Tutorial: Create production ML pipelines with Python SDK v2 (preview) in a Jupyter notebook

Learning Objectives - By the end of this two part tutorial, you should be able to use Azure Machine Learning (Azure ML) to productionize your ML project.

This means you will be able to leverage the AzureML Python SDK to:

- connect to your Azure ML workspace
- create Azure ML data assets
- create reusable Azure ML components
- create, validate and run Azure ML pipelines
- deploy the newly-trained model as an endpoint
- call the Azure ML endpoint for inferencing

Motivations - This tutorial is intended to introduce Azure ML to data scientists who want to scale up or publish their ML projects. By completing a familiar end-to-end project, which starts by loading the data and ends by creating and calling an online inference endpoint, the user should become familiar with the core concepts of Azure ML and their most common usage. Each step of this tutorial can be modified or performed in other ways that might have security or scalability advantages. We will cover some of those in the Part II of this tutorial, however, we suggest the reader use the provide links in each section to learn more on each topic.

Requirements - In order to benefit from this tutorial, you need to have:

- basic understanding of Machine Learning projects workflow
- an Azure subscription. If you don't have an Azure subscription, create a free account before you begin.
- a working Azure ML workspace. A workspace can be created via Azure Portal, Azure CLI, or Python SDK. Read more.
- a Python environmnet
- installed Azure Machine Learning Python SDK v2

Introduction

In this tutorial, you'll create an Azure ML pipeline to train a model for credit default prediction. The pipeline handles the data preparation, training and registering the trained model. You'll then run the pipeline, deploy the model and use it.

The image below shows the pipeline as you'll see it in the AzureML portal once submitted. It's a rather simple pipeline we'll use to walk you through the AzureML SDK v2.

The two steps are first data preparation and second training.

Screenshot that shows the AML Pipeline

An AzureML pipeline that runs from local components, requires several dependent files. Fo better understanding of the project structure, we produce all these dependencies in the notebook cells. By the end of this tutorial, the project structure should look like:

```
e2e-ds-experience
    components
        data_prep
            data_prep.py
        train
            train.py
            train.yml
    dependencies
        conda.yaml
    deploy
        sample-request.json
    media
        metrics.jpg
        pipeline-overview.jpg
        user-logs.jpg
    e2e-ml-workflow.ipynb
```

After running this notebook, you should be able to create the project direcetly in the IDE of your choice, instead.

Set up the pipeline resources

The Azure ML framework can be used from CLI, Python SDK, or studio interface. In this example, you'll use the AzureML Python SDK v2 to create a pipeline.

Before creating the pipeline, you'll set up the resources the pipeline will use:

- The dataset for training
- The software environment to run the pipeline

Connect to the workspace

Before we dive in the code, you'll need to connect to your Azure ML workspace. The workspace is the top-level resource for Azure Machine Learning, providing a centralized place to work with all the artifacts you create when you use Azure Machine Learning.

We are using `DefaultAzureCredential` to get access to workspace. `DefaultAzureCredential` should be capable of handling most Azure SDK authentication scenarios.

Reference for more available credentials if it does not work for you: configure credential example, azure-identity reference doc.

```
# Handle to the workspace
from azure.ai.ml import MLClient
```

```
# Authentication package
from azure.identity import DefaultAzureCredential,
InteractiveBrowserCredential

try:
    credential = DefaultAzureCredential()
    # Check if given credential can get token successfully.
    credential.get_token("https://management.azure.com/.default")
except Exception as ex:
    # Fall back to InteractiveBrowserCredential in case
DefaultAzureCredential not work
    credential = InteractiveBrowserCredential()
```

In the next cell, enter your Subscription ID, Resource Group name and Workspace name. To find your Subscription ID:

1. In the upper right Azure Machine Learning Studio toolbar, select your workspace name.
2. At the bottom, select **View all properties in Azure Portal**
3. Copy the value from Azure Portal into the code.

```
# Get a handle to the workspace
ml_client = MLClient(
    credential=credential,
    subscription_id="<SUBSCRIPTION_ID>",
    resource_group_name="<RESOURCE_GROUP>",
    workspace_name="<AML_WORKSPACE_NAME>",
)
```

The result is a handler to the workspace that you'll use to manage other resources and jobs.

> [!IMPORTANT] Creating MLClient will not connect to the workspace. The client initialization is lazy, it will wait for the first time it needs to make a call (in the notebook below, that will happen during dataset registration).

