-Train_Model-Trained_model
Version 1 (latest)
Refresh
Unregister
New version
Details
Models
Datasheet
Attributes
Properties
File
Datastore
workspaceblobstore
Relative path
path/Trained_model
Profile
No profile generated

Create event driven machine learning workflows (Preview)

4/24/2020 • 5 minutes to read • Edit Online

Azure Event Grid supports Azure Machine Learning events. You can subscribe and consume events such as run status changed, run completion, model registration, model deployment, and data drift detection within a workspace.

For more information on event types, see Azure Machine Learning integration with Event Grid and the Azure Machine Learning event grid schema.

Use Event Grid to enable common scenarios such as:

- Send emails on run failure and run completion
- Use an Azure function after a model is registered
- Streaming events from Azure Machine Learning to various of endpoints
- Trigger an ML pipeline when drift is detected

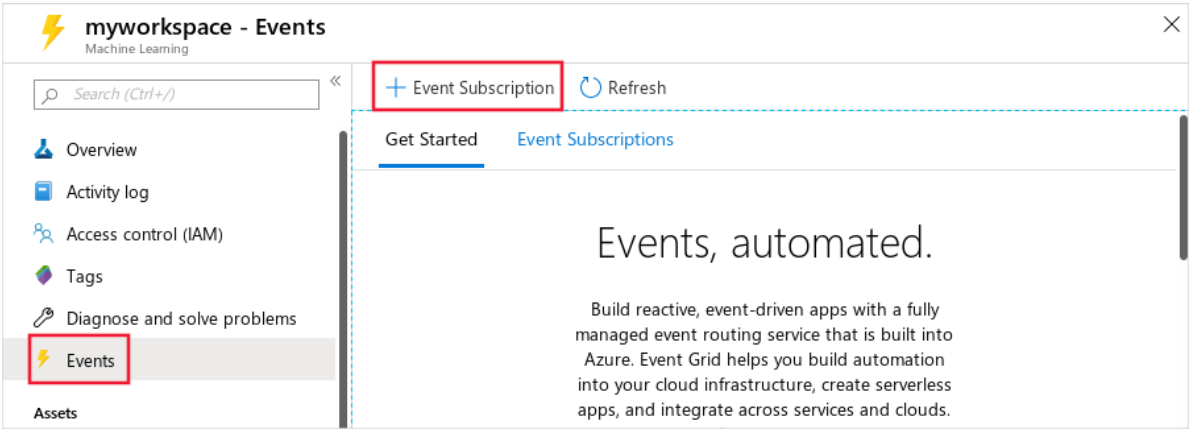
> **NOTE**
> Currently, runStatusChanged events only trigger when the run status is **failed**

Prerequisites

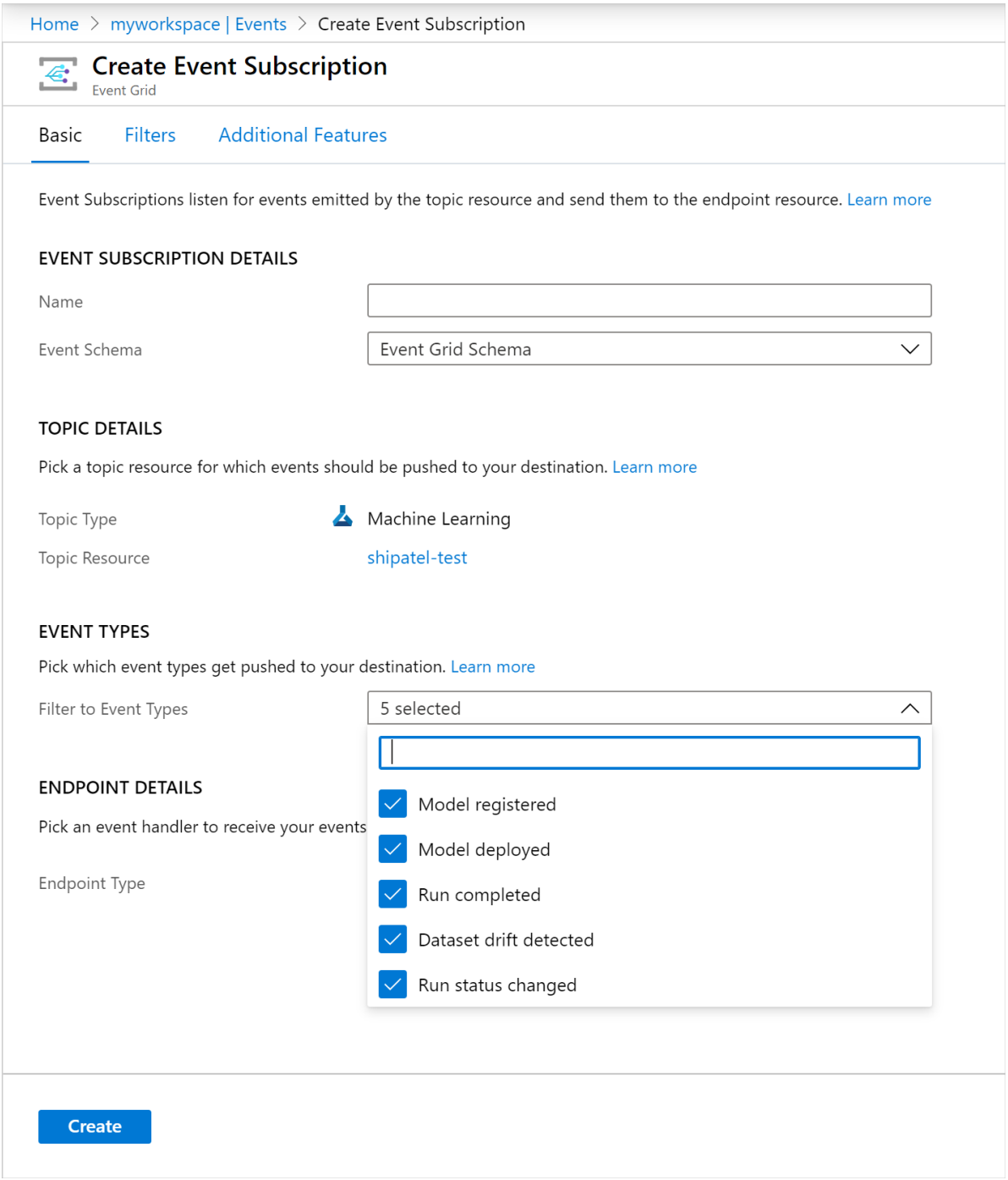
- Contributor or owner access to the Azure Machine Learning workspace you will create events for.

Configure EventGrid using the Azure portal

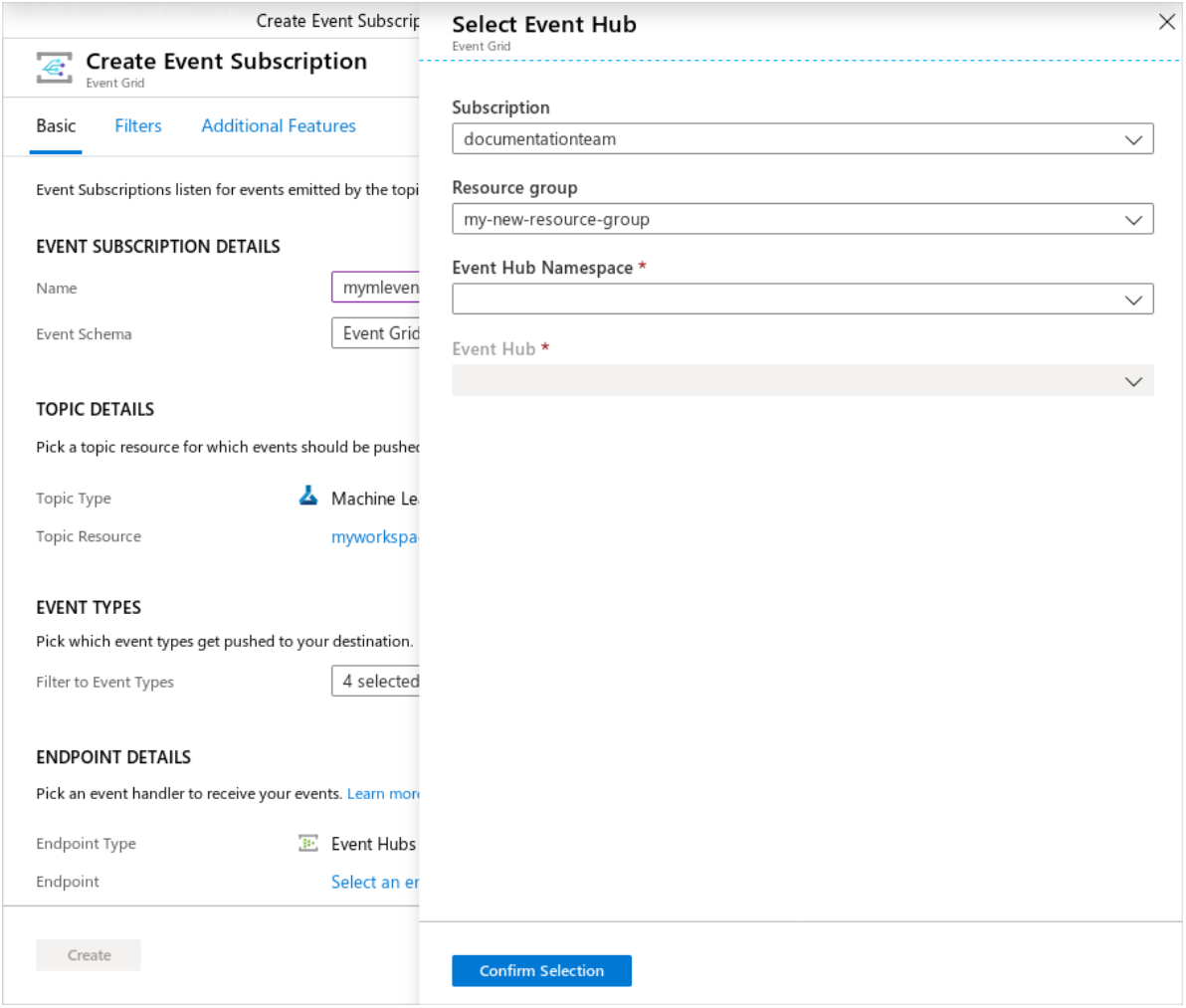
1. Open the Azure portal and go to your Azure Machine Learning workspace.
2. From the left bar, select **Events** and then select **Event Subscriptions**.



3. Select the event type to consume. For example, the following screenshot has selected **Model registered**, **Model deployed**, **Run completed**, and **Dataset drift detected**:



4. Select the endpoint to publish the event to. In the following screenshot, **Event hub** is the selected endpoint:



Once you have confirmed your selection, click **Create**. After configuration, these events will be pushed to your endpoint.

Configure EventGrid using the CLI

You can either install the latest Azure CLI, or use the Azure Cloud Shell that is provided as part of your Azure subscription.

To install the Event Grid extension, use the following command from the CLI:

```
az add extension --name eventgrid
```

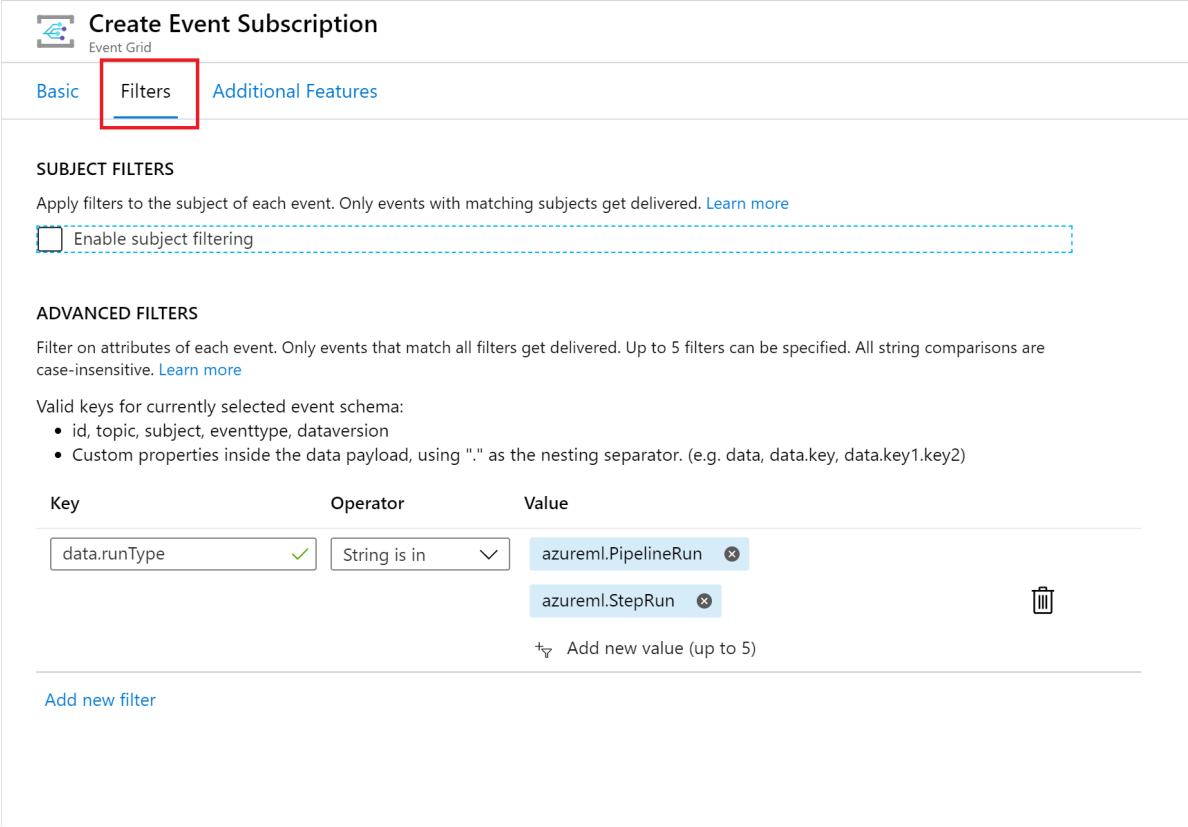
The following example demonstrates how to select an Azure subscription and creates e a new event subscription for Azure Machine Learning:

```
# Select the Azure subscription that contains the workspace
az account set --subscription "<name or ID of the subscription>"

# Subscribe to the machine learning workspace.
az eventgrid event-subscription create \
  --name {eventGridFilterName} \
  --source-resource-id "/subscriptions/{subId}/resourceGroups/{rgName}/
\providers/Microsoft.MachineLearningServices/workspaces/{wsName}" \
  --endpoint {event handler endpoint} \
  --included-event-types Microsoft.MachineLearningServices.ModelRegistered \
  --subject-begins-with "models/mymodelname"
```

Filter Events

When setting up your events, you can apply filters to only trigger on specific event data. In the example below, for run status changed events, you can filter by run types. The event only triggers when the criteria is met. Refer to the Azure Machine Learning event grid schema to learn about event data you can filter by.

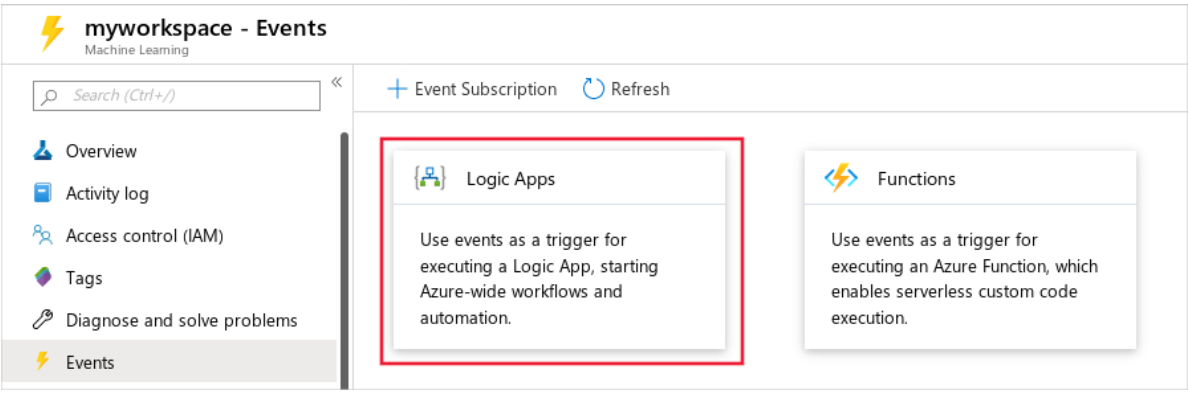
1. Go to the Azure portal, select a new subscription or an existing one.
2. Select the filters tab and scroll down to Advanced filters. For the **Key** and **Value**, provide the property types you want to filter by. Here you can see the event will only trigger when the run type is a pipeline run or pipeline step run.



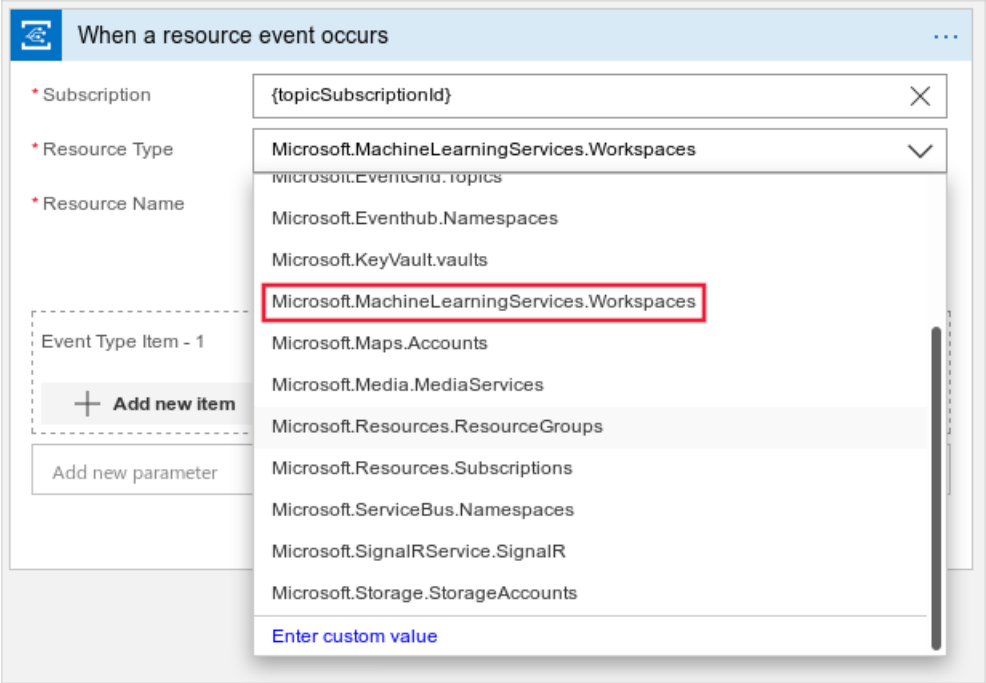
Sample: Send email alerts

Use Azure Logic Apps to configure emails for all your events. Customize with conditions and specify recipients to enable collaboration and awareness across teams working together.

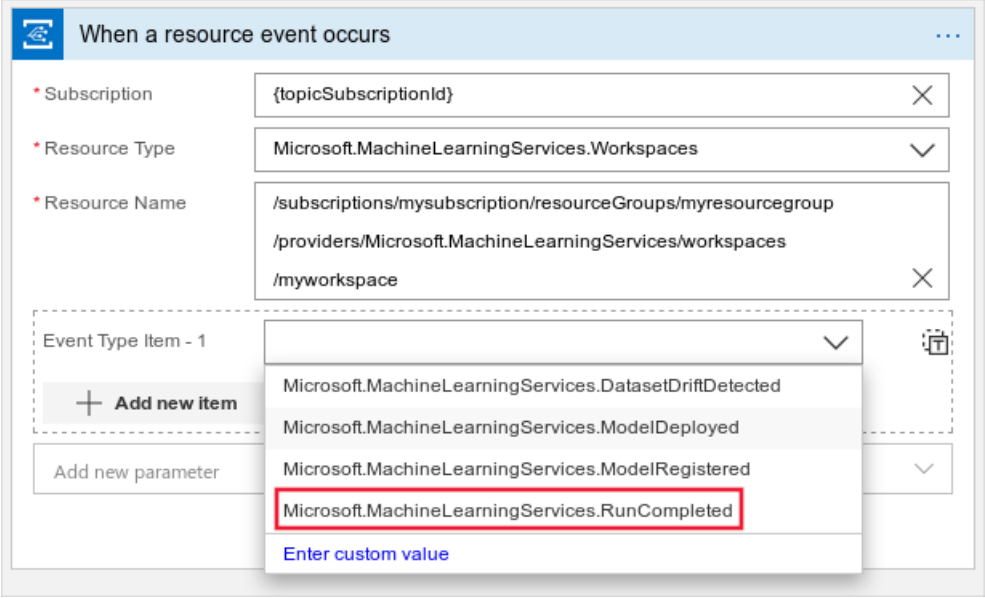
1. In the Azure portal, go to your Azure Machine Learning workspace and select the events tab from the left bar. From here, select **Logic apps**.



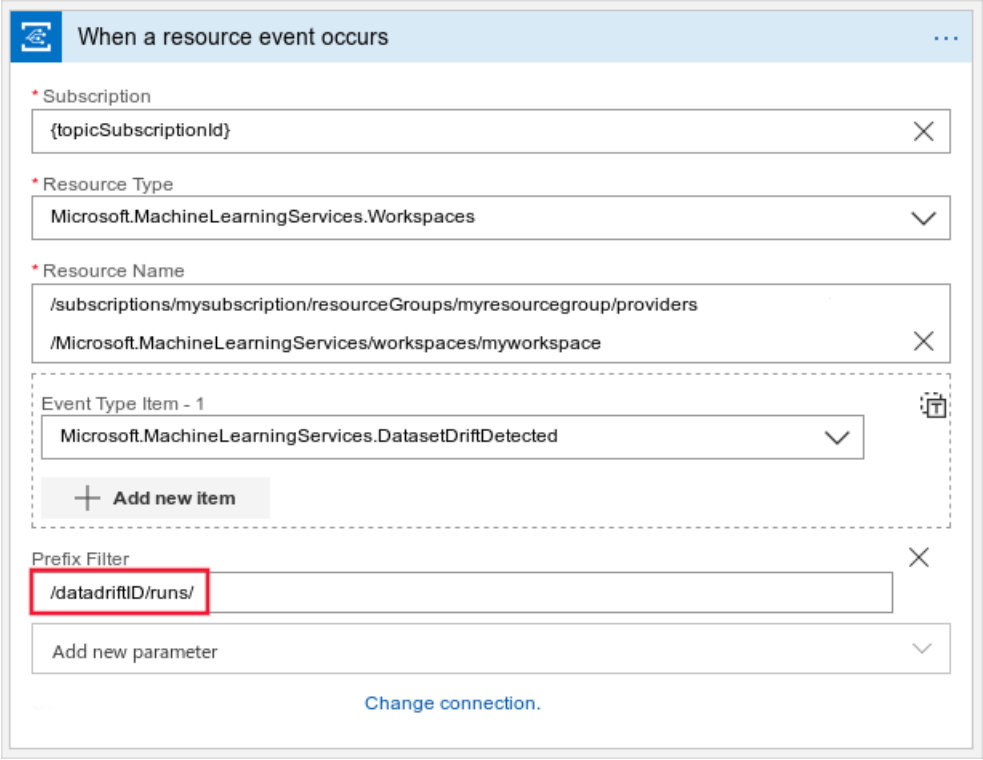
2. Sign into the Logic App UI and select Machine Learning service as the topic type.



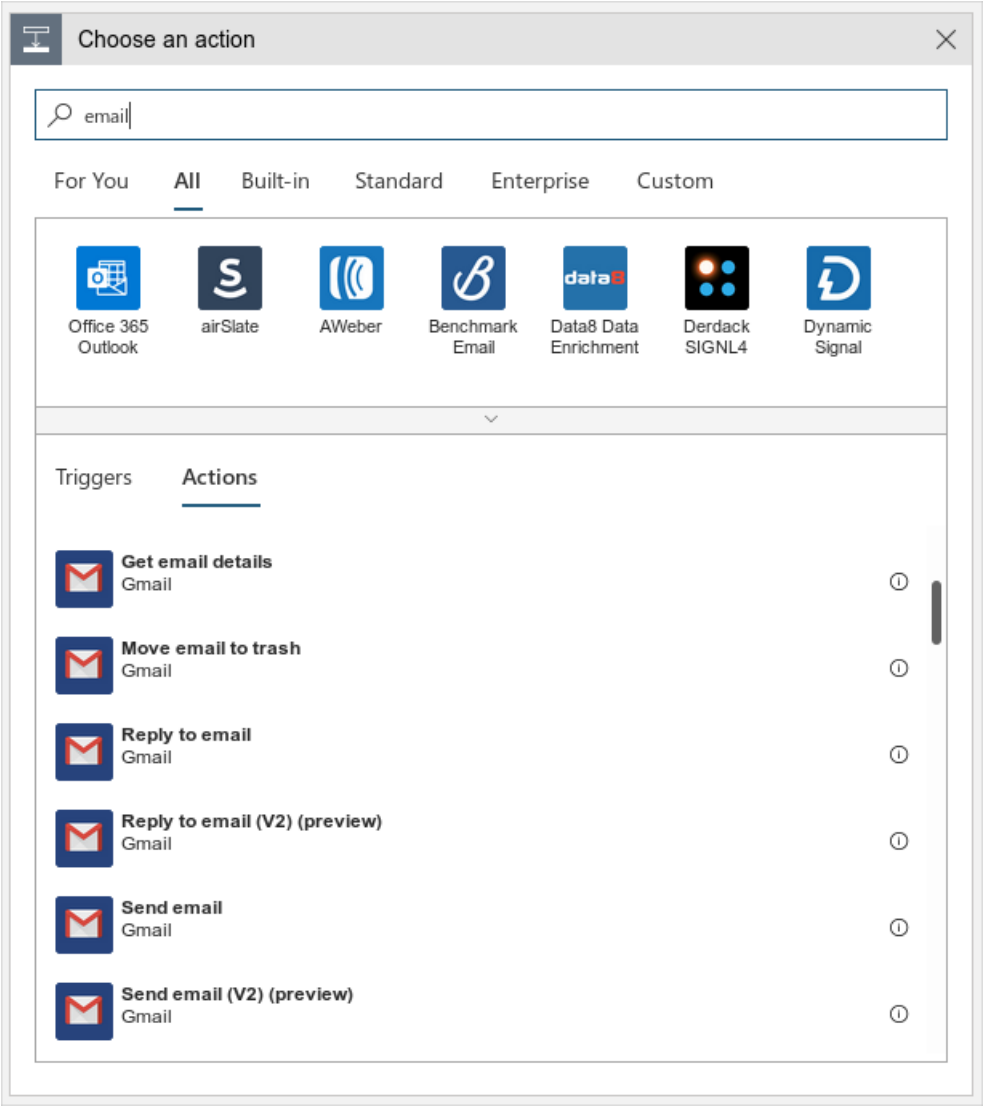
3. Select which event(s) to be notified for. For example, the following screenshot **RunCompleted**.



4. You can use the filtering method in the section above or add filters to only trigger the logic app on a subset of event types. In the following screenshot, a **prefix filter** of **/datadriftID/runs/** is used.

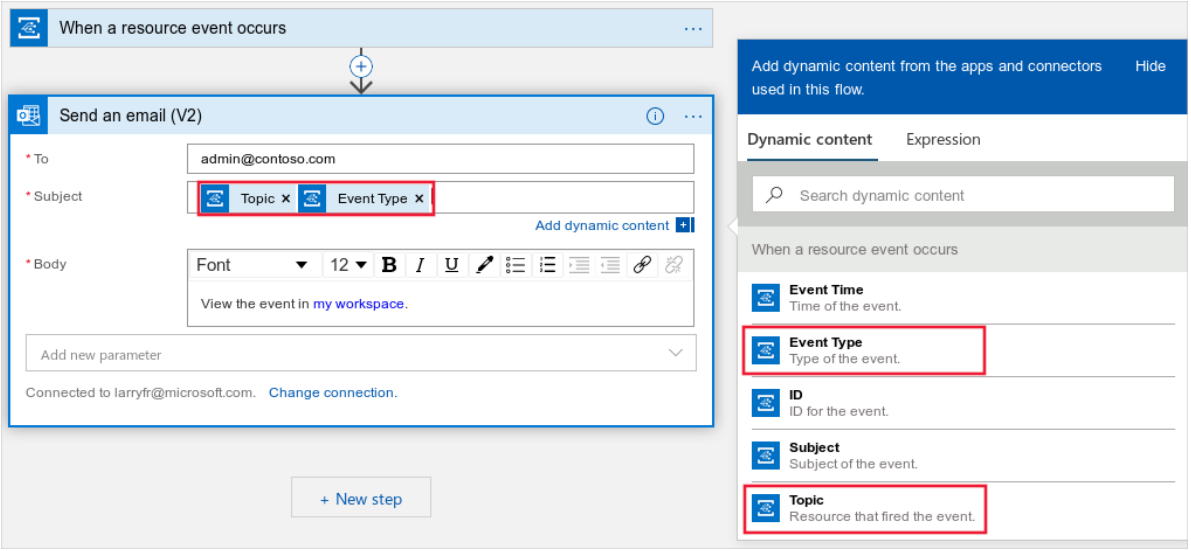


5. Next, add a step to consume this event and search for email. There are several different mail accounts you can use to receive events. You can also configure conditions on when to send an email alert.

