Introduction

Completed100 XP

- 2 minutes

After training, you want to deploy a machine learning model in order to integrate the model with an application. In Azure Machine Learning, you can easily deploy a model to a batch or online endpoint when you register the model with **MLflow**.

Imagine you're a data scientist, working for a company that creates an application for health care practitioners to help diagnose diabetes in patients. The practitioners enter a patient's medical information and expect a response from the application, indicating whether a patient is likely to have diabetes or not.

You expect to regularly retrain the model that predicts diabetes. Whenever you have more training data, you want to retrain the model to produce a better performing model. Every time the model is retrained, you want to update the model that is deployed to the endpoint and integrated with the application. By doing so, you're providing the practitioners with the latest version of the model anytime they use the application.

Log models with MLflow

Completed100 XP

- 9 minutes

To train a machine learning model, you may choose to use an open source framework that best suits your needs. After training, you want to deploy your model. **MLflow** is an open source platform that streamlines machine learning deployment, regardless of the type of model you trained and the framework you used.

MLflow is integrated with Azure Machine Learning. The integration with Azure Machine Learning allows you to easily deploy models that you train and track with Mlflow. For example, when you have an MLflow model, you can opt for the no-code deployment in Azure Machine Learning.

Note

Some types of models are currently not supported by Azure Machine Learning and MLflow. In that case, you can register a `custom` model. Learn more about **how to work with (custom) models in Azure Machine Learning**.

Why use MLflow?

When you train a machine learning model with Azure Machine Learning, you can use MLflow to register your model. MLflow standardizes the packaging of models, which means that an MLflow model can easily be imported or exported across different workflows.

For example, imagine training a model in an Azure Machine Learning workspace used for development. If you want to export the model to another workspace used for production, you can use an MLflow model to easily do so.

When you train and log a model, you store all relevant artifacts in a directory. When you register the model, an `MLmodel` file is created in that directory. The `MLmodel` file contains the model's metadata, which allows for model traceability.

You can register models with MLflow by enabling autologging, or by using custom logging.

Note

MLflow allows you to log a model as an artifact, or as a model. When you log a model as an artifact, the model is treated as a file. When you log a model as a model, you're adding information to the registered model that enables you to use the model directly in pipelines or deployments. Learn more about **the difference between an artifact and a model**

Use autologging to log a model

When you train a model, you can include `mlflow.autolog()` to enable autologging. MLflow's autologging automatically logs parameters, metrics, artifacts, and the model you train. The model is logged when the `.fit()` method is called. The framework you use to train your model is identified and included as the **flavor** of your model.

Optionally, you can specify which flavor you want your model to be identified as by using `mlflow.<flavor>.autolog()`. Some common flavors that you can use with autologging are:

- Keras: `mlflow.keras.autolog`
- Scikit-learn: `mlflow.sklearn.autolog()`
- LightGBM: `mlflow.lightgbm.autolog`
- XGBoost: `mlflow.xgboost.autolog`
- TensorFlow: `mlflow.tensorflow.autolog`
- PyTorch: `mlflow.pytorch.autolog`

- ONNX: `mlflow.onnx.autolog`

Tip

Explore the complete list of **MLflow's built-in model flavors**.

When you use autologging, an output folder is created which includes all necessary model artifacts, including the `MLmodel` file that references these files and includes the model's metadata.

Manually log a model

When you want to have more control over how the model is logged, you can use `autolog` (for your parameters, metrics, and other artifacts), and set `log_models=False`. MLflow doesn't automatically log the model, and you can add it manually.

As logging the model allows you to easily deploy the model, you may want to customize the model's expected inputs and outputs. The schemas of the expected inputs and outputs are defined as the signature in the `MLmodel` file. If you deploy your model and the inputs don't match the defined schema in the signature, you may encounter errors.

Therefore, you may want to customize the signature to alter the deployed model's behavior.

Customize the signature

The model signature defines the schema of the model's inputs and outputs. The signature is stored in JSON format in the `MLmodel` file, together with other metadata of the model.

The model signature can be inferred from datasets or created manually by hand.

To log a model with a signature that is inferred from your training dataset and model predictions, you can use `infer_signature()`. For example, the following example takes the training dataset to infer the schema of the inputs, and the model's predictions to infer the schema of the output:

PythonCopy

```python
import pandas as pd
from sklearn import datasets
from sklearn.ensemble import RandomForestClassifier
import mlflow
import mlflow.sklearn
```

```python
from mlflow.models.signature import infer_signature

iris = datasets.load_iris()
iris_train = pd.DataFrame(iris.data, columns=iris.feature_names)
clf = RandomForestClassifier(max_depth=7, random_state=0)
clf.fit(iris_train, iris.target)

# Infer the signature from the training dataset and model's predictions
signature = infer_signature(iris_train, clf.predict(iris_train))

# Log the scikit-learn model with the custom signature
mlflow.sklearn.log_model(clf, "iris_rf", signature=signature)
```