Register data from an external url

The data you use for training is usually in one of the locations below:

- Local machine
- Web
- Big Data Storage services (for example, Azure Blob, Azure Data Lake Storage, SQL)

Azure ML uses a `Data` object to register a reusable definition of data, and consume data within a pipeline. In the section below, you'll consume some data from web url as one example. `Data` assets ets from other sources can be created as well.

```
from azure.ai.ml.entities import Data
from azure.ai.ml.constants import AssetTypes

web_path = "https://archive.ics.uci.edu/ml/machine-learning-
```

```
databases/00350/default%20of%20credit%20card%20clients.xls"

credit_data = Data(
    name="creditcard_defaults",
    path=web_path,
    type=AssetTypes.URI_FILE,
    description="Dataset for credit card defaults",
    tags={"source_type": "web", "source": "UCI ML Repo"},
    version="1.0.0",
)
```

This code just created a `Data` asset, it is ready to be consumed as an input by the pipeline that you'll define in the next sections. In addition, you can register the data to your workspace so it becomes reusable across pipelines.

Registering the data asset will enable you to:

- reuse and share the data asset in future pipelines
- use versions to track the modification to the data asset
- use the data asset from Azure ML designer, which is Azure ML's GUI for pipeline authoring

Since this is the first time that you're making a call to the workspace, you may be asked to authenticate. Once the authentication is complete, you'll then see the dataset registration completion message.

```
credit_data = ml_client.data.create_or_update(credit_data)
print(
    f"Dataset with name {credit_data.name} was registered to 
workspace, the dataset version is {credit_data.version}"
)
```

Create a job environment for pipeline steps

So far, you've created a development environment on the compute instance, your development machine. You'll also need an environment to use for each step of the pipeline. Each step can have its own environment, or you can use some common environments for multiple steps.

In this example, you'll create a conda environment for your jobs, using a conda yaml file. First, create a directory to store the file in.

```
import os

dependencies_dir = "./dependencies"
os.makedirs(dependencies_dir, exist_ok=True)
```

Now, create the file in the dependencies directory.

```
%%writefile {dependencies_dir}/conda.yaml
name: model-env
```

```
channels:
  - conda-forge
dependencies:
  - python=3.8
  - numpy=1.21.2
  - pip=21.2.4
  - scikit-learn=0.24.2
  - scipy=1.7.1
  - pandas>=1.1,<1.2
  - pip:
    - inference-schema[numpy-support]==1.3.0
    - xlrd==2.0.1
    - mlflow== 1.26.1
    - azureml-mlflow==1.42.0
```

The specification contains some usual packages, that you'll use in your pipeline (numpy, pip).

Use the *yaml* file to create and register this custom environment in your workspace:

```
from azure.ai.ml.entities import Environment

custom_env_name = "aml-scikit-learn"

pipeline_job_env = Environment(
    name=custom_env_name,
    description="Custom environment for Credit Card Defaults 
pipeline",
    tags={"scikit-learn": "0.24.2"},
    conda_file=os.path.join(dependencies_dir, "conda.yaml"),
    image="mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest",
    version="0.1.0",
)
pipeline_job_env = 
ml_client.environments.create_or_update(pipeline_job_env)

print(
    f"Environment with name {pipeline_job_env.name} is registered to 
workspace, the environment version is {pipeline_job_env.version}"
)
```

Build the training pipeline

Now that you have all assets required to run your pipeline, it's time to build the pipeline itself, using the Azure ML Python SDK v2.

Azure ML pipelines are reusable ML workflows that usually consist of several components. The typical life of a component is:

- Write the yaml specification of the component.

- Optionally, register the component with a name and version in your workspace, to make it reusable and shareable.
- Load that component from the pipeline code.
- Implement the pipeline using this component inputs, outputs and parameters.
- Submit the pipeline.

Create component 1: data prep (using programmatic definition)

Let's start by creating the first component. This component handles the preprocessing of the data. The preprocessing task is performed in the *data_prep.py* python file.

First create a source folder for the data_prep component:

```
import os

data_prep_src_dir = "./components/data_prep"
os.makedirs(data_prep_src_dir, exist_ok=True)
```

This script performs the simple task of splitting the data into train and test datasets.

MLFlow will be used to log the parameters and metrics during our pipeline run.

