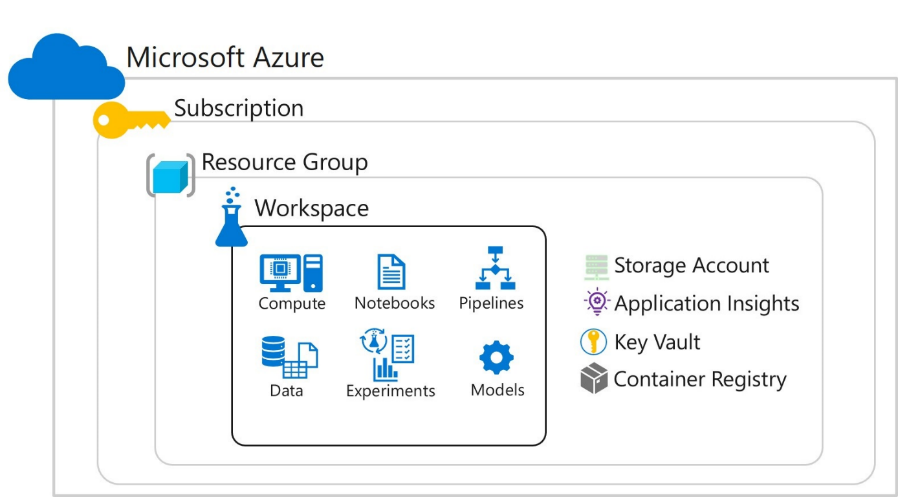
**Build and operate ML Solutions**



pip install azureml-sdk

Workspaces

# Creating a Workspace

ws = Workspace.create(name='aml-workspace',

subscription\_id='123456-abc-123...',0

resource\_group='aml-resources',

create\_resource\_group=True,

location='eastus')

GET JSON

{

"subscription\_id": "1234567-abcde-890-fgh...",

"resource\_group": "aml-resources",

"workspace\_name": "aml-workspace"

}

ws = Workspace.from\_config()

OR

ws = Workspace.get(name='aml-workspace',

subscription\_id='1234567-abcde-890-fgh...',

resource\_group='aml-resources')

Experiments

# Inline

# create an experiment variable

experiment = Experiment(workspace = ws, name = "my-experiment")

# start the experiment

run = experiment.start\_logging()

# experiment code goes here

# Log the row count

run.log('observations', row\_count)

* **log**: Record a single named value.
* **log\_list**: Record a named list of values.
* **log\_row**: Record a row with multiple columns.
* **log\_table**: Record a dictionary as a table.
* **log\_image**: Record an image file or a plot.

run.upload\_file(name='outputs/sample.csv', path\_or\_stream='./sample.csv')

# end the experiment

run.complete()

RunDetails(run).show()

Run.get\_metrics()

files = run.get\_file\_names()

# Script

# Get the experiment run context

run = Run.get\_context()

# Set regularization hyperparameter

parser = argparse.ArgumentParser()

parser.add\_argument('--reg-rate', type=float, dest='reg\_rate', default=0.01)

args = parser.parse\_args()

reg = args.reg\_rate

#log metrics

joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()

THEN

# 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',

arguments = ['--reg-rate', 0.1],

environment=sklearn\_env)

# Submit the experiment

experiment = Experiment(workspace=ws, name='training-experiment')

run = experiment.submit(config=script\_config)

run.wait\_for\_completion()

run.get\_file\_names()

THEN

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')

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')

run.download\_file(name='outputs/model.pkl', output\_file\_path='model.pkl')

for model in Model.list(ws):

# Get model name and auto-generated version

print(model.name, 'version:', model.version

Datastores

# General

In Azure Machine Learning, *datastores* are abstractions for cloud data sources. They encapsulate the information required to connect to data sources. You can access datastores directly in code by using the Azure Machine Learning SDK, and use it to upload or download data.

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)

**Note**: For a full list of supported datastores, see the [Azure Machine Learning documentation](https://aka.ms/AA70zfl).

# 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…')

# Get a reference

blob\_store = Datastore.get(ws, datastore\_name='blob\_data')

# Set default

ws.set\_default\_datastore('blob\_data')

When working with data files, although CSV format is very common, Parquet format generally results in better performance.

# Datasets

## Tabular

csv\_paths = [(blob\_ds, 'data/files/current\_data.csv'),

(blob\_ds, 'data/files/archive/\*.csv')]

tab\_ds = Dataset.Tabular.from\_delimited\_files(path=csv\_paths)

tab\_ds = tab\_ds.register(workspace=ws, name='csv\_table')

### Script argument

script\_config = ScriptRunConfig(source\_directory='my\_dir',

script='script.py',

arguments=['--ds', tab\_ds],

environment=env)

THEN

parser.add\_argument('--ds', type=str, dest='dataset\_id')

args = parser.parse\_args()

dataset = Dataset.get\_by\_id(ws, id=args.dataset\_id)

data = dataset.to\_pandas\_dataframe()

### Named input

script\_config = ScriptRunConfig(source\_directory='my\_dir',

script='script.py',

arguments=['--ds', tab\_ds.as\_named\_input('my\_dataset')],

environment=env)

THEN

parser.add\_argument('--ds', type=str, dest='ds\_id')

args = parser.parse\_args()

run = Run.get\_context()

dataset = run.input\_datasets['my\_dataset']

data = dataset.to\_pandas\_dataframe()

## File

file\_ds = Dataset.File.from\_files(path=(blob\_ds, 'data/files/images/\*.jpg'))

file\_ds = file\_ds.register(workspace=ws, name='img\_files')

#set create\_new\_version=True to overwrite

### Script argument

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.

script\_config = ScriptRunConfig(source\_directory='my\_dir',

script='script.py',

arguments=['--ds', file\_ds.as\_download()],

environment=env)