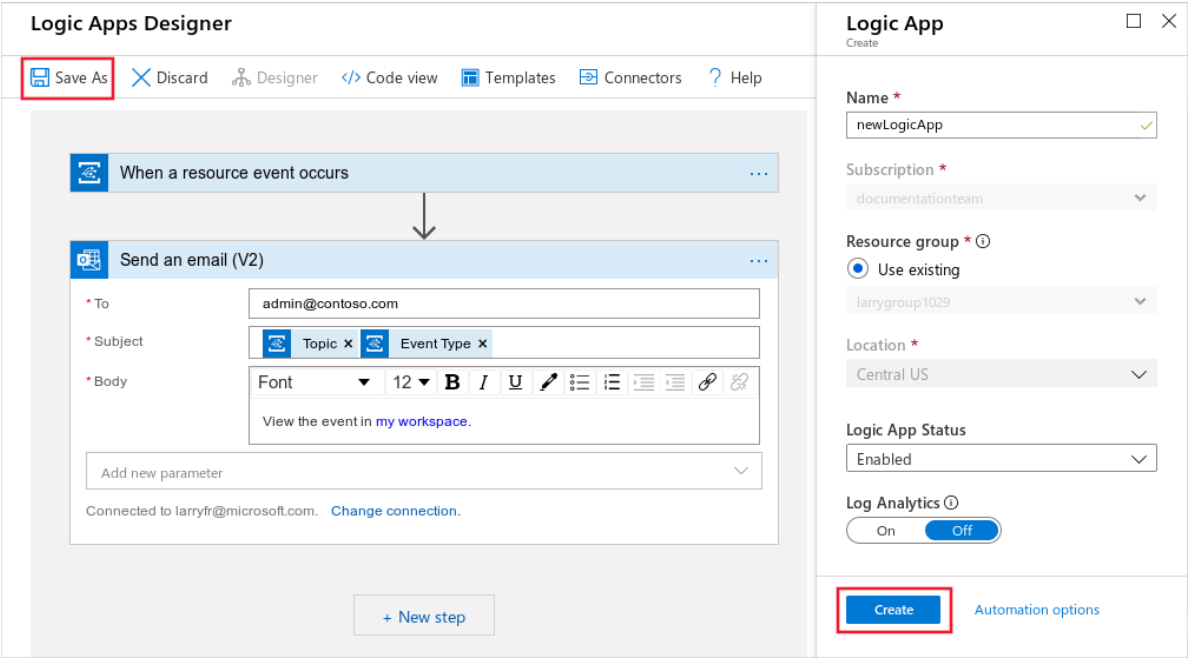


6. Select **Send an email** and fill in the parameters. In the subject, you can include the **Event Type** and **Topic**

to help filter events. You can also include a link to the workspace page for runs in the message body.



7. To save this action, select **Save As** on the left corner of the page. From the right bar that appears, confirm creation of this action.



Sample: Trigger retraining when data drift occurs

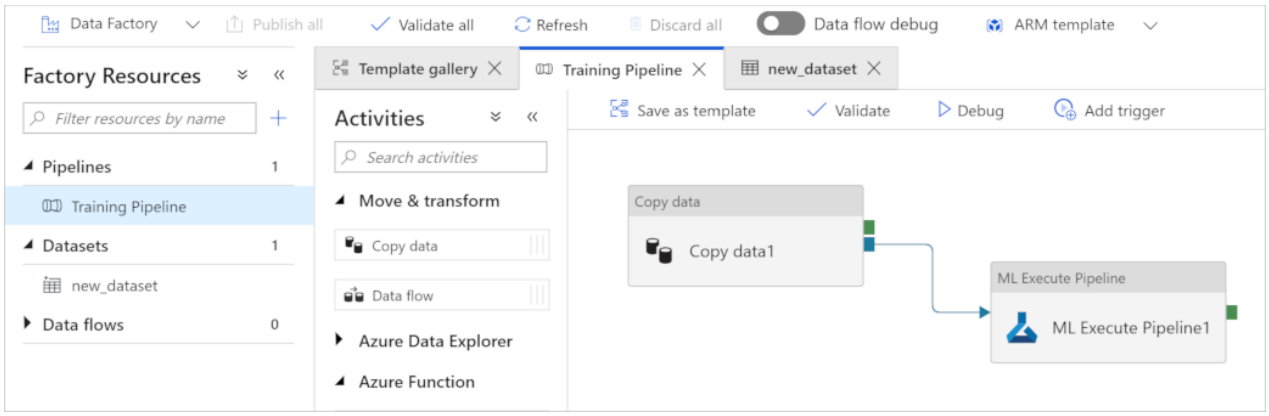
Models go stale over time, and not remain useful in the context it is running in. One way to tell if it's time to retrain the model is detecting data drift.

This example shows how to use event grid with an Azure Logic App to trigger retraining. The example triggers an Azure Data Factory pipeline when data drift occurs between a model's training and serving datasets.

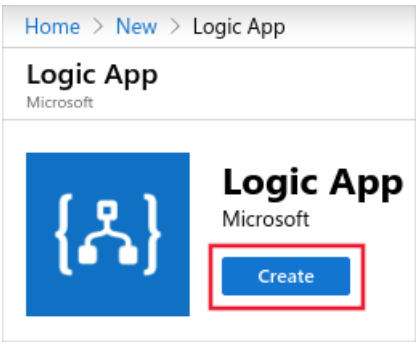
Before you begin, perform the following actions:

- Set up a dataset monitor to detect data drift in a workspace
- Create a published Azure Data Factory pipeline.

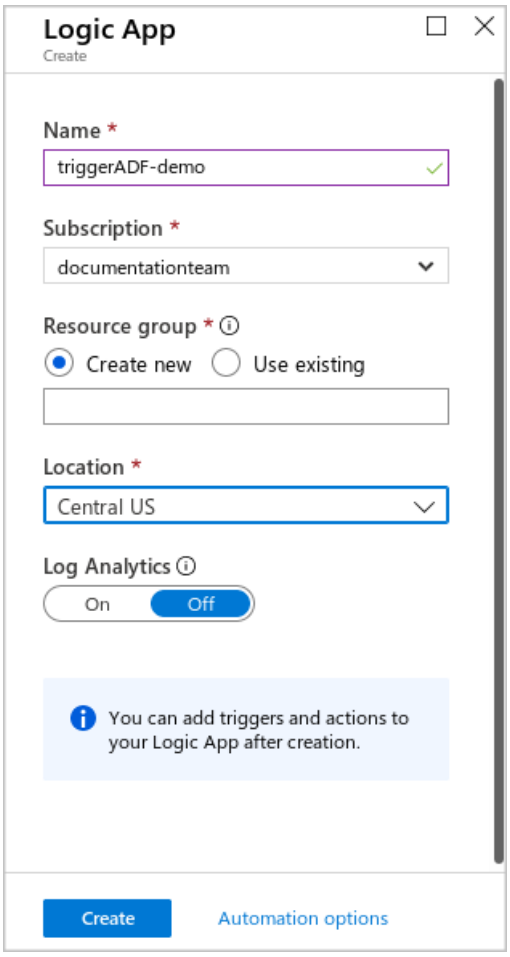
In this example, a simple Data Factory pipeline is used to copy files into a blob store and run a published Machine Learning pipeline. For more information on this scenario, see how to set up a Machine Learning step in Azure Data Factory



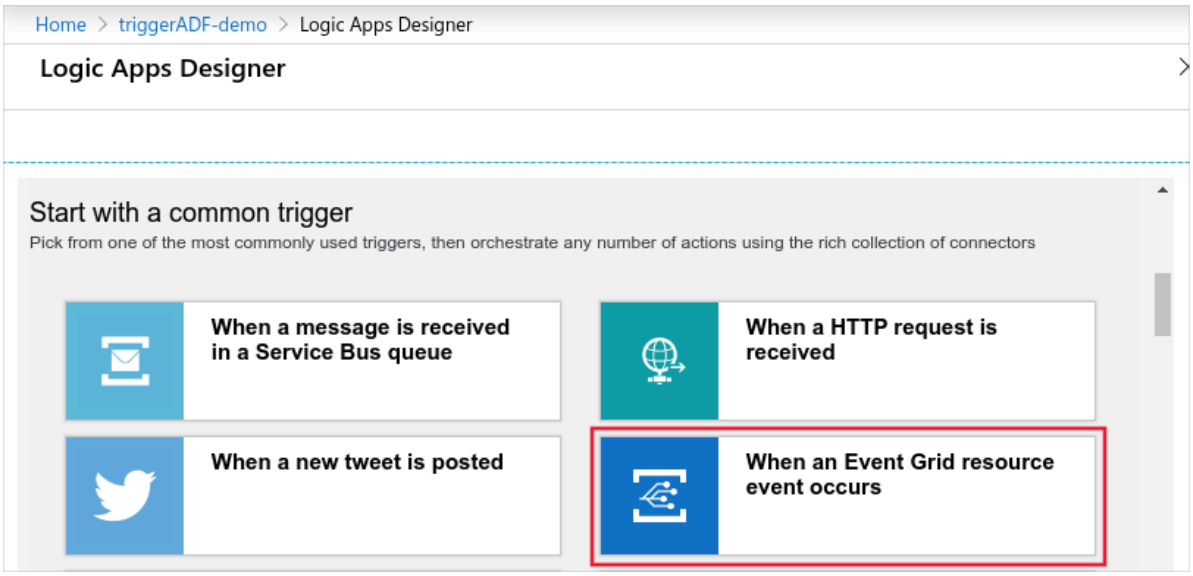
1. Start with creating the logic app. Go to the Azure portal, search for Logic Apps, and select create.



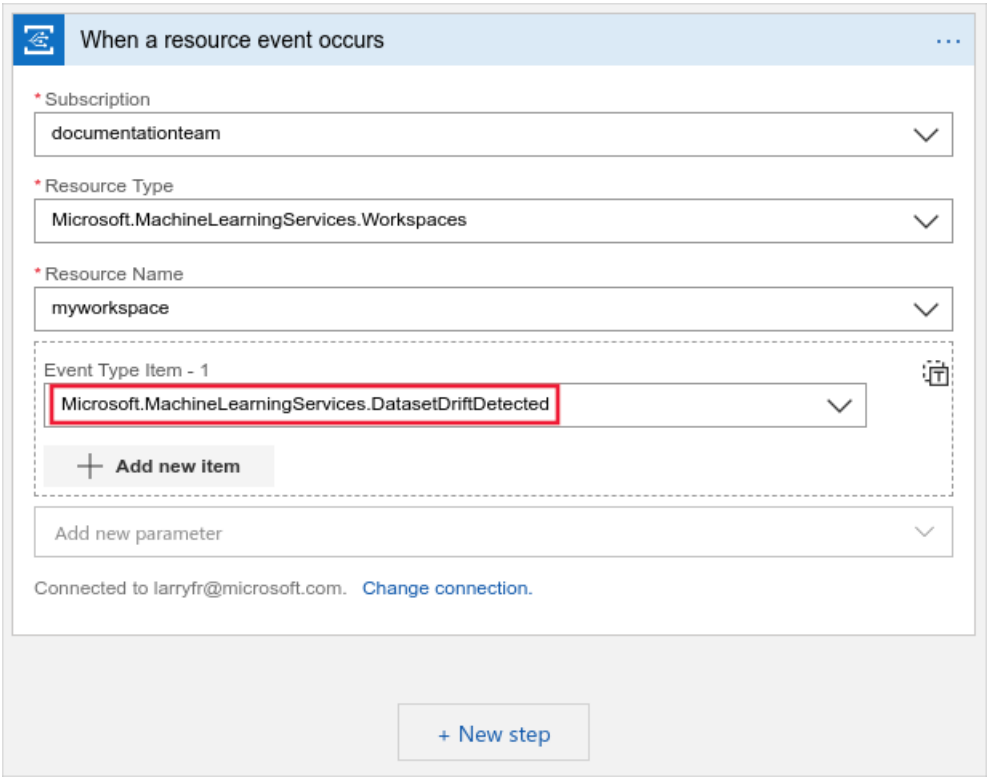
2. Fill in the requested information. To simplify the experience, use the same subscription and resource group as your Azure Data Factory Pipeline and Azure Machine Learning workspace.



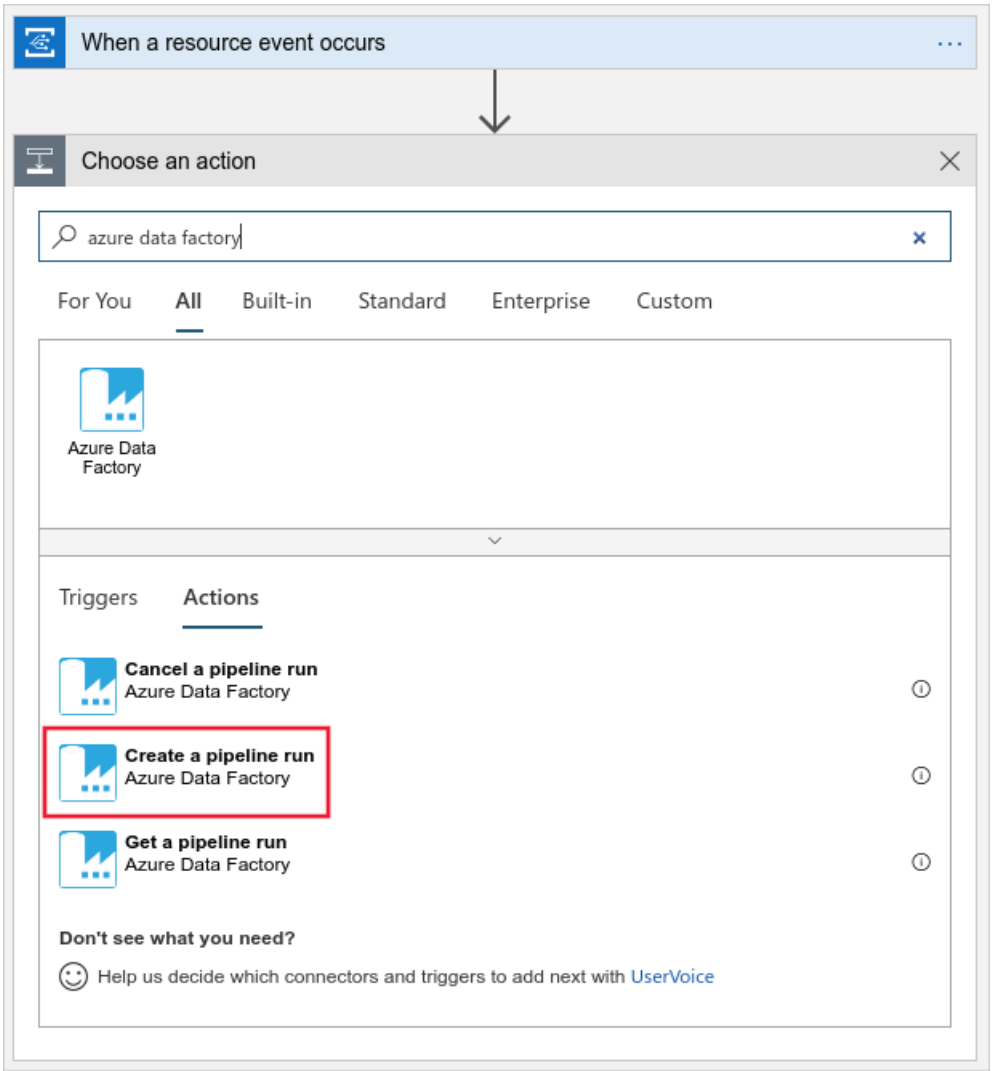
3. Once you have created the logic app, select **When an Event Grid resource event occurs**.



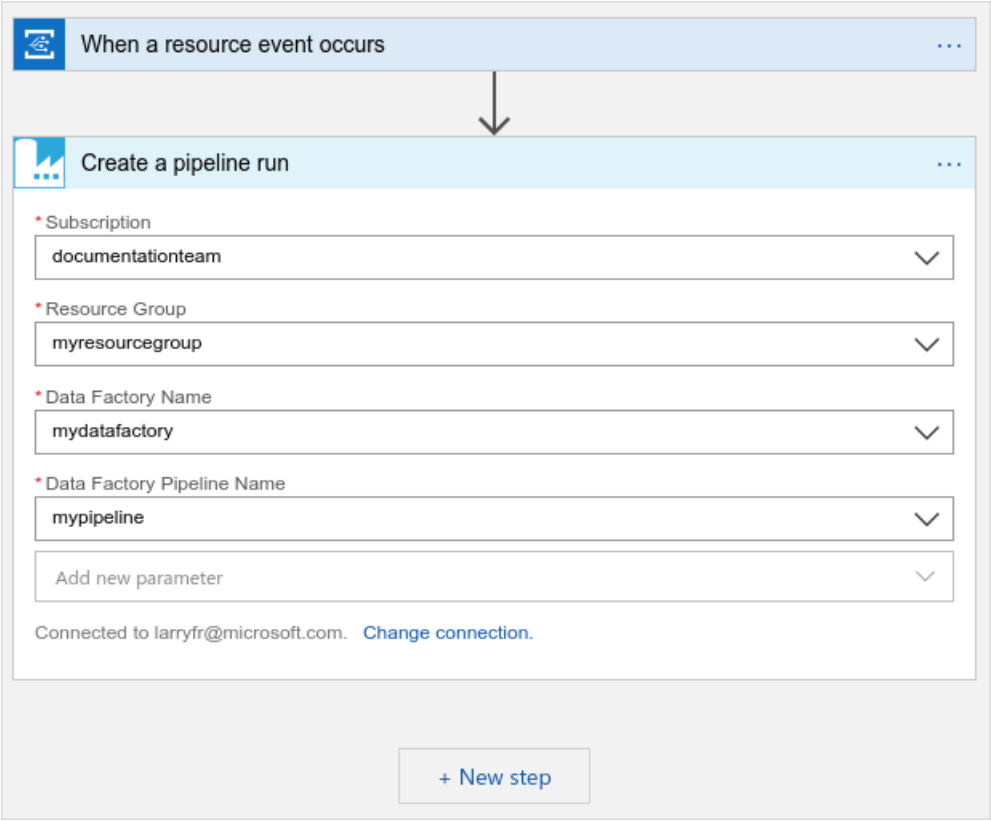
4. Login and fill in the details for the event. Set the **Resource Name** to the workspace name. Set the **Event Type** to **DatasetDriftDetected**.



5. Add a new step, and search for **Azure Data Factory**. Select **Create a pipeline run**.

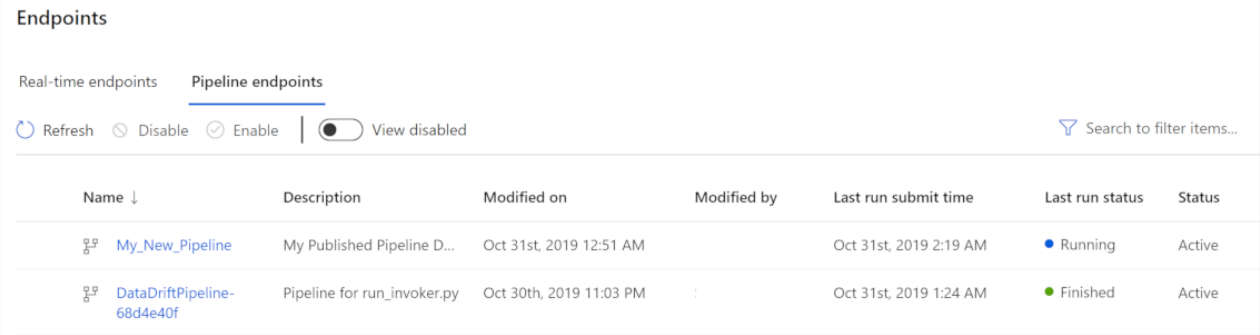


6. Login and specify the published Azure Data Factory pipeline to run.



7. Save and create the logic app using the **save** button on the top left of the page. To view your app, go to your workspace in the Azure portal and click on **Events**.

Now the data factory pipeline is triggered when drift occurs. View details on your data drift run and machine learning pipeline on the new workspace portal.



Sample: Deploy a model based on tags

An Azure Machine Learning model object contains parameters you can pivot deployments on such as model name, version, tag, and property. The model registration event can trigger an endpoint and you can use an Azure Function to deploy a model based on the value of those parameters.

For an example, see the https://github.com/Azure-Samples/MachineLearningSamples-NoCodeDeploymentTriggeredByEventGrid repository and follow the steps in the **readme** file.

Next steps

- To learn more about available events, see the Azure Machine Learning event schema

Use the CLI extension for Azure Machine Learning

4/17/2020 • 15 minutes to read • Edit Online

APPLIES TO: Basic edition Enterprise edition (Upgrade to Enterprise edition)

The Azure Machine Learning CLI is an extension to the Azure CLI, a cross-platform command-line interface for the Azure platform. This extension provides commands for working with Azure Machine Learning. It allows you to automate your machine learning activities. The following list provides some example actions that you can do with the CLI extension:

- Run experiments to create machine learning models
- Register machine learning models for customer usage
- Package, deploy, and track the lifecycle of your machine learning models

The CLI is not a replacement for the Azure Machine Learning SDK. It is a complementary tool that is optimized to handle highly parameterized tasks which suit themselves well to automation.

Prerequisites

- To use the CLI, you must have an Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the free or paid version of Azure Machine Learning today.
- To use the CLI commands in this document from your **local environment**, you need the Azure CLI.

  If you use the Azure Cloud Shell, the CLI is accessed through the browser and lives in the cloud.

Full reference docs

Find the full reference docs for the azure-cli-ml extension of Azure CLI.

Connect the CLI to your Azure subscription

> **IMPORTANT**
>
> If you are using the Azure Cloud Shell, you can skip this section. The cloud shell automatically authenticates you using the account you log into your Azure subscription.

There are several ways that you can authenticate to your Azure subscription from the CLI. The most basic is to interactively authenticate using a browser. To authenticate interactively, open a command line or terminal and use the following command:

```
az login
```

If the CLI can open your default browser, it will do so and load a sign-in page. Otherwise, you need to open a browser and follow the instructions on the command line. The instructions involve browsing to https://aka.ms/devicelogin and entering an authorization code.

TIP

After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the Azure portal by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.

To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see Use multiple Azure Subscriptions.

For other methods of authenticating, see Sign in with Azure CLI.

Install the extension

To install the Machine Learning CLI extension, use the following command:

```
az extension add -n azure-cli-ml
```

TIP

Example files you can use with the commands below can be found here.

When prompted, select `y` to install the extension.

To verify that the extension has been installed, use the following command to display a list of ML-specific subcommands:

```
az ml -h
```

Update the extension

To update the Machine Learning CLI extension, use the following command:

```
az extension update -n azure-cli-ml
```

Remove the extension

To remove the CLI extension, use the following command:

```
az extension remove -n azure-cli-ml
```

Resource management

The following commands demonstrate how to use the CLI to manage resources used by Azure Machine Learning.

- If you do not already have one, create a resource group:

```
az group create -n myresourcegroup -l westus2
```

- Create an Azure Machine Learning workspace:

```
az ml workspace create -w myworkspace -g myresourcegroup
```

> **TIP**
> This command creates a basic edition workspace. To create an enterprise workspace, use the `--sku enterprise` switch with the `az ml workspace create` command. For more information on Azure Machine Learning editions, see What is Azure Machine Learning.

For more information, see az ml workspace create.

- Attach a workspace configuration to a folder to enable CLI contextual awareness.

```
az ml folder attach -w myworkspace -g myresourcegroup
```

This command creates a `.azureml` subdirectory that contains example runconfig and conda environment files. It also contains a `config.json` file that is used to communicate with your Azure Machine Learning workspace.

For more information, see az ml folder attach.

- Attach an Azure blob container as a Datastore.

```
az ml datastore attach-blob  -n datastorename -a accountname -c containername
```

For more information, see az ml datastore attach-blob.

- Upload files to a Datastore.

```
az ml datastore upload  -n datastorename -p sourcepath
```

For more information, see az ml datastore upload.

- Attach an AKS cluster as a Compute Target.

```
az ml computetarget attach aks -n myaks -i myaksresourceid -g myresourcegroup -w myworkspace
```

For more information, see az ml computetarget attach aks

- Create a new AMLcompute target.

```
az ml computetarget create amlcompute -n cpu --min-nodes 1 --max-nodes 1 -s STANDARD_D3_V2
```

For more information, see az ml computetarget create amlcompute.

Run experiments

- Start a run of your experiment. When using this command, specify the name of the runconfig file (the

text before *.runconfig if you are looking at your file system) against the -c parameter.

```
az ml run submit-script -c sklearn -e testexperiment train.py
```

> **TIP**
> The `az ml folder attach` command creates a `.azureml` subdirectory, which contains two example runconfig files.
>
> If you have a Python script that creates a run configuration object programmatically, you can use RunConfig.save() to save it as a runconfig file.
>
> The full runconfig schema can be found in this JSON file. The schema is self-documenting through the `description` key of each object. Additionally, there are enums for possible values, and a template snippet at the end.

For more information, see az ml run submit-script.

- View a list of experiments:

```
az ml experiment list
```

For more information, see az ml experiment list.

Dataset management

The following commands demonstrate how to work with datasets in Azure Machine Learning:

- Register a dataset:

```
az ml dataset register -f mydataset.json
```

For information on the format of the JSON file used to define the dataset, use `az ml dataset register --show-template`.

For more information, see az ml dataset register.

- Archive an active or deprecated dataset:

```
az ml dataset archive -n dataset-name
```

For more information, see az ml dataset archive.

- Deprecate a dataset:

```
az ml dataset deprecate -d replacement-dataset-id -n dataset-to-deprecate
```

For more information, see az ml dataset deprecate.

- List all datasets in a workspace:

```
az ml dataset list
```

For more information, see az ml dataset list.

- Get details of a dataset:

```
az ml dataset show -n dataset-name
```

For more information, see az ml dataset show.

- Reactivate an archived or deprecated dataset:

```
az ml dataset reactivate -n dataset-name
```

For more information, see az ml dataset reactivate.

- Unregister a dataset:

```
az ml dataset unregister -n dataset-name
```

For more information, see az ml dataset unregister.

Environment management

The following commands demonstrate how to create, register, and list Azure Machine Learning environments for your workspace:

- Create scaffolding files for an environment:

```
az ml environment scaffold -n myenv -d myenvdirectory
```

For more information, see az ml environment scaffold.

- Register an environment:

```
az ml environment register -d myenvdirectory
```

For more information, see az ml environment register.

- List registered environments:

```
az ml environment list
```

For more information, see az ml environment list.

- Download a registered environment:

```
az ml environment download -n myenv -d downloaddirectory
```

For more information, see az ml environment download.

Environment configuration schema

If you used the `az ml environment scaffold` command, it generates a template `azureml_environment.json` file that can be modified and used to create custom environment configurations with the CLI. The top level

object loosely maps to the `Environment` class in the Python SDK.

```
{
    "name": "testenv",
    "version": null,
    "environmentVariables": {
        "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
    },
    "python": {
        "userManagedDependencies": false,
        "interpreterPath": "python",
        "condaDependenciesFile": null,
        "baseCondaEnvironment": null
    },
    "docker": {
        "enabled": false,
        "baseImage": "mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04",
        "baseDockerfile": null,
        "sharedVolumes": true,
        "shmSize": "2g",
        "arguments": [],
        "baseImageRegistry": {
            "address": null,
            "username": null,
            "password": null
        }
    },
    "spark": {
        "repositories": [],
        "packages": [],
        "precachePackages": true
    },
    "databricks": {
        "mavenLibraries": [],
        "pypiLibraries": [],
        "rcranLibraries": [],
        "jarLibraries": [],
        "eggLibraries": []
    },
    "inferencingStackVersion": null
}
```

The following table details each top-level field in the JSON file, it's type, and a description. If an object type is linked to a class from the Python SDK, there is a loose 1:1 match between each JSON field and the public variable name in the Python class. In some cases the field may map to a constructor argument rather than a class variable. For example, the `environmentVariables` field maps to the `environment_variables` variable in the `Environment` class.

JSON FIELD	TYPE	DESCRIPTION
`name`	`string`	Name of the environment. Do not start name with **Microsoft** or **AzureML**.
`version`	`string`	Version of the environment.
`environmentVariables`	`{string: string}`	A hash-map of environment variable names and values.

JSON FIELD	TYPE	DESCRIPTION
`python`	`PythonSection`	Object that defines the Python environment and interpreter to use on target compute resource.
`docker`	`DockerSection`	Defines settings to customize the Docker image built to the environment's specifications.
`spark`	`SparkSection`	The section configures Spark settings. It is only used when framework is set to PySpark.
`databricks`	`DatabricksSection`	Configures Databricks library dependencies.
`inferencingStackVersion`	`string`	Specifies the inferencing stack version added to the image. To avoid adding an inferencing stack, leave this field `null`. Valid value: "latest".

ML pipeline management

The following commands demonstrate how to work with machine learning pipelines:

- Create a machine learning pipeline:

  ```
  az ml pipeline create -n mypipeline -y mypipeline.yml
  ```

  For more information, see az ml pipeline create.

  For more information on the pipeline YAML file, see Define machine learning pipelines in YAML.

- Run a pipeline:

  ```
  az ml run submit-pipeline -n myexperiment -y mypipeline.yml
  ```

  For more information, see az ml run submit-pipeline.

  For more information on the pipeline YAML file, see Define machine learning pipelines in YAML.

- Schedule a pipeline:

  ```
  az ml pipeline create-schedule -n myschedule -e myexpereiment -i mypipelineid -y myschedule.yml
  ```

  For more information, see az ml pipeline create-schedule.