Script for data preparation.

This script will be used to create a component using the 'command' class.

```
%%writefile {data_prep_src_dir}/data_prep.py
import os
import argparse
import pandas as pd
from sklearn.model_selection import train_test_split
import logging
import mlflow


def main():
    """Main function of the script."""

    # input and output arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("--data", type=str, help="path to input data")
    parser.add_argument("--test_train_ratio", type=float,
required=False, default=0.25)
    parser.add_argument("--train_data", type=str, help="path to train
data")
    parser.add_argument("--test_data", type=str, help="path to test
data")
    args = parser.parse_args()

    # Start Logging
    mlflow.start_run()

    print(" ".join(f"{k}={v}" for k, v in vars(args).items()))
    print("input data:", args.data)

    credit_df = pd.read_excel(args.data, header=1, index_col=0)

    mlflow.log_metric("num_samples", credit_df.shape[0])
    mlflow.log_metric("num_features", credit_df.shape[1] - 1)

    credit_train_df, credit_test_df = train_test_split(
        credit_df,
        test_size=args.test_train_ratio,
    )

    # output paths are mounted as folder, therefore, we are adding a
filename to the path
    credit_train_df.to_csv(os.path.join(args.train_data, "data.csv"),
index=False)

    credit_test_df.to_csv(os.path.join(args.test_data, "data.csv"),
index=False)

    # Stop Logging
    mlflow.end_run()


if __name__ == "__main__":
    main()
```

Now that you have a script that can perform the desired task, we can create an Azure ML Component from it. Azure ML support various types of components for performing ML tasks, such as running scripts, data transfer, etc.

A component can be created by calling the component instantiators, or directly writing the defining yaml file.

You'll use the general purpose **command** that can run command line actions. This command line action can be directly calling system commands or running a script. The inputs/outputs are accessible in the command via the `${{ ... }}` notation. For the second component of this tutorial you will use `yaml` definitions.

```
from azure.ai.ml import command
from azure.ai.ml import Input, Output

data_prep_component = command(
    name="data_prep_credit_defaults",
    display_name="Data preparation for training",
    description="reads a .xl input, split the input to train and
test",
    inputs={
        "data": Input(type="uri_folder"),
        "test_train_ratio": Input(type="number"),
    },
    outputs=dict(
        train_data=Output(type="uri_folder", mode="rw_mount"),
        test_data=Output(type="uri_folder", mode="rw_mount"),
    ),
    # The source folder of the component
    code=data_prep_src_dir,
    command="""python data_prep.py \
            --data ${{inputs.data}} --test_train_ratio $
{{inputs.test_train_ratio}} \
            --train_data ${{outputs.train_data}} --test_data $
{{outputs.test_data}} \
            """,
    environment=f"{pipeline_job_env.name}:{pipeline_job_env.version}",
)
```

Optionally, register the component in the workspace for future re-use. **command()** is a component builder, in order to fetch the component itself, we need to call the **.component** property from it.

```
# Now we register the component to the workspace
data_prep_component =
ml_client.create_or_update(data_prep_component.component)

# Create (register) the component in your workspace
print(
    f"Component {data_prep_component.name} with Version
{data_prep_component.version} is registered"
)
```

Create component 2: training (using yaml definition)

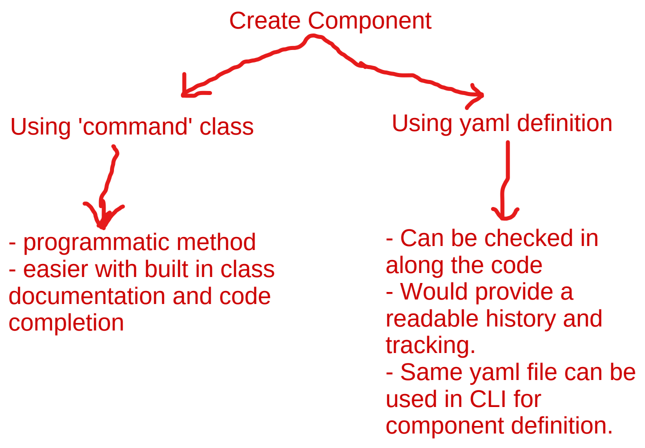
The second component that you'll create will consume the training and test data, train a tree based model and return the output model. You'll use Azure ML logging capabilities to record and visualize the learning progress.