THEN

parser.add\_argument('--ds', type=str, dest='ds\_ref')

args = parser.parse\_args()

run = Run.get\_context()

imgs = glob.glob(args.ds\_ref + "/\*.jpg")

### Named input

As with tabular datasets, if you use a named input, you still need to include a script argument for the dataset, even though you don’t actually use it to retrieve the dataset.

script\_config = ScriptRunConfig(source\_directory='my\_dir',

script='script.py',

arguments=['--ds', file\_ds.as\_named\_input('my\_ds').as\_download()],

environment=env)

THEN

parser.add\_argument('--ds', type=str, dest='ds\_ref')

args = parser.parse\_args()

run = Run.get\_context()

dataset = run.input\_datasets['my\_ds']

imgs= glob.glob(dataset + "/\*.jpg")

## Retrieving

After registering a dataset, you can retrieve it by using any of the following techniques:

* The **datasets** dictionary attribute of a **Workspace** object.
* The **get\_by\_name** or **get\_by\_id** method of the **Dataset** class.

# 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')

You can retrieve a specific version of a dataset by specifying the **version** parameter in the **get\_by\_name** method of the **Dataset** class.

img\_ds = Dataset.get\_by\_name(workspace=ws, name='img\_files', version=2)

Environments

Python code runs in the context of a *virtual environment* that defines the version of the Python runtime to be used as well as the installed packages available to the code. In most Python installations, packages are installed and managed in environments using **Conda** or **pip**. To improve portability, we usually create environments in docker containers that are in turn be hosted in compute targets, such as your development computer, virtual machines, or clusters in the cloud.

# Creation

## From a specification file

For example, you could save the following Conda configuration settings in a file named **conda.yml**:

name: py\_env

dependencies:

- numpy

- pandas

- scikit-learn

- pip:

- azureml-defaults

THEN

env = Environment.from\_conda\_specification(name='training\_environment',

file\_path='./conda.yml')

## From an existing Conda environment

env = Environment.from\_existing\_conda\_environment(name='training\_environment',

conda\_environment\_name='py\_env')

OR

myenv = Environment.get(workspace=ws, name='AzureML-Minimal').clone(my\_env\_name)

## By specifying packages

env = Environment('training\_environment')

deps = CondaDependencies.create(conda\_packages=['scikit-learn','pandas','numpy'],

pip\_packages=['azureml-defaults'])

OR

conda\_dep = CondaDependencies()

conda\_dep.add\_pip\_package("numpy==1.18.1")

env.python.conda\_dependencies = deps

# Configuring containers

Usually, environments for experiment script are created in containers. The following code configures a script-based experiment to host the **env** environment created previously in a container (this is the default unless you use a **DockerConfiguration** with a **use\_docker** attribute of **False**, in which case the environment is created directly in the compute target)

docker\_config = DockerConfiguration(use\_docker=True)

script\_config = ScriptRunConfig(source\_directory='my\_folder',

script='my\_script.py',

environment=env,

docker\_runtime\_config=docker\_config)

Azure Machine Learning uses a library of base images for containers, choosing the appropriate base for the compute target you specify (for example, including Cuda support for GPU-based compute). If you have created custom container images and registered them in a container registry, you can override the default base images and use your own by modifying the attributes of the environment's **docker** property.

env.docker.base\_image='my-base-image'

env.docker.base\_image\_registry='myregistry.azurecr.io/myimage'

OR

env.docker.base\_image = None

env.docker.base\_dockerfile = './Dockerfile'

By default, Azure machine Learning handles Python paths and package dependencies. If your image already includes an installation of Python with the dependencies you need, you can override this behavior by setting **python.user\_managed\_dependencies** to **True** and setting an explicit Python path for your installation.

env.python.user\_managed\_dependencies=True

env.python.interpreter\_path = '/opt/miniconda/bin/python'

# Registering and reusing

env.register(workspace=ws)

training\_env = Environment.get(workspace=ws, name='training\_environment')

Compute targets

In Azure Machine Learning, *Compute Targets* are physical or virtual computers on which experiments are run.

# Types of compute

* **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, which may be your physical workstation or a virtual machine such as an Azure Machine Learning *compute instance* on which you are running a notebook. Local compute is generally a great choice during development and testing with low to moderate volumes of data.
* **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. This is a cost-effective way to run experiments that need to handle large volumes of data or use parallel processing to distribute the workload and reduce the time it takes to run.
* **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.

In Azure Machine Learning studio, 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.

You can run individual processes on the compute target that best fits its needs. For example, by using GPU-based compute to train deep learning models, and switching to lower-cost CPU-only compute to test and register the trained model.

# Creation

The most common ways to create or attach a compute target are to use the **Compute** page in Azure Machine Learning studio, or to use the Azure Machine Learning SDK to provision compute targets in code.

## With the SDK

# 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)

The priority for the virtual machines (VMs) is set to dedicated, meaning they are reserved for use in this cluster (the alternative is to specify *lowpriority*, which has a lower cost but means that the VMs can be preempted if a higher-priority workload requires the compute).

## Attaching an unmanaged compute (e.g., Databricks cluster)

# Specify a name for the compute (unique within the workspace)

compute\_name = 'db\_cluster'

# Define configuration for existing Azure Databricks cluster

db\_workspace\_name = 'db\_workspace'

db\_resource\_group = 'db\_resource\_group'

db\_access\_token = '1234-abc-5678-defg-90...'

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)

# Checking for an existing compute target

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

script\_config = ScriptRunConfig(source\_directory='my\_dir',

script='script.py',

environment=training\_env,

compute\_target=compute\_name)

OR

compute\_name = "aml-cluster"

training\_cluster = ComputeTarget(workspace=ws, name=compute\_name)

training\_env = Environment.get(workspace=ws, name='training\_environment')

script\_config = ScriptRunConfig(source\_directory='my\_dir',

script='script.py',

environment=training\_env,

compute\_target=training\_cluster)

Pipelines

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.