  For more information on the pipeline schedule YAML file, see Define machine learning pipelines in YAML.

Model registration, profiling, deployment

The following commands demonstrate how to register a trained model, and then deploy it as a production service:

- Register a model with Azure Machine Learning:

  ```
  az ml model register -n mymodel -p sklearn_regression_model.pkl
  ```

  For more information, see az ml model register.

- **OPTIONAL** Profile your model to get optimal CPU and memory values for deployment.

  ```
  az ml model profile -n myprofile -m mymodel:1 --ic inferenceconfig.json -d "{\"data\":
  [[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}" -t myprofileresult.json
  ```

  For more information, see az ml model profile.

- Deploy your model to AKS

  ```
  az ml model deploy -n myservice -m mymodel:1 --ic inferenceconfig.json --dc deploymentconfig.json
  --ct akscomputetarget
  ```

  For more information on the inference configuration file schema, see Inference configuration schema.

  For more information on the deployment configuration file schema, see Deployment configuration schema.

  For more information, see az ml model deploy.

Inference configuration schema

The entries in the `inferenceconfig.json` document map to the parameters for the InferenceConfig class. The following table describes the mapping between entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`entryScript`	`entry_script`	Path to a local file that contains the code to run for the image.
`sourceDirectory`	`source_directory`	Optional. Path to folders that contain all files to create the image, which makes it easy to access any files within this folder or subfolder. You can upload an entire folder from your local machine as dependencies for the Webservice. Note: your entry_script, conda_file, and extra_docker_file_steps paths are relative paths to the source_directory path.
`environment`	`environment`	Optional. Azure Machine Learning environment.

You can include full specifications of an Azure Machine Learning environment in the inference configuration file. If this environment doesn't exist in your workspace, Azure Machine Learning will create it. Otherwise, Azure Machine Learning will update the environment if necessary. The following JSON is an example:

```
{
    "entryScript": "score.py",
    "environment": {
        "docker": {
            "arguments": [],
            "baseDockerfile": null,
            "baseImage": "mcr.microsoft.com/azureml/base:intelmpi2018.3-ubuntu16.04",
            "enabled": false,
            "sharedVolumes": true,
            "shmSize": null
        },
        "environmentVariables": {
            "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
        },
        "name": "my-deploy-env",
        "python": {
            "baseCondaEnvironment": null,
            "condaDependencies": {
                "channels": [
                    "conda-forge"
                ],
                "dependencies": [
                    "python=3.6.2",
                    {
                        "pip": [
                            "azureml-defaults",
                            "azureml-telemetry",
                            "scikit-learn",
                            "inference-schema[numpy-support]"
                        ]
                    }
                ],
                "name": "project_environment"
            },
            "condaDependenciesFile": null,
            "interpreterPath": "python",
            "userManagedDependencies": false
        },
        "version": "1"
    }
}
```

You can also use an existing Azure Machine Learning environment in separated CLI parameters and remove the "environment" key from the inference configuration file. Use -e for the environment name, and --ev for the environment version. If you don't specify --ev, the latest version will be used. Here is an example of an inference configuration file:

```
{
    "entryScript": "score.py",
    "sourceDirectory": null
}
```

The following command demonstrates how to deploy a model using the previous inference configuration file (named myInferenceConfig.json).

It also uses the latest version of an existing Azure Machine Learning environment (named AzureML-Minimal).

```
az ml model deploy -m mymodel:1 --ic myInferenceConfig.json -e AzureML-Minimal --dc
deploymentconfig.json
```

Deployment configuration schema

Local deployment configuration schema

The entries in the `deploymentconfig.json` document map to the parameters for LocalWebservice.deploy_configuration. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`computeType`	NA	The compute target. For local targets, the value must be `local`.
`port`	`port`	The local port on which to expose the service's HTTP endpoint.

This JSON is an example deployment configuration for use with the CLI:

```
{
    "computeType": "local",
    "port": 32267
}
```

Azure Container Instance deployment configuration schema

The entries in the `deploymentconfig.json` document map to the parameters for AciWebservice.deploy_configuration. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`computeType`	NA	The compute target. For ACI, the value must be `ACI`.
`containerResourceRequirements`	NA	Container for the CPU and memory entities.
`cpu`	`cpu_cores`	The number of CPU cores to allocate. Defaults, `0.1`
`memoryInGB`	`memory_gb`	The amount of memory (in GB) to allocate for this web service. Default, `0.5`
`location`	`location`	The Azure region to deploy this Webservice to. If not specified the Workspace location will be used. More details on available regions can be found here: ACI Regions
`authEnabled`	`auth_enabled`	Whether to enable auth for this Webservice. Defaults to False
`sslEnabled`	`ssl_enabled`	Whether to enable SSL for this Webservice. Defaults to False.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`appInsightsEnabled`	`enable_app_insights`	Whether to enable AppInsights for this Webservice. Defaults to False
`sslCertificate`	`ssl_cert_pem_file`	The cert file needed if SSL is enabled
`sslKey`	`ssl_key_pem_file`	The key file needed if SSL is enabled
`cname`	`ssl_cname`	The cname for if SSL is enabled
`dnsNameLabel`	`dns_name_label`	The dns name label for the scoring endpoint. If not specified a unique dns name label will be generated for the scoring endpoint.

The following JSON is an example deployment configuration for use with the CLI:

```
{
    "computeType": "aci",
    "containerResourceRequirements":
    {
        "cpu": 0.5,
        "memoryInGB": 1.0
    },
    "authEnabled": true,
    "sslEnabled": false,
    "appInsightsEnabled": false
}
```

Azure Kubernetes Service deployment configuration schema

The entries in the `deploymentconfig.json` document map to the parameters for AksWebservice.deploy_configuration. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`computeType`	NA	The compute target. For AKS, the value must be `aks`.
`autoScaler`	NA	Contains configuration elements for autoscale. See the autoscaler table.
`autoscaleEnabled`	`autoscale_enabled`	Whether to enable autoscaling for the web service. If `numReplicas` = `0`, `True`; otherwise, `False`.
`minReplicas`	`autoscale_min_replicas`	The minimum number of containers to use when autoscaling this web service. Default, `1`.
`maxReplicas`	`autoscale_max_replicas`	The maximum number of containers to use when autoscaling this web service. Default, `10`.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`refreshPeriodInSeconds`	`autoscale_refresh_seconds`	How often the autoscaler attempts to scale this web service. Default, `1`.
`targetUtilization`	`autoscale_target_utilization`	The target utilization (in percent out of 100) that the autoscaler should attempt to maintain for this web service. Default, `70`.
`dataCollection`	NA	Contains configuration elements for data collection.
`storageEnabled`	`collect_model_data`	Whether to enable model data collection for the web service. Default, `False`.
`authEnabled`	`auth_enabled`	Whether or not to enable key authentication for the web service. Both `tokenAuthEnabled` and `authEnabled` cannot be `True`. Default, `True`.
`tokenAuthEnabled`	`token_auth_enabled`	Whether or not to enable token authentication for the web service. Both `tokenAuthEnabled` and `authEnabled` cannot be `True`. Default, `False`.
`containerResourceRequirements`	NA	Container for the CPU and memory entities.
`cpu`	`cpu_cores`	The number of CPU cores to allocate for this web service. Defaults, `0.1`
`memoryInGB`	`memory_gb`	The amount of memory (in GB) to allocate for this web service. Default, `0.5`
`appInsightsEnabled`	`enable_app_insights`	Whether to enable Application Insights logging for the web service. Default, `False`.
`scoringTimeoutMs`	`scoring_timeout_ms`	A timeout to enforce for scoring calls to the web service. Default, `60000`.
`maxConcurrentRequestsPerContainer`	`replica_max_concurrent_requests`	The maximum concurrent requests per node for this web service. Default, `1`.
`maxQueueWaitMs`	`max_request_wait_time`	The maximum time a request will stay in thee queue (in milliseconds) before a 503 error is returned. Default, `500`.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
`numReplicas`	`num_replicas`	The number of containers to allocate for this web service. No default value. If this parameter is not set, the autoscaler is enabled by default.
`keys`	NA	Contains configuration elements for keys.
`primaryKey`	`primary_key`	A primary auth key to use for this Webservice
`secondaryKey`	`secondary_key`	A secondary auth key to use for this Webservice
`gpuCores`	`gpu_cores`	The number of GPU cores (per-container replica) to allocate for this Webservice. Default is 1. Only supports whole number values.
`livenessProbeRequirements`	NA	Contains configuration elements for liveness probe requirements.
`periodSeconds`	`period_seconds`	How often (in seconds) to perform the liveness probe. Default to 10 seconds. Minimum value is 1.
`initialDelaySeconds`	`initial_delay_seconds`	Number of seconds after the container has started before liveness probes are initiated. Defaults to 310
`timeoutSeconds`	`timeout_seconds`	Number of seconds after which the liveness probe times out. Defaults to 2 seconds. Minimum value is 1
`successThreshold`	`success_threshold`	Minimum consecutive successes for the liveness probe to be considered successful after having failed. Defaults to 1. Minimum value is 1.
`failureThreshold`	`failure_threshold`	When a Pod starts and the liveness probe fails, Kubernetes will try failureThreshold times before giving up. Defaults to 3. Minimum value is 1.
`namespace`	`namespace`	The Kubernetes namespace that the webservice is deployed into. Up to 63 lowercase alphanumeric ('a'-'z', '0'-'9') and hyphen ('-') characters. The first and last characters can't be hyphens.

The following JSON is an example deployment configuration for use with the CLI:

```
{
    "computeType": "aks",
    "autoScaler":
    {
        "autoscaleEnabled": true,
        "minReplicas": 1,
        "maxReplicas": 3,
        "refreshPeriodInSeconds": 1,
        "targetUtilization": 70
    },
    "dataCollection":
    {
        "storageEnabled": true
    },
    "authEnabled": true,
    "containerResourceRequirements":
    {
        "cpu": 0.5,
        "memoryInGB": 1.0
    }
}
```

Next steps

- Command reference for the Machine Learning CLI extension.
- Train and deploy machine learning models using Azure Pipelines

Algorithm & module reference for Azure Machine Learning designer (preview)

4/16/2020 • 2 minutes to read • Edit Online

This reference content provides the technical background on each of the machine learning algorithms and modules available in Azure Machine Learning designer (preview).

Each module represents a set of code that can run independently and perform a machine learning task, given the required inputs. A module might contain a particular algorithm, or perform a task that is important in machine learning, such as missing value replacement, or statistical analysis.

For help with choosing algorithms, see

- How to select algorithms
- Azure Machine Learning Algorithm Cheat Sheet

> **TIP**
> In any pipeline in the designer, you can get information about a specific module. Select the module, then select the **more help** link in the **Quick Help** pane.

Data preparation modules

FUNCTIONALITY	DESCRIPTION	MODULE
Data input and output	Move data from cloud sources into your pipeline. Write your results or intermediate data to Azure Storage, a SQL database, or Hive, while running a pipeline, or use cloud storage to exchange data between pipelines.	Enter Data Manually Export Data Import Data
Data transformation	Operations on data that are unique to machine learning, such as normalizing or binning data, dimensionality reduction, and converting data among various file formats.	Add Columns Add Rows Apply Math Operation Apply SQL Transformation Clean Missing Data Clip Values Convert to CSV Convert to Dataset Convert to Indicator Values Edit Metadata Join Data Normalize Data Partition and Sample Remove Duplicate Rows SMOTE Select Columns Transform Select Columns in Dataset Split Data

FUNCTIONALITY	DESCRIPTION	MODULE
Feature Selection	Select a subset of relevant, useful features to use in building an analytical model.	Filter Based Feature Selection Permutation Feature Importance
Statistical Functions	Provide a wide variety of statistical methods related to data science.	Summarize Data

Machine learning algorithms

FUNCTIONALITY	DESCRIPTION	MODULE
Regression	Predict a value.	Boosted Decision Tree Regression Decision Forest Regression Linear Regression Neural Network Regression
Clustering	Group data together.	K-Means Clustering
Classification	Predict a class. Choose from binary (two-class) or multiclass algorithms.	Multiclass Boosted Decision Tree Multiclass Decision Forest Multiclass Logistic Regression Multiclass Neural Network One vs. All Multiclass Two-Class Averaged Perceptron Two-Class Boosted Decision Tree Two-Class Decision Forest Two-Class Logistic Regression Two-Class Neural Network Two Class Support Vector Machine

Modules for building and evaluating models

FUNCTIONALITY	DESCRIPTION	MODULE
Model training	Run data through the algorithm.	Train Clustering Model Train Model Tune Model Hyperparameters
Model Scoring and Evaluation	Measure the accuracy of the trained model.	Apply Transformation Assign Data to Clusters Cross Validate Model Evaluate Model Score Model
Python language	Write code and embed it in a module to integrate Python with your pipeline.	Create Python Model Execute Python Script
R language	Write code and embed it in a module to integrate R with your pipeline.	Execute R Script
Text Analytics	Provide specialized computational tools for working with both structured and unstructured text.	Extract N Gram Features from Text Feature Hashing Preprocess Text Latent Dirichlet Allocation

FUNCTIONALITY	DESCRIPTION	MODULE
Recommendation	Build recommendation models.	Evaluate Recommender Score SVD Recommender Train SVD Recommender
Anomaly Detection	Build anomaly detection models.	PCA-Based Anomaly Detection Train Anomaly Detection Model

Web Service

Learn about the web service modules which are necessary for real-time inference in Azure Machine Learning designer.

Error messages

Learn about the error messages and exception codes you might encounter using modules in Azure Machine Learning designer.

Next steps

- Tutorial: Build a model in designer to predict auto prices

Azure machine learning monitoring data reference

3/8/2020 • 5 minutes to read • Edit Online

Learn about the data and resources collected by Azure Monitor from your Azure Machine Learning workspace. See Monitoring Azure Machine Learning for details on collecting and analyzing monitoring data.

Resource logs

The following table lists the properties for Azure Machine Learning resource logs when they're collected in Azure Monitor Logs or Azure Storage.

AmlComputeJobEvents table

PROPERTY	DESCRIPTION
TimeGenerated	Time when the log entry was generated
OperationName	Name of the operation associated with the log event
Category	Name of the log event, AmlComputeClusterNodeEvent
JobId	ID of the Job submitted
ExperimentId	ID of the Experiment
ExperimentName	Name of the Experiment
CustomerSubscriptionId	SubscriptionId where Experiment and Job as submitted
WorkspaceName	Name of the machine learning workspace
ClusterName	Name of the Cluster
ProvisioningState	State of the Job submission
ResourceGroupName	Name of the resource group
JobName	Name of the Job
ClusterId	ID of the cluster
EventType	Type of the Job event, e.g., JobSubmitted, JobRunning, JobFailed, JobSucceeded, etc.
ExecutionState	State of the job (the Run), e.g., Queued, Running, Succeeded, Failed
ErrorDetails	Details of job error
CreationApiVersion	Api version used to create the job

PROPERTY	DESCRIPTION
ClusterResourceGroupName	Resource group name of the cluster
TFWorkerCount	Count of TF workers
TFParameterServerCount	Count of TF parameter server
ToolType	Type of tool used
RunInContainer	Flag describing if job should be run inside a container
JobErrorMessage	detailed message of Job error
NodeId	ID of the node created where job is running

AmlComputeClusterEvents table

PROPERTY	DESCRIPTION
TimeGenerated	Time when the log entry was generated
OperationName	Name of the operation associated with the log event
Category	Name of the log event, AmlComputeClusterNodeEvent
ProvisioningState	Provisioning state of the cluster
ClusterName	Name of the cluster
ClusterType	Type of the cluster
CreatedBy	User who created the cluster
CoreCount	Count of the cores in the cluster
VmSize	Vm size of the cluster
VmPriority	Priority of the nodes created inside a cluster Dedicated/LowPriority
ScalingType	Type of cluster scaling manual/auto
InitialNodeCount	Initial node count of the cluster
MinimumNodeCount	Minimum node count of the cluster
MaximumNodeCount	Maximum node count of the cluster
NodeDeallocationOption	How the node should be deallocated
Publisher	Publisher of the cluster type

PROPERTY	DESCRIPTION
Offer	Offer with which the cluster is created
Sku	Sku of the Node/VM created inside cluster
Version	Version of the image used while Node/VM is created
SubnetId	SubnetId of the cluster
AllocationState	Cluster allocation state
CurrentNodeCount	Current node count of the cluster
TargetNodeCount	Target node count of the cluster while scaling up/down
EventType	Type of event during cluster creation.
NodeIdleTimeSecondsBeforeScaleDown	Idle time in seconds before cluster is scaled down
PreemptedNodeCount	Preempted node count of the cluster
IsResizeGrow	Flag indicating that cluster is scaling up
VmFamilyName	Name of the VM family of the nodes that can be created inside cluster
LeavingNodeCount	Leaving node count of the cluster
UnusableNodeCount	Unusable node count of the cluster
IdleNodeCount	Idle node count of the cluster
RunningNodeCount	Running node count of the cluster
PreparingNodeCount	Preparing node count of the cluster
QuotaAllocated	Allocated quota to the cluster
QuotaUtilized	Utilized quota of the cluster
AllocationStateTransitionTime	Transition time from one state to another
ClusterErrorCodes	Error code received during cluster creation or scaling
CreationApiVersion	Api version used while creating the cluster

AmlComputeClusterNodeEvents table

PROPERTY	DESCRIPTION
TimeGenerated	Time when the log entry was generated

PROPERTY	DESCRIPTION
OperationName	Name of the operation associated with the log event
Category	Name of the log event, AmlComputeClusterNodeEvent
ClusterName	Name of the cluster
NodeId	ID of the cluster node created
VmSize	Vm size of the node
VmFamilyName	Vm family to which the node belongs
VmPriority	Priority of the node created Dedicated/LowPriority
Publisher	Publisher of the vm image, e.g., microsoft-dsvm
Offer	Offer associated with the VM creation
Sku	Sku of the Node/VM created
Version	Version of the image used while Node/VM is created
ClusterCreationTime	Time when cluster was created
ResizeStartTime	Time when cluster scale up/down started
ResizeEndTime	Time when cluster scale up/down ended
NodeAllocationTime	Time when Node was allocated
NodeBootTime	Time when Node was booted up
StartTaskStartTime	Time when task was assigned to a node and started
StartTaskEndTime	Time when task assigned to a node ended
TotalE2ETimeInSeconds	Total time node was active

Metrics

The following tables list the platform metrics collected for Azure Machine Learning All metrics are stored in the namespace **Azure Machine Learning Workspace**.

Model

METRIC	UNIT	DESCRIPTION
Model deploy failed	Count	The number of model deployments that failed.
Model deploy started	Count	The number of model deployments started.

METRIC	UNIT	DESCRIPTION
Model deploy succeeded	Count	The number of model deployments that succeeded.
Model register failed	Count	The number of model registrations that failed.
Model register succeeded	Count	The number of model registrations that succeeded.

Quota

Quota information is for Azure Machine Learning compute only.

METRIC	UNIT	DESCRIPTION
Active cores	Count	The number of active compute cores.
Active nodes	Count	The number of active nodes.
Idle cores	Count	The number of idle compute cores.
Idle nodes	Count	The number of idle compute nodes.
Leaving cores	Count	The number of leaving cores.
Leaving nodes	Count	The number of leaving nodes.
Preempted cores	Count	The number of preempted cores.
Preempted nodes	Count	The number of preempted nodes.
Quota utilization percentage	Percent	The percentage of quota used.
Total cores	Count	The total cores.
Total nodes	Count	The total nodes.
Unusable cores	Count	The number of unusable cores.
Unusable nodes	Count	The number of unusable nodes.

The following are dimensions that can be used to filter quota metrics:

DIMENSION	METRIC(S) AVAILABLE WITH	DESCRIPTION
Cluster Name	All quota metrics	The name of the compute instance.
Vm Family Name	Quota utilization percentage	The name of the VM family used by the cluster.
Vm Priority	Quota utilization percentage	The priority of the VM.

Run

Information on training runs.

METRIC	UNIT	DESCRIPTION
Completed runs	Count	The number of completed runs.
Failed runs	Count	The number of failed runs.
Started runs	Count	The number of started runs.

The following are dimensions that can be used to filter run metrics:

DIMENSION	DESCRIPTION
ComputeType	The compute type that the run used.
PipelineStepType	The type of PipelineStep used in the run.
PublishedPipelineId	The ID of the published pipeline used in the run.
RunType	The type of run.

The valid values for the RunType dimension are:

VALUE	DESCRIPTION
Experiment	Non-pipeline runs.
PipelineRun	A pipeline run, which is the parent of a StepRun.
StepRun	A run for a pipeline step.
ReusedStepRun	A run for a pipeline step that reuses a previous run.

See Also

- See Monitoring Azure Machine Learning for a description of monitoring Azure Machine Learning.
- See Monitoring Azure resources with Azure Monitor for details on monitoring Azure resources.

Define machine learning pipelines in YAML

4/17/2020 • 9 minutes to read • Edit Online

Learn how to define your machine learning pipelines in YAML. When using the machine learning extension for the Azure CLI, many of the pipeline-related commands expect a YAML file that defines the pipeline.

The following table lists what is and is not currently supported when defining a pipeline in YAML:

STEP TYPE	SUPPORTED?
PythonScriptStep	Yes
AdlaStep	Yes
AzureBatchStep	Yes
DatabricksStep	Yes
DataTransferStep	Yes
AutoMLStep	No
HyperDriveStep	No
ModuleStep	Yes
MPIStep	No
EstimatorStep	No

Pipeline definition

A pipeline definition uses the following keys, which correspond to the Pipelines class:

YAML KEY	DESCRIPTION
`name`	The description of the pipeline.
`parameters`	Parameter(s) to the pipeline.
`data_reference`	Defines how and where data should be made available in a run.
`default_compute`	Default compute target where all steps in the pipeline run.
`steps`	The steps used in the pipeline.

Parameters

The `parameters` section uses the following keys, which correspond to the PipelineParameter class:

YAML KEY	DESCRIPTION
`type`	The value type of the parameter. Valid types are `string`, `int`, `float`, `bool`, or `datapath`.
`default`	The default value.

Each parameter is named. For example, the following YAML snippet defines three parameters named `NumIterationsParameter`, `DataPathParameter`, and `NodeCountParameter`:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        NumIterationsParameter:
            type: int
            default: 40
        DataPathParameter:
            type: datapath
            default:
                datastore: workspaceblobstore
                path_on_datastore: sample2.txt
        NodeCountParameter:
            type: int
            default: 4
```

Data reference

The `data_references` section uses the following keys, which correspond to the DataReference:

YAML KEY	DESCRIPTION
`datastore`	The datastore to reference.
`path_on_datastore`	The relative path in the backing storage for the data reference.

Each data reference is contained in a key. For example, the following YAML snippet defines a data reference stored in the key named `employee_data`:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        PipelineParam1:
            type: int
            default: 3
    data_references:
        employee_data:
            datastore: adftestadla
            path_on_datastore: "adla_sample/sample_input.csv"
```

Steps

Steps define a computational environment, along with the files to run on the environment. To define the type of a step, use the `type` key:

STEP TYPE	DESCRIPTION
`AdlaStep`	Runs a U-SQL script with Azure Data Lake Analytics. Corresponds to the AdlaStep class.
`AzureBatchStep`	Runs jobs using Azure Batch. Corresponds to the AzureBatchStep class.
`DatabricsStep`	Adds a Databricks notebook, Python script, or JAR. Corresponds to the DatabricksStep class.
`DataTransferStep`	Transfers data between storage options. Corresponds to the DataTransferStep class.
`PythonScriptStep`	Runs a Python script. Corresponds to the PythonScriptStep class.

ADLA step

YAML KEY	DESCRIPTION
`script_name`	The name of the U-SQL script (relative to the `source_directory`).
`compute_target`	The Azure Data Lake compute target to use for this step.
`parameters`	Parameters to the pipeline.
`inputs`	Inputs can be InputPortBinding, DataReference, PortDataReference, PipelineData, Dataset, DatasetDefinition, or PipelineDataset.
`outputs`	Outputs can be either PipelineData or OutputPortBinding.
`source_directory`	Directory that contains the script, assemblies, etc.
`priority`	The priority value to use for the current job.
`params`	Dictionary of name-value pairs.
`degree_of_parallelism`	The degree of parallelism to use for this job.
`runtime_version`	The runtime version of the Data Lake Analytics engine.
`allow_reuse`	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains an ADLA Step definition:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        PipelineParam1:
            type: int
            default: 3
    data_references:
        employee_data:
            datastore: adftestadla
            path_on_datastore: "adla_sample/sample_input.csv"
    default_compute: adlacomp
    steps:
        Step1:
            runconfig: "D:\\Yaml\\default_runconfig.yml"
            parameters:
                NUM_ITERATIONS_2:
                    source: PipelineParam1
                NUM_ITERATIONS_1: 7
            type: "AdlaStep"
            name: "MyAdlaStep"
            script_name: "sample_script.usql"
            source_directory: "D:\\scripts\\Adla"
            inputs:
                employee_data:
                    source: employee_data
            outputs:
                OutputData:
                    destination: Output4
                    datastore: adftestadla
                    bind_mode: mount
```

Azure Batch step

YAML KEY	DESCRIPTION
`compute_target`	The Azure Batch compute target to use for this step.
`inputs`	Inputs can be InputPortBinding, DataReference, PortDataReference, PipelineData, Dataset, DatasetDefinition, or PipelineDataset.
`outputs`	Outputs can be either PipelineData or OutputPortBinding.
`source_directory`	Directory that contains the module binaries, executable, assemblies, etc.
`executable`	Name of the command/executable that will be ran as part of this job.
`create_pool`	Boolean flag to indicate whether to create the pool before running the job.
`delete_batch_job_after_finish`	Boolean flag to indicate whether to delete the job from the Batch account after it's finished.
`delete_batch_pool_after_finish`	Boolean flag to indicate whether to delete the pool after the job finishes.
`is_positive_exit_code_failure`	Boolean flag to indicate if the job fails if the task exits with a positive code.

YAML KEY	DESCRIPTION
`vm_image_urn`	If `create_pool` is `True`, and VM uses `VirtualMachineConfiguration`.
`pool_id`	The ID of the pool where the job will run.
`allow_reuse`	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains an Azure Batch step definition:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        PipelineParam1:
            type: int
            default: 3
    data_references:
        input:
            datastore: workspaceblobstore
            path_on_datastore: "input.txt"
    default_compute: testbatch
    steps:
        Step1:
            runconfig: "D:\\Yaml\\default_runconfig.yml"
            parameters:
                NUM_ITERATIONS_2:
                    source: PipelineParam1
                NUM_ITERATIONS_1: 7
            type: "AzureBatchStep"
            name: "MyAzureBatchStep"
            pool_id: "MyPoolName"
            create_pool: true
            executable: "azurebatch.cmd"
            source_directory: "D:\\scripts\\AureBatch"
            allow_reuse: false
            inputs:
                input:
                    source: input
            outputs:
                output:
                    destination: output
                    datastore: workspaceblobstore
```

Databricks step

YAML KEY	DESCRIPTION
`compute_target`	The Azure Databricks compute target to use for this step.
`inputs`	Inputs can be InputPortBinding, DataReference, PortDataReference, PipelineData, Dataset, DatasetDefinition, or PipelineDataset.
`outputs`	Outputs can be either PipelineData or OutputPortBinding.
`run_name`	The name in Databricks for this run.

YAML KEY	DESCRIPTION
`source_directory`	Directory that contains the script and other files.
`num_workers`	The static number of workers for the Databricks run cluster.
`runconfig`	The path to a `.runconfig` file. This file is a YAML representation of the RunConfiguration class. For more information on the structure of this file, see runconfigschema.json.
`allow_reuse`	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a Databricks step:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        PipelineParam1:
            type: int
            default: 3
    data_references:
        adls_test_data:
            datastore: adftestadla
            path_on_datastore: "testdata"
        blob_test_data:
            datastore: workspaceblobstore
            path_on_datastore: "dbtest"
    default_compute: mydatabricks
    steps:
        Step1:
            runconfig: "D:\\Yaml\\default_runconfig.yml"
            parameters:
                NUM_ITERATIONS_2:
                    source: PipelineParam1
                NUM_ITERATIONS_1: 7
            type: "DatabricksStep"
            name: "MyDatabrickStep"
            run_name: "DatabricksRun"
            python_script_name: "train-db-local.py"
            source_directory: "D:\\scripts\\Databricks"
            num_workers: 1
            allow_reuse: true
            inputs:
                blob_test_data:
                    source: blob_test_data
            outputs:
                OutputData:
                    destination: Output4
                    datastore: workspaceblobstore
                    bind_mode: mount
```

Data transfer step

YAML KEY	DESCRIPTION
`compute_target`	The Azure Data Factory compute target to use for this step.

YAML KEY	DESCRIPTION
`source_data_reference`	Input connection that serves as the source of data transfer operations. Supported values are InputPortBinding, DataReference, PortDataReference, PipelineData, Dataset, DatasetDefinition, or PipelineDataset.
`destination_data_reference`	Input connection that serves as the destination of data transfer operations. Supported values are PipelineData and OutputPortBinding.
`allow_reuse`	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a data transfer step:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        PipelineParam1:
            type: int
            default: 3
    data_references:
        adls_test_data:
            datastore: adftestadla
            path_on_datastore: "testdata"
        blob_test_data:
            datastore: workspaceblobstore
            path_on_datastore: "testdata"
    default_compute: adftest
    steps:
        Step1:
            runconfig: "D:\\Yaml\\default_runconfig.yml"
            parameters:
                NUM_ITERATIONS_2:
                    source: PipelineParam1
                NUM_ITERATIONS_1: 7
            type: "DataTransferStep"
            name: "MyDataTransferStep"
            adla_compute_name: adftest
            source_data_reference:
                adls_test_data:
                    source: adls_test_data
            destination_data_reference:
                blob_test_data:
                    source: blob_test_data
```

Python script step

YAML KEY	DESCRIPTION
`inputs`	Inputs can be InputPortBinding, DataReference, PortDataReference, PipelineData, Dataset, DatasetDefinition, or PipelineDataset.
`outputs`	Outputs can be either PipelineData or OutputPortBinding.
`script_name`	The name of the Python script (relative to `source_directory`).

