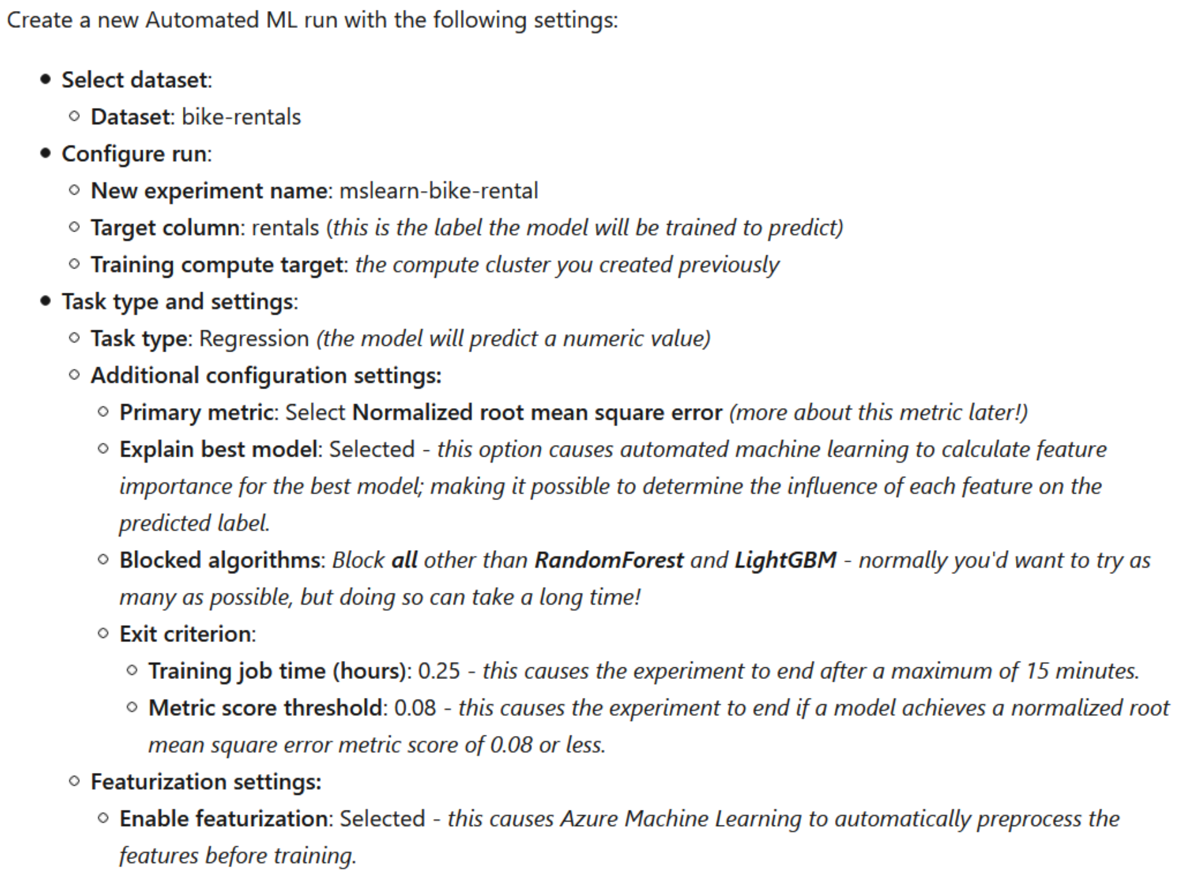
The automated machine learning capability in Azure Machine Learning supports *supervised* machine learning models



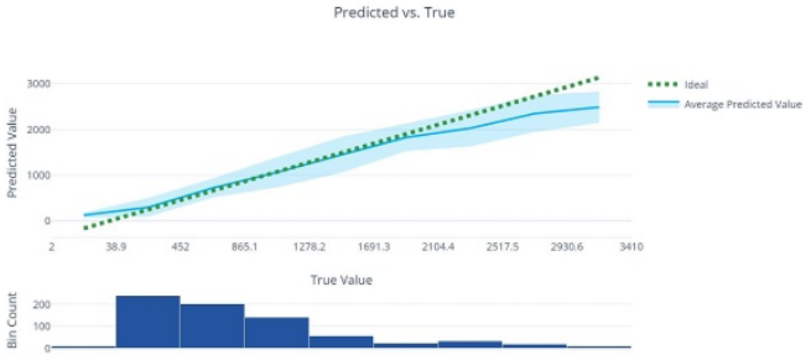
Create a new Automated ML run with the following settings:

- **Select dataset**:
  - **Dataset**: bike-rentals
- **Configure run**:
  - **New experiment name**: mslearn-bike-rental
  - **Target column**: rentals (*this is the label the model will be trained to predict)*
  - **Training compute target**: *the compute cluster you created previously*
- **Task type and settings**:
  - **Task type**: Regression *(the model will predict a numeric value)*
  - **Additional configuration settings:**
    - **Primary metric**: Select **Normalized root mean square error** *(more about this metric later!)*
    - **Explain best model**: Selected - *this option causes automated machine learning to calculate feature importance for the best model; making it possible to determine the influence of each feature on the predicted label.*
    - **Blocked algorithms**: *Block* ***all*** *other than* ***RandomForest*** *and* ***LightGBM*** *- normally you'd want to try as many as possible, but doing so can take a long time!*
    - **Exit criterion**:
      - **Training job time (hours)**: 0.25 - *this causes the experiment to end after a maximum of 15 minutes.*
      - **Metric score threshold**: 0.08 - *this causes the experiment to end if a model achieves a normalized root mean square error metric score of 0.08 or less.*
  - **Featurization settings:**
    - **Enable featurization**: Selected - *this causes Azure Machine Learning to automatically preprocess the features before training.*

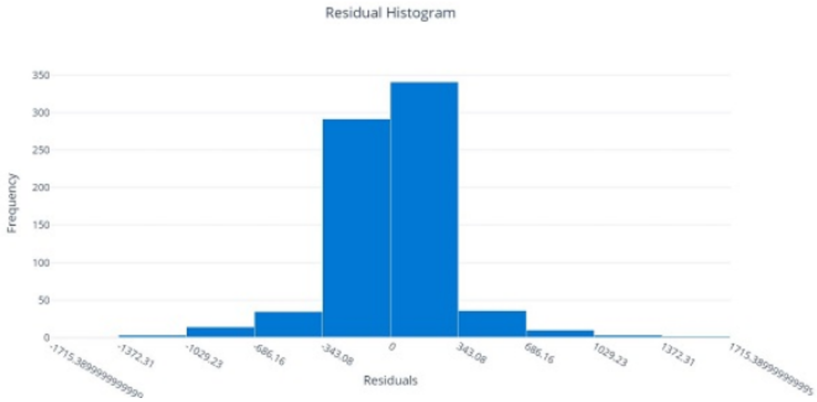
When the run status changes to *Running*, view the **Models** tab and observe as each possible combination of training algorithm and pre-processing steps is tried and the performance of the resulting model is evaluated. The page will automatically refresh periodically,

showing the *residuals* (differences between predicted and actual values) as a histogram.

The **Predicted vs. True** chart should show a diagonal trend in which the predicted value correlates closely to the true value. A dotted line shows how a perfect model should perform, and the closer the line for your model's average predicted value is to this, the better its performance. A histogram below the line chart shows the distribution of true values.



The **Residual Histogram** shows the frequency of residual value ranges. Residuals represent variance between predicted and true values that can't be explained by the model - in other words, errors; so what you should hope to see is that the most frequently occurring residual values are clustered around 0 (in other words, most of the errors are small), with fewer errors at the extreme ends of the scale.



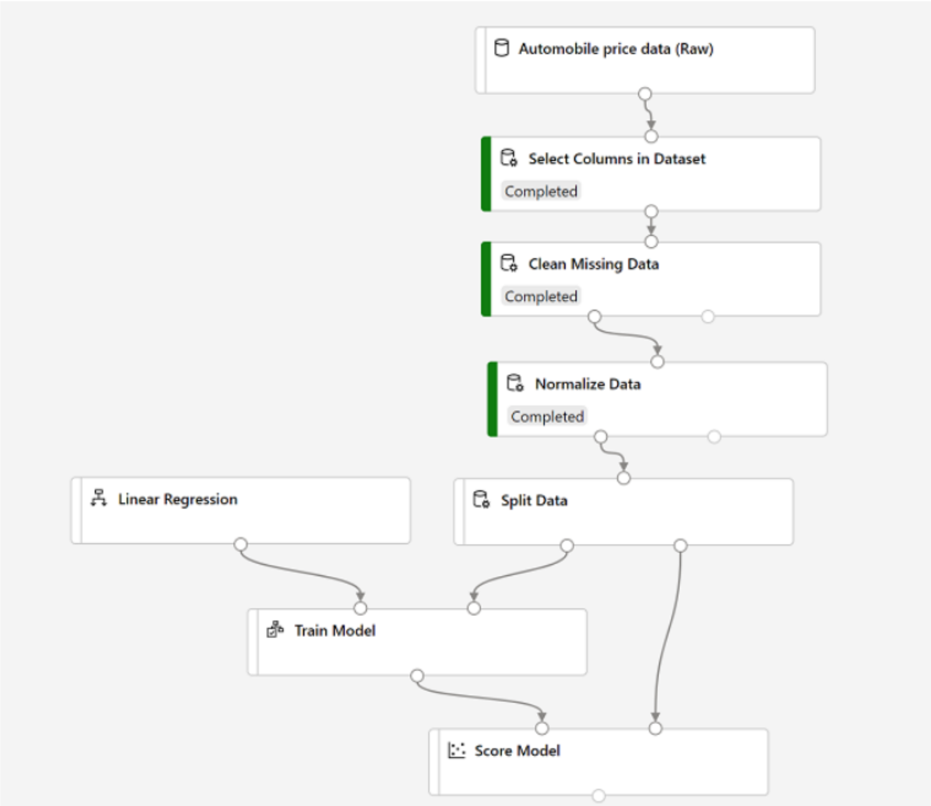
In Azure Machine Learning studio, view the **Endpoints** page and select the **predict-rentals** real-time endpoint. Then select the **Consume** tab and note the following information there. You need this information to connect to your deployed service from a client application.

- The REST endpoint for your service
- the Primary Key for your service

If you don't intend to experiment with it further, you should delete the endpoint to avoid accruing unnecessary Azure usage. You should also stop the training cluster and compute instance resources until you need them again.

From <https://docs.microsoft.com/en-us/learn/modules/use-automated-machine-learning/summary>

7. To test the trained model, we need to use it to *score* the validation dataset we held back when we split the original data - in other words, predict labels for the features in the validation dataset. Expand the **Model Scoring & Evaluation** section and drag a **Score Model** module to the canvas, below the **Train Model** module. Then connect the output of the **Train Model** module to the **Trained model** (left) input of the **Score Model** module; and drag the **Results dataset2** (right) output of the **Split Data** module to the **Dataset** (right) input of the **Score Model** module.
8. Ensure your pipeline looks like this:



Create an inference pipeline

- 8 minutes

After creating and running a pipeline to train the model, you need a second pipeline that performs the same data transformations for new data, and then uses the trained model to *inference* (in other words, predict) label values based on its features. This will form the basis for a predictive service that you can publish for applications to use.

Create and run an inference pipeline

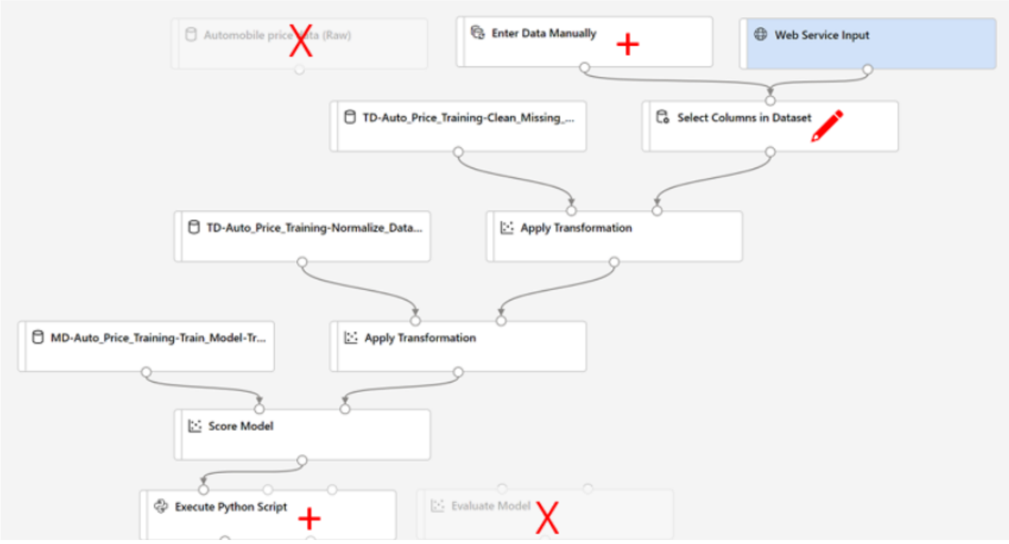
1. In Azure Machine Learning Studio, click the **Designer** page to view all of the pipelines you have created. Then open the **Auto Price Training** pipeline you created previously.

2. In the **Create inference pipeline** drop-down list, click **Real-time inference pipeline**. After a few seconds, a new version of your pipeline named **Auto Price Training-real time inference** will be opened.

   *If the pipeline does not include **Web Service Input** and **Web Service Output** modules, go back to the **Designer** page and then re-open the **Auto Price Training-real time inference** pipeline.*

3. Rename the new pipeline to **Predict Auto Price**, and then review the new pipeline. It contains a web service input for new data to be submitted, and a web service output to return results. Some of the transformations and training steps have been encapsulated in this pipeline so that the statistics from your training data will be used to normalize any new data values, and the trained model will be used to score the new data.

   You are going to make the following changes to the inference pipeline:


Automobile price data (Raw)
X
Enter Data Manually
+
Web Service Input
TD-Auto_Price_Training-Clean_Missing_...
Select Columns in Dataset
TD-Auto_Price_Training-Normalize_Data...
Apply Transformation
MD-Auto_Price_Training-Train_Model-Tr...
Apply Transformation
Score Model
Execute Python Script
+
Evaluate Model
X

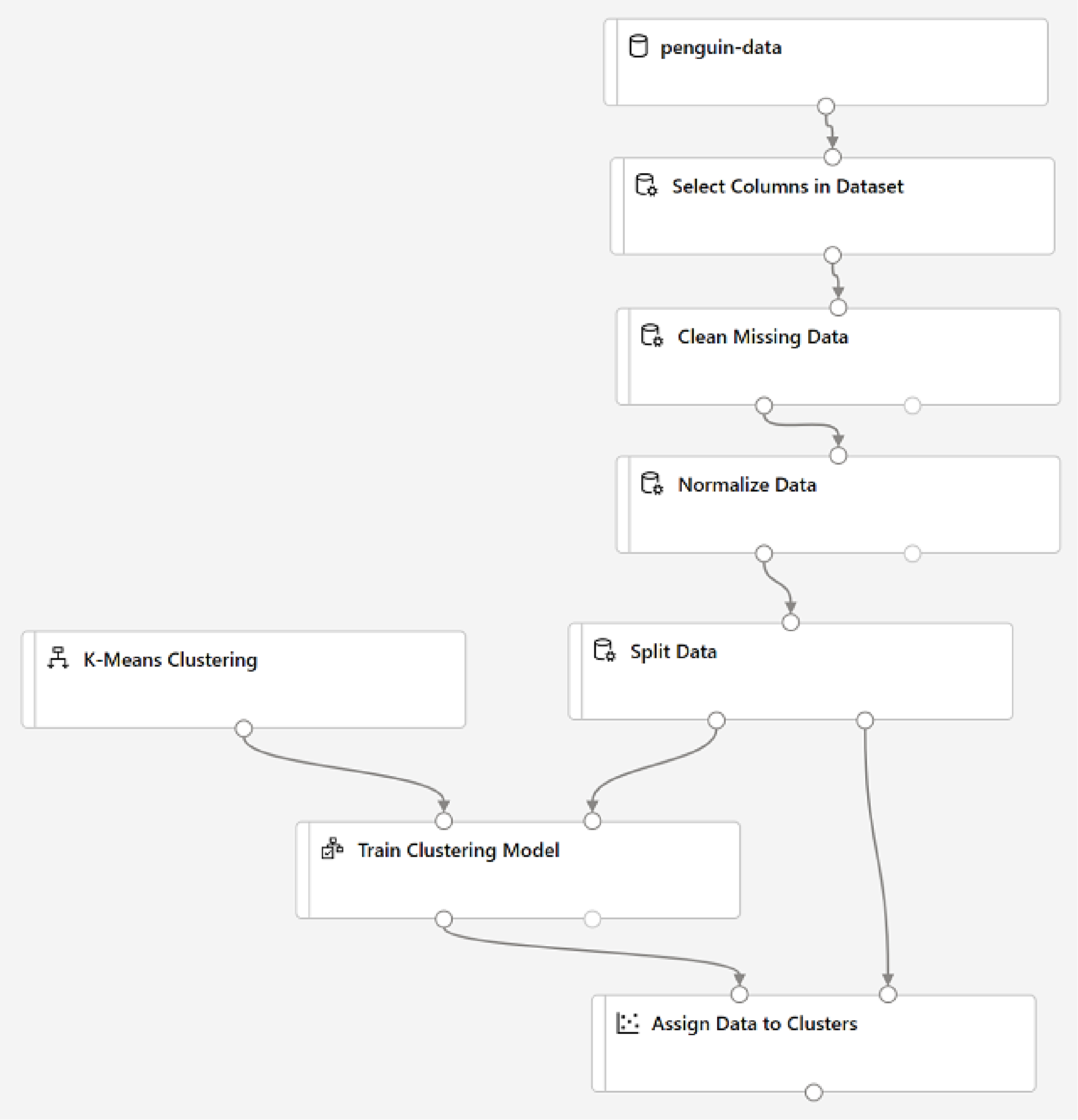
The inference pipeline includes the **Evaluate Model** module, which is not useful when predicting from new data, so delete this module.

Connect the new **Enter Data Manually** module to the same **dataset** input of the **Select Columns in Dataset** module as the **Web Service Input**.

The inference pipeline includes the **Evaluate Model** module, which is not useful when predicting from new data, so delete this module.