You used the `command` class to create your first component. This time you'll use the yaml definition to define the second component. Each method has its own advantages. A yaml definition can actually be checked-in along the code, and would provide a readable history tracking. Also, the same yaml file can be used in the CLI for component deficnition. The programmatic method using `command` can be easier with built-in class documentation and code completion.

Create the directory for this component:

```
import os

train_src_dir = "./components/train"
os.makedirs(train_src_dir, exist_ok=True)
```


Create Component
Using 'command' class
Using yaml definition
- programmatic method
- easier with built in class documentation and code completion
- Can be checked in along the code
- Would provide a readable history and tracking.
- Same yaml file can be used in CLI for component definition.

Create the training script in the directory:

```
%%writefile {train_src_dir}/train.py
import argparse
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report
import os
import pandas as pd
import mlflow


def select_first_file(path):
    """Selects first file in folder, use under assumption there is
only one file in folder
    Args:
        path (str): path to directory or file to choose
    Returns:
        str: full path of selected file
    """
    files = os.listdir(path)
    return os.path.join(path, files[0])


# Start Logging
mlflow.start_run()

# enable autologging
mlflow.sklearn.autolog()

os.makedirs("./outputs", exist_ok=True)


def main():
    """Main function of the script."""

    # input and output arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("--train_data", type=str, help="path to train
data")
    parser.add_argument("--test_data", type=str, help="path to test
data")
    parser.add_argument("--n_estimators", required=False, default=100,
type=int)
    parser.add_argument("--learning_rate", required=False,
default=0.1, type=float)
    parser.add_argument("--registered_model_name", type=str,
help="model name")
    parser.add_argument("--model", type=str, help="path to model
file")
```

```
    args = parser.parse_args()

    # paths are mounted as folder, therefore, we are selecting the
file from folder
    train_df = pd.read_csv(select_first_file(args.train_data))

    # Extracting the label column
    y_train = train_df.pop("default payment next month")

    # convert the dataframe values to array
    X_train = train_df.values

    # paths are mounted as folder, therefore, we are selecting the
file from folder
    test_df = pd.read_csv(select_first_file(args.test_data))

    # Extracting the label column
    y_test = test_df.pop("default payment next month")

    # convert the dataframe values to array
    X_test = test_df.values

    print(f"Training with data of shape {X_train.shape}")

    clf = GradientBoostingClassifier(
        n_estimators=args.n_estimators,
learning_rate=args.learning_rate
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)

    print(classification_report(y_test, y_pred))

    # Registering the model to the workspace
    print("Registering the model via MLFlow")
    mlflow.sklearn.log_model(
        sk_model=clf,
        registered_model_name=args.registered_model_name,
        artifact_path=args.registered_model_name,
    )

    # Saving the model to a file
    mlflow.sklearn.save_model(
        sk_model=clf,
        path=os.path.join(args.model, "trained_model"),
    )

    # Stop Logging
    mlflow.end_run()


if __name__ == "__main__":
    main()
```

As you can see in this training script, once the model is trained, the model file is saved and registered to the workspace. Now you can use the registered model in inferencing endpoints.

For the environment of this step, you'll use one of the built-in (curated) Azure ML environments. The tag `azureml`, tells the system to use look for the name in curated environments.

First, create the *yaml* file describing the component:

```
%%writefile {train_src_dir}/train.yml
# <component>
name: train_credit_defaults_model
display_name: Train Credit Defaults Model
# version: 1 # Not specifying a version will automatically update the
version
type: command
inputs:
  train_data:
    type: uri_folder
  test_data:
    type: uri_folder
  learning_rate:
    type: number
  registered_model_name:
    type: string
outputs:
  model:
    type: uri_folder
code: .
environment:
  # for this step, we'll use an AzureML curate environment
  azureml:AzureML-sklearn-1.0-ubuntu20.04-py38-cpu:1
command: >-
  python train.py
  --train_data ${{inputs.train_data}}
  --test_data ${{inputs.test_data}}
  --learning_rate ${{inputs.learning_rate}}
  --registered_model_name ${{inputs.registered_model_name}}
  --model ${{outputs.model}}
# </component>
```

Once the `yaml` file and the script are ready, you can create your component using `load_component()`.

```python
# importing the Component Package
from azure.ai.ml import load_component

# Loading the component from the yml file
train_component = load_component(source=os.path.join(train_src_dir,
"train.yml"))
```

Now create and register the component:

```python
# Now we register the component to the workspace
train_component = ml_client.create_or_update(train_component)

# Create (register) the component in your workspace
print(
    f"Component {train_component.name} with Version
{train_component.version} is registered"
)
```

Create the pipeline from components

Now that both your components are defined and registered, you can start implementing the pipeline.