Alternatively, you can create the signature manually:

PythonCopy

```python
from mlflow.models.signature import ModelSignature
from mlflow.types.schema import Schema, ColSpec

# Define the schema for the input data
input_schema = Schema([
  ColSpec("double", "sepal length (cm)"),
  ColSpec("double", "sepal width (cm)"),
  ColSpec("double", "petal length (cm)"),
  ColSpec("double", "petal width (cm)"),
])

# Define the schema for the output data
output_schema = Schema([ColSpec("long")])

# Create the signature object
signature = ModelSignature(inputs=input_schema, outputs=output_schema)
```

Logging MLflow models

- Article
- 02/16/2024
- 10 contributors

Feedback

In this article

This article describes how to log your trained models (or artifacts) as MLflow models. It explores the different ways to customize how MLflow packages your models, and how it runs those models.

Why logging models instead of artifacts?

From artifacts to models in MLflow describes the difference between logging artifacts or files, as compared to logging MLflow models.

An MLflow model is also an artifact. However, that model has a specific structure that serves as a contract between the person that created the model and the person that intends to use it. This contract helps build a bridge between the artifacts themselves and their meanings.

Model logging has these advantages:

- You can directly load models, for inference, with `mlflow.<flavor>.load_model`, and you can use the `predict` function
- Pipeline inputs can use models directly
- You can deploy models without indication of a scoring script or an environment
- Swagger is automatically enabled in deployed endpoints, and the Azure Machine Learning studio can use the **Test** feature
- You can use the Responsible AI dashboard

This section describes how to use the model's concept in Azure Machine Learning with MLflow:

Logging models using autolog

You can use MLflow autolog functionality. Autolog allows MLflow to instruct the framework in use to log all the metrics, parameters, artifacts, and models that the framework considers relevant. By default, if autolog is enabled, most models are logged. In some situations, some flavors might not log a model. For instance, the PySpark flavor doesn't log models that exceed a certain size.

Use either `mlflow.autolog()` or `mlflow.<flavor>.autolog()` to activate autologging. This example uses `autolog()` to log a classifier model trained with XGBoost:

PythonCopy

```python
import mlflow
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score

mlflow.autolog()

model = XGBClassifier(use_label_encoder=False, eval_metric="logloss")
model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

Tip

If use Machine Learning pipelines, for example **Scikit-Learn pipelines**, use the `autolog` functionality of that pipeline flavor to log models. Model logging automatically happens when the `fit()` method is called on the pipeline object. The **Training and tracking an XGBoost classifier with MLflow notebook** demonstrates how to log a model with preprocessing, using pipelines.

Logging models with a custom signature, environment or samples

The MLflow `mlflow.<flavor>.log_model` method can manually log models. This workflow can control different aspects of the model logging.

Use this method when:

- You want to indicate pip packages or a conda environment that differ from those that are automatically detected
- You want to include input examples
- You want to include specific artifacts in the needed package
- `autolog` does not correctly infer your signature. This matters when you deal with tensor inputs, where the signature needs specific shapes
- The autolog behavior does not cover your purpose for some reason

This code example logs a model for an XGBoost classifier:

PythonCopy

```python
import mlflow
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score
from mlflow.models import infer_signature
from mlflow.utils.environment import _mlflow_conda_env

mlflow.autolog(log_models=False)

model = XGBClassifier(use_label_encoder=False, eval_metric="logloss")
model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

# Signature
signature = infer_signature(X_test, y_test)

# Conda environment
custom_env =_mlflow_conda_env(
    additional_conda_deps=None,
    additional_pip_deps=["xgboost==1.5.2"],
    additional_conda_channels=None,
)
```

```python
# Sample
input_example = X_train.sample(n=1)