From <https://docs.microsoft.com/en-us/learn/modules/create-regression-model-azure-machine-learning-designer/inference-pipeline>

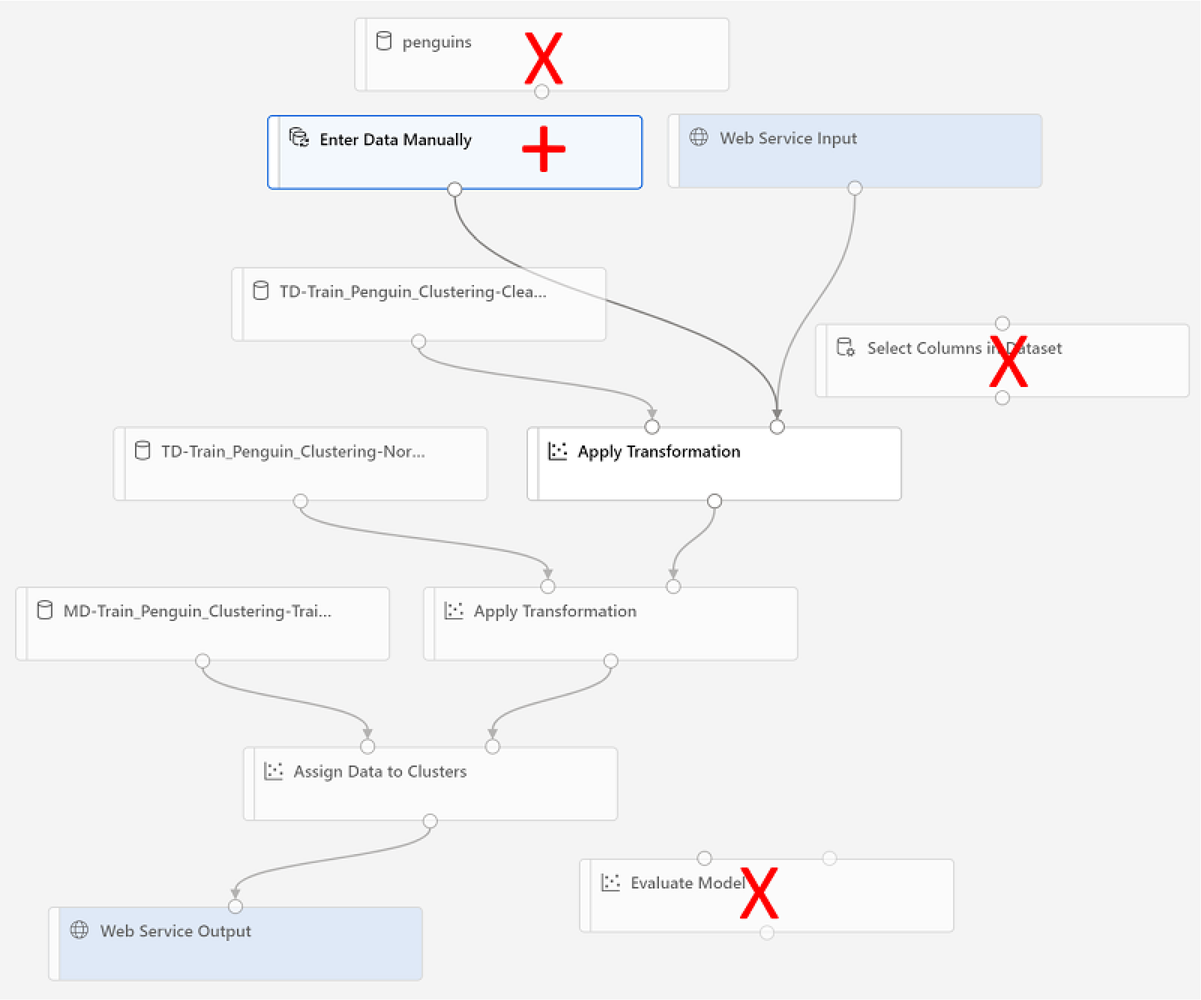
GO THROUGH THIS LINK ONCE TILL END



, select the **Evaluate Model** module and in the settings pane, on the **Outputs + Logs** tab, under **Data outputs** in the **Evaluation results** section, use the **Visualize** icon to view the performance metrics. These metrics can help data scientists assess how well the model separates the clusters. They include a row of metrics for each cluster, and a summary row for a combined evaluation. The metrics in each row are:

- **Average Distance to Other Center**: This indicates how close, on average, each point in the cluster is to the centroids of all other clusters.
- **Average Distance to Cluster Center**: This indicates how close, on average, each point in the cluster is to the centroid of the cluster.
- **Number of Points**: The number of points assigned to the cluster.
- **Maximal Distance to Cluster Center**: The maximum of the distances between each point and the centroid of that point's cluster. If this number is high, the cluster may be widely dispersed.

This statistic in combination with the **Average Distance to Cluster Center** helps you determine the cluster's *spread*.



Can read Cnn from this

a data analysis project is designed to establish insights around a particular scenario or to test a hypothesis.

They might use the data to test a hypothesis that only students who study for a minimum number of hours can expect to achieve a passing grade; or even prepare the data to train a machine learning model that predicts a student's grade based on their study habits.

Data exploration and analysis is typically an *iterative* process, in which the data scientist takes a sample of data, and performs the following kinds of task to analyze it and test hypotheses:

- understand the data, and how the sample might be expected to represent the real-world population of data, allowing for random variation.

- Visualize data to determine relationships between variables, and in the case of a machine learning project, identify *features* that are potentially predictive of the *label*.
- Derive new features from existing ones that might better encapsulate relationships within the data
-
- A *training* dataset to which we'll apply an algorithm that determines a function encapsulating the relationship between the feature values and the known label values.
- A *validation* or *test* dataset that we can use to evaluate the model by using it to generate predictions for the label and comparing them to the actual known label values.

Machine learning is based in statistics and math, and it's important to be aware of specific terms that statisticians and mathematicians (and therefore data scientists) use. You can think of the difference between a *predicted* label value and the *actual* label value as a measure of ***error***. However, in practice, the "actual" values are based on sample observations (which themselves may be subject to some random variance). To make it clear that we're comparing a *predicted* value ($\hat{y}$) with an *observed* value (y) we refer to the difference between them as the ***residuals***. We can summarize the residuals for all of the validation data predictions to calculate the overall ***loss*** in the model as a measure of its predictive performance.

R^2 (R-Squared) (sometimes known as *coefficient of determination*) is the correlation between ***x*** and ***y*** squared. This produces a value between 0 and 1 that measures the amount of variance that can be explained by the model. Generally, the closer this value is to 1, the better the model predicts.

A classification algorithm is used to fit a subset of the data to a function that can calculate the probability for each class label from the feature values. The remaining data is used to evaluate the model by comparing the predictions it generates from the features to the known class labels

The neuron itself encapsulates a function that calculates a weighted sum of ***x***, ***w***, and ***b***. This function is in turn enclosed in an *activation function* that constrains the result (often to a value between 0 and 1) to determine whether or not the neuron passes an output onto the next layer of neurons in the network.

Processing the training features as a batch improves the efficiency of the training process by processing multiple observations simultaneously as a matrix of features with vectors of weights and biases. Linear algebraic functions that operate wth matrices and vectors also feature in 3D graphics processing, which is why computers with graphic processing units (GPUs) provide significantly better performance for deep learning model training than central processing unit (CPU) only computers.

. The loss is calculated using a function, which operates on the results from the final layer of the network, which is also a function. The final layer of network operates on the outputs from the previous layers, which are also functions. So in effect, the entire model from the input layer right through to the loss calculation is just one big nested function. Functions have a few really useful characteristics, including:

- You can conceptualize a function as a plotted line comparing its output with each of its variables.

- You can use differential calculus to calculate the *derivative* of the function at any point with respect to its variables.

optimizer -tells wic direction to move
We use an *optimizer* to apply this same trick for all of the weight and bias variables in the model and determine in which direction we need to adjust them (up or down) to reduce the overall amount of loss in the model.

Learning rate

by how much should the optimizer adjust the weights and bias values

The output of the convolution is typically passed to an activation function, which is often a *Rectified Linear Unit* (ReLU) function that ensures negative values are set to 0:
The resulting matrix is a *feature map* of feature values that can be used to train a machine learning model.

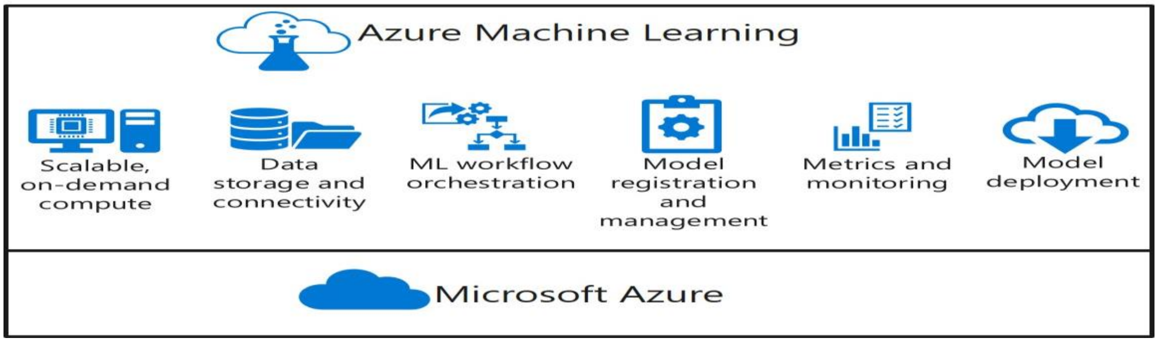
> **Note**: The values in the feature map can be greater than the maximum value for a pixel (255), so if you wanted to visualize the feature map as an image you would need to *normalize* the feature values between 0 and 255.

One of the most difficult challenges in a CNN is the avoidance of *overfitting*,
One technique you can use to mitigate overfitting is to include layers in which the training process randomly eliminates (or "drops") feature maps.
Other techniques you can use to mitigate overfitting include randomly flipping, mirroring, or skewing the training images to generate data that varies between training epochs.

Conceptually, this neural network consists of two distinct sets of layers:

- A set of layers from the base model that perform *feature extraction*.
- A fully connected layer that takes the extracted features and uses them for class *prediction*.

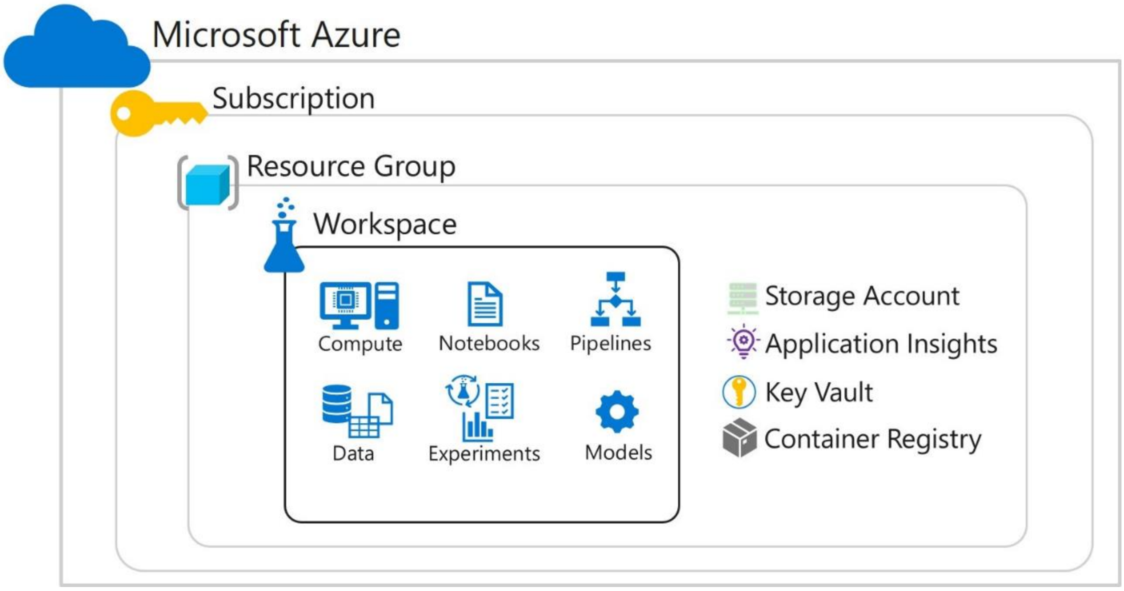
By separating the network into these types of layers, we can take the feature extraction layers from a model that has already been trained and append one or more layers to use the extracted features for prediction of the appropriate class labels for your images. This approach enables you to keep the pre-trained weights for the feature extraction layers, which means you only need to train the prediction layers you have added.


Azure Machine Learning
Scalable, on-demand compute
Data storage and connectivity
ML workflow orchestration
Model registration and management
Metrics and monitoring
Model deployment
Microsoft Azure

You can use workspaces to group machine learning assets based on projects, deployment environments

A workspace is a context for the experiments, data, compute targets, and other assets associated with a machine learning workload.

The Azure resources created alongside a workspace



he assets in a workspace include:

- Compute targets for development, training, and deployment.
- Data for experimentation and model training.
- Notebooks containing shared code and documentation.
- Experiments, including run history with logged metrics and outputs.
- Pipelines that define orchestrated multi-step processes.
- Models that you have trained.

m.e.n.d.c

Role-Based Access Control

You can assign role-based authorization policies to a workspace, enabling you to manage permissions that restrict what actions specific Azure Active Directory (AAD) principals can perform.

```
from azureml.core import Workspace

   ws = Workspace.create(name='aml-workspace',
                subscription_id='123456-abc-123...',
                resource_group='aml-resources',
                create_resource_group=True,
                location='eastus'
               )
```

- you could use the following command (which assumes a resource group named *aml-resources* has already been created):

```
    az ml workspace create -w 'aml-workspace' -g 'aml-resources'
az ml computetarget list -g 'aml-resources' -w 'aml-workspace'
```

-g parameter specifies the name of the resource group in which the Azure Machine Learning workspace specified in the **-w** parameter is defined. These parameters are shortened aliases for **--resource-group** and **--workspace-name**.

```
pip install azureml-sdk[notebooks,automl,explain]
```

Azure Machine Learning studio is a web-based tool for managing an Azure Machine Learning workspace. It enables you to create, manage, and view all of the assets in your workspace and provides the following graphical tools:

- *Designer*: A drag and drop interface for "no code" machine learning model development.
- *Automated Machine Learning*: A wizard interface that enables you to train a model using a combination of algorithms and data preprocessing techniques to find the best model for your data.

. The easiest way to connect to a workspace is to use a workspace configuration file, which includes the Azure subscription, resource group, and workspace details as shown here:

JSON

```
{
    "subscription_id": "1234567-abcde-890-fgh...",
    "resource_group": "aml-resources",
    "workspace_name": "aml-workspace"
}
```

o connect to the workspace using the configuration file, you can use the **from_config** method of the **Workspace** class in the SDK, as shown here:

Python

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

By default, the **from_config** method looks for a file named **config.json** in the folder containing the Python code file, but you can specify another path if necessary.

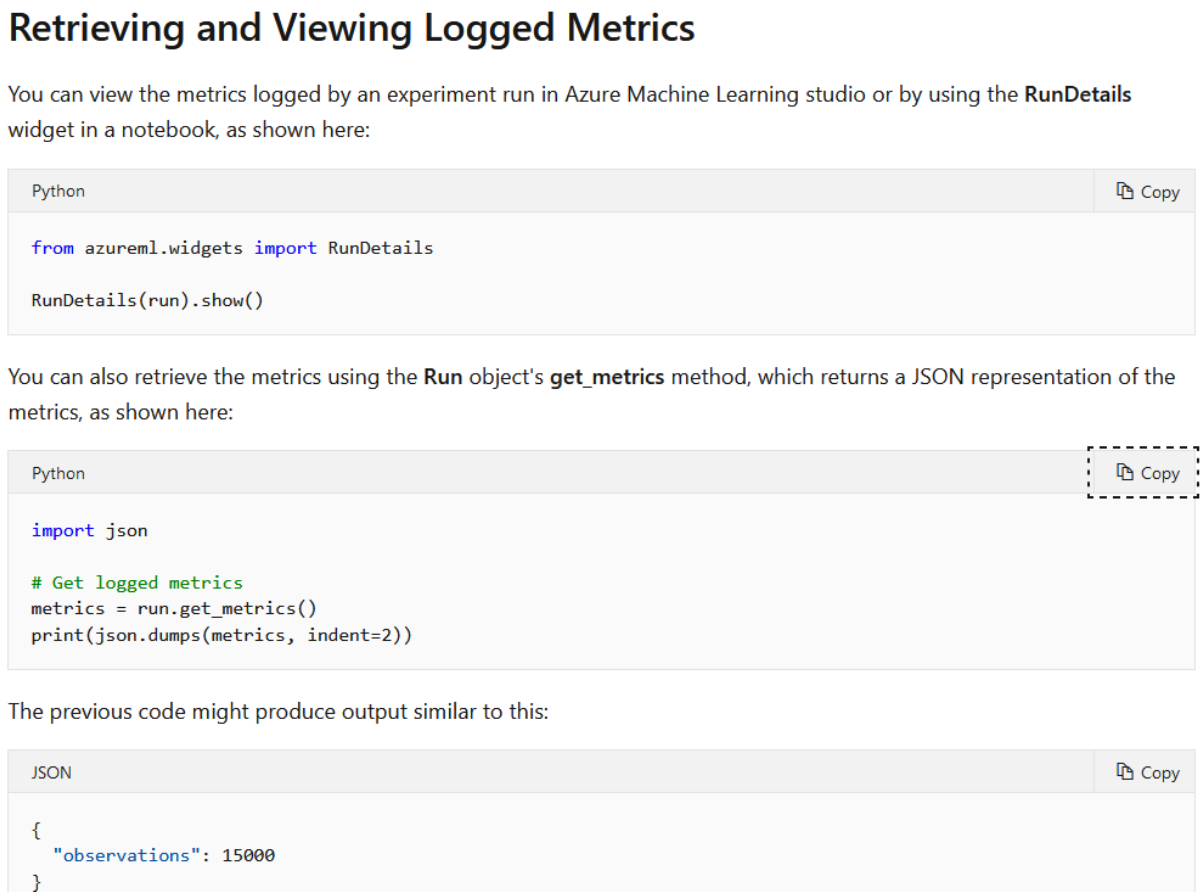
```
from azureml.core import Workspace
ws = Workspace.get(name='aml-workspace',
              subscription_id='1234567-abcde-890-fgh...',
              resource_group='aml-resources')
```

an experiment is a named process, usually the running of a script or a pipeline, that can generate metrics and outputs and be tracked in the Azure Machine Learning workspace.