YAML KEY	DESCRIPTION
`source_directory`	Directory that contains the script, Conda environment, etc.
`runconfig`	The path to a `.runconfig` file. This file is a YAML representation of the RunConfiguration class. For more information on the structure of this file, see runconfig.json.
`allow_reuse`	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a Python script step:

```
pipeline:
    name: SamplePipelineFromYaml
    parameters:
        PipelineParam1:
            type: int
            default: 3
    data_references:
        DataReference1:
            datastore: workspaceblobstore
            path_on_datastore: testfolder/sample.txt
    default_compute: cpu-cluster
    steps:
        Step1:
            runconfig: "D:\\Yaml\\default_runconfig.yml"
            parameters:
                NUM_ITERATIONS_2:
                    source: PipelineParam1
                NUM_ITERATIONS_1: 7
            type: "PythonScriptStep"
            name: "MyPythonScriptStep"
            script_name: "train.py"
            allow_reuse: True
            source_directory: "D:\\scripts\\PythonScript"
            inputs:
                InputData:
                    source: DataReference1
            outputs:
                OutputData:
                    destination: Output4
                    datastore: workspaceblobstore
                    bind_mode: mount
```

Pipeline with multiple steps

YAML KEY	DESCRIPTION
`steps`	Sequence of one or more PipelineStep definitions. Note that the `destination` of one step's `outputs` become the keys to the `inputs` of the .

```
pipeline:
    name: SamplePipelineFromYAML
    description: Sample multistep YAML pipeline
    data_references:
        TitanicDS:
            dataset_name: 'titanic_ds'
            bind_mode: download
    default_compute: cpu-cluster
    steps:
        Dataprep:
            type: "PythonScriptStep"
            name: "DataPrep Step"
            compute: cpu-cluster
            runconfig: ".\\default_runconfig.yml"
            script_name: "prep.py"
            arguments:
            - '--train_path'
            - output:train_path
            - '--test_path'
            - output:test_path
            allow_reuse: True
            inputs:
                titanic_ds:
                    source: TitanicDS
                    bind_mode: download
            outputs:
                train_path:
                    destination: train_csv
                    datastore: workspaceblobstore
                test_path:
                    destination: test_csv
        Training:
            type: "PythonScriptStep"
            name: "Training Step"
            compute: cpu-cluster
            runconfig: ".\\default_runconfig.yml"
            script_name: "train.py"
            arguments:
            - "--train_path"
            - input:train_path
            - "--test_path"
            - input:test_path
            inputs:
                train_path:
                    source: train_csv
                    bind_mode: download
                test_path:
                    source: test_csv
                    bind_mode: download
```

Schedules

When defining the schedule for a pipeline, it can be either datastore-triggered or recurring based on a time interval. The following are the keys used to define a schedule:

YAML KEY	DESCRIPTION
`description`	A description of the schedule.
`recurrence`	Contains recurrence settings, if the schedule is recurring.

YAML KEY	DESCRIPTION
`pipeline_parameters`	Any parameters that are required by the pipeline.
`wait_for_provisioning`	Whether to wait for provisioning of the schedule to complete.
`wait_timeout`	The number of seconds to wait before timing out.
`datastore_name`	The datastore to monitor for modified/added blobs.
`polling_interval`	How long, in minutes, between polling for modified/added blobs. Default value: 5 minutes. Only supported for datastore schedules.
`data_path_parameter_name`	The name of the data path pipeline parameter to set with the changed blob path. Only supported for datastore schedules.
`continue_on_step_failure`	Whether to continue execution of other steps in the submitted PipelineRun if a step fails. If provided, will override the `continue_on_step_failure` setting of the pipeline.
`path_on_datastore`	Optional. The path on the datastore to monitor for modified/added blobs. The path is under the container for the datastore, so the actual path the schedule monitors is container/`path_on_datastore`. If none, the datastore container is monitored. Additions/modifications made in a subfolder of the `path_on_datastore` are not monitored. Only supported for datastore schedules.

The following example contains the definition for a datastore-triggered schedule:

```
Schedule:
      description: "Test create with datastore"
      recurrence: ~
      pipeline_parameters: {}
      wait_for_provisioning: True
      wait_timeout: 3600
      datastore_name: "workspaceblobstore"
      polling_interval: 5
      data_path_parameter_name: "input_data"
      continue_on_step_failure: None
      path_on_datastore: "file/path"
```

When defining a **recurring schedule**, use the following keys under `recurrence`:

YAML KEY	DESCRIPTION
`frequency`	How often the schedule recurs. Valid values are `"Minute"`, `"Hour"`, `"Day"`, `"Week"`, or `"Month"`.
`interval`	How often the schedule fires. The integer value is the number of time units to wait until the schedule fires again.

YAML KEY	DESCRIPTION
`start_time`	The start time for the schedule. The string format of the value is `YYYY-MM-DDThh:mm:ss`. If no start time is provided, the first workload is run instantly and future workloads are run based on the schedule. If the start time is in the past, the first workload is run at the next calculated run time.
`time_zone`	The time zone for the start time. If no time zone is provided, UTC is used.
`hours`	If `frequency` is `"Day"` or `"Week"`, you can specify one or more integers from 0 to 23, separated by commas, as the hours of the day when the pipeline should run. Only `time_of_day` or `hours` and `minutes` can be used.
`minutes`	If `frequency` is `"Day"` or `"Week"`, you can specify one or more integers from 0 to 59, separated by commas, as the minutes of the hour when the pipeline should run. Only `time_of_day` or `hours` and `minutes` can be used.
`time_of_day`	If `frequency` is `"Day"` or `"Week"`, you can specify a time of day for the schedule to run. The string format of the value is `hh:mm`. Only `time_of_day` or `hours` and `minutes` can be used.
`week_days`	If `frequency` is `"Week"`, you can specify one or more days, separated by commas, when the schedule should run. Valid values are `"Monday"`, `"Tuesday"`, `"Wednesday"`, `"Thursday"`, `"Friday"`, `"Saturday"`, and `"Sunday"`.

The following example contains the definition for a recurring schedule:

```
Schedule:
    description: "Test create with recurrence"
    recurrence:
        frequency: Week # Can be "Minute", "Hour", "Day", "Week", or "Month".
        interval: 1 # how often fires
        start_time: 2019-06-07T10:50:00
        time_zone: UTC
        hours:
        - 1
        minutes:
        - 0
        time_of_day: null
        week_days:
        - Friday
    pipeline_parameters:
        'a': 1
    wait_for_provisioning: True
    wait_timeout: 3600
    datastore_name: ~
    polling_interval: ~
    data_path_parameter_name: ~
    continue_on_step_failure: None
    path_on_datastore: ~
```

Next steps

Learn how to use the CLI extension for Azure Machine Learning.

Azure Machine Learning release notes

4/19/2020 • 87 minutes to read • Edit Online

In this article, learn about Azure Machine Learning releases. For the full SDK reference content, visit the Azure Machine Learning's **main SDK for Python** reference page.

See the list of known issues to learn about known bugs and workarounds.

2020-04-13

Azure Machine Learning SDK for Python v1.3.0

- Bug fixes and improvements
  - azureml-automl-core
    - Added additional telemetry around post-training operations.
    - Speeds up automatic ARIMA training by using conditional sum of squares (CSS) training for series of length longer than 100. Note that the length used is stored as the constant ARIMA_TRIGGER_CSS_TRAINING_LENGTH w/in the TimeSeriesInternal class at /src/azureml-automl-core/azureml/automl/core/shared/constants.py
    - The user logging of forecasting runs was improved, now more information on what phase is currently running will be shown in the log
    - Disallowed target_rolling_window_size to be set to values less then 2
  - azureml-automl-runtime
    - Improved the error message shown when duplicated timestamps are found.
    - Disallowed target_rolling_window_size to be set to values less then 2.
    - Fixed the lag imputation failure. The issue was caused by the insufficient number of observations needed to seasonally decompose a series. The "de-seasonalized" data is used to compute a partial autocorrelation function (PACF) to determine the lag length.
    - Enabled column purpose featurization customization for forecasting tasks by featurization config. Numerical and Categorical as column purpose for forecasting tasks are now supported.
    - Enabled drop column featurization customization for forecasting tasks by featurization config.
    - Enabled imputation customization for forecasting tasks by featurization config. Constant value imputation for target column and mean, median, most_frequent and constant value imputation for training data are now supported.
  - azureml-contrib-pipeline-steps
    - Accept string compute names to be passed to ParallelRunConfig
  - azureml-core
    - Added Environment.clone(new_name) API to create a copy of Environment object
    - Environment.docker.base_dockerfile accepts filepath. If able to resolve a file, the content will be read into base_dockerfile environment property
    - Automatically reset mutually exclusive values for base_image and base_dockerfile when user manually sets a value in Environment.docker
    - Dataset: fixed dataset download failure if data path containing unicode characters
    - Dataset: improved dataset mount caching mechanism to respect the minimum disk space requirement in Azure Machine Learning Compute, which avoids making the node unusable and causing the job to be canceled
    - Added user_managed flag in RSection which indicates whether the environment is managed by

user or by AzureML.
    - Dataset: we add an index for the timeseries column when you access a timeseries dataset as a pandas dataframes, which is used to speed up access to timeseries based data access. Previously, the index was given the same name as the timestamp column, confusing users about which is the actual timestamp column and which is the index. We now don't give any specific name to the index since it should not be used as a column.
  - **azureml-dataprep**
    - Fixed dataset authentication issue in sovereign cloud
    - Fixed `Dataset.to_spark_dataframe` failure for datasets created from Azure PostgreSQL datastores
  - **azureml-interpret**
    - Added global scores to visualization if local importance values are sparse
    - Updated azureml-interpret to use interpret-community 0.9.*
    - Fixed issue with downloading explanation that had sparse evaluation data
    - Added support of sparse format of the explanation object in AutoML
  - **azureml-pipeline-core**
    - Support ComputeInstance as compute target in pipelines
  - **azureml-train-automl-client**
    - Added additional telemetry around post-training operations.
    - Fixed the regression in early stopping
    - Deprecated azureml.dprep.Dataflow as a valid type for input data.
    - Changing default AutoML experiment timeout to 6 days.
  - **azureml-train-automl-runtime**
    - Added additional telemetry around post-training operations.
    - added sparse automl e2e support
  - **azureml-opendatasets**
    - Added additional telemetry for service monitor.
    - Enable frontdoor for blob to increase stability

2020-03-23

Azure Machine Learning SDK for Python v1.2.0

- **Breaking changes**
  - Drop support for python 2.7
- **Bug fixes and improvements**
  - **azure-cli-ml**
    - Adds "--subscription-id" to `az ml model/computetarget/service` commands in the CLI
    - Adding support for passing customer managed key(CMK) vault_url, key_name and key_version for ACI deployment
  - **azureml-automl-core**
    - Enabled customized imputation with constant value for both X and y data forecasting tasks.
    - Fixed the issue in with showing error messages to user.
  - **azureml-automl-runtime**
    - Fixed the issue in with forecasting on the data sets, containing grains with only one row
    - Decreased the amount of memory required by the forecasting tasks.
    - Added better error messages if time column has incorrect format.
    - Enabled customized imputation with constant value for both X and y data forecasting tasks.
  - **azureml-core**

    - Add support for loading ServicePrincipal from environment variables: AZUREML_SERVICE_PRINCIPAL_ID, AZUREML_SERVICE_PRINCIPAL_TENANT_ID, and AZUREML_SERVICE_PRINCIPAL_PASSWORD
    - Introduced a new parameter `support_multi_line` to `Dataset.Tabular.from_delimited_files`: By default (`support_multi_line=False`), all line breaks, including those in quoted field values, will be interpreted as a record break. Reading data this way is faster and more optimized for parallel execution on multiple CPU cores. However, it may result in silently producing more records with misaligned field values. This should be set to `True` when the delimited files are known to contain quoted line breaks.
    - Added the ability to register ADLS Gen2 in the Azure Machine Learning CLI
    - Renamed parameter 'fine_grain_timestamp' to 'timestamp' and parameter 'coarse_grain_timestamp' to 'partition_timestamp' for the with_timestamp_columns() method in TabularDataset to better reflect the usage of the parameters.
    - Increased max experiment name length to 255.
  - **azureml-interpret**
    - Updated azureml-interpret to interpret-community 0.7.*
  - **azureml-sdk**
    - Changing to dependencies with compatible version Tilde for the support of patching in pre-release and stable releases.

2020-03-11

Azure Machine Learning SDK for Python v1.1.5

- **Feature deprecation**
  - **Python 2.7**
    - Last version to support python 2.7
- **Breaking changes**
  - **Semantic Versioning 2.0.0**
    - Starting with version 1.1 Azure ML Python SDK adopts Semantic Versioning 2.0.0. Read more here. All subsequent versions will follow new numbering scheme and semantic versioning contract.
- **Bug fixes and improvements**
  - **azure-cli-ml**
    - Change the endpoint CLI command name from 'az ml endpoint aks' to 'az ml endpoint realtime' for consistency.
    - update CLI installation instructions for stable and experimental branch CLI
    - Single instance profiling was fixed to produce a recommendation and was made available in core sdk.
  - **azureml-automl-core**
    - Enabled the Batch mode inference (taking multiple rows once) for automl ONNX models
    - Improved the detection of frequency on the data sets, lacking data or containing irregular data points
    - Added the ability to remove data points not complying with the dominant frequency.
    - Changed the input of the constructor to take a list of options to apply the imputation options for corresponding columns.
    - The error logging has been improved.
  - **azureml-automl-runtime**
    - Fixed the issue with the error thrown if the grain which was not present in the training set

appeared in the test set
    - Removed the y_query requirement during scoring on forecasting service
    - Fixed the issue with forecasting when the data set contains short grains with long time gaps.
    - Fixed the issue when the auto max horizon is turned on and the date column contains dates in form of strings. Proper conversion and error messages were added for when conversion to date is not possible
    - Using native NumPy and SciPy for serializing and deserializing intermediate data for FileCacheStore (used for local AutoML runs)
    - Fixed a bug where failed child runs could get stuck in Running state.
    - Increased speed of featurization.
    - Fixed the frequency check during scoring, now the forecasting tasks do not require strict frequency equivalence between train and test set.
    - Changed the input of the constructor to take a list of options to apply the imputation options for corresponding columns.
    - Fixed errors related to lag type selection.
    - Fixed the unclassified error raised on the data sets, having grains with the single row
    - Fixed the issue with frequency detection slowness.
    - Fixes a bug in AutoML exception handling that caused the real reason for training failure to be replaced by an AttributeError.
- **azureml-cli-common**
    - Single instance profiling was fixed to produce a recommendation and was made available in core sdk.
- **azureml-contrib-mir**
    - Adds functionality in the MirWebservice class to retrieve the Access Token
    - Use token auth for MirWebservice by default during MirWebservice.run() call - Only refresh if call fails
    - Mir webservice deployment now requires proper Skus [Standard_DS2_v2, Standard_F16, Standard_A2_v2] instead of [Ds2v2, A2v2, and F16] respectively.
- **azureml-contrib-pipeline-steps**
    - Optional parameter side_inputs added to ParallelRunStep. This parameter can be used to mount folder on the container. Currently supported types are DataReference and PipelineData.
    - Parameters passed in ParallelRunConfig can be overwritten by passing pipeline parameters now. New pipeline parameters supported aml_mini_batch_size, aml_error_threshold, aml_logging_level, aml_run_invocation_timeout (aml_node_count and aml_process_count_per_node are already part of earlier release).
- **azureml-core**
    - Deployed AzureML Webservices will now default to `INFO` logging. This can be controlled by setting the `AZUREML_LOG_LEVEL` environment variable in the deployed service.
    - Python sdk uses discovery service to use 'api' endpoint instead of 'pipelines'.
    - Swap to the new routes in all SDK calls
    - Changes routing of calls to the ModelManagementService to a new unified structure
        - Made workspace update method publicly available.
        - Added image_build_compute parameter in workspace update method to allow user updating the compute for image build
    - Added deprecation messages to the old profiling workflow. Fixed profiling cpu and memory limits
    - Added RSection as part of Environment to run R jobs
    - Added validation to `Dataset.mount` to raise error when source of the dataset is not accessible or does not contain any data.

    - Added `--grant-workspace-msi-access` as an additional parameter for the Datastore CLI for registering Azure Blob Container which will allow you to register Blob Container that is behind a VNet
    - Single instance profiling was fixed to produce a recommendation and was made available in core sdk.
    - Fixed the issue in aks.py _deploy
    - Validates the integrity of models being uploaded to avoid silent storage failures.
    - User may now specify a value for the auth key when regenerating keys for webservices.
    - Fixed bug where uppercase letters cannot be used as dataset's input name
  - **azureml-defaults**
    - `azureml-dataprep` will now be installed as part of `azureml-defaults`. It is no longer required to install dataprep[fuse] manually on compute targets to mount datasets.
  - **azureml-interpret**
    - Updated azureml-interpret to interpret-community 0.6.*
    - Updated azureml-interpret to depend on interpret-community 0.5.0
    - Added azureml-style exceptions to azureml-interpret
    - Fixed DeepScoringExplainer serialization for keras models
  - **azureml-mlflow**
    - Add support for sovereign clouds to azureml.mlflow
  - **azureml-pipeline-core**
    - Pipeline batch scoring notebook now uses ParallelRunStep
    - Fixed a bug where PythonScriptStep results could be incorrectly reused despite changing the arguments list
    - Added the ability to set columns' type when calling the parse_* methods on `PipelineOutputFileDataset`
  - **azureml-pipeline-steps**
    - Moved the `AutoMLStep` to the `azureml-pipeline-steps` package. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.
    - Added documentation example for dataset as PythonScriptStep input
  - **azureml-tensorboard**
    - updated azureml-tensorboard to support tensorflow 2.0
    - Show correct port number when using a custom Tensorboard port on a Compute Instance
  - **azureml-train-automl-client**
    - Fixed an issue where certain packages may be installed at incorrect versions on remote runs.
    - fixed FeaturizationConfig overriding issue that filters custom featurization config.
  - **azureml-train-automl-runtime**
    - Fixed the issue with frequency detection in the remote runs
    - Moved the `AutoMLStep` in the `azureml-pipeline-steps` package. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.
  - **azureml-train-core**
    - Supporting PyTorch version 1.4 in the PyTorch Estimator

2020-03-02

Azure Machine Learning SDK for Python v1.1.2rc0 (Pre-release)

- Bug fixes and improvements
  - **azureml-automl-core**

    - Enabled the Batch mode inference (taking multiple rows once) for automl ONNX models
    - Improved the detection of frequency on the data sets, lacking data or containing irregular data points
    - Added the ability to remove data points not complying with the dominant frequrncy.
  - **azureml-automl-runtime**
    - Fixed the issue with the error thrown if the grain which was not present in the training set appeared in the test set
    - Removed the y_query requirement during scoring on forecasting service
  - **azureml-contrib-mir**
    - Adds functionality in the MirWebservice class to retrieve the Access Token
  - **azureml-core**
    - Deployed AzureML Webservices will now default to `INFO` logging. This can be controlled by setting the `AZUREML_LOG_LEVEL` environment variable in the deployed service.
    - Fix iterating on `Dataset.get_all` to return all datasets registered with the workspace.
    - Improve error message when invalid type is passed to `path` argument of dataset creation APIs.
    - Python sdk uses discovery service to use 'api' endpoint instead of 'pipelines'.
    - Swap to the new routes in all SDK calls
    - Changes routing of calls to the ModelManagementService to a new unified structure
      - Made workspace update method publicly available.
      - Added image_build_compute parameter in workspace update method to allow user updating the compute for image build
    - Added deprecation messages to the old profiling workflow. Fixed profiling cpu and memory limits
  - **azureml-interpret**
    - update azureml-interpret to interpret-community 0.6.*
  - **azureml-mlflow**
    - Add support for sovereign clouds to azureml.mlflow
  - **azureml-pipeline-steps**
    - Moved the `AutoMLStep` to the `azureml-pipeline-steps package`. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.
  - **azureml-train-automl-client**
    - Fixed an issue where certain packages may be installed at incorrect versions on remote runs.
  - **azureml-train-automl-runtime**
    - Fixed the issue with frequency detection in the remote runs
    - Moved the `AutoMLStep` to the `azureml-pipeline-steps package`. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.
  - **azureml-train-core**
    - Moved the `AutoMLStep` to the `azureml-pipeline-steps package`. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.

2020-02-18

Azure Machine Learning SDK for Python v1.1.1rc0 (Pre-release)

- **Bug fixes and improvements**
  - **azure-cli-ml**
    - Single instance profiling was fixed to produce a recommendation and was made available in core sdk.
  - **azureml-automl-core**
    - The error logging has been improved.

- **azureml-automl-runtime**
  - Fixed the issue with forecasting when the data set contains short grains with long time gaps.
  - Fixed the issue when the auto max horizon is turned on and the date column contains dates in form of strings. We added proper conversion and sensible error if conversion to date is not possible
  - Using native NumPy and SciPy for serializing and deserializing intermediate data for FileCacheStore (used for local AutoML runs)
  - Fixed a bug where failed child runs could get stuck in Running state.
- **azureml-cli-common**
  - Single instance profiling was fixed to produce a recommendation and was made available in core sdk.
- **azureml-core**
  - Added `--grant-workspace-msi-access` as an additional parameter for the Datastore CLI for registering Azure Blob Container which will allow you to register Blob Container that is behind a VNet
  - Single instance profiling was fixed to produce a recommendation and was made available in core sdk.
  - Fixed the issue in aks.py _deploy
  - Validates the integrity of models being uploaded to avoid silent storage failures.
- **azureml-interpret**
  - added azureml-style exceptions to azureml-interpret
  - fixed DeepScoringExplainer serialization for keras models
- **azureml-pipeline-core**
  - Pipeline batch scoring notebook now uses ParallelRunStep
- **azureml-pipeline-steps**
  - Moved the `AutoMLStep` in the `azureml-pipeline-steps` package. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.
- **azureml-contrib-pipeline-steps**
  - Optional parameter side_inputs added to ParallelRunStep. This parameter can be used to mount folder on the container. Currently supported types are DataReference and PipelineData.
- **azureml-tensorboard**
  - updated azureml-tensorboard to support tensorflow 2.0
- **azureml-train-automl-client**
  - fixed FeaturizationConfig overriding issue that filters custom featurization config.
- **azureml-train-automl-runtime**
  - Moved the `AutoMLStep` in the `azureml-pipeline-steps` package. Deprecated the `AutoMLStep` within `azureml-train-automl-runtime`.
- **azureml-train-core**
  - Supporting PyTorch version 1.4 in the PyTorch Estimator

2020-02-04

Azure Machine Learning SDK for Python v1.1.0rc0 (Pre-release)

- **Breaking changes**
  - **Semantic Versioning 2.0.0**
    - Starting with version 1.1 Azure ML Python SDK adopts Semantic Versioning 2.0.0. Read more here. All subsequent versions will follow new numbering scheme and semantic versioning contract.

- Bug fixes and improvements
  - azureml-automl-runtime
    - Increased speed of featurization.
    - Fixed the frequency check during scoring, now in the forecasting tasks we do not require strict frequency equivalence between train and test set.
  - azureml-core
    - User may now specify a value for the auth key when regenerating keys for webservices.
  - azureml-interpret
    - Updated azureml-interpret to depend on interpret-community 0.5.0
  - azureml-pipeline-core
    - Fixed a bug where PythonScriptStep results could be incorrectly reused despite changing the arguments list
  - azureml-pipeline-steps
    - Added documentation example for dataset as PythonScriptStep input
  - azureml-contrib-pipeline-steps
    - Parameters passed in ParallelRunConfig can be overwritten by passing pipeline parameters now. New pipeline parameters supported aml_mini_batch_size, aml_error_threshold, aml_logging_level, aml_run_invocation_timeout (aml_node_count and aml_process_count_per_node are already part of earlier release).

2020-01-21

Azure Machine Learning SDK for Python v1.0.85

- New features
  - azureml-core
    - Get the current core usage and quota limitation for AmlCompute resources in a given workspace and subscription
  - azureml-contrib-pipeline-steps
    - Enable user to pass tabular dataset as intermediate result from previous step to parallelrunstep
- Bug fixes and improvements
  - azureml-automl-runtime
    - Removed the requirement of y_query column in the request to the deployed forecasting service.
    - The 'y_query' was removed from the Dominick's Orange Juice notebook service request section.
    - Fixed the bug preventing forecasting on the deployed models, operating on data sets with date time columns.
    - Added Matthews Correlation Coefficient as a classification metric, for both binary and multiclass classification.
  - azureml-contrib-interpret
    - Removed text explainers from azureml-contrib-interpret as text explanation has been moved to the interpret-text repo which will be released soon.
  - azureml-core
    - Dataset: usages for file dataset no longer depends on numpy and pandas to be installed in the python env.
    - Changed LocalWebservice.wait_for_deployment() to check the status of the local Docker container before trying to ping its health endpoint, greatly reducing the amount of time it takes to report a failed deployment.

- Fixed the initialization of an internal property used in LocalWebservice.reload() when the service object is created from an existing deployment using the LocalWebservice() constructor.
- Edited error message for clarification.
- Added a new method called get_access_token() to AksWebservice that will return AksServiceAccessToken object, which contains access token, refresh after timestamp, expiry on timestamp and token type.
- Deprecated existing get_token() method in AksWebservice as the new method returns all of the information this method returns.
- Modified output of az ml service get-access-token command. Renamed token to accessToken and refreshBy to refreshAfter. Added expiryOn and tokenType properties.
- Fixed get_active_runs

- **azureml-explain-model**
  - updated shap to 0.33.0 and interpret-community to 0.4.*
- **azureml-interpret**
  - updated shap to 0.33.0 and interpret-community to 0.4.*
- **azureml-train-automl-runtime**
  - Added Matthews Correlation Coefficient as a classification metric, for both binary and multiclass classification.
  - Deprecate preprocess flag from code and replaced with featurization -featurization is on by default

2020-01-06

Azure Machine Learning SDK for Python v1.0.83

- **New features**
  - Dataset: Add two options `on_error` and `out_of_range_datetime` for `to_pandas_dataframe` to fail when data has error values instead of filling them with `None`.
  - Workspace: Added the `hbi_workspace` flag for workspaces with sensitive data that enables further encryption and disables advanced diagnostics on workspaces. We also added support for bringing your own keys for the associated Cosmos DB instance, by specifying the `cmk_keyvault` and `resource_cmk_uri` parameters when creating a workspace, which creates a Cosmos DB instance in your subscription while provisioning your workspace. Read more here.
- **Bug fixes and improvements**
  - **azureml-automl-runtime**
    - Fixed a regression that caused a TypeError to be raised when running AutoML on Python versions below 3.5.4.
  - **azureml-core**
    - Fixed bug in `datastore.upload_files` where relative path that didn't start with `./` was not able to be used.
    - Added deprecation messages for all Image class codepaths
    - Fixed Model Management URL construction for Mooncake region.
    - Fixed issue where models using source_dir couldn't be packaged for Azure Functions.
    - Added an option to Environment.build_local() to push an image into AzureML workspace container registry
    - Updated the SDK to use new token library on azure synapse in a back compatible manner.
  - **azureml-interpret**
    - Fixed bug where None was returned when no explanations were available for download. Now raises an exception, matching behavior elsewhere.

- azureml-pipeline-steps
  - Disallowed passing `DatasetConsumptionConfig`s to `Estimator`'s `inputs` parameter when the `Estimator` will be used in an `EstimatorStep`.
- azureml-sdk
  - Added AutoML client to azureml-sdk package, enabling remote AutoML runs to be submitted without installing the full AutoML package.
- azureml-train-automl-client
  - Corrected alignment on console output for automl runs
  - Fixed a bug where incorrect version of pandas may be installed on remote amlcompute.

2019-12-23

Azure Machine Learning SDK for Python v1.0.81

- Bug fixes and improvements
  - azureml-contrib-interpret
    - defer shap dependency to interpret-community from azureml-interpret
  - azureml-core
    - Compute target can now be specified as a parameter to the corresponding deployment config objects. This is specifically the name of the compute target to deploy to, not the SDK object.
    - Added CreatedBy information to Model and Service objects. May be accessed through *.created_by*
    - Fixed ContainerImage.run(), which was not correctly setting up the Docker container's HTTP port.
    - Make `azureml-dataprep` optional for `az ml dataset register` cli command
    - Fixed a bug where `TabularDataset.to_pandas_dataframe` would incorrectly fall back to an alternate reader and print out a warning.
  - azureml-explain-model
    - defer shap dependency to interpret-community from azureml-interpret
  - azureml-pipeline-core
    - Added new pipeline step `NotebookRunnerStep`, to run a local notebook as a step in pipeline.
    - Removed deprecated get_all functions for PublishedPipelines, Schedules, and PipelineEndpoints
  - azureml-train-automl-client
    - Started deprecation of data_script as an input to AutoML.