# Log the model manually
mlflow.xgboost.log_model(model,
                         artifact_path="classifier",
                         conda_env=custom_env,
                         signature=signature,
                         input_example=input_example)
```

Note

- `autolog` has the `log_models=False` configuration. This prevents automatic MLflow model logging. Automatic MLflow model logging happens later, as a manual process
- Use the `infer_signature` method to try to infer the signature directly from inputs and outputs
- The `mlflow.utils.environment._mlflow_conda_env` method is a private method in the MLflow SDK. In this example, it makes the code simpler, but use it with caution. It may change in the future. As an alternative, you can generate the YAML definition manually as a Python dictionary.

Logging models with a different behavior in the predict method

When logging a model with either `mlflow.autolog` or `mlflow.<flavor>.log_model`, the model flavor determines how to execute the inference, and what the model returns. MLflow doesn't enforce any specific behavior about the generation of `predict` results. In some scenarios, you might want to do some preprocessing or post-processing before and after your model executes.

In this situation, implement machine learning pipelines that directly move from inputs to outputs. Although this implementation is possible, and sometimes encouraged to improve performance, it might become challenging to achieve. In those cases, it can help to customize how your model handles inference as explained in next section.

Logging custom models

MLflow supports many machine learning frameworks, including

- CatBoost
- FastAI
- h2o
- Keras
- LightGBM
- MLeap
- MXNet Gluon
- ONNX
- Prophet

- PyTorch
- Scikit-Learn
- spaCy
- Spark MLLib
- statsmodels
- TensorFlow
- XGBoost

However, you might need to change the way a flavor works, log a model not natively supported by MLflow or even log a model that uses multiple elements from different frameworks. In these cases, you might need to create a custom model flavor.

To solve the problem, MLflow introduces the `pyfunc` flavor (starting from a Python function). This flavor can log any object as a model, as long as that object satisfies two conditions:

- You implement the method `predict` method, at least
- The Python object inherits from `mlflow.pyfunc.PythonModel`

Tip

Serializable models that implement the Scikit-learn API can use the Scikit-learn flavor to log the model, regardless of whether the model was built with Scikit-learn. If you can persist your model in Pickle format, and the object has the `predict()` and `predict_proba()` methods (at least), you can use `mlflow.sklearn.log_model()` to log the model inside a MLflow run.

- Using a model wrapper
- Using artifacts
- Using a model loader

Your model might be composed of multiple pieces that need to be loaded. You might not have a way to serialize it as a Pickle file. In those cases, the `PythonModel` supports indication of an arbitrary list of **artifacts**. Each artifact is packaged along with your model.

Use this technique when:

- You can't serialize your model in Pickle format, or you have a better serialization format available
- Your model has one, or many, artifacts must be referenced to load the model
- You might want to persist some inference configuration properties - for example, the number of items to recommend
- You want to customize the way the model loads, and how the `predict` function works

This code sample shows how to log a custom model, using artifacts:

PythonCopy

```python
encoder_path = 'encoder.pkl'
joblib.dump(encoder, encoder_path)

model_path = 'xgb.model'
model.save_model(model_path)

mlflow.pyfunc.log_model("classifier",
                        python_model=ModelWrapper(),
                        artifacts={
                            'encoder': encoder_path,
                            'model': model_path
                        },
                        signature=signature)
```

Note

- The model is not saved as a pickle. Instead, the code saved the model with save method of the framework that you used
- The model wrapper is `ModelWrapper()`, but the model is not passed as a parameter to the constructor A new dictionary parameter - `artifacts` - has keys as the artifact names, and values as the path in the local file system where the artifact is stored

The corresponding model wrapper then would look like this:

PythonCopy

```python
from mlflow.pyfunc import PythonModel, PythonModelContext

class ModelWrapper(PythonModel):
    def load_context(self, context: PythonModelContext):
        import pickle
        from xgboost import XGBClassifier
        from sklearn.preprocessing import OrdinalEncoder

        self._encoder = pickle.loads(context.artifacts["encoder"])
        self._model = XGBClassifier(use_label_encoder=False,
eval_metric="logloss")
        self._model.load_model(context.artifacts["model"])

    def predict(self, context: PythonModelContext, data):
        return self._model.predict_proba(data)
```

The complete training routine would look like this:

PythonCopy

```python
import mlflow
from xgboost import XGBClassifier
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import accuracy_score
from mlflow.models import infer_signature

mlflow.xgboost.autolog(log_models=False)

encoder = OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=np.nan)
X_train['thal'] = encoder.fit_transform(X_train['thal'].to_frame())
```

```
X_test['thal'] = encoder.transform(X_test['thal'].to_frame())

model = XGBClassifier(use_label_encoder=False, eval_metric="logloss")
model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)
y_probs = model.predict_proba(X_test)

accuracy = accuracy_score(y_test, y_probs.argmax(axis=1))
mlflow.log_metric("accuracy", accuracy)

encoder_path = 'encoder.pkl'
joblib.dump(encoder, encoder_path)
model_path = "xgb.model"
model.save_model(model_path)

signature = infer_signature(X, y_probs)
mlflow.pyfunc.log_model("classifier",
                        python_model=ModelWrapper(),
                        artifacts={
                            'encoder': encoder_path,
                            'model': model_path
                        },
                        signature=signature)
```

Understand the MLflow model format

Completed100 XP

- 8 minutes

MLflow uses the MLModel format to store all relevant model assets in a folder or directory. One essential file in the directory is the `MLmodel` file. The `MLmodel` file is the single source of truth about how the model should be loaded and used.

Explore the MLmodel file format

The `MLmodel` file may include:

- `artifact_path`: During the training job, the model is logged to this path.
- `flavor`: The machine learning library with which the model was created.
- `model_uuid`: The unique identifier of the registered model.
- `run_id`: The unique identifier of job run during which the model was created.
- `signature`: Specifies the schema of the model's inputs and outputs:
  - `inputs`: Valid input to the model. For example, a subset of the training dataset.
  - `outputs`: Valid model output. For example, model predictions for the input dataset.

An example of a MLmodel file created for a computer vision model trained with `fastai` may look like:

ymlCopy