When you submit an experiment, you use its *run context* to initialize and end the experiment run that is tracked in Azure Machine Learning,

```
from azureml.core import Experiment
# create an experiment variable
experiment = Experiment(workspace = ws, name = "my-experiment")
# start the experiment
run = experiment.start_logging()
# experiment code goes here
# end the experiment
run.complete()
```

```
# load the dataset and count the rows
data = pd.read_csv('data.csv')
row_count = (len(data))
# Log the row count
run.log('observations', row_count)
# Complete the experiment
run.complete()
```



Retrieving and Viewing Logged Metrics

You can view the metrics logged by an experiment run in Azure Machine Learning studio or by using the **RunDetails** widget in a notebook, as shown here:

```
from azureml.widgets import RunDetails

RunDetails(run).show()
```

You can also retrieve the metrics using the **Run** object's **get_metrics** method, which returns a JSON representation of the metrics, as shown here:

```
import json

# Get logged metrics
metrics = run.get_metrics()
print(json.dumps(metrics, indent=2))
```

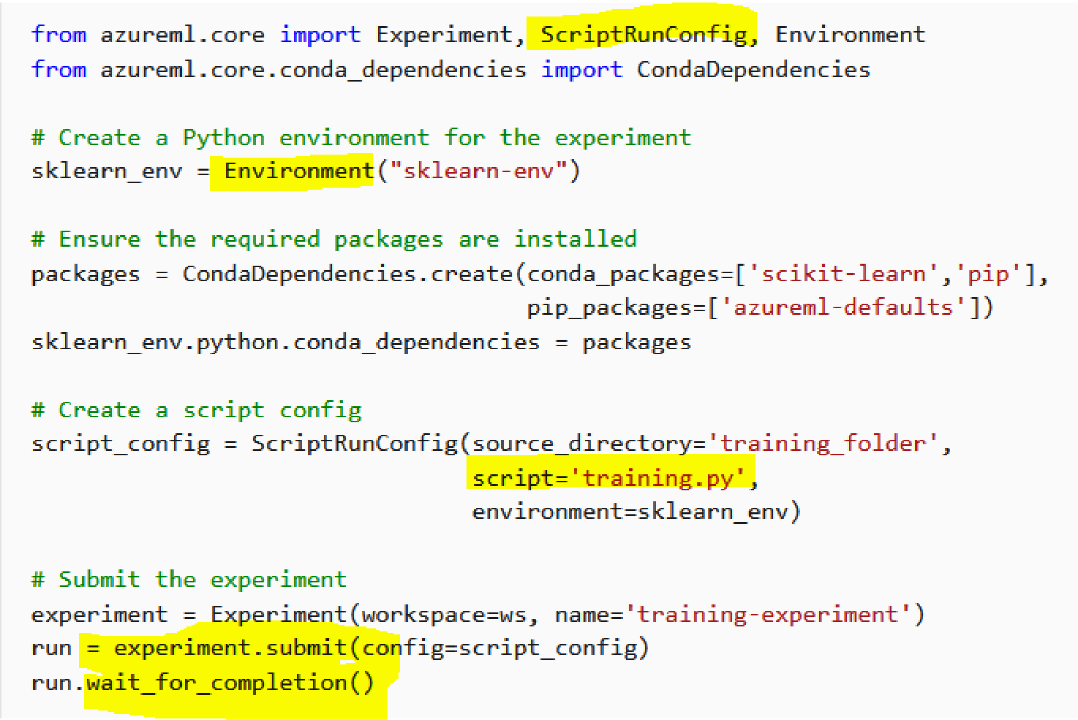
The previous code might produce output similar to this:

```
{
  "observations": 15000
}
```

run.upload_file(name='outputs/sample.csv', path_or_stream='./sample.csv')

Read This page
https://docs.microsoft.com/en-us/learn/modules/intro-to-azure-machine-learning-service/4-azure-ml-experiment

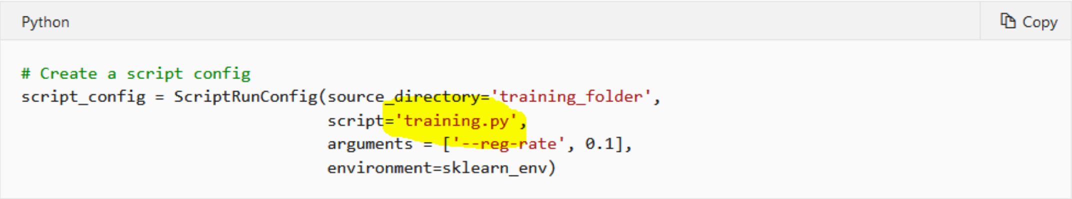
# Get the experiment run context
run = Run.get_context()
# Prepare the dataset
# Train a logistic regression model
# calculate accuracy
run.log('Accuracy', np.float(acc))
# Save the trained model
run.complete()



```
from azureml.core import Experiment, ScriptRunConfig, Environment
from azureml.core.conda_dependencies import CondaDependencies

# Create a Python environment for the experiment
sklearn_env = Environment("sklearn-env")

# Ensure the required packages are installed
packages = CondaDependencies.create(conda_packages=['scikit-learn','pip'],
                                    pip_packages=['azureml-defaults'])
sklearn_env.python.conda_dependencies = packages

# Create a script config
script_config = ScriptRunConfig(source_directory='training_folder',
                                script='training.py',
                                environment=sklearn_env)

# Submit the experiment
experiment = Experiment(workspace=ws, name='training-experiment')
run = experiment.submit(config=script_config)
run.wait_for_completion()
```

Passing arguments to an experiment script

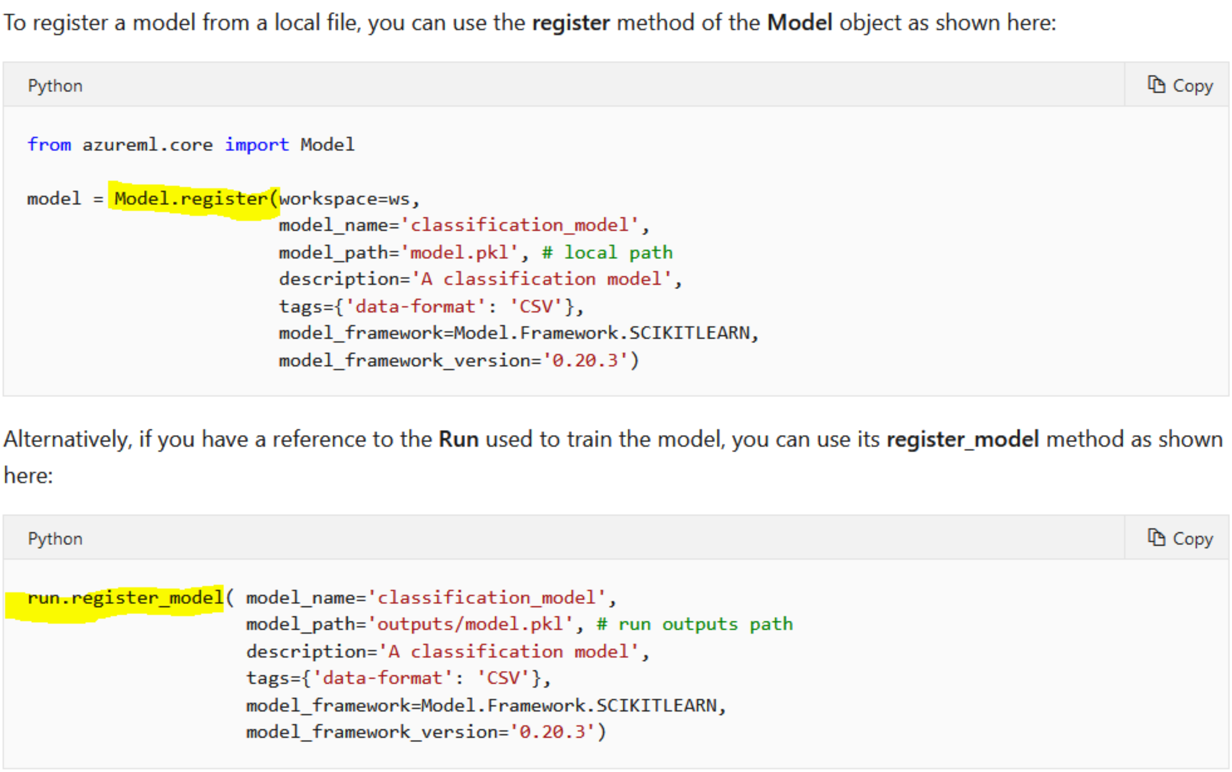
To pass parameter values to a script being run in an experiment, you need to provide an **arguments** value containing a list of comma-separated arguments and their values to the **ScriptRunConfig**, like this:



```
# Create a script config
script_config = ScriptRunConfig(source_directory='training_folder',
                                script='training.py',
                                arguments = ['--reg-rate', 0.1],
                                environment=sklearn_env)
```

# Download a named model file
run.download_file(name='outputs/model.pkl', output_file_path='model.pkl')

Model registration enables you to track multiple versions of a model, and retrieve models for *inferencing* (predicting label values from new data). When you register a model, you can specify a name, description, tags, framework (such as Scikit-Learn or PyTorch), framework version, custom properties, and other useful metadata.



To register a model from a local file, you can use the **register** method of the **Model** object as shown here:

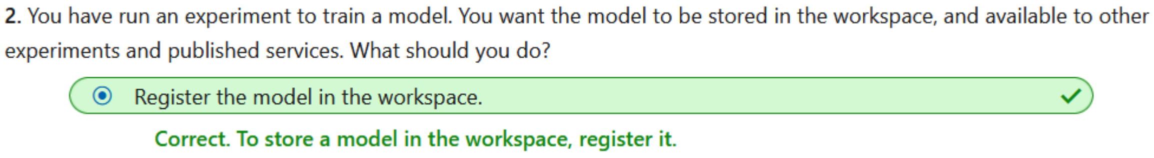
```python
from azureml.core import Model

model = Model.register(workspace=ws,
                       model_name='classification_model',
                       model_path='model.pkl', # local path
                       description='A classification model',
                       tags={'data-format': 'CSV'},
                       model_framework=Model.Framework.SCIKITLEARN,
                       model_framework_version='0.20.3')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown here:

```python
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model',
                    tags={'data-format': 'CSV'},
                    model_framework=Model.Framework.SCIKITLEARN,
                    model_framework_version='0.20.3')
```

for model in Model.list(ws):
    # Get model name and auto-generated version
    print(model.name, 'version:', model.version)

From <https://docs.microsoft.com/en-us/learn/modules/train-local-model-with-azure-mls/4-register-model>



- **2.** You have run an experiment to train a model. You want the model to be stored in the workspace, and available to other experiments and published services. What should you do?

  ● Register the model in the workspace. ✓

  Correct. To store a model in the workspace, register it.

3rd

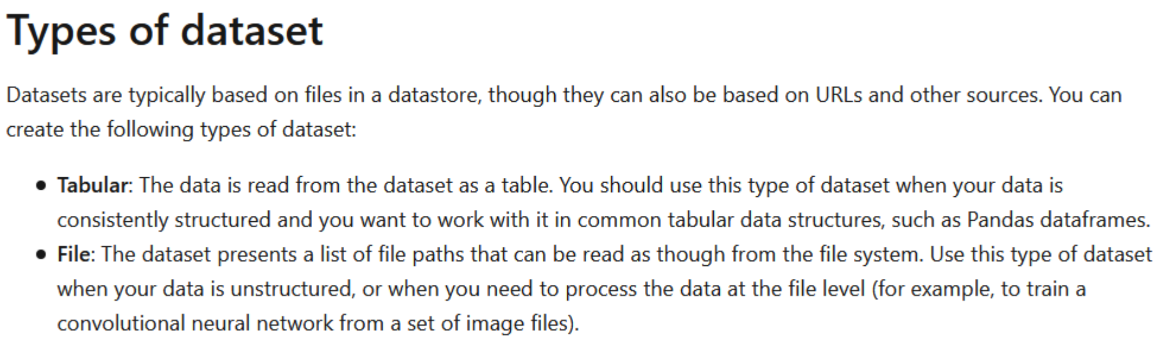
From <https://docs.microsoft.com/en-us/learn/modules/work-with-data-in-aml/1-introduction>

- Create and use datastores
- Create and use datasets

Types of Datastore

Azure Machine Learning supports the creation of datastores for multiple kinds of Azure data source, including:

- Azure Storage (blob and file containers)
- Azure Data Lake stores
- Azure SQL Database
- Azure Databricks file system (DBFS)

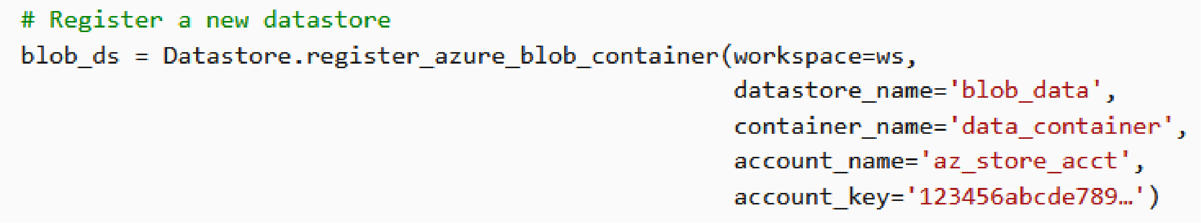


Types of dataset

Datasets are typically based on files in a datastore, though they can also be based on URLs and other sources. You can create the following types of dataset:

- **Tabular**: The data is read from the dataset as a table. You should use this type of dataset when your data is consistently structured and you want to work with it in common tabular data structures, such as Pandas dataframes.
- **File**: The dataset presents a list of file paths that can be read as though from the file system. Use this type of dataset when your data is unstructured, or when you need to process the data at the file level (for example, to train a convolutional neural network from a set of image files).

every workspace has two built-in datastores (an Azure Storage blob container, and an Azure Storage file container) that are used as system storage by Azure Machine Learning. There's also a third datastore that gets added to your workspace if you make use of the open datasets provided as samples



```
# Register a new datastore
blob_ds = Datastore.register_azure_blob_container(workspace=ws,
                                                  datastore_name='blob_data',
                                                  container_name='data_container',
                                                  account_name='az_store_acct',
                                                  account_key='123456abcde789…')
```

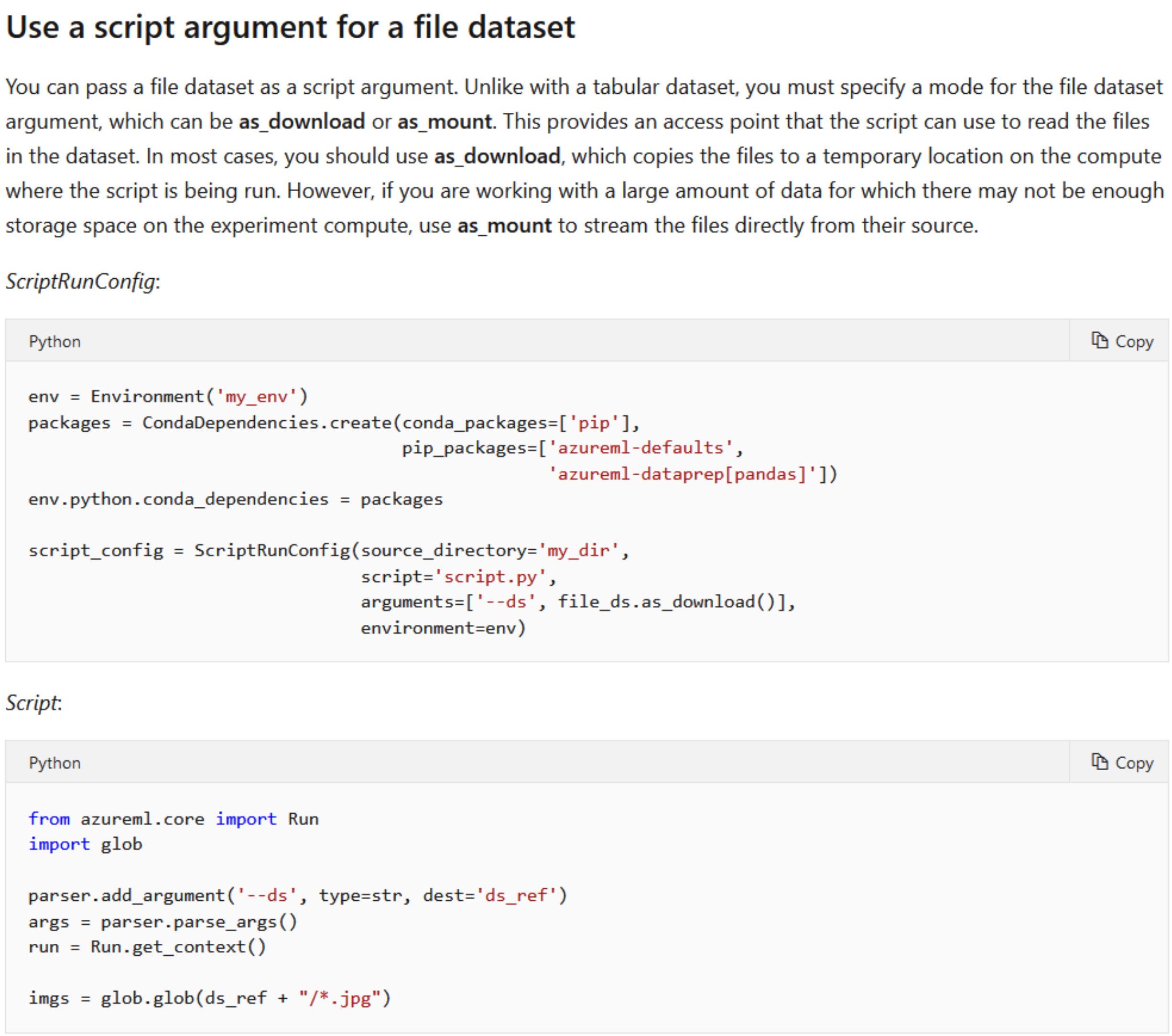
You can get a reference to any datastore by using the **Datastore.get()** method as shown here:
Python
blob_store = **Datastore.get**(ws, datastore_name='blob_data')

To change the default datastore, use the **set_default_datastore()** method:
ws.set_default_datastore('blob_data')
default_store = ws.get_default_datastore()

```
from azureml.core import Dataset
blob_ds = ws.get_default_datastore()
tab_ds = Dataset.Tabular.from_delimited_files(path=csv_paths)
tab_ds = tab_ds.register(workspace=ws, name='csv_table')

# Load the workspace from the saved config file
ws = Workspace.from_config()
# Get a dataset from the workspace datasets collection
ds1 = ws.datasets['csv_table']
# Get a dataset by name from the datasets class
ds2 = Dataset.get_by_name(ws, 'img_files')
```

Datasets can be *versioned*, enabling you to track historical versions of datasets

```
file_ds = Dataset.File.from_files(path=img_paths)
file_ds = file_ds.register(workspace=ws, name='img_files', create_new_version=True)
img_ds = Dataset.get_by_name(workspace=ws, name='img_files', version=2)

df = tab_ds.to_pandas_dataframe()
```



Use a script argument for a file dataset

You can pass a file dataset as a script argument. Unlike with a tabular dataset, you must specify a mode for the file dataset argument, which can be **as_download** or **as_mount**. This provides an access point that the script can use to read the files in the dataset. In most cases, you should use **as_download**, which copies the files to a temporary location on the compute where the script is being run. However, if you are working with a large amount of data for which there may not be enough storage space on the experiment compute, use **as_mount** to stream the files directly from their source.

ScriptRunConfig:

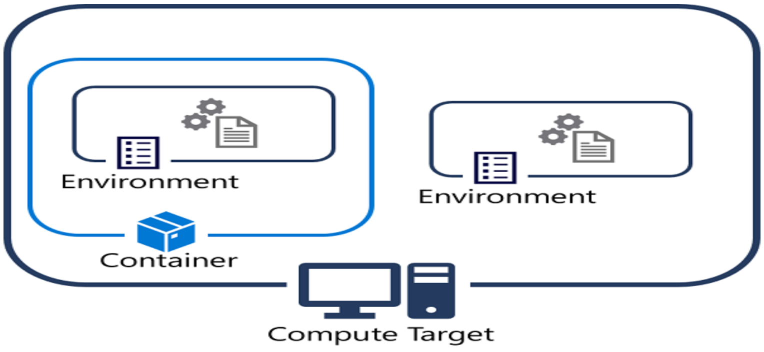
```python
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pandas]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', file_ds.as_download()],
                                environment=env)
```

Script:

```python
from azureml.core import Run
import glob

parser.add_argument('--ds', type=str, dest='ds_ref')
args = parser.parse_args()
run = Run.get_context()

imgs = glob.glob(ds_ref + "/*.jpg")
```

If name input used
 arguments=['--ds', tab_ds.as_named_input('my_dataset')],

When using Azure blob storage, *premium* level storage may provide improved I/O performance for large datasets. However, this option will increase cost and may limit replication options for data redundancy



Creating an environment

Creating an environment from a specification file
Creating an environment from an existing Conda environment
Creating an environment by specifying packages

```
from azureml.core import Environment

env = Environment.from_existing_conda_environment(name='training_environment',
conda_environment_name ='py_env')

env = Environment('training_environment')
env = Environment.from_conda_specification(name='training_environment',
file_path='./conda.yml')
env.python.conda_dependencies = deps


env.docker.enabled = True
deps = CondaDependencies.create(conda_packages=['scikit-learn','pandas','pip'],
pip_packages=['azureml-defaults'])
env.python.conda_dependencies = deps
```

f you have created custom container images and registered them in a container registry, you can override the default base images and use your own.