An Azure Machine Learning pipeline consists of one or more *steps* that perform tasks.

* **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.

**Note**: For a full list of supported step types, see [azure.pipeline.steps package documentation](https://aka.ms/AA70rrh).

# Creation

# Step to run a Python script

step1 = PythonScriptStep(name = 'prepare data',

source\_directory = 'scripts',

script\_name = 'data\_prep.py',

compute\_target = 'aml-cluster')

# Step to train a model

step2 = PythonScriptStep(name = 'train model',

source\_directory = 'scripts',

script\_name = 'train\_model.py',

compute\_target = 'aml-cluster')

# Construct the pipeline

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)

## Pass data between steps

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

1. Define a named **OutputFileDatasetConfig** object that references a location in a datastore. If no explicit datastore is specified, the default datastore is used.
2. Pass the **OutputFileDatasetConfig** object as a script argument in steps that run scripts.
3. Include code in those scripts to write to the **OutputFileDatasetConfig** argument as an output or read it as an input.

# Define a PipelineData object to pass data between steps

data\_store = ws.get\_default\_datastore()

prepped\_data = OutputFileDatasetConfig('prepped')

# Step to run a Python script

step1 = PythonScriptStep(name = 'prepare data',

source\_directory = 'scripts',

script\_name = 'data\_prep.py',

compute\_target = 'aml-cluster',

# Script arguments include PipelineData

arguments = ['--raw-ds', raw\_ds.as\_named\_input('raw\_data'),

'--out\_folder', prepped\_data])

# Step to run an estimator

step2 = PythonScriptStep(name = 'train model',

source\_directory = 'scripts',

script\_name = 'train\_model.py',

compute\_target = 'aml-cluster',

# Pass as script argument

arguments=['--training-data', prepped\_data.as\_input()])

THEN

# Get the experiment run context

run = Run.get\_context()

# Get arguments

parser = argparse.ArgumentParser()

parser.add\_argument('--raw-ds', type=str, dest='raw\_dataset\_id')

parser.add\_argument('--out\_folder', type=str, dest='folder')

args = parser.parse\_args()

output\_folder = args.folder

# 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']]

# Save prepped data to the PipelineData location

os.makedirs(output\_folder, exist\_ok=True)

output\_path = os.path.join(output\_folder, 'prepped\_data.csv')

prepped\_df.to\_csv(output\_path)

# Reuse pipeline steps

## Managing step output reuse

By default, the step output from a previous pipeline run is reused without rerunning the step provided the script, source directory, and other parameters for the step have not changed. Step reuse can reduce the time it takes to run a pipeline, but it can lead to stale results when changes to downstream data sources have not been accounted for. To control reuse for an individual step, you can set the **allow\_reuse** parameter in the step configuration, like this:

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)

## Forcing all steps to run

When you have multiple steps, you can force all of them to run regardless of individual reuse configuration by setting the **regenerate\_outputs** parameter when submitting the pipeline experiment:

pipeline\_run = experiment.submit(train\_pipeline, regenerate\_outputs=True)

# Publish pipelines

published\_pipeline = pipeline.publish(name='training\_pipeline',

description='Model training pipeline',

version='1.0')

OR

# 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')

THEN

rest\_endpoint = published\_pipeline.endpoint

# Using a published pipeline

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"]

## Use pipeline parameters

reg\_param = PipelineParameter(name='reg\_rate', default\_value=0.01)

...

step2 = PythonScriptStep(name = 'train model',

source\_directory = 'scripts',

script\_name = 'data\_prep.py',

compute\_target = 'aml-cluster',

# Pass parameter as script argument

arguments=['--in\_folder', prepped\_data,

'--reg', reg\_param],

inputs=[prepped\_data])

THEN

response = requests.post(rest\_endpoint,

headers=auth\_header,

json={"ExperimentName": "run\_training\_pipeline",

"ParameterAssignments": {"reg\_rate": 0.1}})

## Schedule pipelines

### Periodic intervals

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)

### Run on data changes

You must create a Schedule that monitors a specified path on a datastore, like this:

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')

Deployment

You can deploy a model as a real-time web service to several kinds of compute target, including local compute, an Azure Machine Learning compute instance, an Azure Container Instance (ACI), an Azure Kubernetes Service (AKS) cluster, an Azure Function, or an Internet of Things (IoT) module. Deployment to a local service, a compute instance, or an ACI is a good choice for testing and development. For production, you should deploy to a target that meets the specific performance, scalability, and security needs of your application architecture.

# 1. Register a trained model

classification\_model = Model.register(workspace=ws,

model\_name='classification\_model',

model\_path='model.pkl', # local path

description='A classification model')

OR

run.register\_model( model\_name='classification\_model',

model\_path='outputs/model.pkl', # run outputs path

description='A classification model')

# 2. Define an inference configuration

## Create an entry script

Create the *entry script* (sometimes referred to as the *scoring script*) for the service as a Python (.py) file. It must include two functions:

* **init()**: Called when the service is initialized.
* **run(raw\_data)**: Called when new data is submitted to the service.

# Called when the service is loaded

def init():

global model

# Get the path to the registered model file and load it

model\_path = os.path.join(os.getenv('AZUREML\_MODEL\_DIR'), 'model.pkl')

#or: model\_path = Model.get\_model\_path('nyc-taxi-fare')

model = joblib.load(model\_path)

# Called when a request is received

def run(raw\_data):

# Get the input data as a numpy array

data = np.array(json.loads(raw\_data)['data'])

# Get a prediction from the model

predictions = model.predict(data)

# Return the predictions as any JSON serializable format

return predictions.tolist()

## Create an environment

service\_env = Environment(name='service-env')

python\_packages = ['scikit-learn', 'numpy'] # whatever packages your entry script uses

for package in python\_packages:

service\_env.python.conda\_dependencies.add\_pip\_package(package)

## Create an InferenceConfig

classifier\_inference\_config = InferenceConfig(source\_directory = 'service\_files',

entry\_script="score.py",

environment=service\_env)