```yml
artifact_path: classifier
```

```yaml
flavors:
  fastai:
    data: model.fastai
    fastai_version: 2.4.1
  python_function:
    data: model.fastai
    env: conda.yaml
    loader_module: mlflow.fastai
    python_version: 3.8.12
model_uuid: e694c68eba484299976b06ab9058f636
run_id: e13da8ac-b1e6-45d4-a9b2-6a0a5cfac537
signature:
  inputs: '[{"type": "tensor",
             "tensor-spec": 
                 {"dtype": "uint8", "shape": [-1, 300, 300, 3]}
           }]'
  outputs: '[{"type": "tensor", 
              "tensor-spec": 
                 {"dtype": "float32", "shape": [-1,2]}
            }]'
```

The most important things to set are the **flavor** and the **signature**.

Choose the flavor

A **flavor** is the machine learning library with which the model was created.

For example, to create an image classification model to detect breast cancer you're using `fastai`. Fastai is a flavor in MLflow that tells you how a model should be persisted and loaded. Because each model flavor indicates how they want to persist and load models, the MLModel format doesn't enforce a single serialization mechanism that all the models need to support. Such a decision allows each flavor to use the methods that provide the best performance or best support according to their best practices - without compromising compatibility with the MLModel standard.

`Python function` flavor is the *default* model interface for models created from an MLflow run. Any MLflow python model can be loaded as a `python_function` model, which allows for workflows like deployment to work with any python model regardless of which framework was used to produce the model. This interoperability is immensely powerful as it reduces the time to operationalize in multiple environments.

An example of the Python function flavor may look like:

ymlCopy

```yaml
artifact_path: pipeline
```

```yaml
flavors:
  python_function:
    env:
      conda: conda.yaml
      virtualenv: python_env.yaml
    loader_module: mlflow.sklearn
    model_path: model.pkl
    predict_fn: predict
    python_version: 3.8.5
  sklearn:
    code: null
    pickled_model: model.pkl
    serialization_format: cloudpickle
    sklearn_version: 1.2.0
mlflow_version: 2.1.0
model_uuid: b8f9fe56972e48f2b8c958a3afb9c85d
run_id: 596d2e7a-c7ed-4596-a4d2-a30755c0bfa5
signature:
  inputs: '[{"name": "age", "type": "long"}, {"name": "sex", "type": "long"},
{"name":
    "cp", "type": "long"}, {"name": "trestbps", "type": "long"}, {"name": "chol",
    "type": "long"}, {"name": "fbs", "type": "long"}, {"name": "restecg", "type":
    "long"}, {"name": "thalach", "type": "long"}, {"name": "exang", "type":
"long"},
    {"name": "oldpeak", "type": "double"}, {"name": "slope", "type": "long"},
{"name":
    "ca", "type": "long"}, {"name": "thal", "type": "string"}]'
  outputs: '[{"name": "target", "type": "long"}]'
```

Configure the signature

Apart from flavors, the `MLmodel` file also contains signatures that serve as data contracts between the model and the server running your model.

There are two types of signatures:

- **Column-based**: used for tabular data with a `pandas.Dataframe` as inputs.
- **Tensor-based**: used for n-dimensional arrays or tensors (often used for unstructured data like text or images), with `numpy.ndarray` as inputs.

As the `MLmodel` file is created when you register the model, the signature also is created when you register the model. When you enable MLflow's autologging, the signature is inferred in the best effort way. If you want the signature to be different, you need to manually log the model.

The signature's inputs and outputs are important when deploying your model. When you use Azure Machine Learning's no-code deployment for MLflow models, the inputs and outputs set in the signature will be enforced. In other words, when you send data to a deployed MLflow model, the expected inputs and outputs need to match the schema as defined in the signature.

Absolutely!

When you're deploying a machine learning model, you often have different requirements based on where you're deploying it. For instance:

1. **Server Deployment**: When deploying to a server, you might want your model to be represented as a Python function. This makes it easier to integrate with existing server-side code or web frameworks. The "python_function" flavor in MLflow allows you to save your model as a Python function along with any associated preprocessing or postprocessing code.

2. **Mobile App Deployment**: On the other hand, if you're deploying to a mobile app, you might need a format that's optimized for mobile devices and frameworks commonly used in mobile development, such as TensorFlow or PyTorch. MLflow supports flavors like "tensorflow" or "pytorch" for this purpose.

Now, let's focus on the "python_function" flavor:

- **Use of Python Function Flavor**: This flavor is useful when you want to deploy your model as a simple Python function. It's handy for server deployment where you may have existing infrastructure or APIs that expect functions as inputs. The Python function can encapsulate the logic of your model along with any preprocessing or postprocessing steps.

For example, let's say you've trained a scikit-learn model to predict housing prices. Using the "python_function" flavor, you can save your trained model as a Python function that takes input features and returns predictions. This function can then be easily integrated into your server-side code or deployed as a standalone microservice.

In essence, the "python_function" flavor in MLflow allows you to persist and load your model in a format that's convenient for server deployment, leveraging the flexibility and simplicity of Python functions.

Register an MLflow model

Completed100 XP

- 7 minutes

In Azure Machine Learning, models are trained in jobs. When you want to find the model's artifacts, you can find it in the job's outputs. To more easily manage your models, you can also store a model in the Azure Machine Learning **model registry**.

The model registry makes it easy to organize and keep track of your trained models. When you register a model, you store and version your model in the workspace.

Registered models are identified by name and version. Each time you register a model with the same name as an existing one, the registry increments the version. You can also add more metadata tags to more easily search for a specific model.

Tip

You can also register models trained outside Azure Machine Learning by providing the local path to the model's artifacts.

There are three types of models you can register:

- **MLflow**: Model trained and tracked with MLflow. Recommended for standard use cases.
- **Custom**: Model type with a custom standard not currently supported by Azure Machine Learning.
- **Triton**: Model type for deep learning workloads. Commonly used for TensorFlow and PyTorch model deployments.

Tip

Learn more about **Deploy deep learning workloads to production with Azure Machine Learning**

Azure Machine Learning integrates well with MLflow, which is why it's a best practice to log and register an MLflow model. Working with MLflow models makes model management and deployment in Azure Machine Learning easier. During deployment, for example, the environment and scoring script are created for you when using an MLflow model.

Register an MLflow model

To register an MLflow model, you can use the studio, the Azure CLI, or the Python SDK.

As a data scientist, you may be most comfortable working with the Python SDK.

To train the model, you can submit a training script as a command job by using the following code:

PythonCopy