```
env.docker.base_image='my-base-image'
env.docker.base_image_registry='myregistry.azurecr.io/myimage'
```

Alternatively, you can have an image created on-demand

```
env.docker.base_image = None
env.docker.base_dockerfile = './Dockerfile'
```

your image already includes an installation of Python with the dependencies you need, you can override this behavior by setting

```
env.python.user_managed_dependencies=True
env.python.interpreter_path = '/opt/miniconda/bin/python'
```

Registering and reusing environments

```
env.register(workspace=ws)
```

Retrieving and using an environment

```
training_env = Environment.get(workspace=ws, name='training_environment')
script_config = ScriptRunConfig(source_directory='my_dir',  script='script.py', environment=
training_env)
```

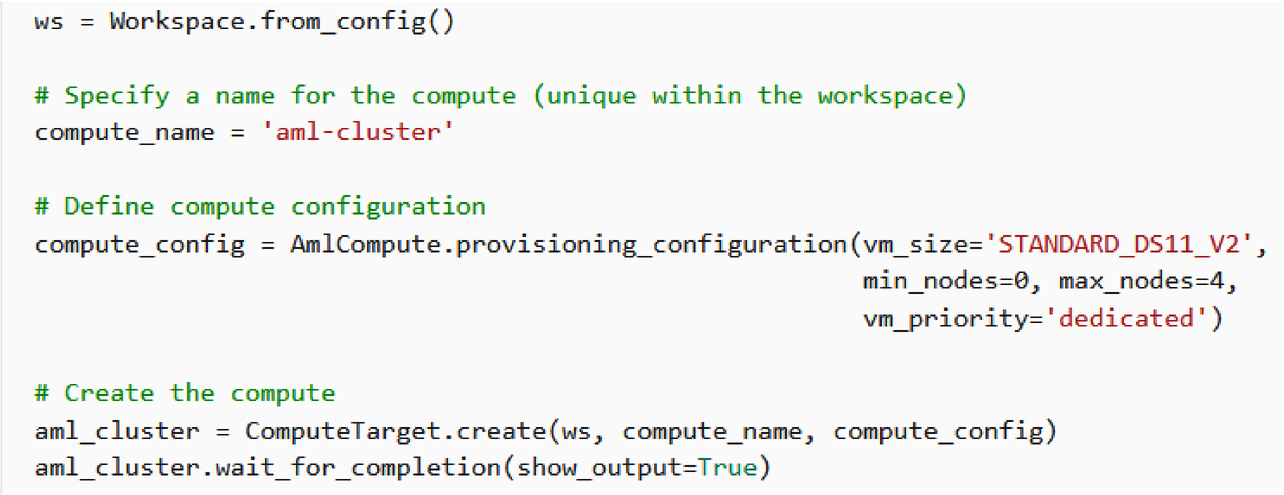
Types of compute

Azure Machine Learning supports multiple types of compute for experimentation and training. This enables you to select the most appropriate type of compute target for your particular needs.

- **Local compute** - You can specify a local compute target for most processing tasks in Azure Machine Learning. This runs the experiment on the same compute target as the code used to initiate the experiment
- **Compute clusters** - For experiment workloads with high scalability requirements, you can use Azure Machine Learning compute clusters; which are multi-node clusters of Virtual Machines that automatically scale up or down to meet demand.
- **Attached compute** - If you already use an Azure-based compute environment for data science, such as a virtual machine or an Azure Databricks cluster, you can attach it to your Azure Machine Learning workspace and use it as a compute target for certain types of workload.
- you can create another type of compute named ***inference clusters*.** This kind of compute represents an Azure Kubernetes Service cluster and can only be used to deploy trained models as inferencing services.

Creating a managed compute target with the SDK

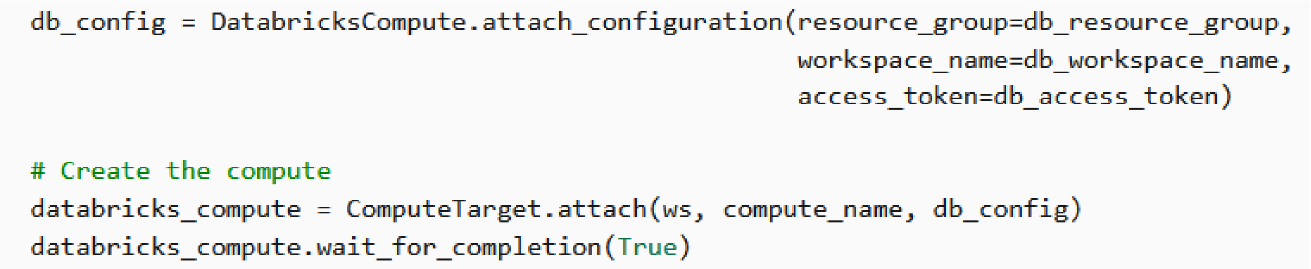
A *managed* compute target is one that is managed by Azure Machine Learning, such as an Azure Machine Learning compute cluster.



```
ws = Workspace.from_config()

# Specify a name for the compute (unique within the workspace)
compute_name = 'aml-cluster'

# Define compute configuration
compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_DS11_V2',
                                                       min_nodes=0, max_nodes=4,
                                                       vm_priority='dedicated')

# Create the compute
aml_cluster = ComputeTarget.create(ws, compute_name, compute_config)
aml_cluster.wait_for_completion(show_output=True)
```

Attaching an unmanaged compute target with the SDK

An *unmanaged* compute target is one that is defined and managed outside of the Azure Machine Learning workspace; for example, an Azure virtual machine or an Azure Databricks cluster.



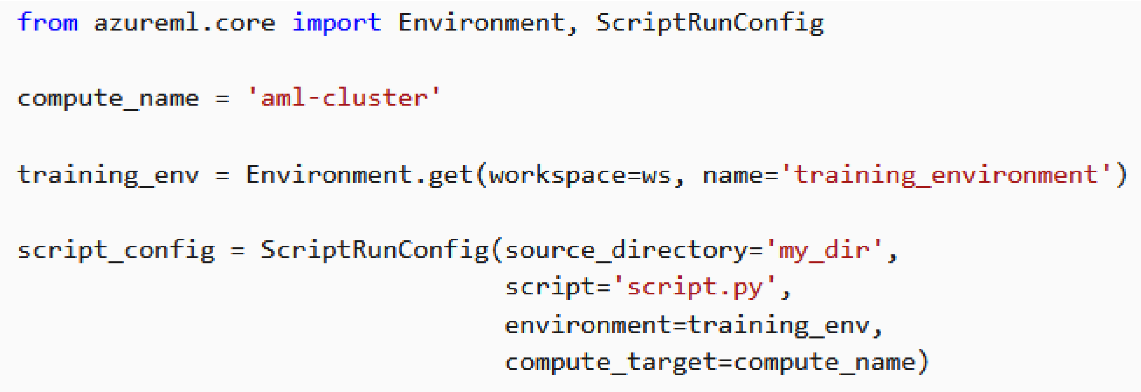
```
db_config = DatabricksCompute.attach_configuration(resource_group=db_resource_group,
                                                   workspace_name=db_workspace_name,
                                                   access_token=db_access_token)

# Create the compute
databricks_compute = ComputeTarget.attach(ws, compute_name, db_config)
databricks_compute.wait_for_completion(True)
```

>

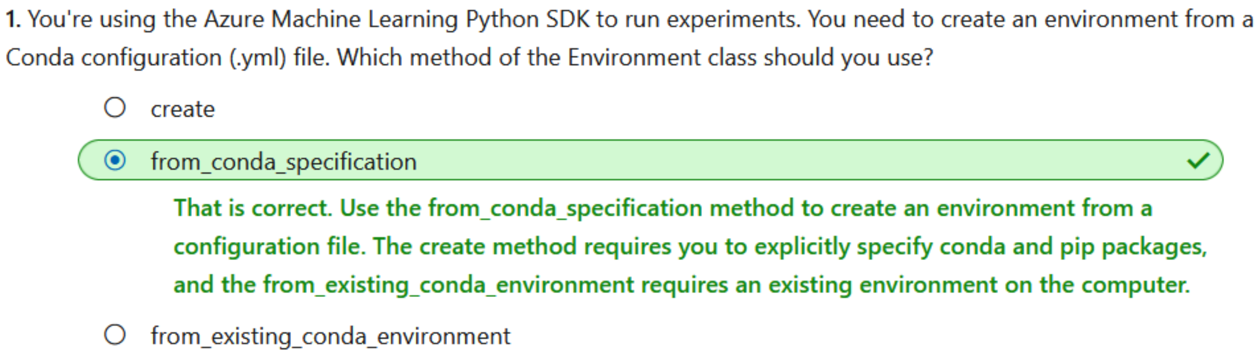
create a new one if there isn't already one with the specified name. To accomplish this, you can catch the **ComputeTargetException** exception

```
compute_name = "aml-cluster"

# Check if the compute target exists
try:
    aml_cluster = ComputeTarget(workspace=ws, name=compute_name)
    print('Found existing cluster.')
except ComputeTargetException:
    # If not, create it
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_DS11_V2',
                                                           max_nodes=4)
    aml_cluster = ComputeTarget.create(ws, compute_name, compute_config)

aml_cluster.wait_for_completion(show_output=True)
```

Use compute targets



```
from azureml.core import Environment, ScriptRunConfig

compute_name = 'aml-cluster'

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=training_env,
                                compute_target=compute_name)
```



1. You're using the Azure Machine Learning Python SDK to run experiments. You need to create an environment from a Conda configuration (.yml) file. Which method of the Environment class should you use?

○ create

● from_conda_specification ✓

That is correct. Use the from_conda_specification method to create an environment from a configuration file. The create method requires you to explicitly specify conda and pip packages, and the from_existing_conda_environment requires an existing environment on the computer.

○ from_existing_conda_environment

A pipeline can be executed as a process by running the pipeline as an experiment. Each step in the pipeline runs on its allocated compute target as part of the overall experiment run.
You can publish a pipeline as a REST endpoint, enabling client applications to initiate a pipeline run

pipeline include:

- **PythonScriptStep**: Runs a specified Python script.
- **DataTransferStep**: Uses Azure Data Factory to copy data between data stores.
- **DatabricksStep**: Runs a notebook, script, or compiled JAR on a databricks cluster.
- **AdlaStep**: Runs a **U-SQL job** in Azure Data Lake Analytics.
- **ParallelRunStep** - Runs a Python script as a **distributed** task on multiple compute nodes.

```
Step1= PythonScriptStep(name ,source_directory , script_name ,compute_target,
arguments ,output)
train_pipeline = Pipeline(workspace = ws, steps = [step1,step2])
# Create an experiment and run the pipeline
experiment = Experiment(workspace = ws, name = 'training-pipeline')
pipeline_run = experiment.submit(train_pipeline)
```

PipelineData step inputs and outputs

To use a **PipelineData** object to pass data between steps, you must:

1. Define a named **PipelineData** object that references a location in a datastore.
2. Pass the **PipelineData** object as a script argument in steps that run scripts (and include code in those scripts to read or write data)
3. Specify the **PipelineData** object as an *input* or *output* for the steps as appropriate.

```
# Get the experiment run context
run = Run.get_context()

# Get input dataset as dataframe
raw_df = run.input_datasets['raw_data'].to_pandas_dataframe()
# code to prep data (in this case, just select specific columns)
prepped_df = raw_df[['col1', 'col2', 'col3']]
```

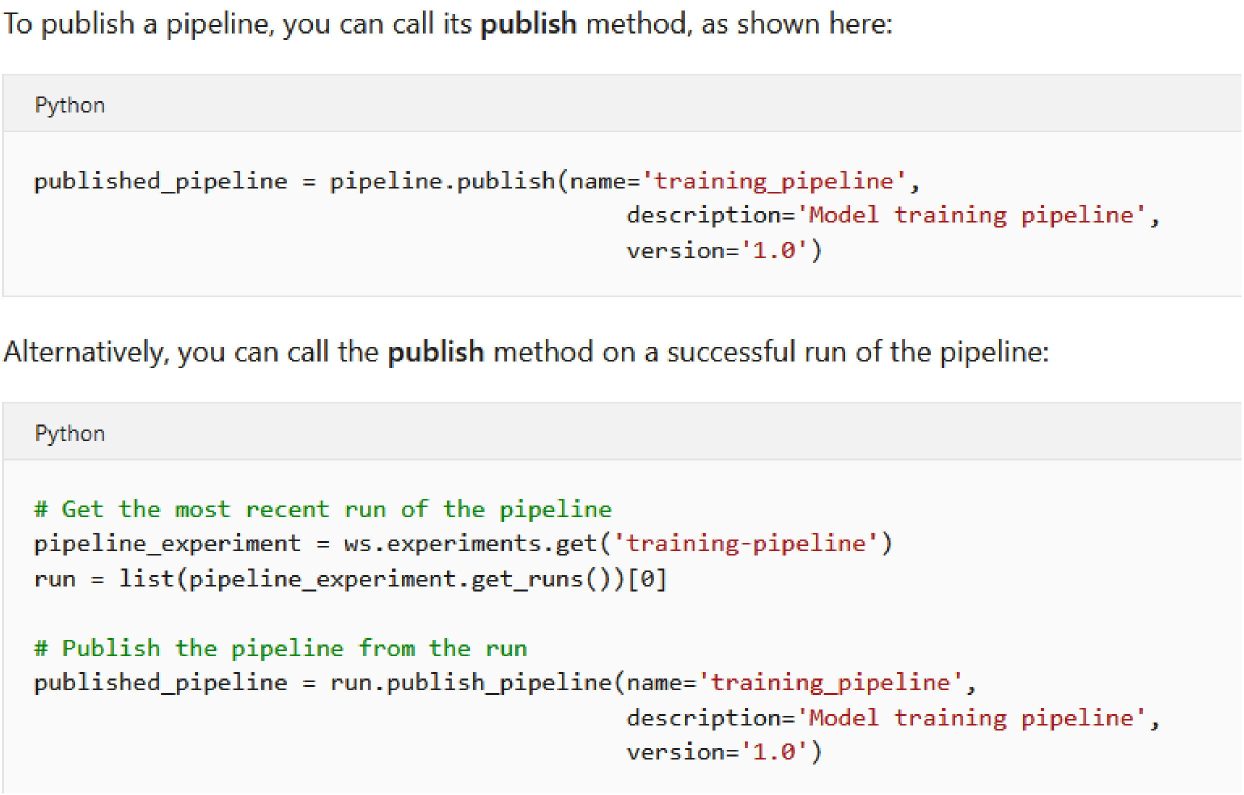
To control reuse for an individual step, you can set the **allow_reuse** parameter in the step configuration,

```
step1 = PythonScriptStep(name = 'prepare data',
                source_directory = 'scripts',
                script_name = 'data_prep.py',
                compute_target = 'aml-cluster',
                runconfig = run_config,
                inputs=[raw_ds.as_named_input('raw_data')],
                outputs=[prepped_data],
                arguments = ['--folder', prepped_data]),
                # Disable step reuse
                allow_reuse = False)
```

When you have multiple steps, you can force all of them to run regardless of individual reuse configuration by setting the **regenerate_outputs** parameter

pipeline_run = experiment.submit(train_pipeline, regenerate_outputs=True)

After you have created a pipeline, you can publish it to create a REST endpoint through which the pipeline



To publish a pipeline, you can call its **publish** method, as shown here:

Python

```
published_pipeline = pipeline.publish(name='training_pipeline',
                                       description='Model training pipeline',
                                       version='1.0')
```

Alternatively, you can call the **publish** method on a successful run of the pipeline:

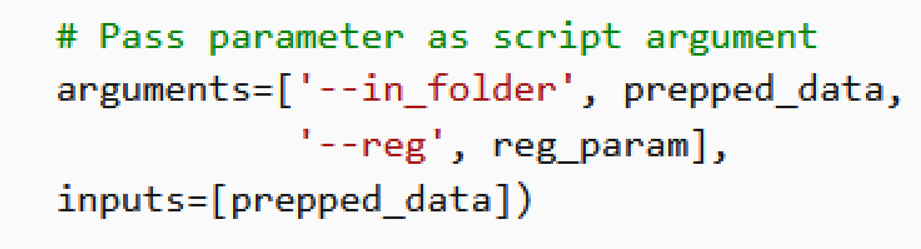
Python

```
# Get the most recent run of the pipeline
pipeline_experiment = ws.experiments.get('training-pipeline')
run = list(pipeline_experiment.get_runs())[0]

# Publish the pipeline from the run
published_pipeline = run.publish_pipeline(name='training_pipeline',
                                          description='Model training pipeline',
                                          version='1.0')
```

To initiate a published endpoint, you make an HTTP request to its REST endpoint, passing an authorization header with a token for a service principal with permission to run the pipeline, and a JSON payload specifying the experiment name. The pipeline is run asynchronously, so the response from a successful REST call includes the run ID. You can use this to track the run in Azure Machine Learning studio.

response = requests.post(rest_endpoint, headers=auth_header,json={"ExperimentName":
"run_training_pipeline"})
run_id = response.json()["Id"]
print(run_id)

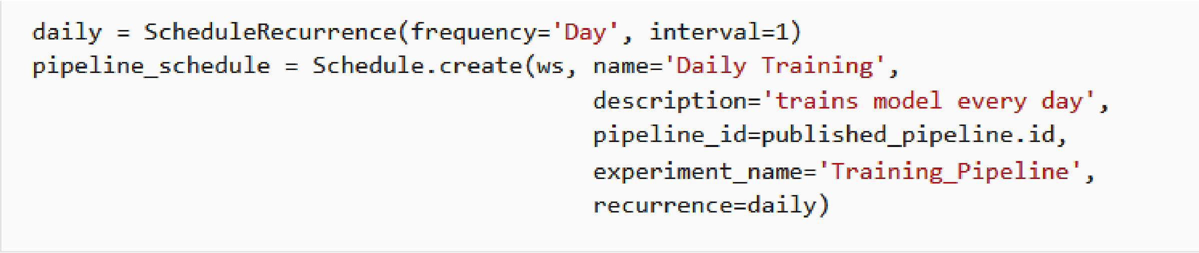
To define parameters for a pipeline, create a **PipelineParameter** object for each parameter
reg_param = PipelineParameter(name='reg_rate', default_value=0.01)