2019-12-09

Azure Machine Learning SDK for Python v1.0.79

- Bug fixes and improvements
  - azureml-automl-core
    - Removed featurizationConfig to be logged
      - Updated logging to log "auto"/"off"/"customized" only.
  - azureml-automl-runtime
    - Added support for pandas. Series and pandas. Categorical for detecting column data type. Previously only supported numpy.ndarray
      - Added related code changes to handle categorical dtype correctly.
    - The forecast function interface was improved: the y_pred parameter was made optional. -The docstrings were improved.
  - azureml-contrib-dataset
    - Fixed a bug where labeled datasets could not be mounted.
  - azureml-core

    - Bug fix for `Environment.from_existing_conda_environment(name, conda_environment_name)`. User can create an instance of Environment that is exact replica of the local environment
    - Changed time series-related Datasets methods to `include_boundary=True` by default.
  - **azureml-train-automl-client**
    - Fixed issue where validation results are not printed when show output is set to false.

2019-11-25

Azure Machine Learning SDK for Python v1.0.76

- **Breaking changes**
  - Azureml-Train-AutoML upgrade issues
    - Upgrading to azureml-train-automl>=1.0.76 from azureml-train-automl<1.0.76 can cause partial installations, causing some automl imports to fail. To resolve this, you can run the setup script found at https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/automated-machine-learning/automl_setup.cmd. Or if you are using pip directly you can:
      - "pip install --upgrade azureml-train-automl"
      - "pip install --ignore-installed azureml-train-automl-client"
    - or you can uninstall the old version before upgrading
      - "pip uninstall azureml-train-automl"
      - "pip install azureml-train-automl"
- **Bug fixes and improvements**
  - **azureml-automl-runtime**
    - AutoML will now take into account both true and false classes when calculating averaged scalar metrics for binary classification tasks.
    - Moved Machine learning and training code in AzureML-AutoML-Core to a new package AzureML-AutoML-Runtime.
  - **azureml-contrib-dataset**
    - When calling `to_pandas_dataframe` on a labeled dataset with the download option, you can now specify whether to overwrite existing files or not.
    - When calling `keep_columns` or `drop_columns` that results in a timeseries, label, or image column being dropped, the corresponding capabilities will be dropped for the dataset as well.
    - Fixed an issue with pytorch loader for the object detection task.
  - **azureml-contrib-interpret**
    - Removed explanation dashboard widget from azureml-contrib-interpret, changed package to reference the new one in interpret_community
    - Updated version of interpret-community to 0.2.0
  - **azureml-core**
    - Improve performance of `workspace.datasets`.
    - Added the ability to register Azure SQL Database Datastore using username and password authentication
    - Fix for loading RunConfigurations from relative paths.
    - When calling `keep_columns` or `drop_columns` that results in a timeseries column being dropped, the corresponding capabilities will be dropped for the dataset as well.
  - **azureml-interpret**
    - updated version of interpret-community to 0.2.0
  - **azureml-pipeline-steps**
    - Documented supported values for `runconfig_pipeline_params` for azure machine learning pipeline

steps.

- **azureml-pipeline-core**
  - Added CLI option to download output in json format for Pipeline commands.
- **azureml-train-automl**
  - Split AzureML-Train-AutoML into 2 packages, an client package AzureML-Train-AutoML-Client and a ML training package AzureML-Train-AutoML-Runtime
- **azureml-train-automl-client**
  - Added a thin client for submitting AutoML experiments without needing to install any machine learning dependencies locally.
  - Fixed logging of automatically detected lags, rolling window sizes and maximal horizons in the remote runs.
- **azureml-train-automl-runtime**
  - Added a new AutoML package to isolate machine learning and runtime components from the client.
- **azureml-contrib-train-rl**
  - Added reinforcement learning support in SDK.
  - Added AmlWindowsCompute support in RL SDK.

2019-11-11

Azure Machine Learning SDK for Python v1.0.74

- **Preview features**
  - **azureml-contrib-dataset**
    - After importing azureml-contrib-dataset, you can call `Dataset.Labeled.from_json_lines` instead of `._Labeled` to create a labeled dataset.
    - When calling `to_pandas_dataframe` on a labeled dataset with the download option, you can now specify whether to overwrite existing files or not.
    - When calling `keep_columns` or `drop_columns` that results in a time series, label, or image column being dropped, the corresponding capabilities will be dropped for the dataset as well.
    - Fixed issues with PyTorch loader when calling `dataset.to_torchvision()`.
- **Bug fixes and improvements**
  - **azure-cli-ml**
    - Added Model Profiling to the preview CLI.
    - Fixes breaking change in Azure Storage causing AzureML CLI to fail.
    - Added Load Balancer Type to MLC for AKS types
  - **azureml-automl-core**
    - Fixed the issue with detection of maximal horizon on time series, having missing values and multiple grains.
    - Fixed the issue with failures during generation of cross validation splits.
    - Replace this section with a message in markdown format to appear in the release notes: -Improved handling of short grains in the forecasting data sets.
    - Fixed the issue with masking of some user information during logging. -Improved logging of the errors during forecasting runs.
    - Adding psutil as a conda dependency to the auto-generated yml deployment file.
  - **azureml-contrib-mir**
    - Fixes breaking change in Azure Storage causing AzureML CLI to fail.
  - **azureml-core**

- Fixes a bug which caused models deployed on Azure Functions to produce 500s.
- Fixed an issue where the amlignore file was not applied on snapshots.
- Added a new API amlcompute.get_active_runs that returns a generator for running and queued runs on a given amlcompute.
- Added Load Balancer Type to MLC for AKS types.
- Added append_prefix bool parameter to download_files in run.py and download_artifacts_from_prefix in artifacts_client. This flag is used to selectively flatten the origin filepath so only the file or folder name is added to the output_directory
- Fix deserialization issue for `run_config.yml` with dataset usage.
- When calling `keep_columns` or `drop_columns` that results in a time series column being dropped, the corresponding capabilities will be dropped for the dataset as well.

- **azureml-interpret**
  - Updated interpret-community version to 0.1.0.3
- **azureml-train-automl**
  - Fixed an issue where automl_step might not print validation issues.
  - Fixed register_model to succeed even if the model's environment is missing dependencies locally.
  - Fixed an issue where some remote runs were not docker enabled.
  - Add logging of the exception that is causing a local run to fail prematurely.
- **azureml-train-core**
  - Consider resume_from runs in the calculation of automated hyperparameter tuning best child runs.
- **azureml-pipeline-core**
  - Fixed parameter handling in pipeline argument construction.
  - Added pipeline description and step type yaml parameter.
  - New yaml format for Pipeline step and added deprecation warning for old format.

2019-11-04

Web experience

The collaborative workspace landing page at https://ml.azure.com has been enhanced and rebranded as the Azure Machine Learning studio (preview).

From the studio, you can train, test, deploy, and manage Azure Machine Learning assets such as datasets, pipelines, models, endpoints, and more.

Access the following web-based authoring tools from the studio:

WEB-BASED TOOL	DESCRIPTION	EDITION
Notebook VM(preview)	Fully managed cloud-based workstation	Basic & Enterprise
Automated machine learning (preview)	No code experience for automating machine learning model development	Enterprise
Designer (preview)	Drag-and-drop machine learning modeling tool formerly known as the the designer	Enterprise

Azure Machine Learning designer enhancements

- Formerly known as the visual interface
- 11 new modules including recommenders, classifiers, and training utilities including feature engineering, cross

validation, and data transformation.

R SDK

Data scientists and AI developers use the Azure Machine Learning SDK for R to build and run machine learning workflows with Azure Machine Learning.

The Azure Machine Learning SDK for R uses the `reticulate` package to bind to the Python SDK. By binding directly to Python, the SDK for R allows you access to core objects and methods implemented in the Python SDK from any R environment you choose.

Main capabilities of the SDK include:

- Manage cloud resources for monitoring, logging, and organizing your machine learning experiments.
- Train models using cloud resources, including GPU-accelerated model training.
- Deploy your models as webservices on Azure Container Instances (ACI) and Azure Kubernetes Service (AKS).

See the package website for complete documentation.

Azure Machine Learning integration with Event Grid

Azure Machine Learning is now a resource provider for Event Grid, you can configure machine learning events through the Azure portal or Azure CLI. Users can create events for run completion, model registration, model deployment and data drift detected. These events can be routed to event handlers supported by Event Grid for consumption. See machine learning event schema, concepts and tutorial articles for more details.

2019-10-31

Azure Machine Learning SDK for Python v1.0.72

- New features
  - Added dataset monitors through the **azureml-datadrift** package, allowing for monitoring time series datasets for data drift or other statistical changes over time. Alerts and events can be triggered if drift is detected or other conditions on the data are met. See our documentation for details.
  - Announcing two new editions (also referred to as a SKU interchangeably) in Azure Machine Learning. With this release you can now create either a Basic or Enterprise Azure Machine Learning workspace. All existing workspaces will be defaulted to the Basic edition, and you can go to the Azure portal or to the studio to upgrade the workspace anytime. You can create either a Basic or Enterprise workspace from the Azure portal. Please read our documentation to learn more. From the SDK, the edition of your workspace can be determined using the "sku" property of your workspace object.
  - We have also made enhancements to Azure Machine Learning Compute - you can now view metrics for your clusters (like total nodes, running nodes, total core quota) in Azure Monitor, besides viewing Diagnostic logs for debugging. In addition you can also view currently running or queued runs on your cluster and details such as the IPs of the various nodes on your cluster. You can view these either in the portal or by using corresponding functions in the SDK or CLI.
  - Preview features
    - We are releasing preview support for disk encryption of your local SSD in Azure Machine Learning Compute. Please raise a technical support ticket to get your subscription whitelisted to use this feature.
    - Public Preview of Azure Machine Learning Batch Inference. Azure Machine Learning Batch Inference targets large inference jobs that are not time-sensitive. Batch Inference provides cost-effective inference compute scaling, with unparalleled throughput for asynchronous applications. It is optimized for high-throughput, fire-and-forget inference over large collections of data.
    - **azureml-contrib-dataset**

- Enabled functionalities for labeled dataset

```
import azureml.core
from azureml.core import Workspace, Datastore, Dataset
import azureml.contrib.dataset
from azureml.contrib.dataset import FileHandlingOption, LabeledDatasetTask

# create a labeled dataset by passing in your JSON lines file
dataset = Dataset._Labeled.from_json_lines(datastore.path('path/to/file.jsonl'),
LabeledDatasetTask.IMAGE_CLASSIFICATION)

# download or mount the files in the `image_url` column
dataset.download()
dataset.mount()

# get a pandas dataframe
from azureml.data.dataset_type_definitions import FileHandlingOption
dataset.to_pandas_dataframe(FileHandlingOption.DOWNLOAD)
dataset.to_pandas_dataframe(FileHandlingOption.MOUNT)

# get a Torchvision dataset
dataset.to_torchvision()
```

- **Bug fixes and improvements**
  - **azure-cli-ml**
    - CLI now supports model packaging.
    - Added dataset CLI. For more information: `az ml dataset --help`
    - Added support for deploying and packaging supported models (ONNX, scikit-learn, and TensorFlow) without an InferenceConfig instance.
    - Added overwrite flag for service deployment (ACI and AKS) in SDK and CLI. If provided, will overwrite the existing service if service with name already exists. If service doesn't exist, will create new service.
    - Models can be registered with two new frameworks, Onnx and Tensorflow. - Model registration accepts sample input data, sample output data and resource configuration for the model.
  - **azureml-automl-core**
    - Training an iteration would run in a child process only when runtime constraints are being set.
    - Added a guardrail for forecasting tasks, to check whether a specified max_horizon will cause a memory issue on the given machine or not. If it will, a guardrail message will be displayed.
    - Added support for complex frequencies like 2 years and 1 month. -Added comprehensible error message if frequency can not be determined.
    - Add azureml-defaults to auto generated conda env to solve the model deployment failure
    - Allow intermediate data in Azure Machine Learning Pipeline to be converted to tabular dataset and used in `AutoMLStep`.
    - Implemented column purpose update for streaming.
    - Implemented transformer parameter update for Imputer and HashOneHotEncoder for streaming.
    - Added the current data size and the minimum required data size to the validation error messages.
    - Updated the minimum required data size for Cross-validation to guarantee a minimum of two samples in each validation fold.
  - **azureml-cli-common**
    - CLI now supports model packaging.
    - Models can be registered with two new frameworks, Onnx and Tensorflow.
    - Model registration accepts sample input data, sample output data and resource configuration for the model.

- **azureml-contrib-gbdt**
  - fixed the release channel for the notebook
  - Added a warning for non AmlCompute compute target that we don't support
  - Added LightGMB Estimator to azureml-contrib-gbdt package
- **azureml-core**
  - CLI now supports model packaging.
  - Add deprecation warning for deprecated Dataset APIs. See Dataset API change notice at https://aka.ms/tabular-dataset.
  - Change `Dataset.get_by_id` to return registration name and version if the dataset is registered.
  - Fix a bug that ScriptRunConfig with dataset as argument cannot be used repeatedly to submit experiment run.
  - Datasets retrieved during a run will be tracked and can be seen in the run details page or by calling `run.get_details()` after the run is complete.
  - Allow intermediate data in Azure Machine Learning Pipeline to be converted to tabular dataset and used in `AutoMLStep`.
  - Added support for deploying and packaging supported models (ONNX, scikit-learn, and TensorFlow) without an InferenceConfig instance.
  - Added overwrite flag for service deployment (ACI and AKS) in SDK and CLI. If provided, will overwrite the existing service if service with name already exists. If service doesn't exist, will create new service.
  - Models can be registered with two new frameworks, Onnx and Tensorflow. Model registration accepts sample input data, sample output data and resource configuration for the model.
  - Added new datastore for Azure Database for MySQL. Added example for using Azure Database for MySQL in DataTransferStep in Azure Machine Learning Pipelines.
  - Added functionality to add and remove tags from experiments Added functionality to remove tags from runs
  - Added overwrite flag for service deployment (ACI and AKS) in SDK and CLI. If provided, will overwrite the existing service if service with name already exists. If service doesn't exist, will create new service.
- **azureml-datadrift**
  - Moved from `azureml-contrib-datadrift` into `azureml-datadrift`
  - Added support for monitoring time series datasets for drift and other statistical measures
  - New methods `create_from_model()` and `create_from_dataset()` to the `DataDriftDetector` class. The `create()` method will be deprecated.
  - Adjustments to the visualizations in Python and UI in the Azure Machine Learning studio.
  - Support weekly and monthly monitor scheduling, in addition to daily for dataset monitors.
  - Support backfill of data monitor metrics to analyze historical data for dataset monitors.
  - Various bug fixes
- **azureml-pipeline-core**
  - azureml-dataprep is no longer needed to submit an Azure Machine Learning Pipeline run from the pipeline `yaml` file.
- **azureml-train-automl**
  - Add azureml-defaults to auto generated conda env to solve the model deployment failure
  - AutoML remote training now includes azureml-defaults to allow reuse of training env for inference.
- **azureml-train-core**
  - Added PyTorch 1.3 support in `PyTorch` estimator

2019-10-21

Visual interface (preview)

- The Azure Machine Learning visual interface (preview) has been overhauled to run on Azure Machine Learning pipelines. Pipelines (previously known as experiments) authored in the visual interface are now fully integrated with the core Azure Machine Learning experience.
  - Unified management experience with SDK assets
  - Versioning and tracking for visual interface models, pipelines, and endpoints
  - Redesigned UI
  - Added batch inference deployment
  - Added Azure Kubernetes Service (AKS) support for inference compute targets
  - New Python-step pipeline authoring workflow
  - New landing page for visual authoring tools
- **New modules**
  - Apply math operation
  - Apply SQL transformation
  - Clip values
  - Summarize data
  - Import from SQL database

2019-10-14

Azure Machine Learning SDK for Python v1.0.69

- **Bug fixes and improvements**
  - **azureml-automl-core**
    - Limiting model explanations to best run rather than computing explanations for every run. Making this behavior change for local, remote and ADB.
    - Added support for on-demand model explanations for UI
    - Added psutil as a dependency of `automl` and included psutil as a conda dependency in amlcompute.
    - Fixed the issue with heuristic lags and rolling window sizes on the forecasting data sets some series of which can cause linear algebra errors
      - Added print out for the heuristically determined parameters in the forecasting runs.
  - **azureml-contrib-datadrift**
    - Added protection while creating output metrics if dataset level drift is not in the first section.
  - **azureml-contrib-interpret**
    - azureml-contrib-explain-model package has been renamed to azureml-contrib-interpret
  - **azureml-core**
    - Added API to unregister datasets. `dataset.unregister_all_versions()`
    - azureml-contrib-explain-model package has been renamed to azureml-contrib-interpret.
  - **azureml-core**
    - Added API to unregister datasets. dataset.unregister_all_versions().
    - Added Dataset API to check data changed time. `dataset.data_changed_time`.
    - Being able to consume `FileDataset` and `TabularDataset` as inputs to `PythonScriptStep`, `EstimatorStep`, and `HyperDriveStep` in Azure Machine Learning Pipeline
    - Performance of `FileDataset.mount` has been improved for folders with a large number of files
    - Being able to consume FileDataset and TabularDataset as inputs to PythonScriptStep,

EstimatorStep, and HyperDriveStep in the Azure Machine Learning Pipeline.
    - Performance of FileDataset.mount() has been improved for folders with a large number of files
    - Added URL to known error recommendations in run details.
    - Fixed a bug in run.get_metrics where requests would fail if a run had too many children
    - Fixed a bug in run.get_metrics where requests would fail if a run had too many children
    - Added support for authentication on Arcadia cluster.
    - Creating an Experiment object gets or creates the experiment in the Azure Machine Learning workspace for run history tracking. The experiment ID and archived time are populated in the Experiment object on creation. Example: experiment = Experiment(workspace, "New Experiment") experiment_id = experiment.id archive() and reactivate() are functions that can be called on an experiment to hide and restore the experiment from being shown in the UX or returned by default in a call to list experiments. If a new experiment is created with the same name as an archived experiment, you can rename the archived experiment when reactivating by passing a new name. There can only be one active experiment with a given name. Example: experiment1 = Experiment(workspace, "Active Experiment") experiment1.archive() # Create new active experiment with the same name as the archived. experiment2. = Experiment(workspace, "Active Experiment") experiment1.reactivate(new_name="Previous Active Experiment") The static method list() on Experiment can take a name filter and ViewType filter. ViewType values are "ACTIVE_ONLY", "ARCHIVED_ONLY" and "ALL" Example: archived_experiments = Experiment.list(workspace, view_type="ARCHIVED_ONLY") all_first_experiments = Experiment.list(workspace, name="First Experiment", view_type="ALL")
    - Support using environment for model deploy, and service update
- **azureml-datadrift**
    - The show attribute of DataDriftDector class won't support optional argument 'with_details' any more. The show attribute will only present data drift coefficient and data drift contribution of feature columns.
    - DataDriftDetector attribute 'get_output' behavior changes:
        - Input parameter start_time, end_time are optional instead of mandatory;
        - Input specific start_time and/or end_time with a specific run_id in the same invoking will result in value error exception because they are mutually exclusive
        - By input specific start_time and/or end_time, only results of scheduled runs will be returned;
        - Parameter 'daily_latest_only' is deprecated.
    - Support retrieving Dataset-based Data Drift outputs.
- **azureml-explain-model**
    - Renames AzureML-explain-model package to AzureML-interpret, keeping the old package for backwards compatibility for now
    - fixed `automl` bug with raw explanations set to classification task instead of regression by default on download from ExplanationClient
    - Add support for `ScoringExplainer` to be created directly using `MimicWrapper`
- **azureml-pipeline-core**
    - Improved performance for large Pipeline creation
- **azureml-train-core**
    - Added TensorFlow 2.0 support in TensorFlow Estimator
- **azureml-train-automl**
    - Creating an Experiment object gets or creates the experiment in the Azure Machine Learning workspace for run history tracking. The experiment ID and archived time are populated in the Experiment object on creation. Example:

```
experiment = Experiment(workspace, "New Experiment")
experiment_id = experiment.id
```

archive() and reactivate() are functions that can be called on an experiment to hide and restore the experiment from being shown in the UX or returned by default in a call to list experiments. If a new experiment is created with the same name as an archived experiment, you can rename the archived experiment when reactivating by passing a new name. There can only be one active experiment with a given name. Example:

```
experiment1 = Experiment(workspace, "Active Experiment")
experiment1.archive()
# Create new active experiment with the same name as the archived.
experiment2 = Experiment(workspace, "Active Experiment")
experiment1.reactivate(new_name="Previous Active Experiment")
```

The static method list() on Experiment can take a name filter and ViewType filter. ViewType values are "ACTIVE_ONLY", "ARCHIVED_ONLY" and "ALL". Example:

```
archived_experiments = Experiment.list(workspace, view_type="ARCHIVED_ONLY")
all_first_experiments = Experiment.list(workspace, name="First Experiment",
view_type="ALL")
```

  - Support using environment for model deploy, and service update.

- azureml-datadrift
  - The show attribute of DataDriftDetector class won't support optional argument 'with_details' any more. The show attribute will only present data drift coefficient and data drift contribution of feature columns.
  - DataDriftDetector function [get_output]https://docs.microsoft.com/python/api/azureml-datadrift/azureml.datadrift.datadriftdetector.datadriftdetector#get-output-start-time-none--end-time-none--run-id-none-) behavior changes:
    - Input parameter start_time, end_time are optional instead of mandatory;
    - Input specific start_time and/or end_time with a specific run_id in the same invoking will result in value error exception because they are mutually exclusive;
    - By input specific start_time and/or end_time, only results of scheduled runs will be returned;
    - Parameter 'daily_latest_only' is deprecated.
  - Support retrieving Dataset-based Data Drift outputs.
- azureml-explain-model
  - Renames AzureML-explain-model package to AzureML-interpret, keeping the old package for backwards compatibility for now.
  - fixed AutoML bug with raw explanations set to classification task instead of regression by default on download from ExplanationClient.
  - Add support for ScoringExplainer to be created directly using MimicWrapper
- azureml-pipeline-core
  - Improved performance for large Pipeline creation.
- azureml-train-core
  - Added TensorFlow 2.0 support in TensorFlow Estimator.
- azureml-train-automl
  - The parent run will no longer be failed when setup iteration failed, as the orchestration already

takes care of it.
    - Added local-docker and local-conda support for AutoML experiments
    - Added local-docker and local-conda support for AutoML experiments.

2019-10-08

New web experience (preview) for Azure Machine Learning workspaces

The Experiment tab in the new workspace portal has been been updated so data scientists can monitor experiments in a more performant way. You can explore the following features:

- Experiment metadata to easily filter and sort your list of experiments
- Simplified and performant experiment details pages which allow you to visualize and compare your runs
- New design to run details pages to understand and monitor your training runs

2019-09-30

Azure Machine Learning SDK for Python v1.0.65

- New features
    - Added curated environments. These environments have been pre-configured with libraries for common machine learning tasks, and have been pre-build and cached as Docker images for faster execution. They appear by default in Workspace's list of environment, with prefix "AzureML".
    - Added curated environments. These environments have been pre-configured with libraries for common machine learning tasks, and have been pre-build and cached as Docker images for faster execution. They appear by default in Workspace's list of environment, with prefix "AzureML".
- azureml-train-automl
- azureml-train-automl
    - Added the ONNX conversion support for the ADB and HDI
- Preview features
    - azureml-train-automl
    - azureml-train-automl
        - Supported BERT and BiLSTM as text featurizer (preview only)
        - Supported featurization customization for column purpose and transformer parameters (preview only)
        - Supported raw explanations when user enables model explanation during training (preview only)
        - Added Prophet for `timeseries` forecasting as a trainable pipeline (preview only)
    - azureml-contrib-datadrift
        - Packages relocated from azureml-contrib-datadrift to azureml-datadrift; the `contrib` package will be removed in a future release
- Bug fixes and improvements
    - azureml-automl-core
        - Introduced FeaturizationConfig to AutoMLConfig and AutoMLBaseSettings
        - Introduced FeaturizationConfig to AutoMLConfig and AutoMLBaseSettings
            - Override Column Purpose for Featurization with given column and feature type
            - Override transformer parameters
        - Added deprecation message for explain_model() and retrieve_model_explanations()

- Added Prophet as a trainable pipeline (preview only)
- Added deprecation message for explain_model() and retrieve_model_explanations().
- Added Prophet as a trainable pipeline (preview only).
- Added support for automatic detection of target lags, rolling window size and maximal horizon. If one of target_lags, target_rolling_window_size or max_horizon is set to 'auto', the heuristics will be applied to estimate the value of corresponding parameter based on training data.
- Fixed forecasting in the case when data set contains one grain column, this grain is of a numeric type and there is a gap between train and test set
- Fixed the error message about the duplicated index in the remote run in forecasting tasks
- Fixed forecasting in the case when data set contains one grain column, this grain is of a numeric type and there is a gap between train and test set.
- Fixed the error message about the duplicated index in the remote run in forecasting tasks.
- Added a guardrail to check whether a dataset is imbalanced or not. If it is, a guardrail message would be written to the console.

- **azureml-core**
  - Added ability to retrieve SAS URL to model in storage through the model object. Ex: model.get_sas_url()
  - Introduce `run.get_details()['datasets']` to get datasets associated with the submitted run
  - Add API `Dataset.Tabular.from_json_lines_files` to create a TabularDataset from JSON Lines files. To learn about this tabular data in JSON Lines files on TabularDataset, please visit https://aka.ms/azureml-data for documentation.
  - Added additional VM size fields (OS Disk, number of GPUs) to the supported_vmsizes () function
  - Added additional fields to the list_nodes () function to show the run, the private and the public IP, the port etc.
  - Ability to specify a new field during cluster provisioning --remotelogin_port_public_access which can be set to enabled or disabled depending on whether you would like to leave the SSH port open or closed at the time of creating the cluster. If you do not specify it, the service will smartly open or close the port depending on whether you are deploying the cluster inside a VNet.
- **azureml-explain-model**
- **azureml-core**
  - Added ability to retrieve SAS URL to model in storage through the model object. Ex: model.get_sas_url()
  - Introduce run.get_details['datasets'] to get datasets associated with the submitted run
  - Add API `Dataset.Tabular`.from_json_lines_files() to create a TabularDataset from JSON Lines files. To learn about this tabular data in JSON Lines files on TabularDataset, please visit https://aka.ms/azureml-data for documentation.
  - Added additional VM size fields (OS Disk, number of GPUs) to the supported_vmsizes() function
  - Added additional fields to the list_nodes() function to show the run, the private and the public IP, the port etc.
  - Ability to specify a new field during cluster provisioning `--remotelogin_port_public_access` which can be set to enabled or disabled depending on whether you would like to leave the SSH port open or closed at the time of creating the cluster. If you do not specify it, the service will smartly open or close the port depending on whether you are deploying the cluster inside a VNet.
- **azureml-explain-model**
  - Improved documentation for Explanation outputs in the classification scenario.
  - Added the ability to upload the predicted y values on the explanation for the evaluation examples. Unlocks more useful visualizations.
  - Added explainer property to MimicWrapper to enable getting the underlying MimicExplainer.

- azureml-pipeline-core
  - Added notebook to describe Module, ModuleVersion and ModuleStep
- azureml-pipeline-steps
  - Added RScriptStep to support R script run via AML pipeline.
  - Fixed metadata parameters parsing in AzureBatchStep which was causing the error message "assignment for parameter SubscriptionId is not specified."
- azureml-train-automl
  - Supported training_data, validation_data, label_column_name, weight_column_name as data input format
  - Added deprecation message for explain_model() and retrieve_model_explanations()
- azureml-pipeline-core
  - Added a notebook to describe Module, ModuleVersion and ModuleStep.
- azureml-pipeline-steps
  - Added RScriptStep to support R script run via AML pipeline.
  - Fixed metadata parameters parsing in AzureBatchStep which was causing the error message "assignment for parameter SubscriptionId is not specified".
- azureml-train-automl
  - Supported training_data, validation_data, label_column_name, weight_column_name as data input format.
  - Added deprecation message for explain_model() and retrieve_model_explanations().

2019-09-16

Azure Machine Learning SDK for Python v1.0.62

- New features
  - Introduced the `timeseries` trait on TabularDataset. This trait enables easy timestamp filtering on data a TabularDataset, such as taking all data between a range of time or the most recent data. To learn about this the `timeseries` trait on TabularDataset, please visit https://aka.ms/azureml-data for documentation or https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/work-with-data/datasets-tutorial/timeseries-datasets/tabular-timeseries-dataset-filtering.ipynb for an example notebook.
  - Enabled training with TabularDataset and FileDataset. Please visit https://aka.ms/dataset-tutorial for an example notebook.
  - azureml-train-core
    - Added `Nccl` and `Gloo` support in PyTorch estimator
- Bug fixes and improvements
  - azureml-automl-core
    - Deprecated the AutoML setting 'lag_length' and the LaggingTransformer.
    - Fixed correct validation of input data if they are specified in a Dataflow format
    - Modified the fit_pipeline.py to generate the graph json and upload to artifacts.
    - Rendered the graph under `userrun` using `Cytoscape`.
  - azureml-core
    - Revisited the exception handling in ADB code and make changes to as per new error handling
    - Added automatic MSI authentication for Notebook VMs.
    - Fixes bug where corrupt or empty models could be uploaded because of failed retries.
    - Fixed the bug where `DataReference` name changes when the `DataReference` mode changes (e.g.