```python
from azure.ai.ml import command

# configure job
```

```
job = command(
    code="./src",
    command="python train-model-signature.py --training_data diabetes.csv",
    environment="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest",
    compute="aml-cluster",
    display_name="diabetes-train-signature",
    experiment_name="diabetes-training"
    )

# submit job
returned_job = ml_client.create_or_update(job)
aml_url = returned_job.studio_url
print("Monitor your job at", aml_url)
```

Once the job is completed and the model is trained, use the job name to find the job run and register the model from its outputs.

PythonCopy

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

job_name = returned_job.name

run_model = Model(
    path=f"azureml://jobs/{job_name}/outputs/artifacts/paths/model/",
    name="mlflow-diabetes",
    description="Model created from run.",
    type=AssetTypes.MLFLOW_MODEL,
)
# Uncomment after adding required details above
ml_client.models.create_or_update(run_model)
```

All registered models are listed in the **Models** page of the Azure Machine Learning studio. The registered model includes the model's output directory. When you log and register an MLflow model, you can find the `MLmodel` file in the artifacts of the registered model.

2.

A data scientist trained a deep learning model with TensorFlow. The deployed model is compute-intensive and needs to use the most optimal inference server for similar workloads. Which model type is compatible with compute-intensive and no-code deployments?

○

○

Triton

○

Explore managed online endpoints

Completed100 XP

- 9 minutes

To make a machine learning model available for other applications, you can deploy the model to a managed online endpoint.

You'll learn how to use managed online endpoints for real-time predictions.

Real-time predictions

To get real-time predictions, you can deploy a model to an endpoint. An **endpoint** is an HTTPS endpoint to which you can send data, and which will return a response (almost) immediately.

Any data you send to the endpoint will serve as the input for the scoring script hosted on the endpoint. The scoring script loads the trained model to predict the label for the new input data, which is also called **inferencing**. The label is then part of the output that's returned.

Managed online endpoint

Within Azure Machine Learning, there are two types of online endpoints:

- **Managed online endpoints**: Azure Machine Learning manages all the underlying infrastructure.
- **Kubernetes online endpoints**: Users manage the Kubernetes cluster which provides the necessary infrastructure.

As a data scientist, you may prefer to work with managed online endpoints to test whether your model works as expected when deployed. If a Kubernetes online endpoint is required for control and scalability, it'll likely be managed by other teams.

If you're using a managed online endpoint, you only need to specify the virtual machine (VM) type and scaling settings. Everything else, such as provisioning compute power and updating the host operating system (OS) is done for you automatically.

Deploy your model

After you create an endpoint in the Azure Machine Learning workspace, you can deploy a model to that endpoint. To deploy your model to a managed online endpoint, you need to specify four things:

- **Model assets** like the model pickle file, or a **registered model** in the Azure Machine Learning workspace.
- **Scoring script** that loads the model.
- **Environment** which lists all necessary packages that need to be installed on the compute of the endpoint.
- **Compute configuration** including the needed **compute size** and **scale settings** to ensure you can handle the amount of requests the endpoint will receive.

Important

When you deploy MLFlow models to an online endpoint, you don't need to provide a scoring script and environment, as both are automatically generated.