```
# Pass parameter as script argument
arguments=['--in_folder', prepped_data,
           '--reg', reg_param],
inputs=[prepped_data])
```

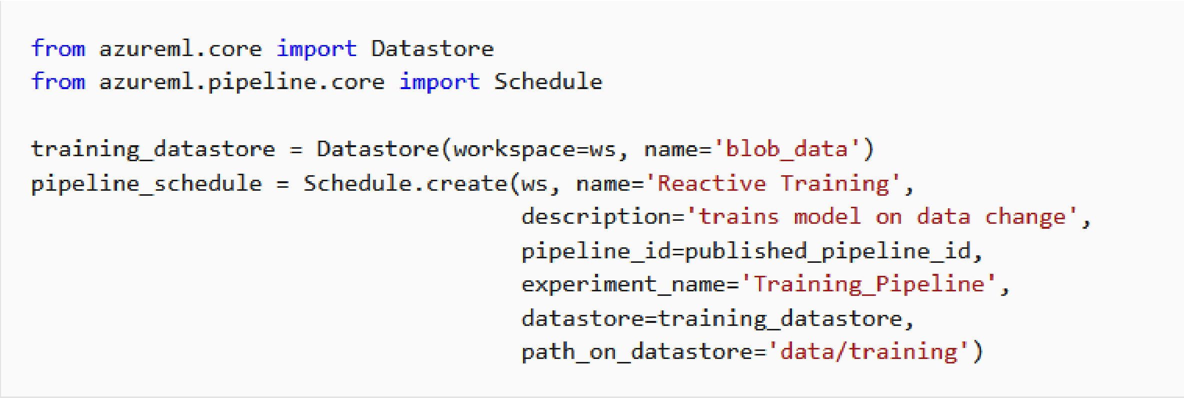
you can pass parameter values in the JSON payload for the REST interface:
response = requests.post(rest_endpoint, headers=auth_header,
json={"ExperimentName": "run_training_pipeline",
"ParameterAssignments": {"reg_rate": 0.1}})

To schedule a pipeline to run at periodic intervals, you must define a **ScheduleRecurrence** that determines the run frequency, and use it to create a **Schedule**.

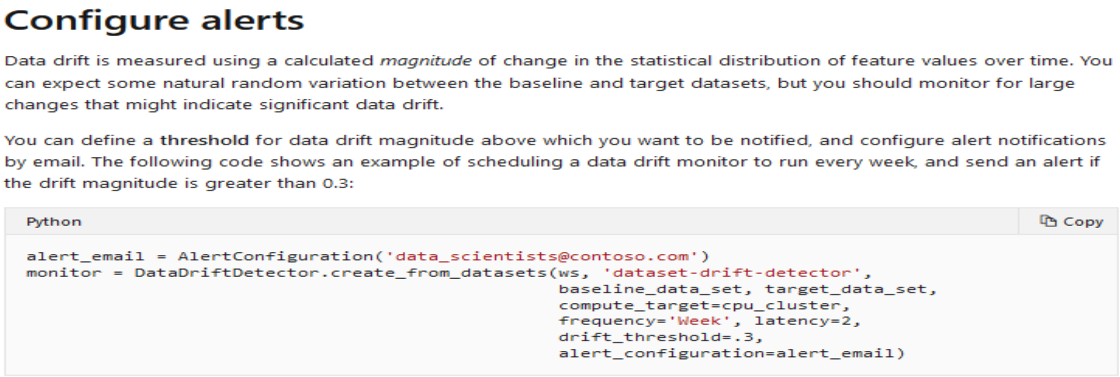


```
daily = ScheduleRecurrence(frequency='Day', interval=1)
pipeline_schedule = Schedule.create(ws, name='Daily Training',
                                        description='trains model every day',
                                        pipeline_id=published_pipeline.id,
                                        experiment_name='Training_Pipeline',
                                        recurrence=daily)
```

To schedule a pipeline to run whenever data changes



```
from azureml.core import Datastore
from azureml.pipeline.core import Schedule

training_datastore = Datastore(workspace=ws, name='blob_data')
pipeline_schedule = Schedule.create(ws, name='Reactive Training',
                                    description='trains model on data change',
                                    pipeline_id=published_pipeline_id,
                                    experiment_name='Training_Pipeline',
                                    datastore=training_datastore,
                                    path_on_datastore='data/training')
```



Configure alerts

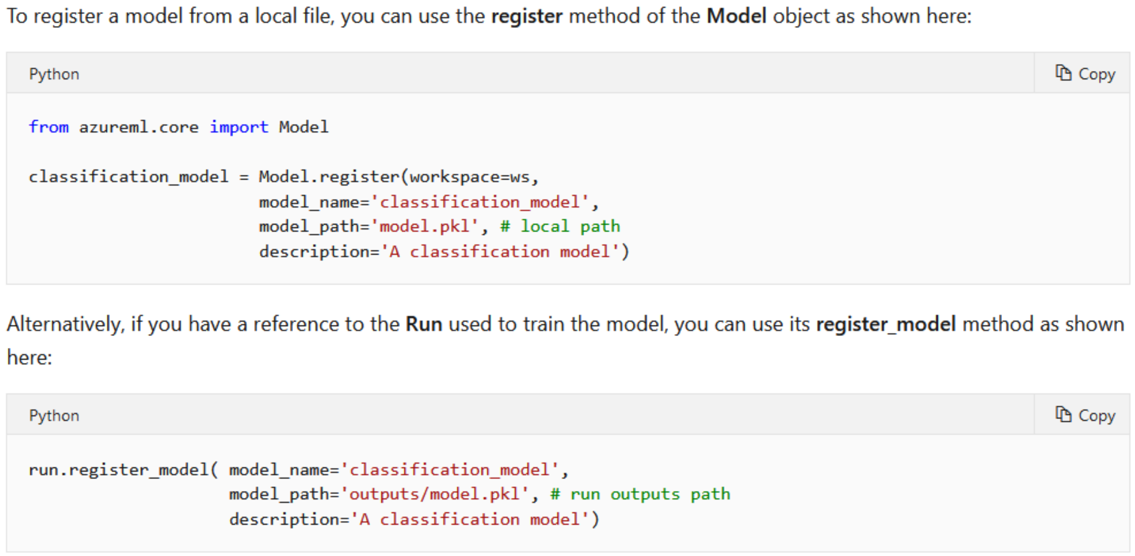
Data drift is measured using a calculated *magnitude* of change in the statistical distribution of feature values over time. You can expect some natural random variation between the baseline and target datasets, but you should monitor for large changes that might indicate significant data drift.

You can define a **threshold** for data drift magnitude above which you want to be notified, and configure alert notifications by email. The following code shows an example of scheduling a data drift monitor to run every week, and send an alert if the drift magnitude is greater than 0.3:

```
alert_email = AlertConfiguration('data_scientists@contoso.com')
monitor = DataDriftDetector.create_from_datasets(ws, 'dataset-drift-detector',
                                                 baseline_data_set, target_data_set,
                                                 compute_target=cpu_cluster,
                                                 frequency='Week', latency=2,
                                                 drift_threshold=.3,
                                                 alert_configuration=alert_email)
```

Deploying a model

Azure Machine Learning uses *containers* as a deployment mechanism, packaging the model



To register a model from a local file, you can use the **register** method of the **Model** object as shown here:

```python
from azureml.core import Model

classification_model = Model.register(workspace=ws,
                       model_name='classification_model',
                       model_path='model.pkl', # local path
                       description='A classification model')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown here:

```python
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model')
```

The model will be deployed as a service that consist of:

- A script to load the model and return predictions for submitted data.
- An environment in which the script will be run.

Combining the script and environment in an InferenceConfig

1.Register a trained model

2. Define an inference configuration

Creating an entry script

Creating an environment

Combining the script and environment in an InferenceConfig

3. Define a deployment configuration

4. Deploy the model

Read this
https://docs.microsoft.com/en-us/learn/modules/register-and-deploy-model-with-amls/2-deploy-model

For testing, you can use the Azure Machine Learning SDK to call a web service through the **run** method of a **WebService** object that references the deployed service. Typically, you send data to the **run** method in JSON format

The response from the **run** method is a JSON collection with a prediction for each case that was submitted in the data.

```
# Call the web service, passing the input data
response = service.run(input_data = json_data)
# Get the predictions
predictions = json.loads(response)
# Print the predicted class for each case.
for i in range(len(x_new)):
   print (x_new[i], predictions[i])
```

You can determine the endpoint of a deployed service in Azure machine Learning studio, or by retrieving the **scoring_uri** property of the **Webservice** object in the SDK

```
endpoint = service.scoring_uri

# Set the content type in the request headers
request_headers = { 'Content-Type':'application/json' }
# Call the service
response = requests.post(url = endpoint, data = json_data,  headers = request_headers)

# Get the predictions from the JSON response
predictions = json.loads(response.json())
```

Authentication

In production, you will likely want to restrict access to your services by applying authentication. There are two kinds of authentication you can use:

- **Key**: Requests are authenticated by specifying the key associated with the service.
- **Token**: Requests are authenticated by providing a JSON Web Token (JWT).

```
primary_key, secondary_key = service.get_keys()

# Set the content type in the request headers
request_headers = { "Content-Type":"application/json",
              "Authorization":"Bearer " + key_or_token }
# Call the service
response = requests.post(url = endpoint, data = json_data,headers = request_headers)
```

Troubleshooting a failed deployment, or an error when consuming a deployed service can be complex.

```
# Get the deployed service
service = AksWebservice(name='classifier-service', workspace=ws)
# Check its state
print(service.state)
print(service.get_logs())#If a service is not healthy, you can review its logs:
```

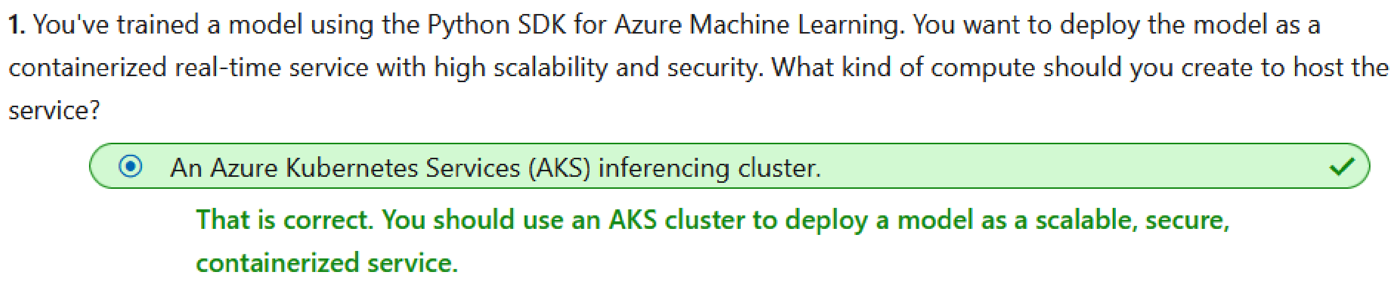
Deployment and runtime errors can be easier to diagnose by deploying the service as a container in a local Docker instance,

```
deployment_config = LocalWebservice.deploy_configuration(port=8890)
service = Model.deploy(ws, 'test-svc', [model], inference_config, deployment_config)
```

service.reload() #reloading the service without redeploying it (something you can only do with a local service):

test the locally deployed service using the SDK:

```
print(service.run(input_data = json_data))
```



1. You've trained a model using the Python SDK for Azure Machine Learning. You want to deploy the model as a containerized real-time service with high scalability and security. What kind of compute should you create to host the service?

● An Azure Kubernetes Services (AKS) inferencing cluster. ✓

That is correct. You should use an AKS cluster to deploy a model as a scalable, secure, containerized service.

To view the **state** of a service, you must use the compue-specific service type (for example **AksWebservice**) and not a generic **WebService** object.

Deploy batch inference pipelines with Azure Machine Learning

In machine learning, *batch inferencing* is used to apply a predictive model to multiple cases asynchronously - usually writing the results to a file or database.

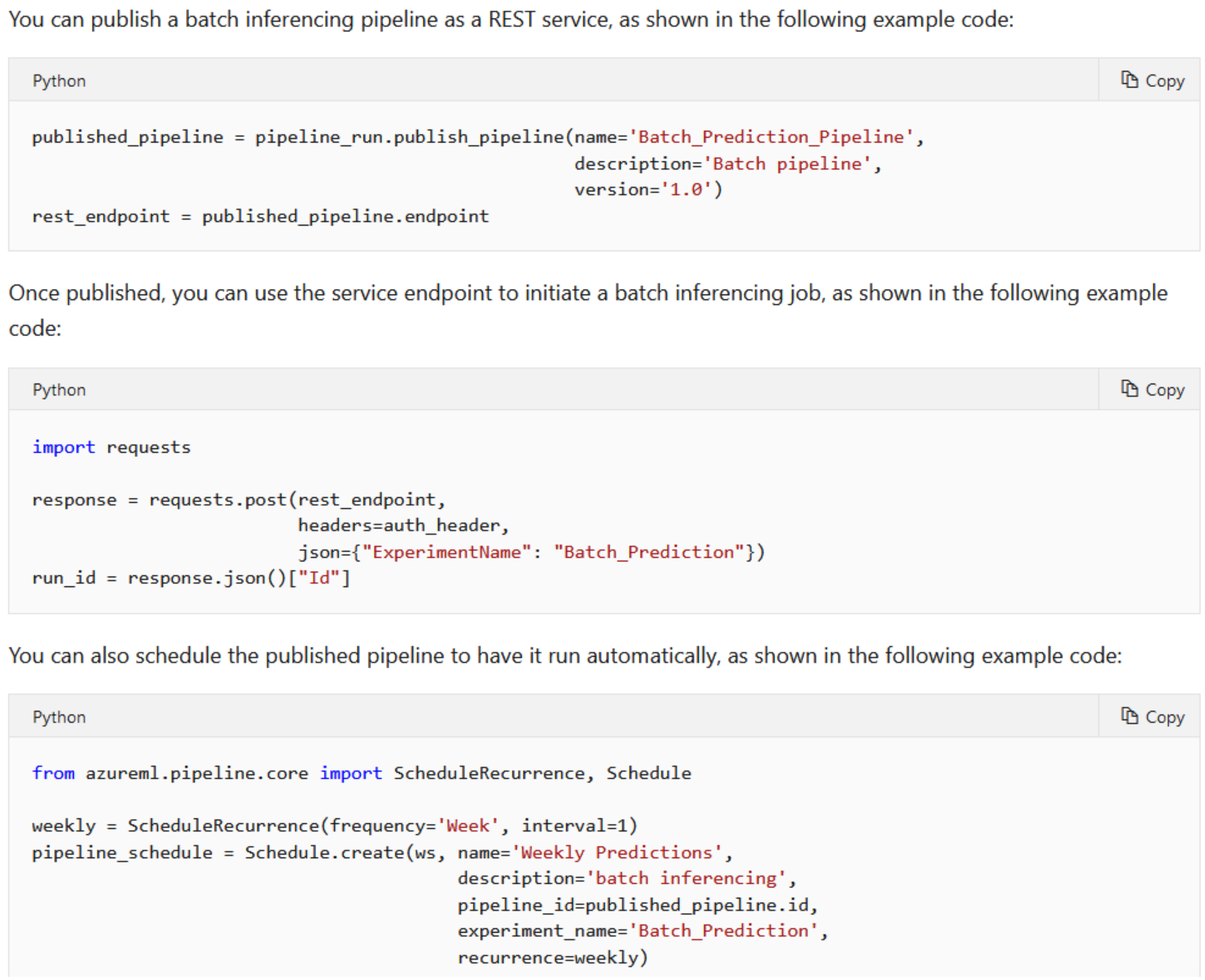
Create a scoring script

Batch inferencing service requires a scoring script to load the model and use it to predict new values. It must include two functions:

- **init()**: Called when the pipeline is initialized.
- **run(mini_batch)**: Called for each batch of data to be processed.

you use the **init** function to load the model from the model registry, and use the **run** function to generate predictions from each batch of data and return the results.

Read this :
https://docs.microsoft.com/en-us/learn/modules/deploy-batch-inference-pipelines-with-azure-machine-learning/2-batch-inference-pipelines



You can publish a batch inferencing pipeline as a REST service, as shown in the following example code:

```python
published_pipeline = pipeline_run.publish_pipeline(name='Batch_Prediction_Pipeline',
                                                    description='Batch pipeline',
                                                    version='1.0')
rest_endpoint = published_pipeline.endpoint
```

Once published, you can use the service endpoint to initiate a batch inferencing job, as shown in the following example code:

```python
import requests

response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "Batch_Prediction"})
run_id = response.json()["Id"]
```

You can also schedule the published pipeline to have it run automatically, as shown in the following example code:

```python
from azureml.pipeline.core import ScheduleRecurrence, Schedule

weekly = ScheduleRecurrence(frequency='Week', interval=1)
pipeline_schedule = Schedule.create(ws, name='Weekly Predictions',
                                        description='batch inferencing',
                                        pipeline_id=published_pipeline.id,
                                        experiment_name='Batch_Prediction',
                                        recurrence=weekly)
```

Discrete hyperparameters

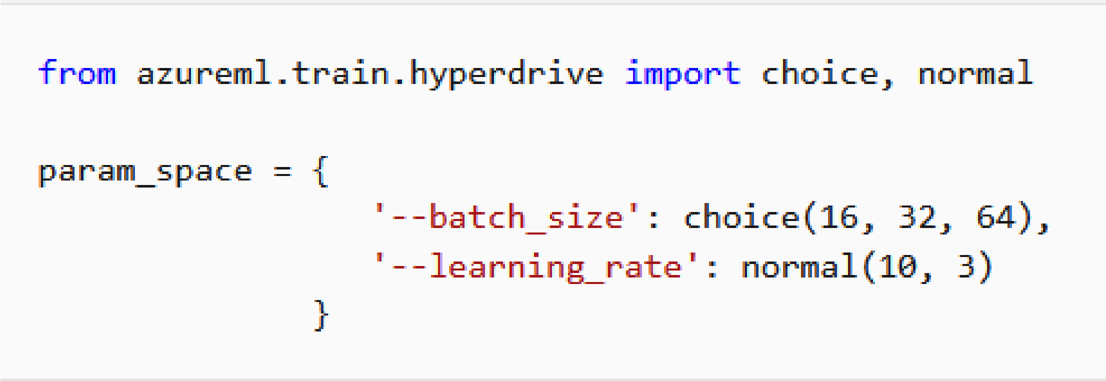
You can also select discrete values from any of the following discrete distributions:

- qnormal
- quniform
- qlognormal
- qloguniform

Continuous hyperparameters

you can use any of the following distribution types:

- normal
- uniform
- lognormal
- loguniform



```
from azureml.train.hyperdrive import choice, normal

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': normal(10, 3)
              }
```

Configuring sampling

Grid sampling

Grid sampling can only be employed when all hyperparameters are discrete,

```
param_space = {
            '--batch_size': choice(16, 32, 64),
            '--learning_rate': choice(0.01, 0.1, 1.0)
         }
param_sampling = GridParameterSampling(param_space)
```

Random sampling

```
param_space = {
            '--batch_size': choice(16, 32, 64),
            '--learning_rate': uniform(0.05, 0.1)
         }

param_sampling = RandomParameterSampling(param_space)
```

Random sampling is used to randomly select a value for each hyperparameter, which can be a mix of discrete and continuous values

Bayesian sampling

Bayesian sampling chooses hyperparameter values based on the Bayesian optimization algorithm, which tries to select parameter combinations that will result in improved performance from the previous selection

```
param_sampling = BayesianParameterSampling(param_space)
```

To help prevent wasting time, you can set an early termination policy that abandons runs that are unlikely to produce a better result than previously completed runs. The policy is evaluated at an *evaluation_interval* you specify, based on each time the target performance metric is logged. You can also set a *delay_evaluation* parameter to avoid evaluating the policy until a minimum number of iterations have been completed.