# 3. Define a deployment configuration

cluster\_name = 'aks-cluster'

compute\_config = AksCompute.provisioning\_configuration(location='eastus')

production\_cluster = ComputeTarget.create(ws, cluster\_name, compute\_config)

production\_cluster.wait\_for\_completion(show\_output=True)

THEN

classifier\_deploy\_config = AksWebservice.deploy\_configuration(cpu\_cores = 1,

memory\_gb = 1)

The code to configure an ACI deployment is similar, except that you do not need to explicitly create an ACI compute target, and you must use the **deploy\_configuration** class from the **azureml.core.webservice.AciWebservice** namespace. Similarly, you can use the **azureml.core.webservice.LocalWebservice** namespace to configure a local Docker-based service. To deploy a model to an Azure Function, you do not need to create a deployment configuration. Instead, you need to package the model based on the type of function trigger you want to use. This functionality is in preview at the time of writing. For more details, see [**Deploy a machine learning model to Azure Functions**](https://aka.ms/AA70rrn) in the Azure Machine Learning documentation.

# 4. Deploy the model

model = ws.models['classification\_model']

service = Model.deploy(workspace=ws,

name = 'classifier-service',

models = [model],

inference\_config = classifier\_inference\_config,

deployment\_config = classifier\_deploy\_config,

deployment\_target = production\_cluster)

service.wait\_for\_deployment(show\_output = True)

For ACI or local services, you can omit the **deployment\_target** parameter (or set it to **None**).

# Real-time inferencing service

## Azure Machine Learning SDK

Typically, you send data to the **run** method in JSON format with the following structure:

{

"data":[

[0.1,2.3,4.1,2.0], // 1st case

[0.2,1.8,3.9,2.1], // 2nd case,

...

]

}

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

# An array of new data cases

x\_new = [[0.1,2.3,4.1,2.0],

[0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document

json\_data = json.dumps({"data": x\_new})

# Call the web service, passing the input data

response = service.run(input\_data = json\_data)

# Get the predictions

predictions = json.loads(response)

## REST endpoint

endpoint = service.scoring\_uri

# An array of new data cases

x\_new = [[0.1,2.3,4.1,2.0],

[0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document

json\_data = json.dumps({"data": x\_new})

# 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

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).

By default, authentication is disabled for ACI services, and set to key-based authentication for AKS services (for which primary and secondary keys are automatically generated). You can optionally configure an AKS service to use token-based authentication (which is not supported for ACI services).

# Get keys

primary\_key, secondary\_key = service.get\_keys()

For token-based authentication, your client application needs to use service-principal authentication to verify its identity through Azure Active Directory (Azure AD) and call the **get\_token** method of the service to retrieve a time-limited token.

To make an authenticated call to the service's REST endpoint, you must include the key or token in the request header like this:

# Set the content type in the request headers

request\_headers = { "Content-Type":"application/json",

"Authorization":"Bearer " + key\_or\_token }

## Troubleshoot

### Check the service state

# Check its state

print(service.state)

To view the **state** of a service, you must use the compute-specific service type (for example **AksWebservice**) and not a generic **WebService** object. For an operational service, the state should be *Healthy*.

### Review service logs

print(service.get\_logs())

### Deploy to a local container

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)

print(service.run(input\_data = json\_data))

You can then troubleshoot runtime issues by making changes to the scoring file that is referenced in the inference configuration, and reloading the service without redeploying it (something you can only do with a local service):

service.reload()

print(service.run(input\_data = json\_data))

# Batch inference pipeline

An asynchronous process that bases its predictions on a batch of observations. The predictions are stored as files or in a database for end users or business applications. There is no need for authentication since users cannot use it to make real-time predictions.

## 1. Register a model

## 2. Create a scoring script

Only the ‘run’ function here differs from what we’ve seen before.

def run(mini\_batch):

# This runs for each batch

resultList = []

# process each file in the batch

for f in mini\_batch:

# Read comma-delimited data into an array

data = np.genfromtxt(f, delimiter=',')

# Reshape into a 2-dimensional array for model input

prediction = model.predict(data.reshape(1, -1))

# Append prediction to results

resultList.append("{}: {}".format(os.path.basename(f), prediction[0]))

return resultList

## 3. Create a pipeline with a ParallelRunStep

Azure Machine Learning provides a type of pipeline step specifically for performing parallel batch inferencing. Using the **ParallelRunStep** class, you can read batches of files from a **File** dataset and write the processing output to a **OutputFileDatasetConfig**. Additionally, you can set the **output\_action** setting for the step to "append\_row", which will ensure that all instances of the step being run in parallel will collate their results to a single output file.

# Get the batch dataset for input

batch\_data\_set = ws.datasets['batch-data']

# Set the output location

output\_dir = OutputFileDatasetConfig(name='inferences')

# Define the parallel run step step configuration

parallel\_run\_config = ParallelRunConfig(

source\_directory='batch\_scripts',

entry\_script="batch\_scoring\_script.py",

mini\_batch\_size="5",

error\_threshold=10,

output\_action="append\_row",

environment=batch\_env,

compute\_target=aml\_cluster,

node\_count=4)

# Create the parallel run step

parallelrun\_step = ParallelRunStep(

name='batch-score',

parallel\_run\_config=parallel\_run\_config,

inputs=[batch\_data\_set.as\_named\_input('batch\_data')],

output=output\_dir,

arguments=[],

allow\_reuse=True

)

# Create the pipeline

pipeline = Pipeline(workspace=ws, steps=[parallelrun\_step])

## 4. Run the pipeline

# Run the pipeline as an experiment

pipeline\_run = Experiment(ws, 'batch\_prediction\_pipeline').submit(pipeline)

pipeline\_run.wait\_for\_completion(show\_output=True)