Here, you'll use *input data*, *split ratio* and *registered model name* as input variables. Then call the components and connect them via their inputs /outputs identifiers. The outputs of each step can be accessed via the `.outputs` property.

The python functions returned by `load_component()` work as any regular python function that we'll use within a pipeline to call each step.

To code the pipeline, we use a specific `@dsl.pipeline` decorator that identifies ~~the Azure~~ ML pipelines. In the decorator, we can specify the pipeline description and default resources like compute (serverless is used here) and storage. Like a python function, pipelines can have inputs, you can then create multiple instances of a single pipeline with different inputs.

Here, we used *input data*, *split ratio* and *registered model name* as input variables. We then call the components and connect them via their inputs /outputs identifiers. The outputs of each step can be accessed via the `.outputs` property.

```python
# the dsl decorator tells the sdk that we are defining an Azure ML
pipeline
from azure.ai.ml import dsl, Input, Output


@dsl.pipeline(
    compute="serverless",
    description="E2E data_perp-train pipeline",
)
def credit_defaults_pipeline(
    pipeline_job_data_input,
    pipeline_job_test_train_ratio,
    pipeline_job_learning_rate,
    pipeline_job_registered_model_name,
):
    # using data_prep_function like a python call with its own inputs
    data_prep_job = data_prep_component(
        data=pipeline_job_data_input,
        test_train_ratio=pipeline_job_test_train_ratio,
    )

    # using train_func like a python call with its own inputs
    train_job = train_component(
        train_data=data_prep_job.outputs.train_data,  # note: using
outputs from previous step
        test_data=data_prep_job.outputs.test_data,  # note: using
outputs from previous step
        learning_rate=pipeline_job_learning_rate,  # note: using a
pipeline input as parameter
        registered_model_name=pipeline_job_registered_model_name,
    )

    # a pipeline returns a dictionary of outputs
    # keys will code for the pipeline output identifier
    return {
        "pipeline_job_train_data": data_prep_job.outputs.train_data,
        "pipeline_job_test_data": data_prep_job.outputs.test_data,
    }
```

Now use your pipeline definition to instantiate a pipeline with your dataset, split rate of choice and the name you picked for your model.

```
registered_model_name = "credit_defaults_model"

# Let's instantiate the pipeline with the parameters of our choice
pipeline = credit_defaults_pipeline(
    pipeline_job_data_input=Input(type="uri_file",
path=credit_data.path),
    pipeline_job_test_train_ratio=0.25,
    pipeline_job_learning_rate=0.05,
    pipeline_job_registered_model_name=registered_model_name,
)
```

Submit the job

It's now time to submit the job to run in Azure ML. This time you'll use `create_or_update` on `ml_client.jobs`.

Here you'll also pass an experiment name. An experiment is a container for all the iterations one does on a certain project. All the jobs submitted under the same experiment name would be listed next to each other in Azure ML studio.

Once completed, the pipeline will register a model in your workspace as a result of training.

```
import webbrowser

# submit the pipeline job
pipeline_job = ml_client.jobs.create_or_update(
    pipeline,
    # Project's name
    experiment_name="e2e_registered_components",
)
# open the pipeline in web browser
webbrowser.open(pipeline_job.studio_url)
```

You can track the progress of your pipeline, by using the link generated in the cell above or in this notebook using the following code:

```
    ml_client.jobs.stream(pipeline_job.name)
```

When you select on each component, you'll see more information about the results of that component. There are two important parts to look for at this stage:

- `Outputs+logs > user_logs > std_log.txt` This section shows the script run sdtout.
- `Outputs+logs > Metric` This section shows different logged metrics. In this example. mlflow `autologging`, has automatically logged the training metrics.

Deploy the model as an online endpoint

Now deploy your machine learning model as a web service in the Azure cloud, an `online endpoint`.