Bandit policy

You can use a bandit policy to stop a run if the target performance metric underperforms the best run so far by a specified margin.

early_termination_policy = BanditPolicy(**slack_amount = 0.2,evaluation_interval=1,delay_evaluation=5)**

Median stopping policy

A median stopping policy abandons runs where the target performance metric is worse than the median of the running averages for all runs.

early_termination_policy = MedianStoppingPolicy(**evaluation_interval=1,delay_evaluation=5**)

Truncation selection policy

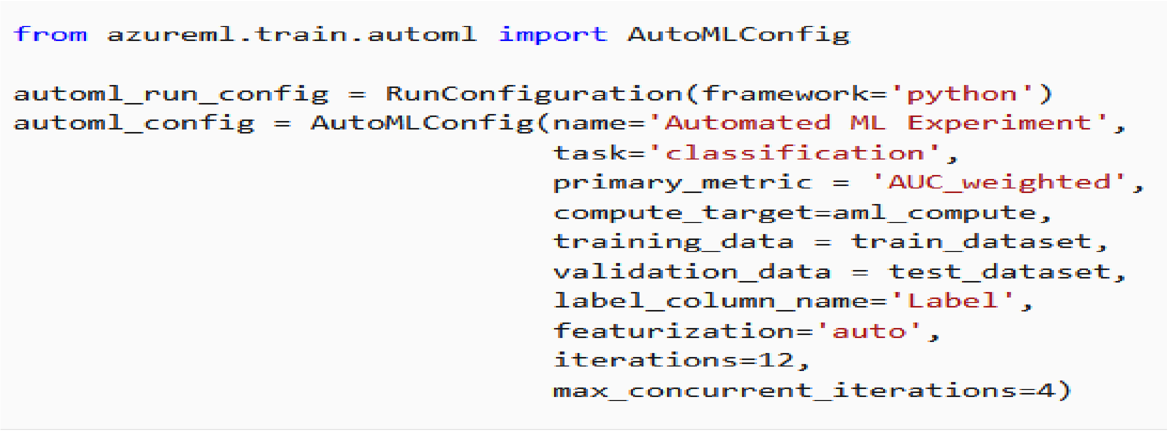
A truncation selection policy cancels the lowest performing *X*% of runs at each evaluation interval based on the *truncation_percentage* value you specify for *X*.

early_termination_policy = TruncationSelectionPolicy(**truncation_percentage=10,evaluation_interval=1, delay_evaluation=5**)

Read this : **https://docs.microsoft.com/en-us/learn/modules/tune-hyperparameters-with-azure-machine-learning/5-hyperdrive-experiment**

Automated machine learning tasks and algorithms

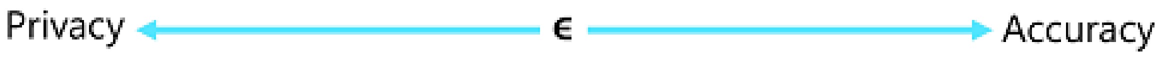
You can choose to block individual algorithms from being selected; which can be useful if you know that your data is not suited to a particular type of algorithm

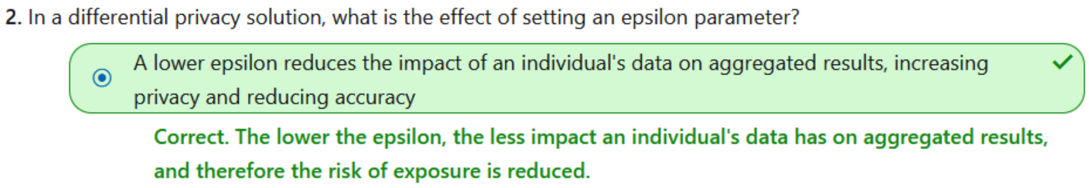


```
from azureml.train.automl import AutoMLConfig

automl_run_config = RunConfiguration(framework='python')
automl_config = AutoMLConfig(name='Automated ML Experiment',
                             task='classification',
                             primary_metric = 'AUC_weighted',
                             compute_target=aml_compute,
                             training_data = train_dataset,
                             validation_data = test_dataset,
                             label_column_name='Label',
                             featurization='auto',
                             iterations=12,
                             max_concurrent_iterations=4)
```

```
get_primary_metrics('classification')

automl_experiment = Experiment(ws, 'automl_experiment')
automl_run = automl_experiment.submit(automl_config)
best_run, fitted_model = automl_run.get_output()
best_run_metrics = best_run.get_metrics()

for step_ in fitted_model.named_steps:
   print(step_)
>
```

The amount of variation caused by adding noise is configurable through a parameter called epsilon
A higher epsilon value results in aggregations that are more true to the actual data distribution, but in which the individual contribution of a single individual to the aggregated value is less obscured by noise.

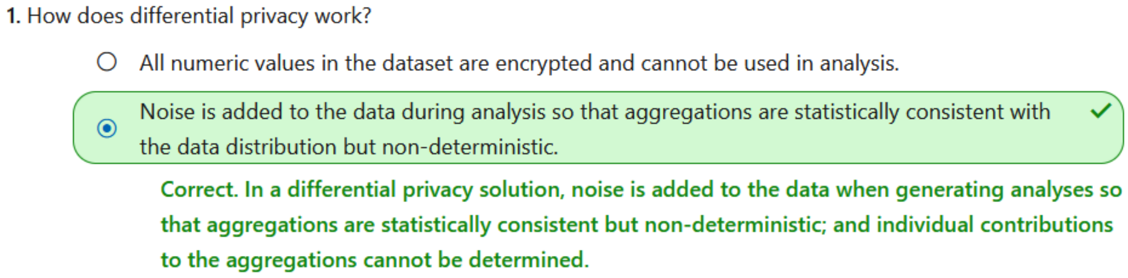




2. In a differential privacy solution, what is the effect of setting an epsilon parameter?

● A lower epsilon reduces the impact of an individual's data on aggregated results, increasing privacy and reducing accuracy ✓

Correct. The lower the epsilon, the less impact an individual's data has on aggregated results, and therefore the risk of exposure is reduced.



1. How does differential privacy work?

○ All numeric values in the dataset are encrypted and cannot be used in analysis.

● Noise is added to the data during analysis so that aggregations are statistically consistent with the data distribution but non-deterministic. ✓

Correct. In a differential privacy solution, noise is added to the data when generating analyses so that aggregations are statistically consistent but non-deterministic; and individual contributions to the aggregations cannot be determined.

Feature importance

Global feature importance

Local feature importance

Local feature importance measures the influence of each feature value for a specific individual prediction.
the Azure Machine Learning SDK to create explainers for models, even if they were not trained using an Azure Machine Learning experiment.

Creating an explainer

To interpret a local model, you must install the **azureml-interpret** package and use it to create an explainer. There are multiple types of explainer, including:

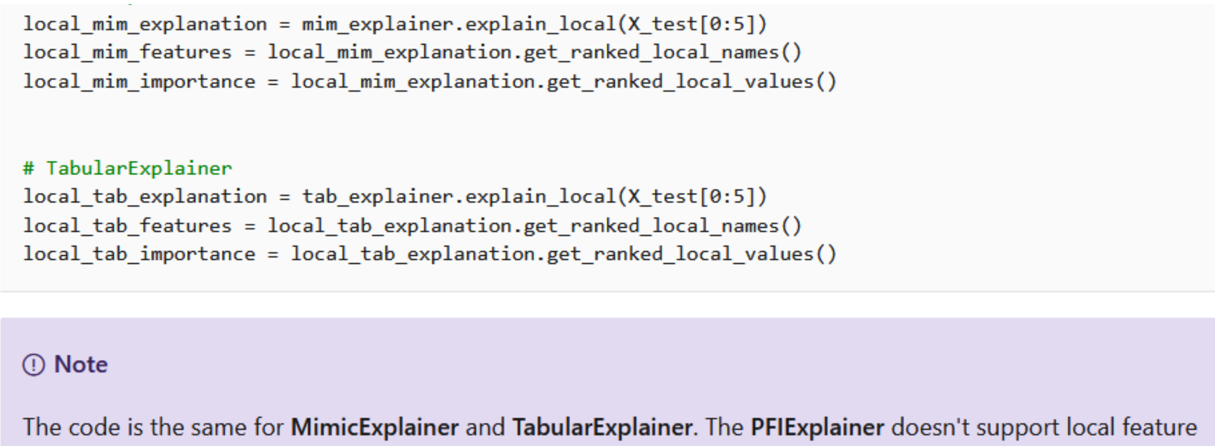
- **MimicExplainer** - An explainer that creates a *global surrogate model* that approximates your trained model and can be used to generate explanations. This explainable model must have the same kind of architecture as your trained model (for example, linear or tree-based).
- **TabularExplainer** - An explainer that acts as a wrapper around various SHAP explainer algorithms, automatically choosing the one that is most appropriate for your model architecture.
- **PFIExplainer** - a *Permutation Feature Importance* explainer that analyzes feature importance by shuffling feature values and measuring the impact on prediction performance.

```
mim_explainer = MimicExplainer(model=loan_model,
                 initialization_examples=X_test, explainable_model = DecisionTreeExplainableModel,
                 features=['loan_amount','income','age','marital_status'], classes=['reject', 'approve'])

tab_explainer = TabularExplainer(model=loan_model,initialization_examples=X_test,
                 features=['loan_amount','income','age','marital_status'],classes=['reject', 'approve'])

pfi_explainer = PFIExplainer(model =
loan_model,features=['loan_amount','income','age','marital_status'],
                 classes=['reject', 'approve'])
```

To retrieve global importance values for the features in your mode, you call the **explain_global()** method of your explainer to get a global explanation, and then use the **get_feature_importance_dict()** method to get a dictionary of the feature importance values.

```
global_mim_explanation = mim_explainer.explain_global(X_train)
global_tab_explanation = tab_explainer.explain_global(X_train)
global_pfi_explanation = pfi_explainer.explain_global(X_train, y_train)
global_pfi_feature_importance = global_pfi_explanation.get_feature_importance_dict()
```

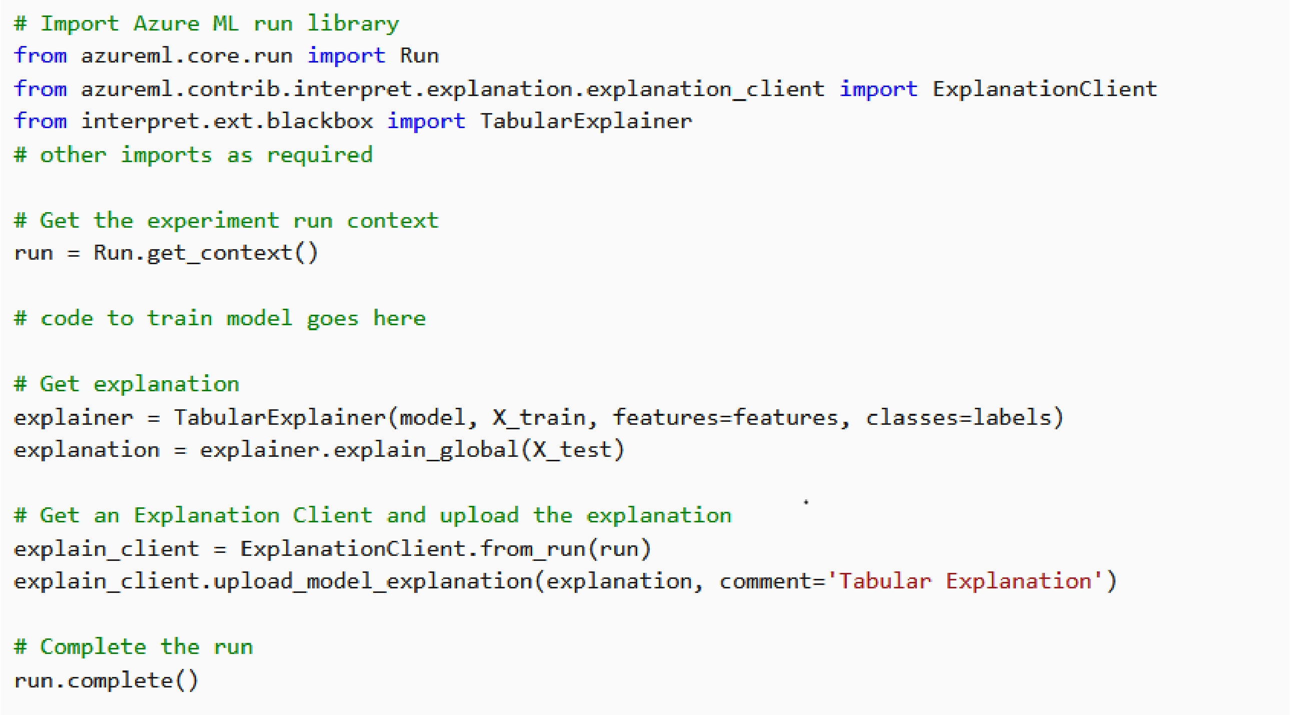
To retrieve local feature importance from a **MimicExplainer** or a **TabularExplainer**, you must call the **explain_local()** method of your explainer, specifying the subset of cases you want to explain. Then you can use the **get_ranked_local_names()** and **get_ranked_local_values()** methods to retrieve dictionaries of the feature names and importance values, ranked by importance



```
local_mim_explanation = mim_explainer.explain_local(X_test[0:5])
local_mim_features = local_mim_explanation.get_ranked_local_names()
local_mim_importance = local_mim_explanation.get_ranked_local_values()


# TabularExplainer
local_tab_explanation = tab_explainer.explain_local(X_test[0:5])
local_tab_features = local_tab_explanation.get_ranked_local_names()
local_tab_importance = local_tab_explanation.get_ranked_local_values()
```

ⓘ Note

The code is the same for **MimicExplainer** and **TabularExplainer**. The **PFIExplainer** doesn't support local feature



```
# Import Azure ML run library
from azureml.core.run import Run
from azureml.contrib.interpret.explanation.explanation_client import ExplanationClient
from interpret.ext.blackbox import TabularExplainer
# other imports as required

# Get the experiment run context
run = Run.get_context()

# code to train model goes here

# Get explanation
explainer = TabularExplainer(model, X_train, features=features, classes=labels)
explanation = explainer.explain_global(X_test)

# Get an Explanation Client and upload the explanation
explain_client = ExplanationClient.from_run(run)
explain_client.upload_model_explanation(explanation, comment='Tabular Explanation')

# Complete the run
run.complete()
```

```
client = ExplanationClient.from_run_id(workspace=ws,
experiment_name=experiment.experiment_name, run_id=run.id)
explanation = client.download_model_explanation()
feature_importances = explanation.get_feature_importance_dict()
```

Model explanations in Azure Machine Learning studio include multiple visualizations that you can use to explore feature importance.

Visualizations are only available for experiment runs that were configured to generate and upload explanations. When using automated machine learning, only the run producing the best model has explanations generated by default.

One way to start evaluating the fairness of a model is to compare *predictions* for each group within a *sensitive feature*

Potential causes of disparity

When you find a disparity between prediction rates or prediction performance metrics across sensitive feature groups, it's worth considering potential causes. These might include:

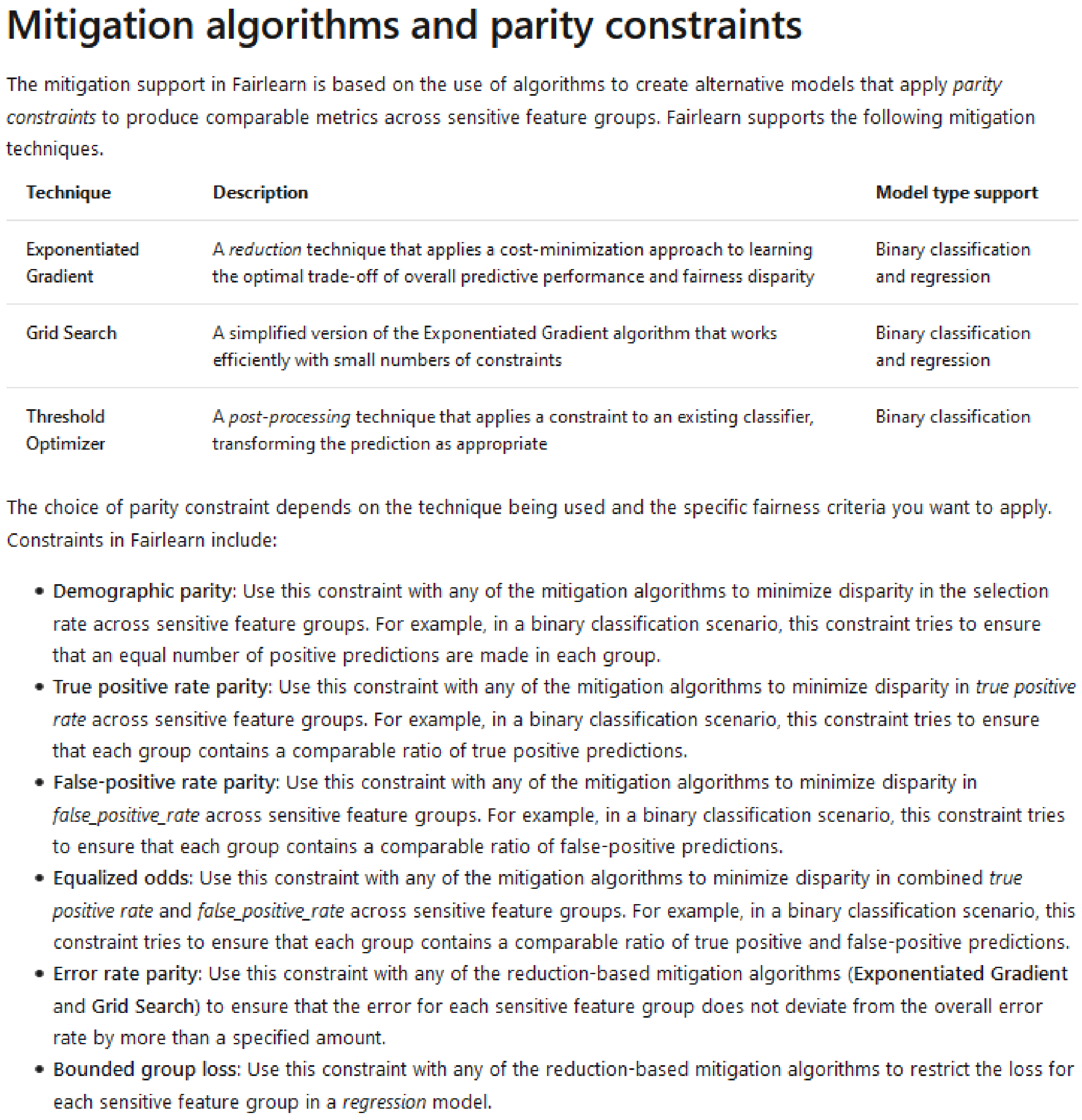
- Data imbalance. Some groups may be overrepresented in the training data, or the data may be skewed so that cases within a specific group aren't representative of the overall population.
- Indirect correlation. The sensitive feature itself may not be predictive of the label, but there may be a hidden correlation between the sensitive feature and some other feature that influences the prediction. For example, there's likely a correlation between age and credit history, and there's likely a correlation between credit history and loan defaults. If the credit history feature is not

included in the training data, the training algorithm may assign a predictive weight to age without accounting for credit history, which might make a difference to loan repayment probability.