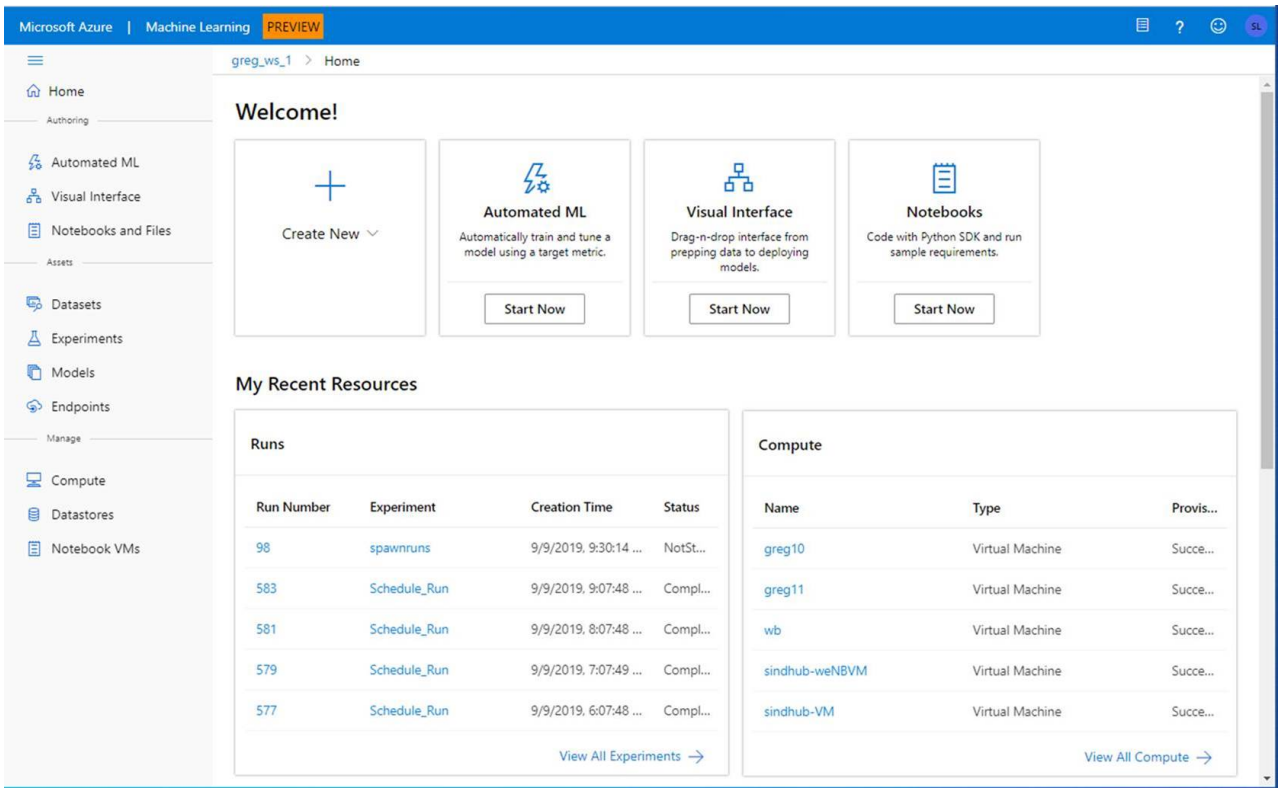
when calling `as_upload`, `as_download`, or `as_mount`).
    - Make `mount_point` and `target_path` optional for `FileDataset.mount` and `FileDataset.download`.
    - Exception that timestamp column cannot be found will be throw out if the time serials related API is called without fine timestamp column assigned or the assigned timestamp columns are dropped.
    - Time serials columns should be assigned with column whose type is Date, otherwise exception is expected
    - Time serials columns assigning API 'with_timestamp_columns' can take None value fine/coarse timestamp column name, which will clear previously assigned timestamp columns.
    - Exception will be thrown out when either coarse grain or fine grained timestamp column is dropped with indication for user that dropping can be done after either excluding timestamp column in dropping list or call with_time_stamp with None value to release timestamp columns
    - Exception will be thrown out when either coarse grain or fine grained timestamp column is not included in keep columns list with indication for user that keeping can be done after either including timestamp column in keep column list or call with_time_stamp with None value to release timestamp columns.
    - Added logging for the size of a registered model.
- **azureml-explain-model**
    - Fixed warning printed to console when "packaging" python package is not installed: "Using older than supported version of lightgbm, please upgrade to version greater than 2.2.1"
    - Fixed download model explanation with sharding for global explanations with many features
    - Fixed mimic explainer missing initialization examples on output explanation
    - Fixed immutable error on set properties when uploading with explanation client using two different types of models
    - Added a get_raw param to scoring explainer .explain() so one scoring explainer can return both engineered and raw values.
- **azureml-train-automl**
    - Introduced public APIs from AutoML for supporting explanations from `automl` explain SDK - Newer way of supporting AutoML explanations by decoupling AutoML featurization and explain SDK - Integrated raw explanation support from azureml explain SDK for AutoML models.
    - Removing azureml-defaults from remote training environments.
    - Changed default cache store location from FileCacheStore based one to AzureFileCacheStore one for AutoML on Azure Databricks code path.
    - Fixed correct validation of input data if they are specified in a Dataflow format
- **azureml-train-core**
    - Reverted source_directory_data_store deprecation.
    - Added ability to override azureml installed package versions.
    - Added dockerfile support in `environment_definition` parameter in estimators.
    - Simplified distributed training parameters in estimators.

```
from azureml.train.dnn import TensorFlow, Mpi, ParameterServer
```

2019-09-09

New web experience (preview) for Azure Machine Learning workspaces

The new web experience enables data scientists and data engineers to complete their end-to-end machine learning lifecycle from prepping and visualizing data to training and deploying models in a single location.



Key features:

Using this new Azure Machine Learning interface, you can now:

- Manage your notebooks or link out to Jupyter
- Run automated ML experiments
- Create datasets from local files, datastores, & web files
- Explore & prepare datasets for model creation
- Monitor data drift for your models
- View recent resources from a dashboard

At the time of this release, the following browsers are supported: Chrome, Firefox, Safari, and Microsoft Edge Preview.

Known issues:

1. Refresh your browser if you see "Something went wrong! Error loading chunk files" when deployment is in progress.
2. Can't delete or rename file in Notebooks and Files. During Public Preview you can use Jupyter UI or Terminal in Notebook VM to perform update file operations. Because it is a mounted network file system all changes you make on Notebook VM are immediately reflected in the Notebook Workspace.
3. To SSH into the Notebook VM:
   a. Find the SSH keys that were created during VM setup. Or, find the keys in the Azure Machine Learning workspace > open Compute tab > locate Notebook VM in the list > open it's properties : copy the keys from the dialog.
   b. Import those public and private SSH keys to your local machine.
   c. Use them to SSH into the Notebook VM.

2019-09-03

Azure Machine Learning SDK for Python v1.0.60

- **New features**
  - Introduced FileDataset, which references single or multiple files in your datastores or public urls. The files can be of any format. FileDataset provides you with the ability to download or mount the files to your compute. To learn about FileDataset, please visit https://aka.ms/file-dataset.
  - Added Pipeline Yaml Support for PythonScript Step, Adla Step, Databricks Step, DataTransferStep, and AzureBatch Step
- **Bug fixes and improvements**
  - **azureml-automl-core**
    - AutoArima is now a suggestable pipeline for preview only.
    - Improved error reporting for forecasting.
    - Improved the logging by using custom exceptions instead of generic in the forecasting tasks.
    - Removed the check on max_concurrent_iterations to be less than total number of iterations.
    - AutoML models now return AutoMLExceptions
    - This release improves the execution performance of automated machine learning local runs.
  - **azureml-core**
    - Introduce Dataset.get_all(workspace), which returns a dictionary of `TabularDataset` and `FileDataset` objects keyed by their registration name.

```
workspace = Workspace.from_config()
all_datasets = Dataset.get_all(workspace)
mydata = all_datasets['my-data']
```

    - Introduce `parition_format` as argument to `Dataset.Tabular.from_delimited_files` and `Dataset.Tabular.from_parquet.files`. The partition information of each data path will be extracted into columns based on the specified format. '{column_name}' creates string column, and '{column_name:yyyy/MM/dd/HH/mm/ss}' creates datetime column, where 'yyyy', 'MM', 'dd', 'HH', 'mm' and 'ss' are used to extract year, month, day, hour, minute, and second for the datetime type. The partition_format should start from the position of first partition key until the end of file path. For example, given the path '../USA/2019/01/01/data.csv' where the partition is by country and time, partition_format='/{Country}/{PartitionDate:yyyy/MM/dd}/data.csv' creates string column 'Country' with value 'USA' and datetime column 'PartitionDate' with value '2019-01-01'.

```
workspace = Workspace.from_config()
all_datasets = Dataset.get_all(workspace)
mydata = all_datasets['my-data']
```

    - Introduce `partition_format` as argument to `Dataset.Tabular.from_delimited_files` and `Dataset.Tabular.from_parquet.files`. The partition information of each data path will be extracted into columns based on the specified format. '{column_name}' creates string column, and '{column_name:yyyy/MM/dd/HH/mm/ss}' creates datetime column, where 'yyyy', 'MM', 'dd', 'HH', 'mm' and 'ss' are used to extract year, month, day, hour, minute, and second for the datetime type. The partition_format should start from the position of first partition key until the end of file path. For example, given the path '../USA/2019/01/01/data.csv' where the partition is by country and time, partition_format='/{Country}/{PartitionDate:yyyy/MM/dd}/data.csv' creates string column 'Country' with value 'USA' and datetime column 'PartitionDate' with value '2019-01-01'.
    - `to_csv_files` and `to_parquet_files` methods have been added to `TabularDataset`. These

methods enable conversion between a `TabularDataset` and a `FileDataset` by converting the data to files of the specified format.

- Automatically log into the base image registry when saving a Dockerfile generated by Model.package().
- 'gpu_support' is no longer necessary; AML now automatically detects and uses the nvidia docker extension when it is available. It will be removed in a future release.
- Added support to create, update, and use PipelineDrafts.
- This release improves the execution performance of automated machine learning local runs.
- Users can query metrics from run history by name.
- Improved the logging by using custom exceptions instead of generic in the forecasting tasks.

- **azureml-explain-model**
  - Added feature_maps parameter to the new MimicWrapper, allowing users to get raw feature explanations.
  - Dataset uploads are now off by default for explanation upload, and can be re-enabled with upload_datasets=True
  - Added "is_law" filtering parameters to explanation list and download functions.
  - Adds method `get_raw_explanation(feature_maps)` to both global and local explanation objects.
  - Added version check to lightgbm with printed warning if below supported version
  - Optimized memory usage when batching explanations
  - AutoML models now return AutoMLExceptions
- **azureml-pipeline-core**
  - Added support to create, update, and use PipelineDrafts - can be used to maintain mutable pipeline definitions and use them interactively to run
- **azureml-train-automl**
  - Created feature to install specific versions of gpu-capable pytorch v1.1.0, cuda toolkit 9.0, pytorch-transformers, which is required to enable BERT/ XLNet in the remote python runtime environment.
- **azureml-train-core**
  - Early failure of some hyperparameter space definition errors directly in the sdk instead of server side.

Azure Machine Learning Data Prep SDK v1.1.14

- **Bug fixes and improvements**
  - Enabled writing to ADLS/ADLSGen2 using raw path and credentials.
  - Fixed a bug that caused `include_path=True` to not work for `read_parquet`.
  - Fixed `to_pandas_dataframe()` failure caused by exception "Invalid property value: hostSecret".
  - Fixed a bug where files could not be read on DBFS in Spark mode.

2019-08-19

Azure Machine Learning SDK for Python v1.0.57

- **New features**
  - Enabled `TabularDataset` to be consumed by AutomatedML. To learn more about `TabularDataset`, please visit https://aka.ms/azureml/howto/createdatasets.

- Bug fixes and improvements
  - automl-client-core-nativeclient
    - Fixed the error, raised when training and/or validation labels (y and y_valid) are provided in the form of pandas dataframe but not as numpy array.
    - Updated interface to create a `RawDataContext` to only require the data and the `AutoMLBaseSettings` object.
    - Allow AutoML users to drop training series that are not long enough when forecasting. - Allow AutoML users to drop grains from the test set that does not exist in the training set when forecasting.
  - azure-cli-ml
    - You can now update the TLS/SSL certificate for the scoring endpoint deployed on AKS cluster both for Microsoft generated and customer certificate.
  - azureml-automl-core
    - Fixed an issue in AutoML where rows with missing labels were not removed properly.
    - Improved error logging in AutoML; full error messages will now always be written to the log file.
    - AutoML has updated its package pinning to include `azureml-defaults`, `azureml-explain-model`, and `azureml-dataprep`. AutoML will no longer warn on package mismatches (except for `azureml-train-automl` package).
    - Fixed an issue in `timeseries` where cv splits are of unequal size causing bin calculation to fail.
    - When running ensemble iteration for the Cross-Validation training type, if we ended up having trouble downloading the models trained on the entire dataset, we were having an inconsistency between the model weights and the models that were being fed into the voting ensemble.
    - Fixed the error, raised when training and/or validation labels (y and y_valid) are provided in the form of pandas dataframe but not as numpy array.
    - Fixed the issue with the forecasting tasks when None was encountered in the Boolean columns of input tables.
    - Allow AutoML users to drop training series that are not long enough when forecasting. - Allow AutoML users to drop grains from the test set that does not exist in the training set when forecasting.
  - azureml-core
    - Fixed issue with blob_cache_timeout parameter ordering.
    - Added external fit and transform exception types to system errors.
    - Added support for Key Vault secrets for remote runs. Add a azureml.core.keyvault.Keyvault class to add, get, and list secrets from the keyvault associated with your workspace. Supported operations are:
      - azureml.core.workspace.Workspace.get_default_keyvault()
      - azureml.core.keyvault.Keyvault.set_secret(name, value)
      - azureml.core.keyvault.Keyvault.set_secrets(secrets_dict)
      - azureml.core.keyvault.Keyvault.get_secret(name)
      - azureml.core.keyvault.Keyvault.get_secrets(secrets_list)
      - azureml.core.keyvault.Keyvault.list_secrets()
    - Additional methods to obtain default keyvault and get secrets during remote run:
      - azureml.core.workspace.Workspace.get_default_keyvault()
      - azureml.core.run.Run.get_secret(name)
      - azureml.core.run.Run.get_secrets(secrets_list)
    - Added additional override parameters to submit-hyperdrive CLI command.
    - Improve reliability of API calls be expanding retries to common requests library exceptions.

    - Add support for submitting runs from a submitted run.
    - Fixed expiring SAS token issue in FileWatcher, which caused files to stop being uploaded after their initial token had expired.
    - Supported importing HTTP csv/tsv files in dataset python SDK.
    - Deprecated the Workspace.setup() method. Warning message shown to users suggests using create() or get()/from_config() instead.
    - Added Environment.add_private_pip_wheel(), which enables uploading private custom python packages `whl` to the workspace and securely using them to build/materialize the environment.
    - You can now update the TLS/SSL certificate for the scoring endpoint deployed on AKS cluster both for Microsoft generated and customer certificate.
  - **azureml-explain-model**
    - Added parameter to add a model ID to explanations on upload.
    - Added `is_raw` tagging to explanations in memory and upload.
    - Added pytorch support and tests for azureml-explain-model package.
  - **azureml-opendatasets**
    - Support detecting and logging auto test environment.
    - Added classes to get US population by county and zip.
  - **azureml-pipeline-core**
    - Added label property to input and output port definitions.
  - **azureml-telemetry**
    - Fixed an incorrect telemetry configuration.
  - **azureml-train-automl**
    - Fixed the bug where on setup failure, error was not getting logged in "errors" field for the setup run and hence was not stored in parent run "errors".
    - Fixed an issue in AutoML where rows with missing labels were not removed properly.
    - Allow AutoML users to drop training series that are not long enough when forecasting.
    - Allow AutoML users to drop grains from the test set that do not exist in the training set when forecasting.
    - Now AutoMLStep passes through `automl` config to backend to avoid any issues on changes or additions of new config parameters.
    - AutoML Data Guardrail is now in public preview. User will see a Data Guardrail report (for classification/regression tasks) after training and also be able to access it through SDK API.
  - **azureml-train-core**
    - Added torch 1.2 support in PyTorch Estimator.
  - **azureml-widgets**
    - Improved confusion matrix charts for classification training.

Azure Machine Learning Data Prep SDK v1.1.12

- **New features**
  - Lists of strings can now be passed in as input to `read_*` methods.
- **Bug fixes and improvements**
  - The performance of `read_parquet` has been significantly improved when running in Spark.
  - Fixed an issue where `column_type_builder` failed in case of a single column with ambiguous date formats.

Azure portal

- **Preview Feature**

- Log and output file streaming is now available for run details pages. The files will stream updates in real time when the preview toggle is turned on.
- Ability to set quota at a workspace level is released in preview. AmlCompute quotas are allocated at the subscription level, but we now allow you to distribute that quota between workspaces and allocate it for fair sharing and governance. Just click on the **Usages+Quotas** blade in the left navigation bar of your workspace and select the **Configure Quotas** tab. Note that you must be a subscription admin to be able to set quotas at the workspace level since this is a cross-workspace operation.

2019-08-05

Azure Machine Learning SDK for Python v1.0.55

- New features
  - Token based authentication is now supported for the calls made to the scoring endpoint deployed on AKS. We will continue to support the current key based authentication and users can use one of these authentication mechanisms at a time.
  - Ability to register a blob storage that is behind the virtual network (VNet) as a datastore.
- Bug fixes and improvements
  - azureml-automl-core
    - Fixes a bug where validation size for CV splits is small and results in bad predicted vs. true charts for regression and forecasting.
    - The logging of forecasting tasks on the remote runs improved, now user is provided with comprehensive error message if the run was failed.
    - Fixed failures of `Timeseries` if preprocess flag is True.
    - Made some forecasting data validation error messages more actionable.
    - Reduced memory consumption of AutoML runs by dropping and/or lazy loading of datasets, especially in between process spawns
  - azureml-contrib-explain-model
    - Added model_task flag to explainers to allow user to override default automatic inference logic for model type
    - Widget changes: Automatically installs with `contrib`, no more `nbextension` install/enable - support explanation with just global feature importance (eg Permutative)
    - Dashboard changes: - Box plots and violin plots in addition to `beeswarm` plot on summary page - Much faster rerendering of `beeswarm` plot on 'Top -k' slider change - helpful message explaining how top-k is computed - Useful customizable messages in place of charts when data not provided
  - azureml-core
    - Added Model.package() method to create Docker images and Dockerfiles that encapsulate models and their dependencies.
    - Updated local webservices to accept InferenceConfigs containing Environment objects.
    - Fixed Model.register() producing invalid models when '.' (for the current directory) is passed as the model_path parameter.
    - Add Run.submit_child, the functionality mirrors Experiment.submit while specifying the run as the parent of the submitted child run.
    - Support configuration options from Model.register in Run.register_model.
    - Ability to run JAR jobs on existing cluster.
    - Now supporting instance_pool_id and cluster_log_dbfs_path parameters.
    - Added support for using an Environment object when deploying a Model to a Webservice. The Environment object can now be provided as a part of the InferenceConfig object.
    - Add appinsifht mapping for new regions - centralus - westus - northcentralus

    - Added documentation for all the attributes in all the Datastore classes.
    - Added blob_cache_timeout parameter to `Datastore.register_azure_blob_container`.
    - Added save_to_directory and load_from_directory methods to azureml.core.environment.Environment.
    - Added the "az ml environment download" and "az ml environment register" commands to the CLI.
    - Added Environment.add_private_pip_wheel method.
  - **azureml-explain-model**
    - Added dataset tracking to Explanations using the Dataset service (preview).
    - Decreased default batch size when streaming global explanations from 10k to 100.
    - Added model_task flag to explainers to allow user to override default automatic inference logic for model type.
  - **azureml-mlflow**
    - Fixed bug in mlflow.azureml.build_image where nested directories are ignored.
  - **azureml-pipeline-steps**
    - Added ability to run JAR jobs on existing Azure Databricks cluster.
    - Added support instance_pool_id and cluster_log_dbfs_path parameters for DatabricksStep step.
    - Added support for pipeline parameters in DatabricksStep step.
  - **azureml-train-automl**
    - Added `docstrings` for the Ensemble related files.
    - Updated docs to more appropriate language for `max_cores_per_iteration` and `max_concurrent_iterations`
    - The logging of forecasting tasks on the remote runs improved, now user is provided with comprehensive error message if the run was failed.
    - Removed get_data from pipeline `automlstep` notebook.
    - Started support `dataprep` in `automlstep`.

Azure Machine Learning Data Prep SDK v1.1.10

- New features
  - You can now request to execute specific inspectors (e.g. histogram, scatter plot, etc.) on specific columns.
  - Added a parallelize argument to `append_columns`. If True, data will be loaded into memory but execution will run in parallel; if False, execution will be streaming but single-threaded.

2019-07-23

Azure Machine Learning SDK for Python v1.0.53

- New features
  - Automated Machine Learning now supports training ONNX models on the remote compute target
  - Azure Machine Learning now provides ability to resume training from a previous run, checkpoint or model files.
    - Learn how to use estimators to resume training from a previous run
- Bug fixes and improvements
  - **automl-client-core-nativeclient**
    - Fix the bug about loosing columns types after the transformation (bug linked);
    - Allow y_query to be an object type containing None(s) at the begin (#459519).
  - **azure-cli-ml**
    - CLI commands "model deploy" and "service update" now accept parameters, config files, or a combination of the two. Parameters have precedence over attributes in files.

- Model description can now be updated after registration

- **azureml-automl-core**
  - Update NimbusML dependency to 1.2.0 version (current latest).
  - Adding support for NimbusML estimators & pipelines to be used within AutoML estimators.
  - Fixing a bug in the Ensemble selection procedure which was unnecessarily growing the resulting ensemble even if the scores remained constant.
  - Enable re-use of some featurizations across CV Splits for forecasting tasks. This speeds up the run-time of the setup run by roughly a factor of n_cross_validations for expensive featurizations like lags and rolling windows.
  - Addressing an issue if time is out of pandas supported time range. We now raise a DataException if time is less than pd.Timestamp.min or greater than pd.Timestamp.max
  - Forecasting now allows different frequencies in train and test sets if they can be aligned. For example, "quarterly starting in January" and at "quarterly starting in October" can be aligned.
  - The property "parameters" was added to the TimeSeriesTransformer.
  - Remove old exception classes.
  - In forecasting tasks, the `target_lags` parameter now accepts a single integer value or a list of integers. If the integer was provided, only one lag will be created. If a list is provided, the unique values of lags will be taken. target_lags=[1, 2, 2, 4] will create lags of one, 2 and 4 periods.
  - Fix the bug about losing columns types after the transformation (bug linked);
  - In `model.forecast(X, y_query)`, allow y_query to be an object type containing None(s) at the begin (#459519).
  - Add expected values to `automl` output
- **azureml-contrib-datadrift**
  - Improvements to example notebook including switch to azureml-opendatasets instead of azureml-contrib-opendatasets and performance improvements when enriching data
- **azureml-contrib-explain-model**
  - Fixed transformations argument for LIME explainer for raw feature importance in azureml-contrib-explain-model package
  - Added segmentations to image explanations in image explainer for the AzureML-contrib-explain-model package
  - Add scipy sparse support for LimeExplainer
  - Added `batch_size` to mimic explainer when `include_local=False`, for streaming global explanations in batches to improve execution time of DecisionTreeExplainableModel
- **azureml-contrib-featureengineering**
  - Fix for calling set_featurizer_timeseries_params(): dict value type change and null check - Add notebook for `timeseries` featurizer
  - Update NimbusML dependency to 1.2.0 version (current latest).
- **azureml-core**
  - Added the ability to attach DBFS datastores in the AzureML CLI
  - Fixed the bug with datastore upload where an empty folder is created if `target_path` started with `/`
  - Fixed `deepcopy` issue in ServicePrincipalAuthentication.
  - Added the "az ml environment show" and "az ml environment list" commands to the CLI.
  - Environments now support specifying a base_dockerfile as an alternative to an already-built base_image.
  - The unused RunConfiguration setting auto_prepare_environment has been marked as deprecated.
  - Model description can now be updated after registration
  - Bugfix: Model and Image delete now provides more information about retrieving upstream

objects that depend on them if delete fails due to an upstream dependency.
    - Fixed bug that printed blank duration for deployments that occur when creating a workspace for some environments.
    - Improved workspace create failure exceptions. Such that users don't see "Unable to create workspace. Unable to find..." as the message and instead see the actual creation failure.
    - Add support for token authentication in AKS webservices.
    - Add `get_token()` method to `Webservice` objects.
    - Added CLI support to manage machine learning datasets.
    - `Datastore.register_azure_blob_container` now optionally takes a `blob_cache_timeout` value (in seconds) which configures blobfuse's mount parameters to enable cache expiration for this datastore. The default is no timeout, i.e. when a blob is read, it will stay in the local cache until the job is finished. Most jobs will prefer this setting, but some jobs need to read more data from a large dataset than will fit on their nodes. For these jobs, tuning this parameter will help them succeed. Take care when tuning this parameter: setting the value too low can result in poor performance, as the data used in an epoch may expire before being used again. This means that all reads will be done from blob storage (i.e. the network) rather than the local cache, which negatively impacts training times.
    - Model description can now properly be updated after registration
    - Model and Image deletion now provides more information about upstream objects that depend on them which causes the delete to fail
    - Improve resource utilization of remote runs using azureml.mlflow.
- **azureml-explain-model**
    - Fixed transformations argument for LIME explainer for raw feature importance in azureml-contrib-explain-model package
    - add scipy sparse support for LimeExplainer
    - added shape linear explainer wrapper, as well as another level to tabular explainer for explaining linear models
    - for mimic explainer in explain model library, fixed error when include_local=False for sparse data input
    - add expected values to `automl` output
    - fixed permutation feature importance when transformations argument supplied to get raw feature importance
    - added `batch_size` to mimic explainer when `include_local=False`, for streaming global explanations in batches to improve execution time of DecisionTreeExplainableModel
    - for model explainability library, fixed blackbox explainers where pandas dataframe input is required for prediction
    - Fixed a bug where `explanation.expected_values` would sometimes return a float rather than a list with a float in it.
- **azureml-mlflow**
    - Improve performance of mlflow.set_experiment(experiment_name)
    - Fix bug in use of InteractiveLoginAuthentication for mlflow tracking_uri
    - Improve resource utilization of remote runs using azureml.mlflow.
    - Improve the documentation of the azureml-mlflow package
    - Patch bug where mlflow.log_artifacts("my_dir") would save artifacts under "my_dir/" instead of ""
- **azureml-opendatasets**
    - Pin `pyarrow` of `opendatasets` to old versions (<0.14.0) because of memory issue newly introduced there.
    - Move azureml-contrib-opendatasets to azureml-opendatasets.

    - Allow open dataset classes to be registered to Azure Machine Learning workspace and leverage AML Dataset capabilities seamlessly.
    - Improve NoaaIsdWeather enrich performance in non-SPARK version significantly.
  - **azureml-pipeline-steps**
    - DBFS Datastore is now supported for Inputs and Outputs in DatabricksStep.
    - Updated documentation for Azure Batch Step with regards to inputs/outputs.
    - In AzureBatchStep, changed *delete_batch_job_after_finish* default value to *true*.
  - **azureml-telemetry**
    - Move azureml-contrib-opendatasets to azureml-opendatasets.
    - Allow open dataset classes to be registered to Azure Machine Learning workspace and leverage AML Dataset capabilities seamlessly.
    - Improve NoaaIsdWeather enrich performance in non-SPARK version significantly.
  - **azureml-train-automl**
    - Updated documentation on get_output to reflect the actual return type and provide additional notes on retrieving key properties.
    - Update NimbusML dependency to 1.2.0 version (current latest).
    - add expected values to `automl` output
  - **azureml-train-core**
    - Strings are now accepted as compute target for Automated Hyperparameter Tuning
    - The unused RunConfiguration setting auto_prepare_environment has been marked as deprecated.

Azure Machine Learning Data Prep SDK v1.1.9

- **New features**
  - Added support for reading a file directly from a http or https url.
- **Bug fixes and improvements**
  - Improved error message when attempting to read a Parquet Dataset from a remote source (which is not currently supported).
  - Fixed a bug when writing to Parquet file format in ADLS Gen 2, and updating the ADLS Gen 2 container name in the path.

2019-07-09

Visual Interface

- **Preview features**
  - Added "Execute R script" module in visual interface.

Azure Machine Learning SDK for Python v1.0.48

- **New features**
  - **azureml-opendatasets**
    - **azureml-contrib-opendatasets** is now available as **azureml-opendatasets**. The old package can still work, but we recommend you using **azureml-opendatasets** moving forward for richer capabilities and improvements.
    - This new package allows you to register open datasets as Dataset in Azure Machine Learning workspace, and leverage whatever functionalities that Dataset offers.
    - It also includes existing capabilities such as consuming open datasets as Pandas/SPARK dataframes, and location joins for some dataset like weather.
- **Preview features**

- HyperDriveConfig can now accept pipeline object as a parameter to support hyperparameter tuning using a pipeline.