To deploy a machine learning service, you usually need:

- The model assets (filed, metadata) that you want to deploy. You've already registered these assets in your training component.
- Some code to run as a service. The code executes the model on a given input request. This entry script receives data submitted to a deployed web service and passes it to the model, then returns the model's response to the client. The script is specific to your model. The entry script must understand the data that the model expects and returns. When using a MLFlow model, as in this tutorial, this script is automatically created for you. Samples of scoring scripts can be found here.

Create a new online endpoint

Now that you have a registered model and an inference script, it's time to create your online endpoint. The endpoint name needs to be unique in the entire Azure region. For this tutorial, you'll create a unique name using [UUID](https://en.wikipedia.org/wiki/Universally_unique_identifier#:~:text=A%20universally%20unique%20identifier%20(UUID,%2C%20for%20practical%20purposes%2C%20unique.).

```
import uuid

# Creating a unique name for the endpoint
online_endpoint_name = "credit-endpoint-" + str(uuid.uuid4())[:8]
```

```
from azure.ai.ml.entities import (
    ManagedOnlineEndpoint,
    ManagedOnlineDeployment,
    Model,
    Environment,
)

# create an online endpoint
endpoint = ManagedOnlineEndpoint(
    name=online_endpoint_name,
    description="this is an online endpoint",
    auth_mode="key",
    tags={
        "training_dataset": "credit_defaults",
        "model_type": "sklearn.GradientBoostingClassifier",
    },
)

endpoint_result = ml_client.begin_create_or_update(endpoint).result()

print(
    f"Endpint {endpoint_result.name} provisioning state: 
{endpoint_result.provisioning_state}"
)
```

Once you've created an endpoint, you can retrieve it as below:

```
endpoint = ml_client.online_endpoints.get(name=online_endpoint_name)

print(
    f'Endpint "{endpoint.name}" with provisioning state 
"{endpoint.provisioning_state}" is retrieved'
)
```

Deploy the model to the endpoint

Once the endpoint is created, deploy the model with the entry script. Each endpoint can have multiple deployments and direct traffic to these deployments can be specified using rules. Here you'll create a single deployment that handles 100% of the incoming traffic. We have chosen a color name for the deployment, for example, *blue*, *green*, *red* deployments, which is arbitrary.

You can check the *Models* page on the Azure ML studio, to identify the latest version of your registered model. Alternatively, the code below will retrieve the latest version number for you to use.

```
# Let's pick the latest version of the model
latest_model_version = max(
    [int(m.version) for m in
ml_client.models.list(name=registered_model_name)]
)
```

Deploy the latest version of the model.

> [!NOTE] Expect this deployment to take approximately 6 to 8 minutes.

```
# picking the model to deploy. Here we use the latest version of our
registered model
model = ml_client.models.get(name=registered_model_name,
version=latest_model_version)


# create an online deployment.
blue_deployment = ManagedOnlineDeployment(
    name="blue",
    endpoint_name=online_endpoint_name,
    model=model,
    instance_type="Standard_F4s_v2",
    instance_count=1,
)

blue_deployment_results =
ml_client.online_deployments.begin_create_or_update(
    blue_deployment
).result()

print(
    f"Deployment {blue_deployment_results.name} provisioning state:
{blue_deployment_results.provisioning_state}"
)
```

Test with a sample query

Now that the model is deployed to the endpoint, you can run inference with it.

Create a sample request file following the design expected in the run method in the score script.

```
deploy_dir = "./deploy"
os.makedirs(deploy_dir, exist_ok=True)
```

```
%%writefile {deploy_dir}/sample-request.json
{
  "input_data": {
    "columns":
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22],
    "index": [0, 1],
    "data": [
            [20000,2,2,1,24,2,2,-1,-1,-2,-
2,3913,3102,689,0,0,0,0,689,0,0,0,0],
            [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1, 10, 9, 8]
        ]
  }
}
```

```
# test the blue deployment with some sample data
ml_client.online_endpoints.invoke(
    endpoint_name=online_endpoint_name,
    request_file="./deploy/sample-request.json",
    deployment_name="blue",
)
```

Clean up resources

If you're not going to use the endpoint, delete it to stop using the resource. Make sure no other deployments are using an endpoint before you delete it.

> [!NOTE] Expect this step to take approximately 6 to 8 minutes.

```
ml_client.online_endpoints.begin_delete(name=online_endpoint_name)
```

Next Steps

Learn more about Azure ML logging.