# Get the outputs from the first (and only) step

prediction\_run = next(pipeline\_run.get\_children())

prediction\_output = prediction\_run.get\_output\_data('inferences')

prediction\_output.download(local\_path='results')

# Find the parallel\_run\_step.txt file

for root, dirs, files in os.walk('results'):

for file in files:

if file.endswith('parallel\_run\_step.txt'):

result\_file = os.path.join(root,file)

# Load and display the results

df = pd.read\_csv(result\_file, delimiter=":", header=None)

df.columns = ["File", "Prediction"]

print(df)

## 5. Publishing a batch inference pipeline

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:

response = requests.post(rest\_endpoint,

headers=auth\_header,

json={"ExperimentName": "Batch\_Prediction"})

run\_id = response.json()["Id"]

Hyperparameter tuning

# Defining a search space

Some hyperparameters require *discrete* values - in other words, you must select the value from a particular set of possibilities. You can define a search space for a discrete parameter using a **choice** from a list of explicit values, which you can define as a Python **list** (choice([10,20,30])), a **range** (choice(range(1,10))), or an arbitrary set of comma-separated values (choice(30,50,100))

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

* qnormal
* quniform
* qlognormal
* qloguniform

Some hyperparameters are *continuous* - in other words you can use any value along a scale. To define a search space for these kinds of value, you can use any of the following distribution types:

* normal
* uniform
* lognormal
* loguniform

The set of hyperparameter values tried during hyperparameter tuning is known as the *search space*. The definition of the range of possible values that can be chosen depends on the type of hyperparameter.

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

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

param\_space = {

'--batch\_size': choice(16, 32, 64),

'--learning\_rate': normal(10, 3)

}

param\_sampling = RandomParameterSampling(param\_space)

## 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\_space = {

'--batch\_size': choice(16, 32, 64),

'--learning\_rate': uniform(0.05, 0.1)

}

param\_sampling = BayesianParameterSampling(param\_space)

You can only use Bayesian sampling with **choice**, **uniform**, and **quniform** parameter expressions, and you can't combine it with an early-termination policy.

# Early termination

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. Early termination is particularly useful for deep learning scenarios where a deep neural network (DNN) is trained.

## 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)

You can also apply a bandit policy using a slack *factor*, which compares the performance metric as a ratio rather than an absolute value.

## 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)

# Running a hyperdrive experiment

Your script ***must***:

* Include an argument for each hyperparameter you want to vary.
* Log the target performance metric. This enables the hyperdrive run to evaluate the performance of the child runs it initiates, and identify the one that produces the best performing model.

# Get regularization hyperparameter

parser = argparse.ArgumentParser()

parser.add\_argument('--regularization', type=float, dest='reg\_rate', default=0.01)

args = parser.parse\_args()

reg = args.reg\_rate

# Get the experiment run context

run = Run.get\_context()

# load the training dataset

data = run.input\_datasets['training\_data'].to\_pandas\_dataframe()

# Separate features and labels, and split for training/validation

X = data[['feature1','feature2','feature3','feature4']].values

y = data['label'].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30)

# Train a logistic regression model with the reg hyperparameter

model = LogisticRegression(C=1/reg, solver="liblinear").fit(X\_train, y\_train)

# calculate and log accuracy

y\_hat = model.predict(X\_test)

acc = np.average(y\_hat == y\_test)

run.log('Accuracy', np.float(acc))

# Save the trained model

os.makedirs('outputs', exist\_ok=True)

joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()

THEN

# Assumes ws, script\_config and param\_sampling are already defined

hyperdrive = HyperDriveConfig(run\_config=script\_config,

hyperparameter\_sampling=param\_sampling,

policy=None,

primary\_metric\_name='Accuracy',

primary\_metric\_goal=PrimaryMetricGoal.MAXIMIZE,

max\_total\_runs=6,

max\_concurrent\_runs=4)

experiment = Experiment(workspace = ws, name = 'hyperdrive\_training')

hyperdrive\_run = experiment.submit(config=hyperdrive)

## Monitoring and reviewing

The experiment will initiate a child run for each hyperparameter combination to be tried, and you can retrieve the logged metrics these runs using the following code:

for child\_run in run.get\_children():

print(child\_run.id, child\_run.get\_metrics())

You can also list all runs in descending order of performance like this:

for child\_run in hyperdrive\_run.get\_children\_sorted\_by\_primary\_metric():

print(child\_run)

To retrieve the best performing run, you can use the following code:

best\_run = hyperdrive\_run.get\_best\_run\_by\_primary\_metric()

Automated Machine Learning

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)

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()

Specify a second *validation data* dataset or dataframe that will be used to validate the trained model. If this is not provided, Azure Machine Learning will apply cross-validation using the training data.

Alternatively:

* Specify a dataset, dataframe, or numpy array of *X* values containing the training features, with a corresponding *y* array of label values.
* Optionally, specify *X\_valid* and *y\_valid* datasets, dataframes, or numpy arrays of *X\_valid* values to be used for validation.

To retrieve the list of metrics available for a particular task type, you can use the **get\_primary\_metrics** function as shown here:

get\_primary\_metrics('classification')

You can find a full list of primary metrics and their definitions in [Understand automated machine learning results](https://aka.ms/AA70rrw). You can monitor automated machine learning experiment runs in Azure Machine Learning studio, or in the Jupyter Notebooks **RunDetails** widget.

Automated machine learning uses scikit-learn pipelines to encapsulate preprocessing steps with the model. You can view the steps in the fitted model you obtained from the best run using the code above like this:

for step\_ in fitted\_model.named\_steps:

print(step\_)

Differential Privacy

One way that an individual can protect their personal data is simply to not participate in a study – this is known as their "opt-out" option. However, there are a few considerations for this as a solution:

* Even if you decide to opt out a study may still produce results that affect you.
* The benefits of participation in the study may outweigh any negative impact.
* The only way for the opt-out option to work for every individual, is for every individual not to take part – which makes the whole study pointless!