- **Bug fixes and improvements**
  - **azureml-train-automl**
    - Fixed the bug about losing columns types after the transformation.
    - Fixed the bug to allow y_query to be an object type containing None(s) at the beginning.
    - Fixed the issue in the Ensemble selection procedure which was unnecessarily growing the resulting ensemble even if the scores remained constant.
    - Fixed the issue with whitelist_models and blacklist_models settings in AutoMLStep.
    - Fixed the issue that prevented the usage of preprocessing when AutoML would have been used in the context of Azure ML Pipelines.
  - **azureml-opendatasets**
    - Moved azureml-contrib-opendatasets to azureml-opendatasets.
    - Allowed open dataset classes to be registered to Azure Machine Learning workspace and leverage AML Dataset capabilities seamlessly.
    - Improved NoaaIsdWeather enrich performance in non-SPARK version significantly.
  - **azureml-explain-model**
    - Updated online documentation for interpretability objects.
    - Added `batch_size` to mimic explainer when `include_local=False`, for streaming global explanations in batches to improve execution time of DecisionTreeExplainableModel for model explainability library.
    - Fixed the issue where `explanation.expected_values` would sometimes return a float rather than a list with a float in it.
    - Added expected values to `automl` output for mimic explainer in explain model library.
    - Fixed permutation feature importance when transformations argument supplied to get raw feature importance.
  - **azureml-core**
    - Added the ability to attach DBFS datastores in the AzureML CLI.
    - Fixed the issue with datastore upload where an empty folder is created if `target_path` started with `/`.
    - Enabled comparison of two datasets.
    - Model and Image delete now provides more information about retrieving upstream objects that depend on them if delete fails due to an upstream dependency.
    - Deprecated the unused RunConfiguration setting in auto_prepare_environment.
  - **azureml-mlflow**
    - Improved resource utilization of remote runs that use azureml.mlflow.
    - Improved the documentation of the azureml-mlflow package.
    - Fixed the issue where mlflow.log_artifacts("my_dir") would save artifacts under "my_dir/artifact-paths" instead of "artifact-paths".
  - **azureml-pipeline-core**
    - Parameter hash_paths for all pipeline steps is deprecated and will be removed in future. By default contents of the source_directory is hashed (except files listed in .amlignore or .gitignore)
    - Continued improving Module and ModuleStep to support compute type specific modules, to prepare for RunConfiguration integration and other changes to unlock compute type specific module usage in pipelines.
  - **azureml-pipeline-steps**
    - AzureBatchStep: Improved documentation with regards to inputs/outputs.

- AzureBatchStep: Changed delete_batch_job_after_finish default value to true.
- **azureml-train-core**
  - Strings are now accepted as compute target for Automated Hyperparameter Tuning.
  - Deprecated the unused RunConfiguration setting in auto_prepare_environment.
  - Deprecated parameters `conda_dependencies_file_path` and `pip_requirements_file_path` in favor of `conda_dependencies_file` and `pip_requirements_file` respectively.
- **azureml-opendatasets**
  - Improve NoaaIsdWeather enrich performance in non-SPARK version significantly.

Azure Machine Learning Data Prep SDK v1.1.8

- **New features**
- Dataflow objects can now be iterated over, producing a sequence of records. See documentation for `Dataflow.to_record_iterator`.
- Dataflow objects can now be iterated over, producing a sequence of records. See documentation for `Dataflow.to_record_iterator`.
- **Bug fixes and improvements**
- Increased the robustness of DataPrep SDK.
- Improved handling of pandas DataFrames with non-string Column Indexes.
- Improved the performance of `to_pandas_dataframe` in Datasets.
- Fixed a bug where Spark execution of Datasets failed when run in a multi-node environment.
- Increased the robustness of DataPrep SDK.
- Improved handling of pandas DataFrames with non-string Column Indexes.
- Improved the performance of `to_pandas_dataframe` in Datasets.
- Fixed a bug where Spark execution of Datasets failed when run in a multi-node environment.

2019-07-01

Azure Machine Learning Data Prep SDK v1.1.7

We reverted a change that improved performance, as it was causing issues for some customers using Azure Databricks. If you experienced an issue on Azure Databricks, you can upgrade to version 1.1.7 using one of the methods below:

1. Run this script to upgrade: `%sh /home/ubuntu/databricks/python/bin/pip install azureml-dataprep==1.1.7`
2. Recreate the cluster, which will install the latest Data Prep SDK version.

2019-06-25

Azure Machine Learning SDK for Python v1.0.45

- **New features**
  - Add decision tree surrogate model to mimic explainer in azureml-explain-model package
  - Ability to specify a CUDA version to be installed on Inferencing images. Support for CUDA 9.0, 9.1, and 10.0.
  - Information about Azure ML training base images are now available at Azure ML Containers GitHub Repository and DockerHub

- Added CLI support for pipeline schedule. Run "az ml pipeline -h" to learn more
- Added custom Kubernetes namespace parameter to AKS webservice deployment configuration and CLI.
- Deprecated hash_paths parameter for all pipeline steps
- Model.register now supports registering multiple individual files as a single model with use of the `child_paths` parameter.

- **Preview features**
  - Scoring explainers can now optionally save conda and pip information for more reliable serialization and deserialization.
  - Bug Fix for Auto Feature Selector.
  - Updated mlflow.azureml.build_image to the new api, patched bugs exposed by the new implementation.
- **Bug fixes and improvements**
  - Removed `paramiko` dependency from azureml-core. Added deprecation warnings for legacy compute target attach methods.
  - Improve performance of run.create_children
  - In mimic explainer with binary classifier, fix the order of probabilities when teacher probability is used for scaling shape values.
  - Improved error handling and message for Automated machine learning.
  - Fixed the iteration timeout issue for Automated machine learning.
  - Improved the time-series transformation performance for Automated machine learning.

2019-06-24

Azure Machine Learning Data Prep SDK v1.1.6

- **New features**
  - Added summary functions for top values (`SummaryFunction.TOPVALUES`) and bottom values (`SummaryFunction.BOTTOMVALUES`).
- **Bug fixes and improvements**
  - Significantly improved the performance of `read_pandas_dataframe` .
  - Fixed a bug that would cause `get_profile()` on a Dataflow pointing to binary files to fail.
  - Exposed `set_diagnostics_collection()` to allow for programmatic enabling/disabling of the telemetry collection.
  - Changed the behavior of `get_profile()` . NaN values are now ignored for Min, Mean, Std, and Sum, which aligns with the behavior of Pandas.

2019-06-10

Azure Machine Learning SDK for Python v1.0.43

- **New features**
  - Azure Machine Learning now provides first-class support for popular machine learning and data analysis framework Scikit-learn. Using `SKLearn` estimator, users can easily train and deploy Scikit-learn models.
    - Learn how to run hyperparameter tuning with Scikit-learn using HyperDrive.
  - Added support for creating ModuleStep in pipelines along with Module and ModuleVersion classes to manage reusable compute units.
  - ACI webservices now support persistent scoring_uri through updates. The scoring_uri will change from IP to FQDN. The Dns Name Label for FQDN can be configured by setting the dns_name_label on deploy_configuration.

- Automated machine learning new features:
  - STL featurizer for forecasting
  - KMeans clustering is enabled for feature sweeping
- AmlCompute Quota approvals just became faster! We have now automated the process to approve your quota requests within a threshold. For more information on how quotas work, learn how to manage quotas.

- **Preview features**
  - Integration with MLflow 1.0.0 tracking through azureml-mlflow package (example notebooks).
  - Submit Jupyter notebook as a run. API Reference Documentation
  - Public Preview of Data Drift Detector through azureml-contrib-datadrift package (example notebooks). Data Drift is one of the top reasons where model accuracy degrades over time. It happens when data served to model in production is different from the data that the model was trained on. AML Data Drift detector helps customer to monitor data drift and sends alert whenever drift is detected.
- **Breaking changes**
- **Bug fixes and improvements**
  - RunConfiguration load and save supports specifying a full file path with full back-compat for previous behavior.
  - Added caching in ServicePrincipalAuthentication, turned off by default.
  - Enable logging of multiple plots under the same metric name.
  - Model class now properly importable from azureml.core (`from azureml.core import Model`).
  - In pipeline steps, `hash_path` parameter is now deprecated. New behavior is to hash complete source_directory, except files listed in .amlignore or .gitignore.
  - In pipeline packages, various `get_all` and `get_all_*` methods have been deprecated in favor of `list` and `list_*`, respectively.
  - azureml.core.get_run no longer requires classes to be imported before returning the original run type.
  - Fixed an issue where some calls to WebService Update did not trigger an update.
  - Scoring timeout on AKS webservices should be between 5ms and 300000ms. Max allowed scoring_timeout_ms for scoring requests has been bumped from 1 min to 5 min.
  - LocalWebservice objects now have `scoring_uri` and `swagger_uri` properties.
  - Moved outputs directory creation and outputs directory upload out of the user process. Enabled run history SDK to run in every user process. This should resolve some synchronization issues experienced by distributed training runs.
  - The name of the azureml log written from the user process name will now include process name (for distributed training only) and PID.

Azure Machine Learning Data Prep SDK v1.1.5

- **Bug fixes and improvements**
  - For interpreted datetime values that have a 2-digit year format, the range of valid years has been updated to match Windows May Release. The range has been changed from 1930-2029 to 1950-2049.
  - When reading in a file and setting `handleQuotedLineBreaks=True`, `\r` will be treated as a new line.
  - Fixed a bug that caused `read_pandas_dataframe` to fail in some cases.
  - Improved performance of `get_profile`.
  - Improved error messages.

2019-05-28

Azure Machine Learning Data Prep SDK v1.1.4

- New features
  - You can now use the following expression language functions to extract and parse datetime values into new columns.
    - `RegEx.extract_record()` extracts datetime elements into a new column.
    - `create_datetime()` creates datetime objects from separate datetime elements.
  - When calling `get_profile()`, you can now see that quantile columns are labeled as (est.) to clearly indicate that the values are approximations.
  - You can now use ** globbing when reading from Azure Blob Storage.
    - e.g. `dprep.read_csv(path='https://yourblob.blob.core.windows.net/yourcontainer/**/data/*.csv')`
- Bug fixes
  - Fixed a bug related to reading a Parquet file from a remote source (Azure Blob).

2019-05-14

Azure Machine Learning SDK for Python v1.0.39

- Changes
  - Run configuration auto_prepare_environment option is being deprecated, with auto prepare becoming the default.

2019-05-08

Azure Machine Learning Data Prep SDK v1.1.3

- New features
  - Added support to read from a PostgresSQL database, either by calling read_postgresql or using a Datastore.
    - See examples in how-to guides:
      - Data Ingestion notebook
      - Datastore notebook
- Bug fixes and improvements
  - Fixed issues with column type conversion:
  - Now correctly converts a boolean or numeric column to a boolean column.
  - Now does not fail when attempting to set a date column to be date type.
  - Improved JoinType types and accompanying reference documentation. When joining two dataflows, you can now specify one of these types of join:
    - NONE, MATCH, INNER, UNMATCHLEFT, LEFTANTI, LEFTOUTER, UNMATCHRIGHT, RIGHTANTI, RIGHTOUTER, FULLANTI, FULL.
  - Improved data type inferencing to recognize more date formats.

2019-05-06

Azure portal

In Azure portal, you can now:

- Create and run automated ML experiments
- Create a Notebook VM to try out sample Jupyter notebooks or your own.
- Brand new Authoring section (Preview) in the Azure Machine Learning workspace, which includes Automated Machine Learning, Visual Interface and Hosted Notebook VMs

    - Automatically create a model using Automated machine learning
    - Use a drag and drop Visual Interface to run experiments
    - Create a Notebook VM to explore data, create models, and deploy services.
- Live chart and metric updating in run reports and run details pages
- Updated file viewer for logs, outputs, and snapshots in Run details pages.
- New and improved report creation experience in the Experiments tab.
- Added ability to download the config.json file from the Overview page of the Azure Machine Learning workspace.
- Support Azure Machine Learning workspace creation from the Azure Databricks workspace.

2019-04-26

Azure Machine Learning SDK for Python v1.0.33

- **New features**
    - The *Workspace.create* method now accepts default cluster configurations for CPU and GPU clusters.
    - If Workspace creation fails, depended resources are cleaned.
    - Default Azure Container Registry SKU was switched to basic.
    - Azure Container Registry is created lazily, when needed for run or image creation.
    - Support for Environments for training runs.

Notebook Virtual Machine

Use a Notebook VM as a secure, enterprise-ready hosting environment for Jupyter notebooks in which you can program machine learning experiments, deploy models as web endpoints and perform all other operations supported by Azure Machine Learning SDK using Python. It provides several capabilities:

- Quickly spin up a preconfigured notebook VM that has the latest version of Azure Machine Learning SDK and related packages.
- Access is secured through proven technologies, such as HTTPS, Azure Active Directory authentication and authorization.
- Reliable cloud storage of notebooks and code in your Azure Machine Learning Workspace blob storage account. You can safely delete your notebook VM without losing your work.
- Preinstalled sample notebooks to explore and experiment with Azure Machine Learning features.
- Full customization capabilities of Azure VMs, any VM type, any packages, any drivers.

2019-04-26

Azure Machine Learning SDK for Python v1.0.33 released.

- Azure ML Hardware Accelerated Models on FPGAs is generally available.
    - You can now use the azureml-accel-models package to:
        - Train the weights of a supported deep neural network (ResNet 50, ResNet 152, DenseNet-121, VGG-16, and SSD-VGG)
        - Use transfer learning with the supported DNN
        - Register the model with Model Management Service and containerize the model
        - Deploy the model to an Azure VM with an FPGA in an Azure Kubernetes Service (AKS) cluster
    - Deploy the container to an Azure Data Box Edge server device
    - Score your data with the gRPC endpoint with this sample

Automated Machine Learning

- Feature sweeping to enable dynamically adding featurizers for performance optimization. New featurizers: work embeddings, weight of evidence, target encodings, text target encoding, cluster distance

- Smart CV to handle train/valid splits inside automated ML
- Few memory optimization changes and runtime performance improvement
- Performance improvement in model explanation
- ONNX model conversion for local run
- Added Subsampling support
- Intelligent Stopping when no exit criteria defined
- Stacked ensembles
- Time Series Forecasting
    - New predict forecast function
    - You can now use rolling-origin cross validation on time series data
    - New functionality added to configure time series lags
    - New functionality added to support rolling window aggregate features
    - New Holiday detection and featurizer when country code is defined in experiment settings
- Azure Databricks
    - Enabled time series forecasting and model explainabilty/interpretability capability
    - You can now cancel and resume (continue) automated ML experiments
    - Added support for multicore processing

MLOps

- **Local deployment & debugging for scoring containers**
  You can now deploy an ML model locally and iterate quickly on your scoring file and dependencies to ensure they behave as expected.
- **Introduced InferenceConfig & Model.deploy()**
  Model deployment now supports specifying a source folder with an entry script, the same as a RunConfig. Additionally, model deployment has been simplified to a single command.
- **Git reference tracking**
  Customers have been requesting basic Git integration capabilities for some time as it helps maintain an end-to-end audit trail. We have implemented tracking across major entities in Azure ML for Git-related metadata (repo, commit, clean state). This information will be collected automatically by the SDK and CLI.
- **Model profiling & validation service**
  Customers frequently complain of the difficulty to properly size the compute associated with their inference service. With our model profiling service, the customer can provide sample inputs and we will profile across 16 different CPU / memory configurations to determine optimal sizing for deployment.
- **Bring your own base image for inference**
  Another common complaint was the difficulty in moving from experimentation to inference RE sharing dependencies. With our new base image sharing capability, you can now reuse your experimentation base images, dependencies and all, for inference. This should speed up deployments and reduce the gap from the inner to the outer loop.
- **Improved Swagger schema generation experience**
  Our previous swagger generation method was error prone and impossible to automate. We have a new in-line way of generating swagger schemas from any Python function via decorators. We have open-sourced this code and our schema generation protocol is not coupled to the Azure ML platform.
- **Azure ML CLI is generally available (GA)**

Models can now be deployed with a single CLI command. We got common customer feedback that no one deploys an ML model from a Jupyter notebook. The **CLI reference documentation** has been updated.

2019-04-22

Azure Machine Learning SDK for Python v1.0.30 released.

The `PipelineEndpoint` was introduce to add a new version of a published pipeline while maintaining same endpoint.

2019-04-17

Azure Machine Learning Data Prep SDK v1.1.2

Note: Data Prep Python SDK will no longer install `numpy` and `pandas` packages. See updated installation instructions.

- **New features**
  - You can now use the Pivot transform.
    - How-to guide: Pivot notebook
  - You can now use regular expressions in native functions.
    - Examples:
      - `dflow.filter(dprep.RegEx('pattern').is_match(dflow['column_name']))`
      - `dflow.assert_value('column_name', dprep.RegEx('pattern').is_match(dprep.value))`
  - You can now use `to_upper` and `to_lower` functions in expression language.
  - You can now see the number of unique values of each column in a data profile.
  - For some of the commonly used reader steps, you can now pass in the `infer_column_types` argument. If it is set to `True`, Data Prep will attempt to detect and automatically convert column types.
    - `inference_arguments` is now deprecated.
  - You can now call `Dataflow.shape`.
- **Bug fixes and improvements**
  - `keep_columns` now accepts an additional optional argument `validate_column_exists`, which checks if the result of `keep_columns` will contain any columns.
  - All reader steps (which read from a file) now accept an additional optional argument `verify_exists`.
  - Improved performance of reading from pandas dataframe and getting data profiles.
  - Fixed a bug where slicing a single step from a Dataflow failed with a single index.

2019-04-15

Azure portal

- You can now resubmit an existing Script run on an existing remote compute cluster.
- You can now run a published pipeline with new parameters on the Pipelines tab.
- Run details now supports a new Snapshot file viewer. You can view a snapshot of the directory when you submitted a specific run. You can also download the notebook that was submitted to start the run.
- You can now cancel parent runs from the Azure portal.

2019-04-08

Azure Machine Learning SDK for Python v1.0.23

- **New features**

- The Azure Machine Learning SDK now supports Python 3.7.
- Azure Machine Learning DNN Estimators now provide built-in multi-version support. For example, `TensorFlow` estimator now accepts a `framework_version` parameter, and users can specify version '1.10' or '1.12'. For a list of the versions supported by your current SDK release, call `get_supported_versions()` on the desired framework class (for example, `TensorFlow.get_supported_versions()`). For a list of the versions supported by the latest SDK release, see the DNN Estimator documentation.

Azure Machine Learning Data Prep SDK v1.1.1

- New features
  - You can read multiple Datastore/DataPath/DataReference sources using read_* transforms.
  - You can perform the following operations on columns to create a new column: division, floor, modulo, power, length.
  - Data Prep is now part of the Azure ML diagnostics suite and will log diagnostic information by default.
    - To turn this off, set this environment variable to true: DISABLE_DPREP_LOGGER
- Bug fixes and improvements
  - Improved code documentation for commonly used classes and functions.
  - Fixed a bug in auto_read_file that failed to read Excel files.
  - Added option to overwrite the folder in read_pandas_dataframe.
  - Improved performance of dotnetcore2 dependency installation, and added support for Fedora 27/28 and Ubuntu 1804.
  - Improved the performance of reading from Azure Blobs.
  - Column type detection now supports columns of type Long.
  - Fixed a bug where some date values were being displayed as timestamps instead of Python datetime objects.
  - Fixed a bug where some type counts were being displayed as doubles instead of integers.

2019-03-25

Azure Machine Learning SDK for Python v1.0.21

- New features
  - The *azureml.core.Run.create_children* method allows low-latency creation of multiple child runs with a single call.

Azure Machine Learning Data Prep SDK v1.1.0

- Breaking changes
  - The concept of the Data Prep Package has been deprecated and is no longer supported. Instead of persisting multiple Dataflows in one Package, you can persist Dataflows individually.
    - How-to guide: Opening and Saving Dataflows notebook
- New features
  - Data Prep can now recognize columns that match a particular Semantic Type, and split accordingly. The STypes currently supported include: email address, geographic coordinates (latitude & longitude), IPv4 and IPv6 addresses, US phone number, and US zip code.
    - How-to guide: Semantic Types notebook
  - Data Prep now supports the following operations to generate a resultant column from two numeric columns: subtract, multiply, divide, and modulo.
  - You can call `verify_has_data()` on a Dataflow to check whether the Dataflow would produce records if executed.
- Bug fixes and improvements

- You can now specify the number of bins to use in a histogram for numeric column profiles.
- The `read_pandas_dataframe` transform now requires the DataFrame to have string- or byte- typed column names.
- Fixed a bug in the `fill_nulls` transform, where values were not correctly filled in if the column was missing.

2019-03-11

Azure Machine Learning SDK for Python v1.0.18

- Changes
  - The azureml-tensorboard package replaces azureml-contrib-tensorboard.
  - With this release, you can set up a user account on your managed compute cluster (amlcompute), while creating it. This can be done by passing these properties in the provisioning configuration. You can find more details in the SDK reference documentation.

Azure Machine Learning Data Prep SDK v1.0.17

- New features
  - Now supports adding two numeric columns to generate a resultant column using the expression language.
- Bug fixes and improvements
  - Improved the documentation and parameter checking for random_split.

2019-02-27

Azure Machine Learning Data Prep SDK v1.0.16

- Bug fix
  - Fixed a Service Principal authentication issue that was caused by an API change.

2019-02-25

Azure Machine Learning SDK for Python v1.0.17

- New features
  - Azure Machine Learning now provides first class support for popular DNN framework Chainer. Using `Chainer` class users can easily train and deploy Chainer models.
    - Learn how to run distributed training with ChainerMN
    - Learn how to run hyperparameter tuning with Chainer using HyperDrive
  - Azure Machine Learning Pipelines added ability to trigger a Pipeline run based on datastore modifications. The pipeline schedule notebook is updated to showcase this feature.
- Bug fixes and improvements
  - We have added support in Azure Machine Learning pipelines for setting the source_directory_data_store property to a desired datastore (such as a blob storage) on RunConfigurations that are supplied to the PythonScriptStep. By default Steps use Azure File store as the backing datastore, which may run into throttling issues when a large number of steps are executed concurrently.

Azure portal

- New features
  - New drag and drop table editor experience for reports. Users can drag a column from the well to the table area where a preview of the table will be displayed. The columns can be rearranged.

- New Logs file viewer
- Links to experiment runs, compute, models, images, and deployments from the activities tab

Azure Machine Learning Data Prep SDK v1.0.15

- New features
  - Data Prep now supports writing file streams from a dataflow. Also provides the ability to manipulate the file stream names to create new file names.
    - How-to guide: Working With File Streams notebook
- Bug fixes and improvements
  - Improved performance of t-Digest on large data sets.
  - Data Prep now supports reading data from a DataPath.
  - One hot encoding now works on boolean and numeric columns.
  - Other miscellaneous bug fixes.

2019-02-11

Azure Machine Learning SDK for Python v1.0.15

- New features
  - Azure Machine Learning Pipelines added AzureBatchStep (notebook), HyperDriveStep (notebook), and time-based scheduling functionality (notebook).
  - DataTranferStep updated to work with Azure SQL Server and Azure database for PostgreSQL (notebook).
- Changes
  - Deprecated `PublishedPipeline.get_published_pipeline` in favor of `PublishedPipeline.get`.
  - Deprecated `Schedule.get_schedule` in favor of `Schedule.get`.

Azure Machine Learning Data Prep SDK v1.0.12

- New features
  - Data Prep now supports reading from an Azure SQL database using Datastore.
- Changes
  - Improved the memory performance of certain operations on large data.
  - `read_pandas_dataframe()` now requires `temp_folder` to be specified.
  - The `name` property on `ColumnProfile` has been deprecated - use `column_name` instead.

2019-01-28

Azure Machine Learning SDK for Python v1.0.10

- Changes:
  - Azure ML SDK no longer has azure-cli packages as dependency. Specifically, azure-cli-core and azure-cli-profile dependencies have been removed from azureml-core. These are the user impacting changes:
    - If you are performing "az login" and then using azureml-sdk, the SDK will do the browser or device code log in one more time. It won't use any credentials state created by "az login".
    - For Azure CLI authentication, such as using "az login", use *azureml.core.authentication.AzureCliAuthentication* class. For Azure CLI authentication, do *pip install azure-cli* in the Python environment where you have installed azureml-sdk.
    - If you are doing "az login" using a service principal for automation, we recommend using *azureml.core.authentication.ServicePrincipalAuthentication* class, as azureml-sdk won't use

credentials state created by azure CLI.

- **Bug fixes**: This release mostly contains minor bug fixes

Azure Machine Learning Data Prep SDK v1.0.8

- **Bug fixes**
  - Improved the performance of getting data profiles.
  - Fixed minor bugs related to error reporting.

Azure portal: new features

- New drag and drop charting experience for reports. Users can drag a column or attribute from the well to the chart area where the system will automatically select an appropriate chart type for the user based on the type of data. Users can change the chart type to other applicable types or add additional attributes.

  Supported Chart Types:

  - Line Chart
  - Histogram
  - Stacked Bar Chart
  - Box Plot
  - Scatter Plot
  - Bubble Plot
- The portal now dynamically generates reports for experiments. When a user submits a run to an experiment, a report will automatically be generated with logged metrics and graphs to allow comparison across different runs.

2019-01-14

Azure Machine Learning SDK for Python v1.0.8

- **Bug fixes**: This release mostly contains minor bug fixes

Azure Machine Learning Data Prep SDK v1.0.7

- **New features**
  - Datastore improvements (documented in Datastore how-to-guide)
    - Added ability to read from and write to Azure File Share and ADLS Datastores in scale-up.
    - When using Datastores, Data Prep now supports using service principal authentication instead of interactive authentication.
    - Added support for wasb and wasbs urls.

2019-01-09

Azure Machine Learning Data Prep SDK v1.0.6

- **Bug fixes**
  - Fixed bug with reading from public readable Azure Blob containers on Spark

2018-12-20

Azure Machine Learning SDK for Python v1.0.6

- **Bug fixes**: This release mostly contains minor bug fixes

Azure Machine Learning Data Prep SDK v1.0.4

- **New features**

- `to_bool` function now allows mismatched values to be converted to Error values. This is the new default mismatch behavior for `to_bool` and `set_column_types`, whereas the previous default behavior was to convert mismatched values to False.
- When calling `to_pandas_dataframe`, there is a new option to interpret null/missing values in numeric columns as NaN.
- Added ability to check the return type of some expressions to ensure type consistency and fail early.
- You can now call `parse_json` to parse values in a column as JSON objects and expand them into multiple columns.

- **Bug fixes**
  - Fixed a bug that crashed `set_column_types` in Python 3.5.2.
  - Fixed a bug that crashed when connecting to Datastore using an AML image.
- **Updates**
  - Example Notebooks for getting started tutorials, case studies, and how-to guides.

2018-12-04: General Availability

Azure Machine Learning is now generally available.

Azure Machine Learning Compute

With this release, we are announcing a new managed compute experience through the Azure Machine Learning Compute. This compute target replaces Azure Batch AI compute for Azure Machine Learning.

This compute target:

- Is used for model training and batch inference/scoring
- Is single- to multi-node compute
- Does the cluster management and job scheduling for the user
- Autoscales by default
- Support for both CPU and GPU resources
- Enables use of low-priority VMs for reduced cost

Azure Machine Learning Compute can be created in Python, using Azure portal, or the CLI. It must be created in the region of your workspace, and cannot be attached to any other workspace. This compute target uses a Docker container for your run, and packages your dependencies to replicate the same environment across all your nodes.

> **WARNING**
> We recommend creating a new workspace to use Azure Machine Learning Compute. There is a remote chance that users trying to create Azure Machine Learning Compute from an existing workspace might see an error. Existing compute in your workspace should continue to work unaffected.

Azure Machine Learning SDK for Python v1.0.2

- **Breaking changes**
  - With this release, we are removing support for creating a VM from Azure Machine Learning. You can still attach an existing cloud VM or a remote on-premises server.
  - We are also removing support for BatchAI, all of which should be supported through Azure Machine Learning Compute now.
- **New**
  - For machine learning pipelines:

    - EstimatorStep
    - HyperDriveStep
    - MpiStep
- Updated
    - For machine learning pipelines:
        - DatabricksStep now accepts runconfig
        - DataTransferStep now copies to and from a SQL datasource
        - Schedule functionality in SDK to create and update schedules for running published pipelines

Azure Machine Learning Data Prep SDK v0.5.2

- Breaking changes
    - `SummaryFunction.N` was renamed to `SummaryFunction.Count`.
- Bug Fixes
    - Use the latest AML Run Token when reading from and writing to datastores on remote runs. Previously, if the AML Run Token is updated in Python, the Data Prep runtime will not be updated with the updated AML Run Token.
    - Additional clearer error messages
    - to_spark_dataframe() will no longer crash when Spark uses `Kryo` serialization
    - Value Count Inspector can now show more than 1000 unique values
    - Random Split no longer fails if the original Dataflow doesn't have a name
- More information
    - Azure Machine Learning Data Prep SDK

Docs and notebooks

- ML Pipelines
    - New and updated notebooks for getting started with pipelines, batch scoping, and style transfer examples: https://aka.ms/aml-pipeline-notebooks
    - Learn how to create your first pipeline
    - Learn how to run batch predictions using pipelines
- Azure Machine Learning compute target
    - Sample notebooks are now updated to use the new managed compute.
    - Learn about this compute

Azure portal: new features

- Create and manage Azure Machine Learning Compute types in the portal.
- Monitor quota usage and request quota for Azure Machine Learning Compute.
- View Azure Machine Learning Compute cluster status in real time.
- Virtual network support was added for Azure Machine Learning Compute and Azure Kubernetes Service creation.
- Rerun your published pipelines with existing parameters.
- New automated machine learning charts for classification models (lift, gains, calibration, feature importance chart with model explainability) and regression models (residuals and feature importance chart with model explainability).
- Pipelines can be viewed in Azure portal

2018-11-20

Azure Machine Learning SDK for Python v0.1.80

- Breaking changes
  - *azureml.train.widgets* namespace has moved to *azureml.widgets*.
  - *azureml.core.compute.AmlCompute* deprecates the following classes - *azureml.core.compute.BatchAICompute* and *azureml.core.compute.DSVMCompute*. The latter class will be removed in subsequent releases. The AmlCompute class has an easier definition now, and simply needs a vm_size and the max_nodes, and will automatically scale your cluster from 0 to the max_nodes when a job is submitted. Our sample notebooks have been updated with this information and should give you usage examples. We hope you like this simplification and lots of more exciting features to come in a later release!