- Societal biases. Subconscious biases in the data collection, preparation, or modeling process may have influenced feature selection or other aspects of model design.

Visualizing metrics in a dashboard

It's often easier to compare metrics visually, so Fairlearn provides an interactive dashboard widge



Mitigation algorithms and parity constraints

The mitigation support in Fairlearn is based on the use of algorithms to create alternative models that apply *parity constraints* to produce comparable metrics across sensitive feature groups. Fairlearn supports the following mitigation techniques.

Technique	Description	Model type support
Exponentiated Gradient	A *reduction* technique that applies a cost-minimization approach to learning the optimal trade-off of overall predictive performance and fairness disparity	Binary classification and regression
Grid Search	A simplified version of the Exponentiated Gradient algorithm that works efficiently with small numbers of constraints	Binary classification and regression
Threshold Optimizer	A *post-processing* technique that applies a constraint to an existing classifier, transforming the prediction as appropriate	Binary classification

The choice of parity constraint depends on the technique being used and the specific fairness criteria you want to apply. Constraints in Fairlearn include:

- **Demographic parity**: Use this constraint with any of the mitigation algorithms to minimize disparity in the selection rate across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that an equal number of positive predictions are made in each group.
- **True positive rate parity**: Use this constraint with any of the mitigation algorithms to minimize disparity in *true positive rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of true positive predictions.
- **False-positive rate parity**: Use this constraint with any of the mitigation algorithms to minimize disparity in *false_positive_rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of false-positive predictions.
- **Equalized odds**: Use this constraint with any of the mitigation algorithms to minimize disparity in combined *true positive rate* and *false_positive_rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of true positive and false-positive predictions.
- **Error rate parity**: Use this constraint with any of the reduction-based mitigation algorithms (**Exponentiated Gradient** and **Grid Search**) to ensure that the error for each sensitive feature group does not deviate from the overall error rate by more than a specified amount.
- **Bounded group loss**: Use this constraint with any of the reduction-based mitigation algorithms to restrict the loss for each sensitive feature group in a *regression* model.

Enable Application Insights

ws = Workspace.from_config()
ws.get_details()['applicationInsights']

When deploying a new real-time service, you can enable Application Insights in the deployment configuration for the service,

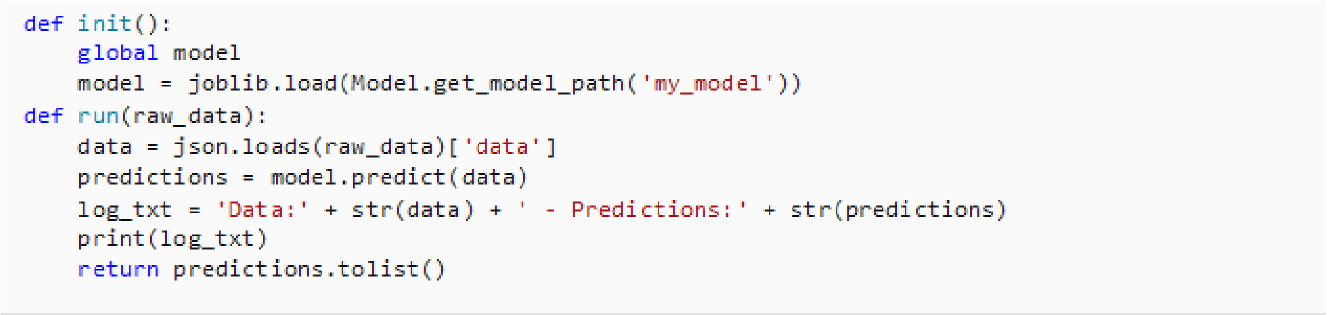
dep_config = AciWebservice.deploy_configuration(cpu_cores = 1,memory_gb = 1, enable_app_insights=True)

If you want to enable Application Insights for a service that is already deployed, you can modify the deployment configuration for Azure Kubernetes Service (AKS) based services .

service = ws.webservices['my-svc']
service.update(enable_app_insights=True)

Write log data

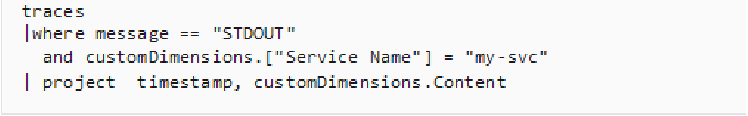
To capture telemetry data for Application insights, you can write any values to the standard output log in the scoring script for your service by using a print statement,



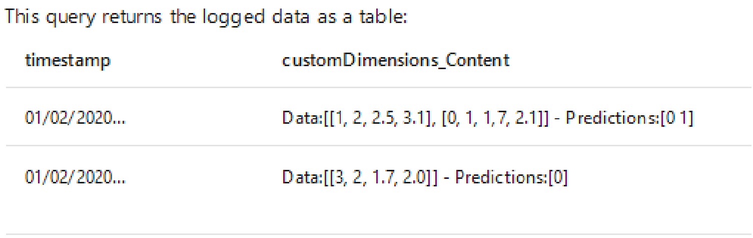
```
def init():
    global model
    model = joblib.load(Model.get_model_path('my_model'))
def run(raw_data):
    data = json.loads(raw_data)['data']
    predictions = model.predict(data)
    log_txt = 'Data:' + str(data) + ' - Predictions:' + str(predictions)
    print(log_txt)
    return predictions.tolist()
```

Azure Machine Learning creates a *custom dimension* in the Application Insights data model for the output you write.

To analyze captured log data, you can use the Log Analytics query interface for Application Insights in the Azure portal. This interface supports a SQL-like query syntax that you can use to extract fields from logged data, including custom dimensions created by your Azure Machine Learning service.



```
traces
|where message == "STDOUT"
  and customDimensions.["Service Name"] = "my-svc"
| project  timestamp, customDimensions.Content
```



This query returns the logged data as a table:

timestamp	customDimensions_Content
01/02/2020...	Data:[[1, 2, 2.5, 3.1], [0, 1, 1,7, 2.1]] - Predictions:[0 1]
01/02/2020...	Data:[[3, 2, 1.7, 2.0]] - Predictions:[0]

over time there may be trends that change the profile of the data, making your model less accurate. suppose a model is trained to predict the expected gas mileage of an automobile based on the number of cylinders, engine size, weight, and other features. Over time, as car manufacturing and engine technologies advance, the typical fuel-efficiency of vehicles might improve dramatically; making the predictions made by the model trained on older data less accurate.

This change in data profiles between training and inferencing is known as *data drift*, and it can be a significant issue for predictive models used in production. It is therefore important to be able to monitor data drift over time, and retrain models as required to maintain predictive accuracy.

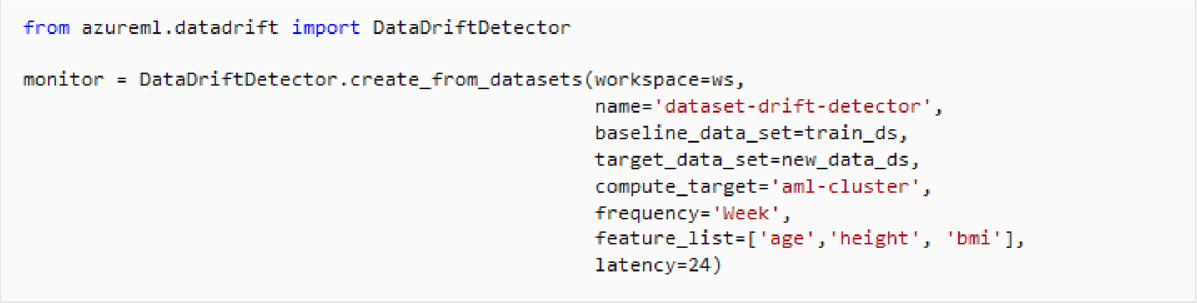
Azure Machine Learning supports data drift monitoring through the use of *datasets*. You can capture new feature data in a dataset and compare it to the dataset with which the model was trained

Monitor data drift by comparing dataset

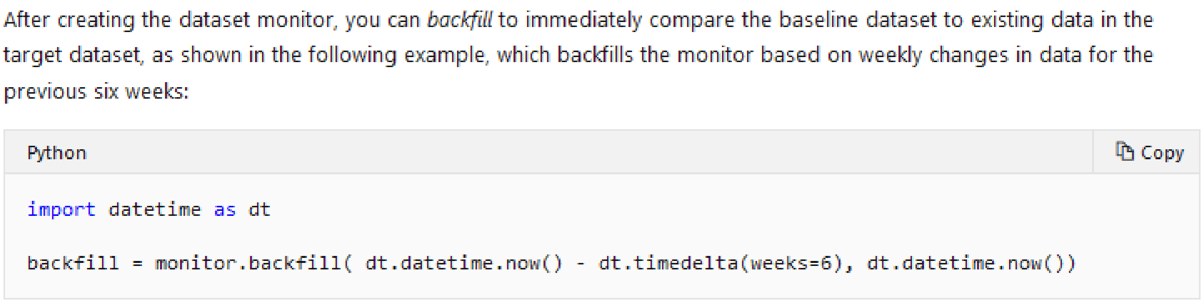
To monitor data drift using registered datasets, you need to register two datasets:

- A *baseline* dataset - usually the original training data.
- A *target* dataset that will be compared to the baseline based on time intervals. This dataset requires a column for each feature you want to compare, and a timestamp column so the rate of data drift can be measured.

After creating these datasets, you can define a *dataset monitor* to detect data drift and trigger alerts if the rate of drift exceeds a specified threshold. You can create dataset monitors using the visual interface in Azure Machine Learning studio, or by using the **DataDriftDetector** class in the Azure Machine Learning SDK



```python
from azureml.datadrift import DataDriftDetector

monitor = DataDriftDetector.create_from_datasets(workspace=ws,
                                                 name='dataset-drift-detector',
                                                 baseline_data_set=train_ds,
                                                 target_data_set=new_data_ds,
                                                 compute_target='aml-cluster',
                                                 frequency='Week',
                                                 feature_list=['age','height', 'bmi'],
                                                 latency=24)
```



After creating the dataset monitor, you can *backfill* to immediately compare the baseline dataset to existing data in the target dataset, as shown in the following example, which backfills the monitor based on weekly changes in data for the previous six weeks:

```python
import datetime as dt

backfill = monitor.backfill( dt.datetime.now() - dt.timedelta(weeks=6), dt.datetime.now())
```

ou can define a schedule to run every **Day**, **Week**, or **Month**.

For dataset monitors, you can specify a **latency**, indicating the number of hours to allow for new data to be collected and added to the target dataset. For deployed model data drift monitors, you can specify a **schedule_start** time value to indicate when the data drift run should start (if omitted, the run will start at the current time).

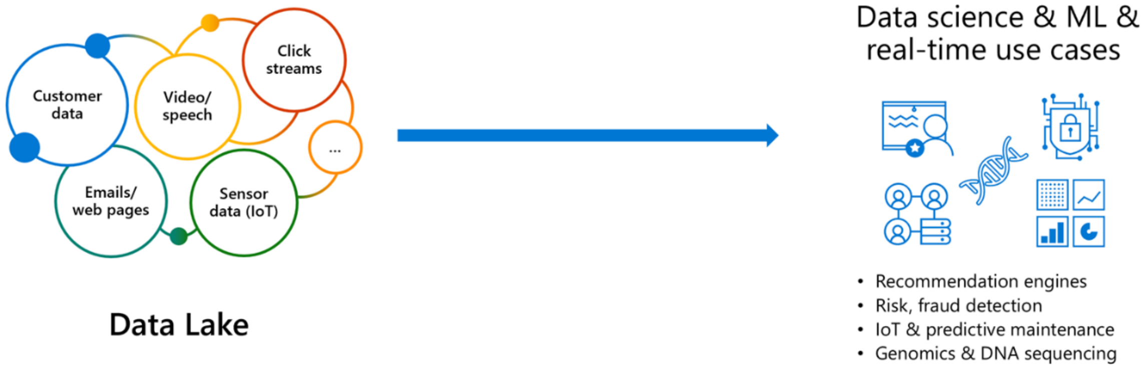
Describe the open source Delta Lake

Delta Lake is a transactional storage layer designed specifically to work with Apache Spark and Databricks File System (DBFS). At the core of Delta Lake is an optimized Spark table. It stores your data as Apache Parquet files in DBFS and maintains a transaction log that efficiently tracks changes to the table.

Data lakes

A data lake is a storage repository that inexpensively stores a vast amount of raw data, both current and historical, in native formats such as XML, JSON, CSV, and Parquet. It may contain operational relational databases with live transactional data.

Enterprises have been spending millions of dollars getting data into data lakes with Apache Spark. The aspiration is to do data science and ML on all that data using Apache Spark.



But the data is not ready for data science & ML. The majority of these projects are failing due to unreliable data!

The challenge with data lakes

Why are these projects struggling with reliability and performance?

To extract meaningful information from a data lake, you must solve problems such as:

- Schema enforcement when new tables are introduced.
- Table repairs when any new data is inserted into the data lake.
- Frequent refreshes of metadata.
- Bottlenecks of small file sizes for distributed computations.
- Difficulty sorting data by an index if data is spread across many files and partitioned.

There are also data reliability challenges with data lakes:

- Failed production jobs leave data in corrupt state requiring tedious recovery.
- Lack of schema enforcement creates inconsistent and low quality data.
- Lack of consistency makes it almost impossible to mix appends and reads, batch and streaming.

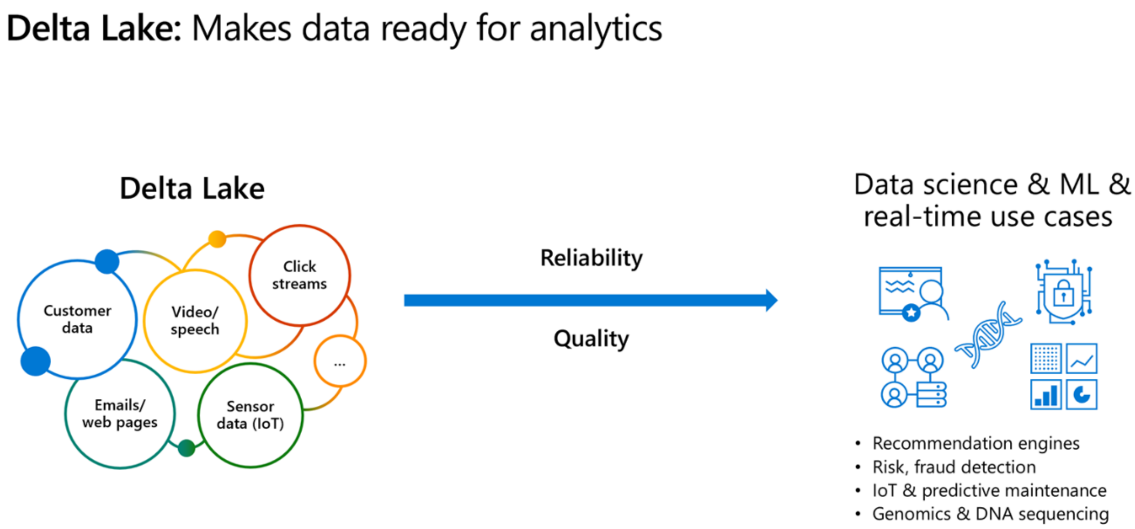
As great as data lakes are at inexpensively storing our raw data, they also bring with them performance challenges:

- **Too many small or very big files** - more time opening & closing files rather than reading contents (worse with streaming).
- **Partitioning also known as "poor man's indexing"**- breaks down if you picked the wrong fields or when data has many dimensions, high cardinality columns.
- **No caching** - cloud storage throughput is low (cloud object storage is 20-50MB/s/core vs 300MB/s/core for local SSDs).

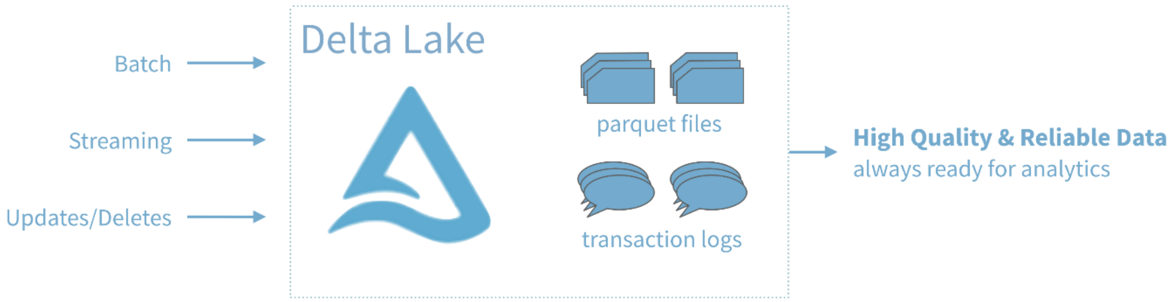
The solution: Delta Lake

Delta Lake is a file format that can help you build a data lake comprised of one or many tables in Delta Lake format. Delta Lake integrates tightly with Apache Spark, and uses an open format that is based on Parquet. Because it is an open-source format, Delta Lake is also supported by other data platforms, including Azure Synapse Analytics.

Delta Lake makes data ready for analytics.



Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.



You can read and write data that's stored in Delta Lake by using Apache Spark SQL batch and streaming APIs. These are the same familiar APIs that you use to work with Hive tables or DBFS directories. Delta Lake provides the following functionality:

ACID Transactions: Data lakes typically have multiple data pipelines reading and writing data concurrently, and data engineers have to go through a tedious process to ensure data integrity, due to the lack of transactions. Delta Lake brings ACID transactions to your data lakes. It provides serializability, the strongest level of isolation level.

Scalable Metadata Handling: In big data, even the metadata itself can be "big data". Delta Lake treats metadata just like data, leveraging Spark's distributed processing power to handle all its metadata. As a result, Delta Lake can handle petabyte-scale tables with billions of partitions and files at ease.

Time Travel (data versioning): Delta Lake provides snapshots of data enabling developers to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments.

Open Format: All data in Delta Lake is stored in Apache Parquet format enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet.

Unified Batch and Streaming Source and Sink: A table in Delta Lake is both a batch table, as well as a streaming source and sink. Streaming data ingest, batch historic backfill, and interactive queries all just work out of the box.