Differential privacy seeks to protect individual data values by adding statistical "noise" to the analysis process. The math involved in adding the noise is complex, but the principle is fairly intuitive – the noise ensures that data aggregations stay statistically consistent with the actual data values allowing for some random variation, but make it impossible to work out the individual values from the aggregated data. In addition, the noise is different for each analysis, so the results are non-deterministic – in other words, two analyses that perform the same aggregation may produce slightly different results.

The amount of variation caused by adding noise is configurable through a parameter called epsilon. This value governs the amount of additional risk that your personal data can be identified through rejecting the opt-out option and participating in a study. A low epsilon value provides the most privacy, at the expense of less accuracy when aggregating the data. 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.

Explainable Machine Learning

Global feature importance quantifies the relative importance of each feature in the test dataset as a whole. It provides a general comparison of the extent to which each feature in the dataset influences prediction. Local feature importance measures the influence of each feature value for a specific individual prediction. Because this is a *classification* model, each feature gets a local importance value for each possible class, indicating the amount of support for that class based on the feature value. Since this is a *binary* classification model, there are only two possible classes (0 and 1). Each feature's support for one class results in correlatively negative level of support for the other. There could be multiple reasons why local importance for an individual prediction varies from global importance for the overall dataset; for example, someone might have a lower income than average, but the loan amount in this case might be unusually small.

For a multi-class classification model, a local importance values for each possible class is calculated for every feature, with the total across all classes always being 0. For example, a model might predict the species of a penguin based on features like its bill length, bill width, flipper length, and weight. Suppose there are three species of penguin, so the model predicts one of three class labels (0, 1, or 2). For an individual prediction, the flipper length feature might have local importance values of 0.5 for class 0, 0.3 for class 1, and -0.8 for class 2 - indicating that the flipper length moderately supports a prediction of class 0, slightly supports a prediction of class 1, and strongly supports a prediction that this particular penguin is ***not*** class 2. For a regression model, there are no classes so the local importance values simply indicate the level of influence each feature has on the predicted scalar label.

# Creation

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.

To retrieve global importance values for the features in your model, 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. 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.

# MimicExplainer

mim\_explainer = MimicExplainer(model=loan\_model,

initialization\_examples=X\_test,

explainable\_model = DecisionTreeExplainableModel,

features=['loan\_amount','income','age','marital\_status'],

classes=['reject', 'approve'])

global\_mim\_explanation = mim\_explainer.explain\_global(X\_train)

global\_mim\_feature\_importance = global\_mim\_explanation.get\_feature\_importance\_dict()

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

tab\_explainer = TabularExplainer(model=loan\_model,

initialization\_examples=X\_test,

features=['loan\_amount','income','age','marital\_status'],

classes=['reject', 'approve'])

global\_tab\_explanation = tab\_explainer.explain\_global(X\_train)

global\_tab\_feature\_importance = global\_tab\_explanation.get\_feature\_importance\_dict()

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()

# PFIExplainer

pfi\_explainer = PFIExplainer(model = loan\_model,

features=['loan\_amount','income','age','marital\_status'],

classes=['reject', 'approve'])

global\_pfi\_explanation = pfi\_explainer.explain\_global(X\_train, y\_train)

global\_pfi\_feature\_importance = global\_pfi\_explanation.get\_feature\_importance\_dict()

The **PFIExplainer** doesn't support local feature importance explanations.

# Upload explanations

To create an explanation in the experiment script, you'll need to ensure that the **azureml-interpret** and **azureml-contrib-interpret** packages are installed in the run environment.

# 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()

# Download explanations

You can view the explanation you created for your model in the **Explanations** tab for the run in Azure Machine learning studio. You can also use the **ExplanationClient** object to download the explanation in Python.

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 Fairness

When we consider the concept of *fairness* concerning predictions made by machine learning models, it helps to be clear about what we mean by "fair". One way to start evaluating the fairness of a model is to compare *predictions* for each group within a *sensitive feature*. For the loan approval model, *Age* is a sensitive feature that we care about, so we could split the data into subsets for each age group and compare the *selection rate* (the proportion of positive predictions) for each group.

Let's say we find that the model predicts that 36% of applicants aged 25 or younger will repay a loan, but it predicts successful repayments for 54% of applicants aged over 25. There's a disparity in predictions of 18%.

At first glance, this comparison seems to confirm that there's bias in the model that discriminates against younger applicants. However, when you consider the population as a whole, it may be that younger people generally earn less than people more established in their careers, have lower levels of savings and assets, and have a higher rate of defaulting on loans.

The important point to consider here is that just because we want to ensure fairness regarding age, it doesn't necessarily follow that age is not a factor in loan repayment probability. It's possible that in general, younger people are less likely to repay a loan than older people. To get the full picture, we need to look a little deeper into the predictive performance of the model for each subset of the population.

# Measuring fairness

When you train a machine learning model using a supervised technique, like regression or classification, you use metrics achieved against hold-out validation data to evaluate the overall predictive performance of the model. For example, you might evaluate a classification model based on *accuracy*, *precision*, or *recall*.

To evaluate the fairness of a model, you can apply the same predictive performance metric to subsets of the data, based on the sensitive features on which your population is grouped, and measure the disparity in those metrics across the subgroups.

For example, suppose the loan approval model exhibits an overall *recall* metric of 0.67 - in other words, it correctly identifies 67% of cases where the applicant repaid the loan. The question is whether or not the model provides a similar rate of correct predictions for different age groups.

To find out, we group the data based on the sensitive feature (*Age*) and measure the predictive performance metric (*recall*) for those groups. Then we can compare the metric scores to determine the disparity between them. Let's say that we find that the recall for validation cases where the applicant is 25 or younger is 0.50, and recall for cases where the applicant is over 25 is 0.83. In other words, the model correctly identified 50% of the people in the 25 or younger age group who successfully repaid a loan (and therefore misclassified 50% of them as loan defaulters), but found 83% of loan repayers in the older age group (misclassifying only 17% of them). The disparity in prediction performance between the groups is 33%, with the model predicting significantly more false negatives for the younger age group.