Azure Machine Learning Data Prep SDK v0.5.1

Learn more about the Data Prep SDK by reading reference docs.

- New Features
  - Created a new DataPrep CLI to execute DataPrep packages and view the data profile for a dataset or dataflow
  - Redesigned SetColumnType API to improve usability
  - Renamed smart_read_file to auto_read_file
  - Now includes skew and kurtosis in the Data Profile
  - Can sample with stratified sampling
  - Can read from zip files that contain CSV files
  - Can split datasets row-wise with Random Split (for example, into test-train sets)
  - Can get all the column data types from a dataflow or a data profile by calling `.dtypes`
  - Can get the row count from a dataflow or a data profile by calling `.row_count`
- Bug Fixes
  - Fixed long to double conversion
  - Fixed assert after any add column
  - Fixed an issue with FuzzyGrouping, where it would not detect groups in some cases
  - Fixed sort function to respect multi-column sort order
  - Fixed and/or expressions to be similar to how `pandas` handles them
  - Fixed reading from dbfs path
  - Made error messages more understandable
  - Now no longer fails when reading on remote compute target using an AML token
  - Now no longer fails on Linux DSVM
  - Now no longer crashes when non-string values are in string predicates
  - Now handles assertion errors when Dataflow should fail correctly
  - Now supports dbutils mounted storage locations on Azure Databricks

2018-11-05

Azure portal

The Azure portal for Azure Machine Learning has the following updates:

- A new **Pipelines** tab for published pipelines.
- Added support for attaching an existing HDInsight cluster as a compute target.

Azure Machine Learning SDK for Python v0.1.74

- Breaking changes

- *Workspace.compute_targets, datastores, experiments, images, models, and *webservices* are properties instead of methods. For example, replace *Workspace.compute_targets()* with *Workspace.compute_targets*.
- *Run.get_context* deprecates *Run.get_submitted_run*. The latter method will be removed in subsequent releases.
- *PipelineData* class now expects a datastore object as a parameter rather than datastore_name. Similarly, *Pipeline* accepts default_datastore rather than default_datastore_name.

- **New features**
  - The Azure Machine Learning Pipelines sample notebook now uses MPI steps.
  - The RunDetails widget for Jupyter notebooks is updated to show a visualization of the pipeline.

Azure Machine Learning Data Prep SDK v0.4.0

- **New features**
  - Type Count added to Data Profile
  - Value Count and Histogram is now available
  - More percentiles in Data Profile
  - The Median is available in Summarize
  - Python 3.7 is now supported
  - When you save a dataflow that contains datastores to a DataPrep package, the datastore information will be persisted as part of the DataPrep package
  - Writing to datastore is now supported
- **Bug fixed**
  - 64-bit unsigned integer overflows are now handled properly on Linux
  - Fixed incorrect text label for plain text files in smart_read
  - String column type now shows up in metrics view
  - Type count now is fixed to show ValueKinds mapped to single FieldType instead of individual ones
  - Write_to_csv no longer fails when path is provided as a string
  - When using Replace, leaving "find" blank will no longer fail

2018-10-12

Azure Machine Learning SDK for Python v0.1.68

- **New features**
  - Multi-tenant support when creating new workspace.
- **Bugs fixed**
  - You no longer need to pin the pynacl library version when deploying web service.

Azure Machine Learning Data Prep SDK v0.3.0

- **New features**
  - Added method transform_partition_with_file(script_path), which allows users to pass in the path of a Python file to execute

2018-10-01

Azure Machine Learning SDK for Python v0.1.65

Version 0.1.65 includes new features, more documentation, bug fixes, and more sample notebooks.

See the list of known issues to learn about known bugs and workarounds.

- **Breaking changes**
  - Workspace.experiments, Workspace.models, Workspace.compute_targets, Workspace.images, Workspace.web_services return dictionary, previously returned list. See azureml.core.Workspace API documentation.
  - Automated Machine Learning removed normalized mean square error from the primary metrics.
- **HyperDrive**
  - Various HyperDrive bug fixes for Bayesian, Performance improvements for get Metrics calls.
  - Tensorflow 1.10 upgrade from 1.9
  - Docker image optimization for cold start.
  - Jobs now report correct status even if they exit with error code other than 0.
  - RunConfig attribute validation in SDK.
  - HyperDrive run object supports cancel similar to a regular run: no need to pass any parameters.
  - Widget improvements for maintaining state of drop-down values for distributed runs and HyperDrive runs.
  - TensorBoard and other log files support fixed for Parameter server.
  - Intel(R) MPI support on service side.
  - Bugfix to parameter tuning for distributed run fix during validation in BatchAI.
  - Context Manager now identifies the primary instance.
- **Azure portal experience**
  - log_table() and log_row() are supported in Run details.
  - Automatically create graphs for tables and rows with 1, 2 or 3 numerical columns and an optional categorical column.
- **Automated Machine Learning**
  - Improved error handling and documentation
  - Fixed run property retrieval performance issues.
  - Fixed continue run issue.
  - Fixed ensembling iteration issues.
  - Fixed training hanging bug on MAC OS.
  - Downsampling macro average PR/ROC curve in custom validation scenario.
  - Removed extra index logic.
  - Removed filter from get_output API.
- **Pipelines**
  - Added a method Pipeline.publish() to publish a pipeline directly, without requiring an execution run first.
  - Added a method PipelineRun.get_pipeline_runs() to fetch the pipeline runs that were generated from a published pipeline.
- **Project Brainwave**
  - Updated support for new AI models available on FPGAs.

Azure Machine Learning Data Prep SDK v0.2.0

Version 0.2.0 includes following features and bug fixes:

- **New features**
  - Support for one-hot encoding
  - Support for quantile transform

- **Bug fixed:**
  - Works with any Tornado version, no need to downgrade your Tornado version
  - Value counts for all values, not just the top three

2018-09 (Public preview refresh)

A new, refreshed release of Azure Machine Learning: Read more about this release: https://azure.microsoft.com/blog/what-s-new-in-azure-machine-learning-service/

Next steps

Read the overview for Azure Machine Learning.

Known issues and troubleshooting Azure Machine Learning

4/24/2020 • 11 minutes to read • Edit Online

This article helps you find and correct errors or failures you may encounter when using Azure Machine Learning.

Diagnostic logs

Sometimes it can be helpful if you can provide diagnostic information when asking for help. To see some logs:

1. Visit Azure Machine Learning studio.
2. On the left-hand side, select **Experiment**
3. Select an experiment.
4. Select a run.
5. On the top, select **Outputs + logs**.

> **NOTE**
> Azure Machine Learning logs information from a variety of sources during training, such as AutoML or the Docker container that runs the training job. Many of these logs are not documented. If you encounter problems and contact Microsoft support, they may be able to use these logs during troubleshooting.

Resource quotas

Learn about the resource quotas you might encounter when working with Azure Machine Learning.

Installation and import

- **Pip Installation: Dependencies are not guaranteed to be consistent with single line installation**:

  This is a known limitation of pip, as it does not have a functioning dependency resolver when you install as a single line. The first unique dependency is the only one it looks at.

  In the following code `azure-ml-datadrift` and `azureml-train-automl` are both installed using a single line pip install.

  ```
  pip install azure-ml-datadrift, azureml-train-automl
  ```

  For this example, let's say `azure-ml-datadrift` requires version > 1.0 and `azureml-train-automl` requires version < 1.2. If the latest version of `azure-ml-datadrift` is 1.3, then both packages get upgraded to 1.3, regardless of the `azureml-train-automl` package requirement for an older version.

  To ensure the appropriate versions are installed for your packages, install using multiple lines like in the following code. Order isn't an issue here, since pip explicitly downgrades as part of the next line call. And so, the appropriate version dependencies are applied.

  ```
  pip install azure-ml-datadrift
  pip install azureml-train-automl
  ```

- **Error message: Cannot uninstall 'PyYAML'**

  Azure Machine Learning SDK for Python: PyYAML is a `distutils` installed project. Therefore, we cannot accurately determine which files belong to it if there is a partial uninstall. To continue installing the SDK while ignoring this error, use:

  ```
  pip install --upgrade azureml-sdk[notebooks,automl] --ignore-installed PyYAML
  ```

- **Databricks failure when installing packages**

  Azure Machine Learning SDK installation fails on Azure Databricks when more packages are installed. Some packages, such as `psutil`, can cause conflicts. To avoid installation errors, install packages by freezing the library version. This issue is related to Databricks and not to the Azure Machine Learning SDK. You might experience this issue with other libraries, too. Example:

  ```
  psutil cryptography==1.5 pyopenssl==16.0.0 ipython==2.2.0
  ```

  Alternatively, you can use init scripts if you keep facing install issues with Python libraries. This approach isn't officially supported. For more information, see Cluster-scoped init scripts.

- **Databricks import error: cannot import name 'Timedelta' from 'pandas._libs.tslibs'**: If you see this error when you use automated machine learning, run the two following lines in your notebook:

  ```
  %sh rm -rf /databricks/python/lib/python3.7/site-packages/pandas-0.23.4.dist-info
  /databricks/python/lib/python3.7/site-packages/pandas
  %sh /databricks/python/bin/pip install pandas==0.23.4
  ```

- **Databricks import error: No module named 'pandas.core.indexes'**: If you see this error when you use automated machine learning:

  1. Run this command to install two packages in your Azure Databricks cluster:

     ```
     scikit-learn==0.19.1
     pandas==0.22.0
     ```

  2. Detach and then reattach the cluster to your notebook.

  If these steps don't solve the issue, try restarting the cluster.

- **Databricks FailToSendFeather**: If you see a `FailToSendFeather` error when reading data on Azure Databricks cluster, refer to the following solutions:

  - Upgrade `azureml-sdk[automl]` package to the latest version.
  - Add `azureml-dataprep` version 1.1.8 or above.
  - Add `pyarrow` version 0.11 or above.

Create and manage workspaces

> **WARNING**
> Moving your Azure Machine Learning workspace to a different subscription, or moving the owning subscription to a new tenant, is not supported. Doing so may cause errors.

- **Azure portal**: If you go directly to view your workspace from a share link from the SDK or the portal, you will not be able to view the normal **Overview** page with subscription information in the extension. You will also not be able to switch into another workspace. If you need to view another workspace, go directly to Azure Machine Learning studio and search for the workspace name.

Set up your environment

- **Trouble creating AmlCompute**: There is a rare chance that some users who created their Azure Machine Learning workspace from the Azure portal before the GA release might not be able to create AmlCompute in that workspace. You can either raise a support request against the service or create a new workspace through the portal or the SDK to unblock yourself immediately.

Work with data

Overloaded AzureFile storage

If you receive an error `Unable to upload project files to working directory in AzureFile because the storage is overloaded`, apply following workarounds.

If you are using file share for other workloads, such as data transfer, the recommendation is to use blobs so that file share is free to be used for submitting runs. You may also split the workload between two different workspaces.

Passing data as input

- **TypeError: FileNotFound: No such file or directory**: This error occurs if the file path you provide isn't where the file is located. You need to make sure the way you refer to the file is consistent with where you mounted your dataset on your compute target. To ensure a deterministic state, we recommend using the abstract path when mounting a dataset to a compute target. For example, in the following code we mount the dataset under the root of the filesystem of the compute target, `/tmp`.

  ```
  # Note the leading / in '/tmp/dataset'
  script_params = {
      '--data-folder': dset.as_named_input('dogscats_train').as_mount('/tmp/dataset'),
  }
  ```

  If you don't include the leading forward slash, '/', you'll need to prefix the working directory e.g. `/mnt/batch/.../tmp/dataset` on the compute target to indicate where you want the dataset to be mounted.

Data labeling projects

ISSUE	RESOLUTION
Only datasets created on blob datastores can be used	this is a known limitation of the current release.
After creation, the project shows "Initializing" for a long time	Manually refresh the page. Initialization should proceed at roughly 20 datapoints per second. The lack of autorefresh is a known issue.
When reviewing images, newly labeled images are not shown	To load all labeled images, choose the **First** button. The **First** button will take you back to the front of the list, but loads all labeled data.
Pressing Esc key while labeling for object detection creates a zero size label on the top-left corner. Submitting labels in this state fails.	Delete the label by clicking on the cross mark next to it.

Azure Machine Learning designer

Known issues:

- **Long compute preparation time**: It may be a few minutes or even longer when you first connect to or create a compute target.

Train models

- **ModuleErrors (No module named)**: If you are running into ModuleErrors while submitting experiments in Azure ML, it means that the training script is expecting a package to be installed but it isn't added. Once you provide the package name, Azure ML will install the package in the environment used for your training run.

  If you are using Estimators to submit experiments, you can specify a package name via `pip_packages` or `conda_packages` parameter in the estimator based on from which source you want to install the package. You can also specify a yml file with all your dependencies using `conda_dependencies_file` or list all your pip requirements in a txt file using `pip_requirements_file` parameter. If you have your own Azure ML Environment object that you want to override the default image used by the estimator, you can specify that environment via the `environment` parameter of the estimator constructor.

  Azure ML also provides framework-specific estimators for Tensorflow, PyTorch, Chainer and SKLearn. Using these estimators will make sure that the core framework dependencies are installed on your behalf in the environment used for training. You have the option to specify extra dependencies as described above.

  Azure ML maintained docker images and their contents can be seen in AzureML Containers. Framework-specific dependencies are listed in the respective framework documentation - Chainer, PyTorch, TensorFlow, SKLearn.

  > **NOTE**
  > If you think a particular package is common enough to be added in Azure ML maintained images and environments please raise a GitHub issue in AzureML Containers.

- **NameError (Name not defined), AttributeError (Object has no attribute)**: This exception should come from your training scripts. You can look at the log files from Azure portal to get more information about the specific name not defined or attribute error. From the SDK, you can use `run.get_details()` to look at the error message. This will also list all the log files generated for your run. Please make sure to take a look at your training script and fix the error before resubmitting your run.

- **Horovod has been shut down**: In most cases if you encounter "AbortedError: Horovod has been shut down" this exception means there was an underlying exception in one of the processes that caused Horovod to shut down. Each rank in the MPI job gets it own dedicated log file in Azure ML. These logs are named `70_driver_logs`. In case of distributed training, the log names are suffixed with `_rank` to make it easier to differentiate the logs. To find the exact error that caused Horovod to shut down, go through all the log files and look for `Traceback` at the end of the driver_log files. One of these files will give you the actual underlying exception.

- **Run or experiment deletion**: Experiments can be archived by using the Experiment.archive method, or from the Experiment tab view in Azure Machine Learning studio client via the "Archive experiment" button. This action hides the experiment from list queries and views, but does not delete it.

  Permanent deletion of individual experiments or runs is not currently supported. For more information on deleting Workspace assets, see Export or delete your Machine Learning service workspace data.

- **Metric Document is too large**: Azure Machine Learning has internal limits on the size of metric objects that can be logged at once from a training run. If you encounter a "Metric Document is too large" error when logging a list-valued metric, try splitting the list into smaller chunks, for example:

  ```
  run.log_list("my metric name", my_metric[:N])
  run.log_list("my metric name", my_metric[N:])
  ```

  Internally, Azure ML concatenates the blocks with the same metric name into a contiguous list.

Automated machine learning

- **Tensor Flow**: Automated machine learning does not currently support tensor flow version 1.13. Installing this version will cause package dependencies to stop working. We are working to fix this issue in a future release.
- **Experiment Charts**: Binary classification charts (precision-recall, ROC, gain curve etc.) shown in automated ML experiment iterations are not rendering correctly in user interface since 4/12. Chart plots are currently showing inverse results, where better performing models are shown with lower results. A resolution is under investigation.
- **Databricks cancel an automated machine learning run**: When you use automated machine learning capabilities on Azure Databricks, to cancel a run and start a new experiment run, restart your Azure Databricks cluster.
- **Databricks >10 iterations for automated machine learning**: In automated machine learning settings, if you have more than 10 iterations, set `show_output` to `False` when you submit the run.
- **Databricks widget for the Azure Machine Learning SDK and automated machine learning**: The Azure Machine Learning SDK widget isn't supported in a Databricks notebook because the notebooks can't parse HTML widgets. You can view the widget in the portal by using this Python code in your Azure Databricks notebook cell:

  ```
  displayHTML("<a href={} target='_blank'>Azure Portal: {}</a>".format(local_run.get_portal_url(),
  local_run.id))
  ```

Deploy & serve models

Take these actions for the following errors:

ERROR	RESOLUTION
Image building failure when deploying web service	Add "pynacl==1.2.1" as a pip dependency to Conda file for image configuration
`['DaskOnBatch:context_managers.DaskOnBatch', 'setup.py']' died with <Signals.SIGKILL: 9>`	Change the SKU for VMs used in your deployment to one that has more memory.
FPGA failure	You will not be able to deploy models on FPGAs until you have requested and been approved for FPGA quota. To request access, fill out the quota request form: https://aka.ms/aml-real-time-ai

Updating Azure Machine Learning components in AKS cluster

Updates to Azure Machine Learning components installed in an Azure Kubernetes Service cluster must be

manually applied.

You can apply these updates by detaching the cluster from the Azure Machine Learning workspace, and then reattaching the cluster to the workspace. If TLS is enabled in the cluster, you will need to supply the TLS/SSL certificate and private key when reattaching the cluster.

```
compute_target = ComputeTarget(workspace=ws, name=clusterWorkspaceName)
compute_target.detach()
compute_target.wait_for_completion(show_output=True)

attach_config = AksCompute.attach_configuration(resource_group=resourceGroup,
cluster_name=kubernetesClusterName)

## If SSL is enabled.
attach_config.enable_ssl(
    ssl_cert_pem_file="cert.pem",
    ssl_key_pem_file="key.pem",
    ssl_cname=sslCname)

attach_config.validate_configuration()

compute_target = ComputeTarget.attach(workspace=ws, name=args.clusterWorkspaceName,
attach_configuration=attach_config)
compute_target.wait_for_completion(show_output=True)
```

If you no longer have the TLS/SSL certificate and private key, or you are using a certificate generated by Azure Machine Learning, you can retrieve the files prior to detaching the cluster by connecting to the cluster using `kubectl` and retrieving the secret `azuremlfessl`.

```
kubectl get secret/azuremlfessl -o yaml
```

> **NOTE**
> Kubernetes stores the secrets in base-64 encoded format. You will need to base-64 decode the `cert.pem` and `key.pem` components of the secrets prior to providing them to `attach_config.enable_ssl`.

Webservices in Azure Kubernetes Service failures

Many webservice failures in Azure Kubernetes Service can be debugged by connecting to the cluster using `kubectl`. You can get the `kubeconfig.json` for an Azure Kubernetes Service Cluster by running

```
az aks get-credentials -g <rg> -n <aks cluster name>
```

Authentication errors

If you perform a management operation on a compute target from a remote job, you will receive one of the following errors:

```
{"code":"Unauthorized","statusCode":401,"message":"Unauthorized","details":
[{"code":"InvalidOrExpiredToken","message":"The request token was either invalid or expired. Please try again
with a valid token."}]}
```

```
{"error":{"code":"AuthenticationFailed","message":"Authentication failed."}}
```

For example, you will receive an error if you try to create or attach a compute target from an ML Pipeline that is submitted for remote execution.

What happened to Azure Machine Learning Workbench?

3/10/2020 • 4 minutes to read • Edit Online

The Azure Machine Learning Workbench application and some other early features were deprecated and replaced in the **September 2018** release to make way for an improved architecture.

To improve your experience, the release contains many significant updates prompted by customer feedback. The core functionality from experiment runs to model deployment hasn't changed. But now, you can use the robust Python SDK, R SDK, and the Azure CLI to accomplish your machine learning tasks and pipelines.

Most of the artifacts that were created in the earlier version of Azure Machine Learning are stored in your own local or cloud storage. These artifacts won't ever disappear.

In this article, you learn about what changed and how it affects your pre-existing work with the Azure Machine Learning Workbench and its APIs.

> **WARNING**
> This article is not for Azure Machine Learning Studio users. It is for Azure Machine Learning customers who have installed the Workbench (preview) application and/or have experimentation and model management preview accounts.

What changed?

The latest release of Azure Machine Learning includes the following features:

- A simplified Azure resources model.
- A new portal UI to manage your experiments and compute targets.
- A new, more comprehensive Python SDK.
- The new expanded Azure CLI extension for machine learning.

The architecture was redesigned for ease of use. Instead of multiple Azure resources and accounts, you only need an Azure Machine Learning Workspace. You can create workspaces quickly in the Azure portal. By using a workspace, multiple users can store training and deployment compute targets, model experiments, Docker images, deployed models, and so on.

Although there are new improved CLI and SDK clients in the current release, the desktop workbench application itself has been retired. Experiments can be managed in the workspace dashboard in Azure Machine Learning studio. Use the dashboard to get your experiment history, manage the compute targets attached to your workspace, manage your models and Docker images, and even deploy web services.

Support timeline

On January 9th, 2019 support for Machine Learning Workbench, Azure Machine Learning Experimentation and Model Management accounts, and their associated SDK and CLI ended.

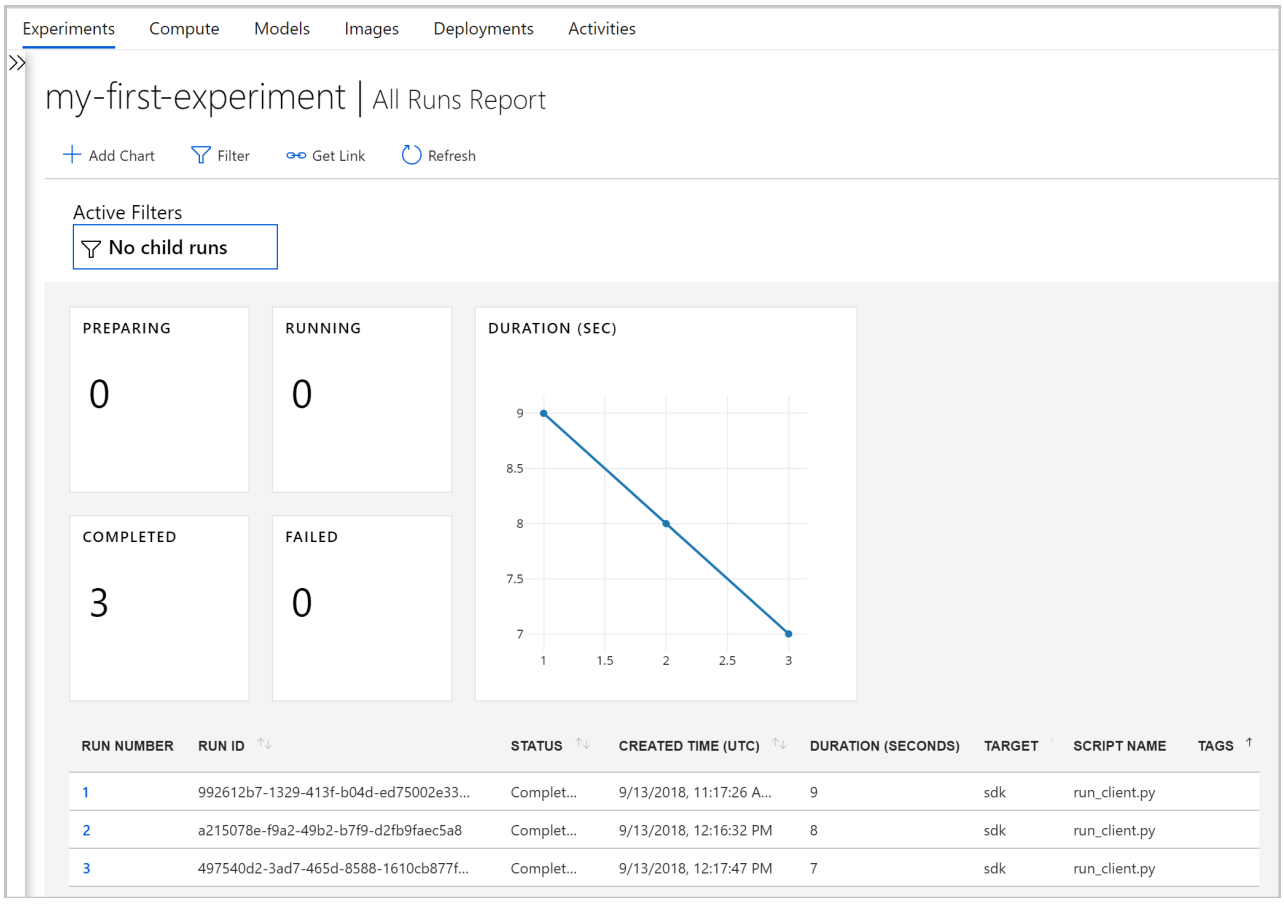
All the latest capabilities are available by using this SDK, the CLI, and the portal.

What about run histories?

Older run histories are no longer accessible, how you can still see your runs in the latest version.

Run histories are now called **experiments**. You can collect your model's experiments and explore them by using the SDK, the CLI, or the Azure Machine Learning studio.

The portal's workspace dashboard is supported on Microsoft Edge, Chrome, and Firefox browsers only:



Start training your models and tracking the run histories using the new CLI and SDK. You can learn how with the Tutorial: train models with Azure Machine Learning.

Will projects persist?

You won't lose any code or work. In the older version, projects are cloud entities with a local directory. In the latest version, you attach local directories to the Azure Machine Learning workspace by using a local config file. See a diagram of the latest architecture.

Much of the project content was already on your local machine. So you just need to create a config file in that directory and reference it in your code to connect to your workspace. To continue using the local directory containing your files and scripts, specify the directory's name in the 'experiment.submit' Python command or using the `az ml project attach` CLI command. For example:

```
run = exp.submit(source_directory=script_folder,
                 script='train.py', run_config=run_config_system_managed)
```

Create a workspace to get started.

What about my registered models and images?

The models that you registered in your old model registry must be migrated to your new workspace if you want to continue to use them. To migrate your models, download the models and re-register them in your new workspace.

The images that you created in your old image registry cannot be directly migrated to the new workspace. In most cases, the model can be deployed without having to create an image. If needed, you can create an image for the model in the new workspace. For more information, see Manage, register, deploy, and monitor machine learning models.

What about deployed web services?

Now that support for the old CLI has ended, you can no longer redeploy models or manage the web services you originally deployed with your Model Management account. However, those web services will continue to work for as long as Azure Container Service (ACS) is still supported.

In the latest version, models are deployed as web services to Azure Container Instances (ACI) or Azure Kubernetes Service (AKS) clusters. You can also deploy to FPGAs and to Azure IoT Edge.

Learn more in these articles:

- Where and how to deploy models
- Tutorial: Deploy models with Azure Machine Learning

Next steps

Learn about the latest architecture for Azure Machine Learning.

For an overview of the service, read What is Azure Machine Learning?.

Create your first experiment with your preferred method:

- Use Python notebooks
- Use R Markdown
- Use automated machine learning
- Use the designer's drag & drop capabilities
- Use the ML extension to the CLI

Use a keyboard to use Azure Machine Learning designer (preview)

2/14/2020 • 2 minutes to read • Edit Online

Learn how to use a keyboard and screen reader to use Azure Machine Learning designer. For a list of keyboard shortcuts that work everywhere in the Azure portal, see Keyboard shortcuts in the Azure portal

This workflow has been tested with Narrator and JAWS, but it should work with other standard screen readers.

Navigate the pipeline graph

The pipeline graph is organized as a nested list. The outer list is a module list, which describes all the modules in the pipeline graph. The inner list is a connection list, which describes all the connections of a specific module.

1. In the module list, use the arrow key to switch modules.
2. Use tab to open the connection list for the target module.
3. Use arrow key to switch between the connection ports for the module.
4. Use "G" to go to the target module.

Edit the pipeline graph

Add a module to the graph

1. Use Ctrl+F6 to switch focus from the canvas to the module tree.
2. Find the desired module in the module tree using standard treeview control.

Edit a module

To connect a module to another module:

1. Use Ctrl + Shift + H when targeting a module in the module list to open the connection helper.
2. Edit the connection ports for the module.

To adjust module properties:

1. Use Ctrl + Shift + E when targeting a module to open the module properties.
2. Edit the module properties.

Navigation shortcuts

KEYSTROKE	DESCRIPTION
Ctrl + F6	Toggle focus between canvas and module tree
Ctrl + F1	Open the information card when focusing on a node in module tree
Ctrl + Shift + H	Open the connection helper when focus is on a node
Ctrl + Shift + E	Open module properties when focus is on a node

KEYSTROKE	DESCRIPTION
Ctrl + G	Move focus to first failed node if the pipeline run failed

Action shortcuts

Use the following shortcuts with the access key. For more information on access keys, see https://en.wikipedia.org/wiki/Access_key.

KEYSTROKE	ACTION
Access key + R	Run
Access key + P	Publish
Access key + C	Clone
Access key + D	Deploy
Access key + I	Create/update inference pipeline
Access key + B	Create/update batch inference pipeline
Access key + K	Open "Create inference pipeline" dropdown
Access key + U	Open "Update inference pipeline" dropdown
Access key + M	Open more(...) dropdown

Next steps

- Turn on high contrast or change theme
- Accessibility related tools at Microsoft