All of these elements are defined in the deployment. The deployment is essentially a set of resources needed to host the model that performs the actual inferencing.

Blue/green deployment

One endpoint can have multiple deployments. One approach is the **blue/green deployment**.

Let's take the example of the restaurant recommender model. After experimentation, you select the best performing model. You use the blue deployment for this first version of the model. As new data is collected, the model can be retrained, and a new version is registered in the Azure Machine Learning workspace. To test the new model, you can use the green deployment for the second version of the model.

Both versions of the model are deployed to the same endpoint, which is integrated with the application. Within the application, a user selects a restaurant, sending a request to the endpoint to get new real-time recommendations of other restaurants the user may like.

When a request is sent to the endpoint, 90% of the traffic can go to the blue deployment*, and 10% of the traffic can go to the *green deployment*. With two versions of the model deployed on the same endpoint, you can easily test the model.

After testing, you can also seamlessly transition to the new version of the model by redirecting 90% of the traffic to the green deployment. If it turns out that the new version doesn't perform better, you can easily roll back to the first version of the model by redirecting most of the traffic back to the blue deployment.

Blue/green deployment allows for multiple models to be deployed to an endpoint. You can decide how much traffic to forward to each deployed model. This way, you can switch to a new version of the model without interrupting service to the consumer.

Tip

Learn more about **safe rollout for online endpoints**

Create an endpoint

To create an online endpoint, you'll use the `ManagedOnlineEndpoint` class, which requires the following parameters:

- `name`: Name of the endpoint. Must be unique in the Azure region.
- `auth_mode`: Use `key` for key-based authentication. Use `aml_token` for Azure Machine Learning token-based authentication.

To create an endpoint, use the following command:

PythonCopy

```python
from azure.ai.ml.entities import ManagedOnlineEndpoint

# create an online endpoint
endpoint = ManagedOnlineEndpoint(
    name="endpoint-example",
    description="Online endpoint",
    auth_mode="key",
)

ml_client.begin_create_or_update(endpoint).result()
```

Deploy your MLflow model to a managed online endpoint

Completed100 XP

- 6 minutes

The easiest way to deploy a model to an online endpoint is to use an **MLflow** model and deploy it to a *managed* online endpoint. Azure Machine Learning will automatically generate the scoring script and environment for MLflow models.

To deploy an MLflow model, you need to have created an endpoint. Then you can deploy the model to the endpoint.

Deploy an MLflow model to an endpoint

When you deploy an MLflow model to a managed online endpoint, you don´t need to have the scoring script and environment.

To deploy an MLflow model, you must have model files stored on a local path or with a registered model. You can log model files when training a model by using MLflow tracking.

In this example, we're taking the model files from a local path. The files are all stored in a local folder called `model`. The folder must include the `MLmodel` file, which describes how the model can be loaded and used.

Tip

Learn more about **the MLmodel format**.

Next to the model, you also need to specify the compute configuration for the deployment:

- `instance_type`: Virtual machine (VM) size to use. Review the list of supported sizes.
- `instance_count`: Number of instances to use.

To deploy (and automatically register) the model, run the following command:

PythonCopy

```python
from azure.ai.ml.entities import Model, ManagedOnlineDeployment
from azure.ai.ml.constants import AssetTypes

# create a blue deployment
model = Model(
    path="./model",
    type=AssetTypes.MLFLOW_MODEL,
    description="my sample mlflow model",
)

blue_deployment = ManagedOnlineDeployment(
    name="blue",
    endpoint_name="endpoint-example",
    model=model,
    instance_type="Standard_F4s_v2",
    instance_count=1,
)

ml_client.online_deployments.begin_create_or_update(blue_deployment).result()
```

Tip

Explore the reference documentation to **create a managed online deployment with the Python SDK v2**.

Since only one model is deployed to the endpoint, you want this model to take 100% of the traffic. When you deploy multiple models to the same endpoint, you can distribute the traffic among the deployed models.

To route traffic to a specific deployment, use the following code:

PythonCopy

```python
# blue deployment takes 100 traffic
endpoint.traffic = {"blue": 100}
ml_client.begin_create_or_update(endpoint).result()
```

To delete the endpoint and all associated deployments, run the command:

PythonCopy

```python
ml_client.online_endpoints.begin_delete(name="endpoint-example")
```

Deploy a model to a managed online endpoint

Completed100 XP

- 8 minutes

You can choose to deploy a model to a managed online endpoint without using the MLflow model format. To deploy a model, you'll need to create the scoring script and define the environment necessary during inferencing.

To deploy a model, you need to have created an endpoint. Then you can deploy the model to the endpoint.

Deploy a model to an endpoint

To deploy a model, you must have:

- Model files stored on local path or registered model.
- A scoring script.
- An execution environment.

The model files can be logged and stored when you train a model.

Create the scoring script

The scoring script needs to include two functions:

- `init()`: Called when the service is initialized.
- `run()`: Called when new data is submitted to the service.