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

# Mitigating bias

Optimizing for fairness in a machine learning model is a *sociotechnical* challenge. In other words, it's not always something you can achieve purely by applying technical corrections to a training algorithm. However, there are some strategies you can adopt to mitigate bias, including:

* Balance training and validation data. You can apply over-sampling or under-sampling techniques to balance data and use stratified splitting algorithms to maintain representative proportions for training and validation.
* Perform extensive feature selection and engineering analysis. Make sure you fully explore the interconnected correlations in your data to try to differentiate features that are directly predictive from features that encapsulate more complex, nuanced relationships. You can use the [model interpretability support in Azure Machine Learning](https://docs.microsoft.com/en-us/learn/modules/explain-machine-learning-models-with-azure-machine-learning/) to understand how individual features influence predictions.
* Evaluate models for disparity based on significant features. You can't easily address the bias in a model if you can't quantify it.
* Trade-off overall predictive performance for the lower disparity in predictive performance between sensitive feature groups. A model that is 99.5% accurate with comparable performance across all groups is often more desirable than a model that is 99.9% accurate but discriminates against a particular subset of cases.

# Analyze model fairness with Fairlearn

**Fairlearn** is a Python package that you can use to analyze models and evaluate disparity between predictions and prediction performance for one or more sensitive features. It works by calculating group metrics for the sensitive features you specify. The metrics themselves are based on standard **scikit-learn** model evaluation metrics, such as *accuracy*, *precision*, or *recall* for classification models.

Fairlearn includes a **MetricFrame** function that enables you to create a dataframe of multiple metrics by the group. It's often easier to compare metrics visually, so Fairlearn provides an interactive dashboard widget that you can use in a notebook to display group metrics for a model.

Fairlearn integrates with Azure Machine Learning by enabling you to run an experiment in which the dashboard metrics are uploaded to your Azure Machine Learning workspace. This enables you to share the dashboard in Azure Machine Learning studio so that your data science team can track and compare disparity metrics for models registered in the workspace.

Monitoring

# Monitor models

To log telemetry in Application Insights from an Azure machine learning service, you must have an Application Insights resource associated with your Azure Machine Learning workspace, and you must configure your service to use it for telemetry logging.

When you create an Azure Machine Learning workspace, you can select an Azure Application Insights resource to associate with it. If you do not select an existing Application Insights resource, a new one is created in the same resource group as your workspace.

ws = Workspace.from\_config()

ws.get\_details()['applicationInsights']

## Enable Application Insights for a service

When deploying a new real-time service, you can enable Application Insights in the deployment configuration for the service, as shown in this example:

dep\_config = AciWebservice.deploy\_configuration(cpu\_cores = 1,

memory\_gb = 1,

enable\_app\_insights=True)

Alternatively, you can update any web service by using the Azure Machine Learning SDK, like this:

service = ws.webservices['my-svc']

service.update(enable\_app\_insights=True)

## Register telemetry

Application Insights automatically captures any information written to the standard output and error logs, and provides a query capability to view data in these logs.

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, as shown in the following example:

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.

## Query logs

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.

For example, the following query returns the **timestamp** and **customDimensions.Content** fields from log traces that have a **message** field value of *STDOUT* (indicating the data is in the standard output log) and a **customDimensions.["Service Name"]** field value of *my-svc*:

traces

|where message == "STDOUT"

and customDimensions.["Service Name"] = "my-svc"

| project timestamp, customDimensions.Content

# Monitor data drift

You typically train a machine learning model using a historical dataset that is representative of the new data that your model will receive for inferencing. However, over time there may be trends that change the profile of the data, making your model less accurate. It is therefore important to be able to monitor data drift over time, and retrain models as required to maintain predictive accuracy.

## Creation

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. 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. 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).

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:

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

You can configure a deployed service to collect new data submitted to the model for inferencing, which is saved in Azure blob storage and can be used as a target dataset for data drift monitoring. See [**Collect data from models in production**](https://aka.ms/AA70zg8) in the Azure Machine Learning documentation for more information.

## Scheduling alerts

You can define a **threshold** for data drift magnitude above which you want to be notified, and configure alert notifications by email.

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)

Security (summary)

# What is role-based access control?

*Azure Role-Based Access Control (Azure RBAC)* is an authorization system that allows fine-grained access management of Azure Machine Learning resources. Azure RBAC roles can be assigned to individuals and groups. This control is applied at the workspace level and can only be changed by administrators or owners of the specific workspace within Azure Machine Learning. An Azure Machine Learning workspace, like other Azure resources, comes with three default roles when it is created. You can add users to the workspace and assign one of these roles:

* **Owners** have full access to the workspace, including the ability to view, create, edit, or delete assets in a workspace. Owners can also change role assignments.
* **Contributors** can view, create, edit, or delete 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.
* **Readers** can only perform read-only actions in the workspace. Readers can list and view assets, including datastore credentials in a workspace. Readers can't create or update these assets.

If the default roles do not meet your organization's need for more selective access control, you can create your own Custom roles. You can make a role available at a specific workspace level, a specific resource group level, or a subscription level by defining the scope of your custom role, which we can see in the example JSON below.

Custom roles can be created by defining possible actions permitted and NotActions to restrict specific activities or access. You can create custom roles using Azure portal, Azure PowerShell, Azure CLI, or the REST API. Note the *Actions* and *NotActions* above which define the permissions for the custom role and the assigned scope that is at the specific workspace level. In this example, the data scientist’s actions are defined through a wildcard (represented by the \* sign) which extends a permission to everything that matches the action string you provide.

# Authentication with Azure AD

Azure Machine Learning relies on **Azure Active Directory (Azure AD)** for authentication and/or communication between other Azure cloud resources. Azure AD is a cloud-based identity and access management service that helps your employees' sign-in and access cloud resources on Azure.