Schema Enforcement: Delta Lake provides the ability to specify your schema and enforce it. This helps ensure that the data types are correct and required columns are present, preventing bad data from causing data corruption.

Schema Evolution: Big data is continuously changing. Delta Lake enables you to make changes to a table schema that can be applied automatically, without the need for cumbersome DDL.

100% Compatible with Apache Spark API: Developers can use Delta Lake with their existing data pipelines with minimal change as it is fully compatible with Spark, the commonly used big data processing engine.

Get started with Delta using Spark APIs

Delta Lake is included with Azure Databricks. You can start using it today. To quickly get started with Delta Lake, do the following:

Instead of parquet...

Python

```
CREATE TABLE ...
USING parquet
...
dataframe
   .write
   .format("parquet")
   .save("/data")
```

... simply say delta

Python

```
CREATE TABLE ...
USING delta
...
dataframe
   .write
   .format("delta")
   .save("/data")
```

Using Delta with your existing Parquet tables

Step 1: Convert Parquet to Delta tables:

Python

```
CONVERT TO DELTA parquet.`path/to/table` [NO STATISTICS]
[PARTITIONED BY (col_name1 col_type1, col_name2 col_type2, ...)]
```

Step 2: Optimize layout for fast queries:

Python

```
OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType)
```

Basic syntax

Two of the core features of Delta Lake are performing upserts (insert/updates) and Time Travel operations. We will explore these concepts more within the notebooks in this module.

To UPSERT means to "UPdate" and "inSERT". In other words, UPSERT is literally TWO operations. It is not supported in traditional data lakes, as running an UPDATE could invalidate data that is accessed by the subsequent INSERT operation.

Using Delta Lake, however, we can do UPSERTS. Delta Lake combines these operations to guarantee atomicity to

- INSERT a row
- if the row already exists, UPDATE the row.

Upsert syntax

Upserting, or merging, in Delta Lake provides fine-grained updates of your data. The following syntax shows how to perform an Upsert:

SQL

```
MERGE INTO customers -- Delta table
USING updates
ON customers.customerId = source.customerId
WHEN MATCHED THEN
   UPDATE SET address = updates.address
WHEN NOT MATCHED
   THEN INSERT (customerId, address) VALUES (updates.customerId, updates.address)
```

See update table data syntax documentation.

Time Travel syntax

Because Delta Lake is version controlled, you have the option to query past versions of the data. Using a single file storage system, you now have access to several versions your historical data, ensuring that your data analysts will be able to replicate their reports (and compare aggregate changes over time) and your data scientists will be able to replicate their experiments.

Other time travel use cases are:

- Re-creating analyses, reports, or outputs (for example, the output of a machine learning model). This could be useful for debugging or auditing, especially in regulated industries.
- Writing complex temporal queries.
- Fixing mistakes in your data.
- Providing snapshot isolation for a set of queries for fast changing tables.

Example of using time travel to reproduce experiments and reports:

SQL

```
SELECT count(*) FROM events
TIMESTAMP AS OF timestamp
SELECT count(*) FROM events
VERSION AS OF version
```

Python

```
spark.read.format("delta").option("timestampAsOf", timestamp_string).load("/events/")
```

If you need to rollback accidental or bad writes:

SQL

```
INSERT INTO my_table
   SELECT * FROM my_table TIMESTAMP AS OF
   date_sub( current_date(), 1)
```

1. What is the Databricks Delta command to display metadata?

○ MSCK DETAIL *tablename*

● DESCRIBE DETAIL *tableName* ✓

Correct. You display metadata by using DESCRIBE DETAIL *tableName*.

○ SHOW SCHEMA *tablename*

2. How do you perform UPSERT in a Delta dataset?

○ Use UPSERT INTO *my-table*

○ Use UPSERT INTO *my-table* /MERGE

● Use MERGE INTO *my-table* USING *data-to-upsert* ✓

That's the correct syntax to perform UPSERT in a Databricks Delta dataset.

3. What optimization does the following command perform: OPTIMIZE Students ZORDER BY Grade?

○ Creates an order-based index on the Grade field to improve filters against that field

○ Ensures that all data backing, for example, Grade=8 is colocated, then updates a graph that routes requests to the appropriate files

● Ensures that all data backing, for example, Grade=8 is colocated, then rewrites the sorted data into new Parquet files ✓

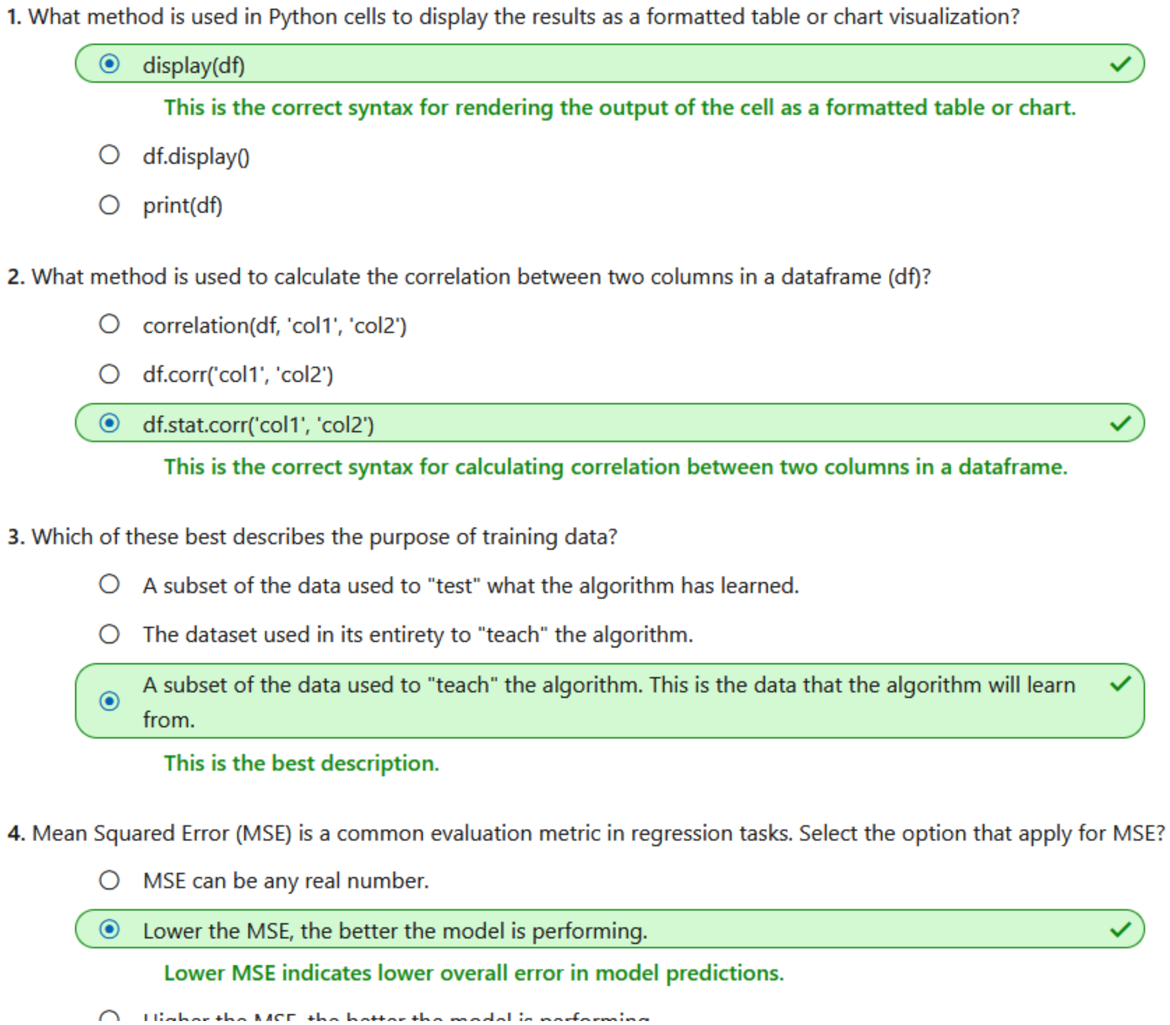
Correct. ZOrdering colocates related information in the same set of files.

4. What size does OPTIMIZE compact small files to?

○ Around 100 MB

● Around 1 GB ✓

Correct. The Spark optimization team determined this value to be a good compromise between speed and performance.



1. What method is used in Python cells to display the results as a formatted table or chart visualization?

● display(df) ✓

This is the correct syntax for rendering the output of the cell as a formatted table or chart.

○ df.display()

○ print(df)

2. What method is used to calculate the correlation between two columns in a dataframe (df)?

○ correlation(df, 'col1', 'col2')

○ df.corr('col1', 'col2')

● df.stat.corr('col1', 'col2') ✓

This is the correct syntax for calculating correlation between two columns in a dataframe.

3. Which of these best describes the purpose of training data?

○ A subset of the data used to "test" what the algorithm has learned.

○ The dataset used in its entirety to "teach" the algorithm.

● A subset of the data used to "teach" the algorithm. This is the data that the algorithm will learn from. ✓

This is the best description.

4. Mean Squared Error (MSE) is a common evaluation metric in regression tasks. Select the option that apply for MSE?

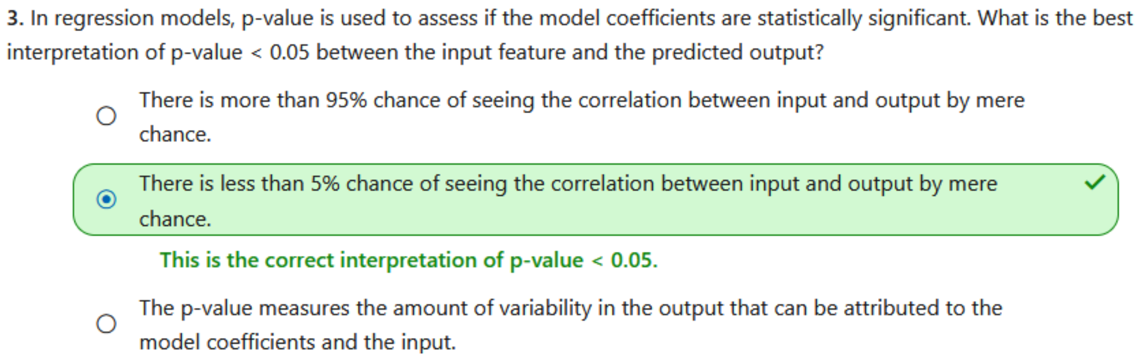
○ MSE can be any real number.

● Lower the MSE, the better the model is performing. ✓

Lower MSE indicates lower overall error in model predictions.

○ Higher the MSE, the better the model is performing.

>



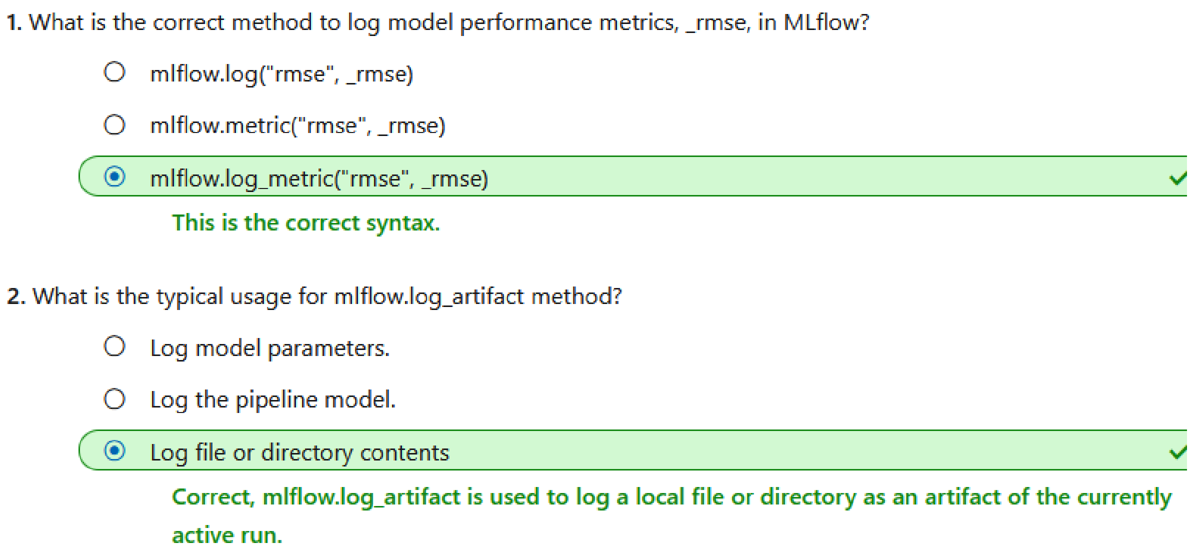
3. In regression models, p-value is used to assess if the model coefficients are statistically significant. What is the best interpretation of p-value < 0.05 between the input feature and the predicted output?

○ There is more than 95% chance of seeing the correlation between input and output by mere chance.

● There is less than 5% chance of seeing the correlation between input and output by mere chance. ✓

This is the correct interpretation of p-value < 0.05.

○ The p-value measures the amount of variability in the output that can be attributed to the model coefficients and the input.



1. What is the correct method to log model performance metrics, _rmse, in MLflow?

○ mlflow.log("rmse", _rmse)

○ mlflow.metric("rmse", _rmse)

◉ mlflow.log_metric("rmse", _rmse) ✓

This is the correct syntax.

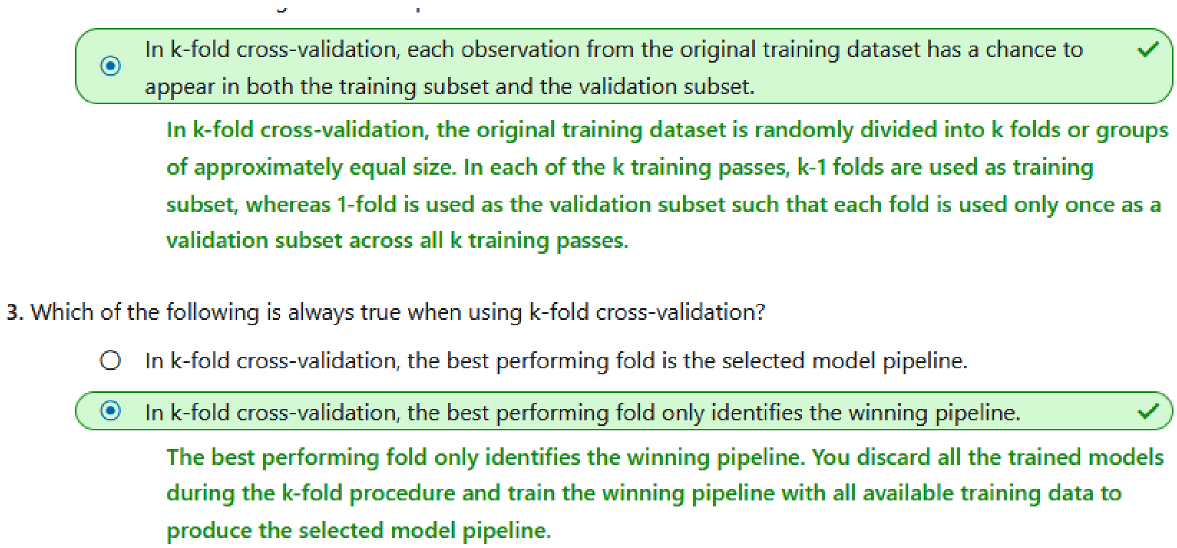
2. What is the typical usage for mlflow.log_artifact method?

○ Log model parameters.

○ Log the pipeline model.

◉ Log file or directory contents ✓

Correct, mlflow.log_artifact is used to log a local file or directory as an artifact of the currently active run.



◉ In k-fold cross-validation, each observation from the original training dataset has a chance to appear in both the training subset and the validation subset. ✓

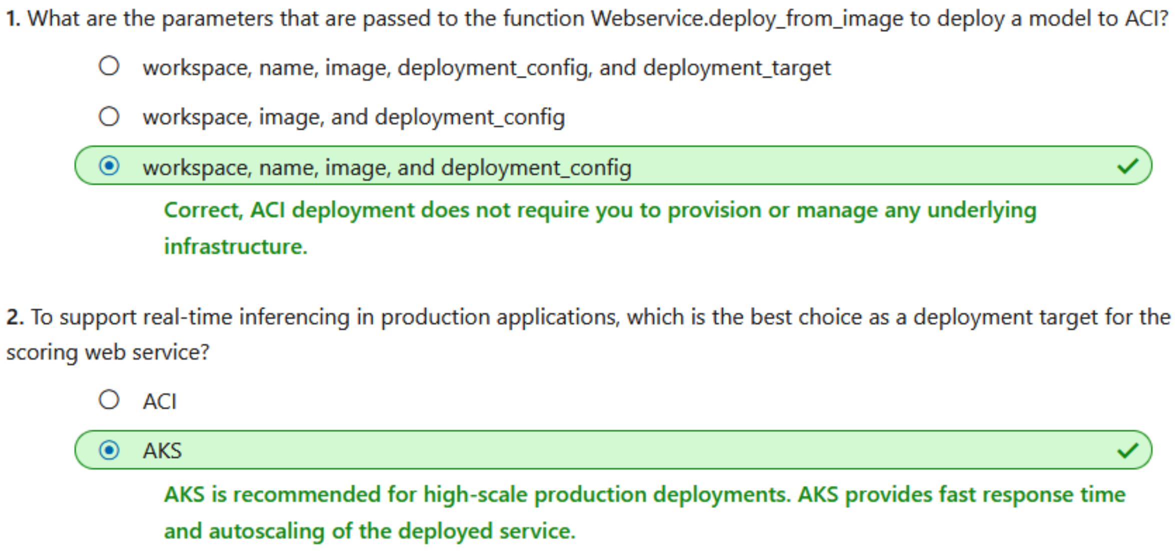
In k-fold cross-validation, the original training dataset is randomly divided into k folds or groups of approximately equal size. In each of the k training passes, k-1 folds are used as training subset, whereas 1-fold is used as the validation subset such that each fold is used only once as a validation subset across all k training passes.

3. Which of the following is always true when using k-fold cross-validation?

○ In k-fold cross-validation, the best performing fold is the selected model pipeline.

◉ In k-fold cross-validation, the best performing fold only identifies the winning pipeline. ✓

The best performing fold only identifies the winning pipeline. You discard all the trained models during the k-fold procedure and train the winning pipeline with all available training data to produce the selected model pipeline.



1. What are the parameters that are passed to the function Webservice.deploy_from_image to deploy a model to ACI?

○ workspace, name, image, deployment_config, and deployment_target

○ workspace, image, and deployment_config

● workspace, name, image, and deployment_config ✓

Correct, ACI deployment does not require you to provision or manage any underlying infrastructure.

2. To support real-time inferencing in production applications, which is the best choice as a deployment target for the scoring web service?

○ ACI

● AKS ✓

AKS is recommended for high-scale production deployments. AKS provides fast response time and autoscaling of the deployed service.

Dumps
https://www.gratisexam.com/microsoft/dp-100-exam/
https://www.itexams.com/exam/DP-100