The **init** function is called when the deployment is created or updated, to load and cache the model from the model registry. The **run** function is called for every time the endpoint is invoked, to generate predictions from the input data. The following example Python script shows this pattern:

PythonCopy

```python
import json
import joblib
import numpy as np
import os

# called when the deployment is created or updated
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')
    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

Your deployment requires an execution environment in which to run the scoring script.

You can create an environment with a Docker image with Conda dependencies, or with a Dockerfile.

To create an environment using a base Docker image, you can define the Conda dependencies in a `conda.yml` file:

ymlCopy

```yml
name: basic-env-cpu
channels:
  - conda-forge
```

```
dependencies:
  - python=3.7
  - scikit-learn
  - pandas
  - numpy
  - matplotlib
```

Then, to create the environment, run the following code:

PythonCopy

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

env = Environment(
    image="mcr.microsoft.com/azureml/openmpi3.1.2-ubuntu18.04",
    conda_file="./src/conda.yml",
    name="deployment-environment",
    description="Environment created from a Docker image plus Conda environment.",
)
ml_client.environments.create_or_update(env)
```

Create the deployment

When you have your model files, scoring script, and environment, you can create the deployment.

To deploy a model to an endpoint, you can specify the compute configuration with two parameters:

- `instance_type`: Virtual machine (VM) size to use. Review the list of supported sizes.
- `instance_count`: Number of instances to use.

To deploy the model, use the `ManagedOnlineDeployment` class and run the following command:

PythonCopy

```
from azure.ai.ml.entities import ManagedOnlineDeployment, CodeConfiguration

model = Model(path="./model",

blue_deployment = ManagedOnlineDeployment(
    name="blue",
    endpoint_name="endpoint-example",
    model=model,
    environment="deployment-environment",
    code_configuration=CodeConfiguration(
        code="./src", scoring_script="score.py"
    ),
    instance_type="Standard_DS2_v2",
    instance_count=1,
)
```

```
ml_client.online_deployments.begin_create_or_update(blue_deployment).result()
```

Tip

Explore the reference documentation to **create a managed online deployment with the Python SDK v2**.

You can deploy multiple models to an endpoint. To route traffic to a specific deployment, use the following code:

PythonCopy

```
# blue deployment takes 100 traffic
endpoint.traffic = {"blue": 100}
ml_client.begin_create_or_update(endpoint).result()
```

To delete the endpoint and all associated deployments, run the command:

PythonCopy

```
ml_client.online_endpoints.begin_delete(name="endpoint-example")
```

Test managed online endpoints

Completed100 XP

- 5 minutes

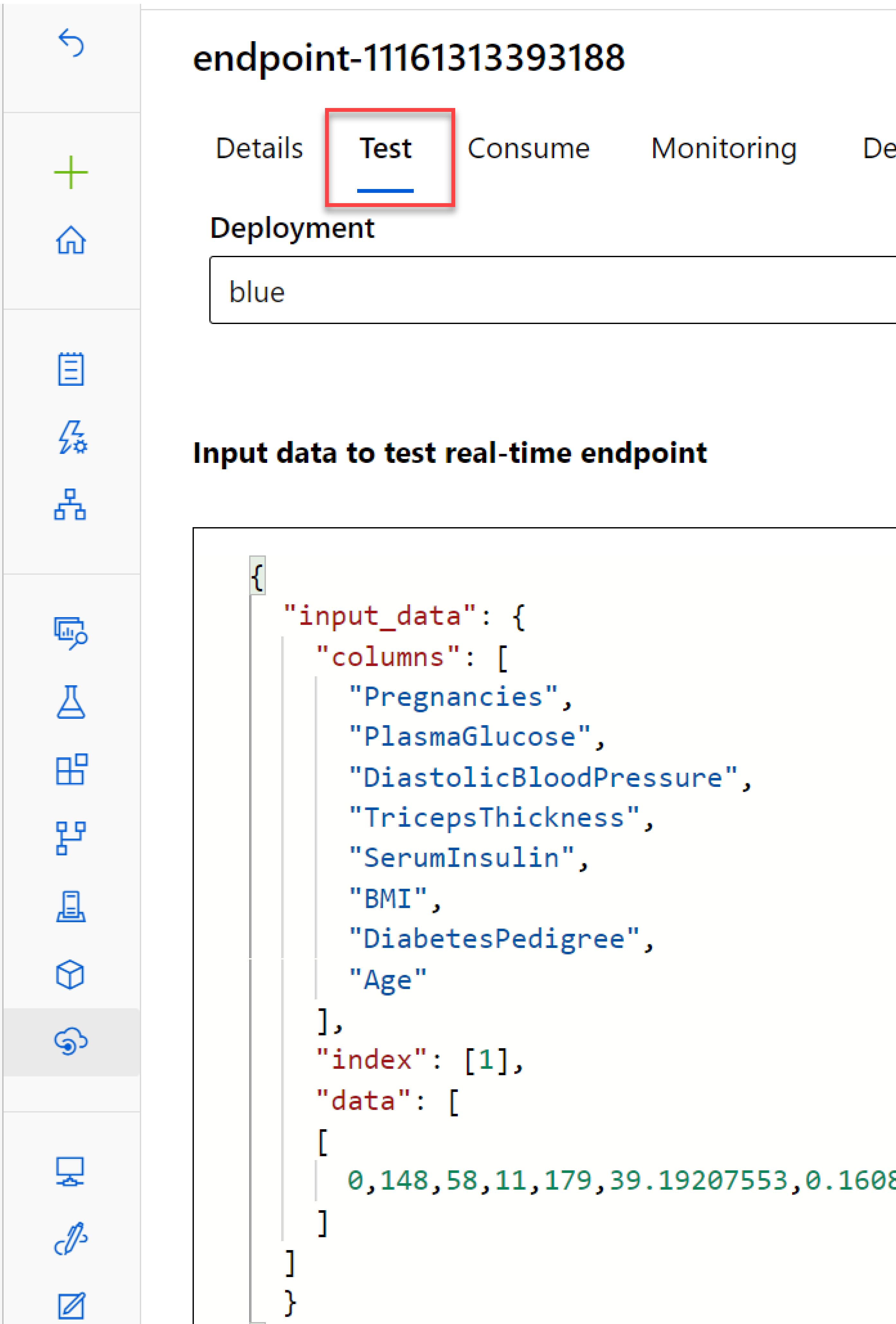
After deploying a real-time service, you can consume it from client applications to predict labels for new data cases.