In general, there are four authentication workflows that you can use when connecting to the workspace:

1. **Interactive**: You use your account in Azure Active Directory to either manually authenticate or obtain an authentication token. Interactive authentication is used during **experimentation and iterative development.** It enables you to control access to resources (such as a web service) on a per-user basis.
2. **Service principal:** You create a service principal account in Azure Active Directory, and use it to authenticate or obtain an authentication token. A service principal is used when you need an automated process to authenticate to the service.
3. **Azure CLI session:** You use an active Azure CLI session to authenticate. Azure CLI authentication is used during **experimentation and iterative development**, or when you need an **automated process to authenticate** to the service using a pre-authenticated session.
4. **Managed identity:** When using the Azure Machine Learning SDK on an Azure Virtual Machine, you can use a managed identity for Azure. This workflow allows the VM to connect to the workspace using the managed identity, without storing credentials in code or prompting the user to authenticate.

# Managed Identities

Managed identities allow you to authenticate services by providing an automatically managed identity for applications or services to use when connecting to Azure cloud services.

There are two types of managed identities:

* **System-assigned:** Some Azure services allow you to enable a managed identity directly on a service instance. When you enable a system-assigned managed identity, an identity is created in Azure AD tied to that service instance's lifecycle. By design, only that Azure resource can use this identity to request tokens from Azure AD, and when the resource is deleted, Azure automatically deletes the identity for you.
* **User-assigned:** You may also create a managed identity as a standalone Azure resource. You can create a user-assigned managed identity and assign it to one or more instances of an Azure service. The identity is managed separately from the resources that use it and will persist if a resource using it is removed. For simplicity, we recommend using system-assigned roles unless you require a custom access solution.

Azure Machine Learning compute clusters can use managed identities to authenticate access to Azure resources within Azure Machine Learning without including credentials in your code. This is useful to quickly provide the minimum required permissions to access resources while securing other critical resources. For example, during machine learning workflow, the workspace needs access to Azure Container Registry for Docker images, and storage accounts for training data. By default, the System-assigned identity is enabled directly on the Azure Machine Learning compute clusters and these resources will all be available to use. Once the compute cluster is deleted, Azure will automatically clean up the credentials and the identity in Azure Active Directory. Compute clusters can also support custom user-assigned identities assigned to multiple resources and will persist after resources are deleted.

# Azure Key Vault

Storing secrets in source code is impractical and a security anti-pattern. This is for multiple reasons:

* Changing passwords means updating source code, which can mean rebuilding and re-publishing applications.
* Hard-coded secrets make it awkward to work with different environments, such as staging and production environments. This also increases the risk of inadvertent modification or destruction of production environment data during development.
* All people with access to the source code gain access to all secrets. This makes it near impossible to ensure that only senior team members have access to sensitive resources. It also means that any sharing, or leak, of your source code also provides outside parties with your security keys.
* Source control, such as git, will typically retain old passwords in history. This means future team members gain access to all historical passwords.

One of the best alternatives to storing secrets in source code is to make them available in the application environment. In this pattern, your application requests secrets from the environment and then uses these to connect to the requisite resources. Azure Key Vault provides secure storage of generic secrets for applications in Azure-hosted environments.

For example, you can use the Azure Shell to set an environmental variable holding Key Store’s name, and save a password to that key store like so:

# export the name of the vault to an environmental variable

export KEY\_VAULT\_NAME=<your-unique-keyvault-name>

# Save a new secret, called ExamplePassword

az keyvault secret set --vault-name $KEY\_VAULT\_NAME --name "ExamplePassword" --value "hVFkk965BuUv"

This password is stored securely and is encrypted. As an example, a Python application using Azure Machine Learning’s SDK can access this key as follows. (Assumes azure-identity and azure-keyvault-secrets have been

pip installed)

# Get the key vault name

keyVaultName = os.environ["KEY\_VAULT\_NAME"]

# Create a client to access the secret

credential = DefaultAzureCredential()

client = SecretClient(vault\_url= f"https://{keyVaultName}.vault.azure.net", credential=credential)

# Get a secret and print it to the console

# Note that printing out passwords is bad practice and only

# performed here for learning purposes

retrieved\_secret = client.get\_secret("ExamplePassword")

print(f"Your secret is '{retrieved\_secret.value}'")

## Working with remote runs

The above provides a generic solution to using Key Vault. Typically, with Azure Machine Learning, you will be executing code through a remote run.

The standard flow for using secrets in this context is:

* Log in to Azure and connect to your workspace,
* Set a secret in Workspace Key Vault,
* Submit a remote run, then
* Within the remote run, get the secret from Key Vault and use it.

run = Run.get\_context()

secret\_value = run.get\_secret(name=" ExamplePassword")

# Virtual networks

We will begin by separating your model training from the wider net to its own virtual network to avoid these problems. To secure the Azure Machine Learning workspace and compute resources, we will use a virtual network (VNet). An Azure VNet is the fundamental building block for your private network in Azure. You can integrate Azure services in your virtual network with the following options:

* **Service endpoints** provide the identity of your virtual network to the Azure service. Once you enable service endpoints in your virtual network, you can add a virtual network rule to secure the Azure service resources to your virtual network. Service endpoints use public IP addresses.
* **Private endpoints** are network interfaces that securely connect you to a service powered by Azure Private Link. Private endpoint uses a private IP address from your VNet, effectively bringing the Azure services into your VNet.

You can connect your on-premises computers and networks to a VNet through a virtual private network (VPN) in several ways. A **Point-to-site VPN** is a connection between a virtual network and a single computer in your network. While a **Site-to-site VPN** can be established between your on-premises VPN device and an Azure VPN Gateway that's deployed in a virtual network. **ExpressRoute** can be used instead of a VPN If you wish to speed up creating private connections to Azure services.