Use the Azure Machine Learning studio

You can list all endpoints in the Azure Machine Learning studio, by navigating to the **Endpoints** page. In the **Real-time endpoints** tab, all endpoints are shown.

You can select an endpoint to review its details and deployment logs.

Additionally, you can use the studio to test the endpoint.



Deployment

blue

Input data to test real-time endpoint

```
{
  "input_data": {
    "columns": [
      "Pregnancies",
      "PlasmaGlucose",
      "DiastolicBloodPressure",
      "TricepsThickness",
      "SerumInsulin",
      "BMI",
      "DiabetesPedigree",
      "Age"
    ],
    "index": [1],
    "data": [
    [
      0,148,58,11,179,39.19207553,0.1608
    ]
  ]
  }
```

Use the Azure Machine Learning Python SDK

For testing, you can also use the Azure Machine Learning Python SDK to invoke an endpoint.

Typically, you send data to deployed model in JSON format with the following structure:

JSONCopy

```json
{
  "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 deployed model is a JSON collection with a prediction for each case that was submitted in the data. The following code sample invokes an endpoint and displays the response:

PythonCopy

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

if response[1]=='1':
    print("Yes")
else:
    print ("No")
```

1.

You've trained a model using the Python SDK for Azure Machine Learning. You want to deploy the model to get real-time predictions. You want to manage the underlying infrastructure used by the endpoint. What kind of endpoint should you create?

○

○

○

A Kubernetes online endpoint.

Correct. You should use Kubernetes online endpoint if you want to manage the underlying Kubernetes clusters.
2.

You're deploying a model as a real-time inferencing service. What functions must the scoring script for the deployment include?

○

○

○

`init()` and `run()`

Introduction BATCH INFERENCING

Completed100 XP

- 3 minutes

Imagine you trained a model to predict the product sales. The model has been trained and tracked in Azure Machine Learning. Every month, you want to use the model to forecast the sales for the upcoming month.

In many production scenarios, long-running tasks that deal with large amounts of data are performed as **batch** operations. In machine learning, ***batch inferencing*** is used to asynchronously apply a predictive model to multiple cases and write the results to a file or database.

Understand and create batch endpoints

Completed100 XP

- 6 minutes

To get a model to generate batch predictions, you can deploy the model to a batch endpoint.

You'll learn how to use batch endpoints for asynchronous batch scoring.

Batch predictions

To get batch predictions, you can deploy a model to an endpoint. An **endpoint** is an HTTPS endpoint that you can call to trigger a batch scoring job. The advantage of such an endpoint is that you can trigger the batch scoring job from another service, such as Azure Synapse Analytics or Azure Databricks. A batch endpoint allows you to integrate the batch scoring with an existing data ingestion and transformation pipeline.

Whenever the endpoint is invoked, a batch scoring job is submitted to the Azure Machine Learning workspace. The job typically uses a **compute cluster** to score multiple inputs. The results can be stored in a datastore, connected to the Azure Machine Learning workspace.

Create a batch endpoint

To deploy a model to a batch endpoint, you'll first have to create the batch endpoint.

To create a batch endpoint, you'll use the `BatchEndpoint` class. Batch endpoint names need to be unique within an Azure region.

To create an endpoint, use the following command:

PythonCopy

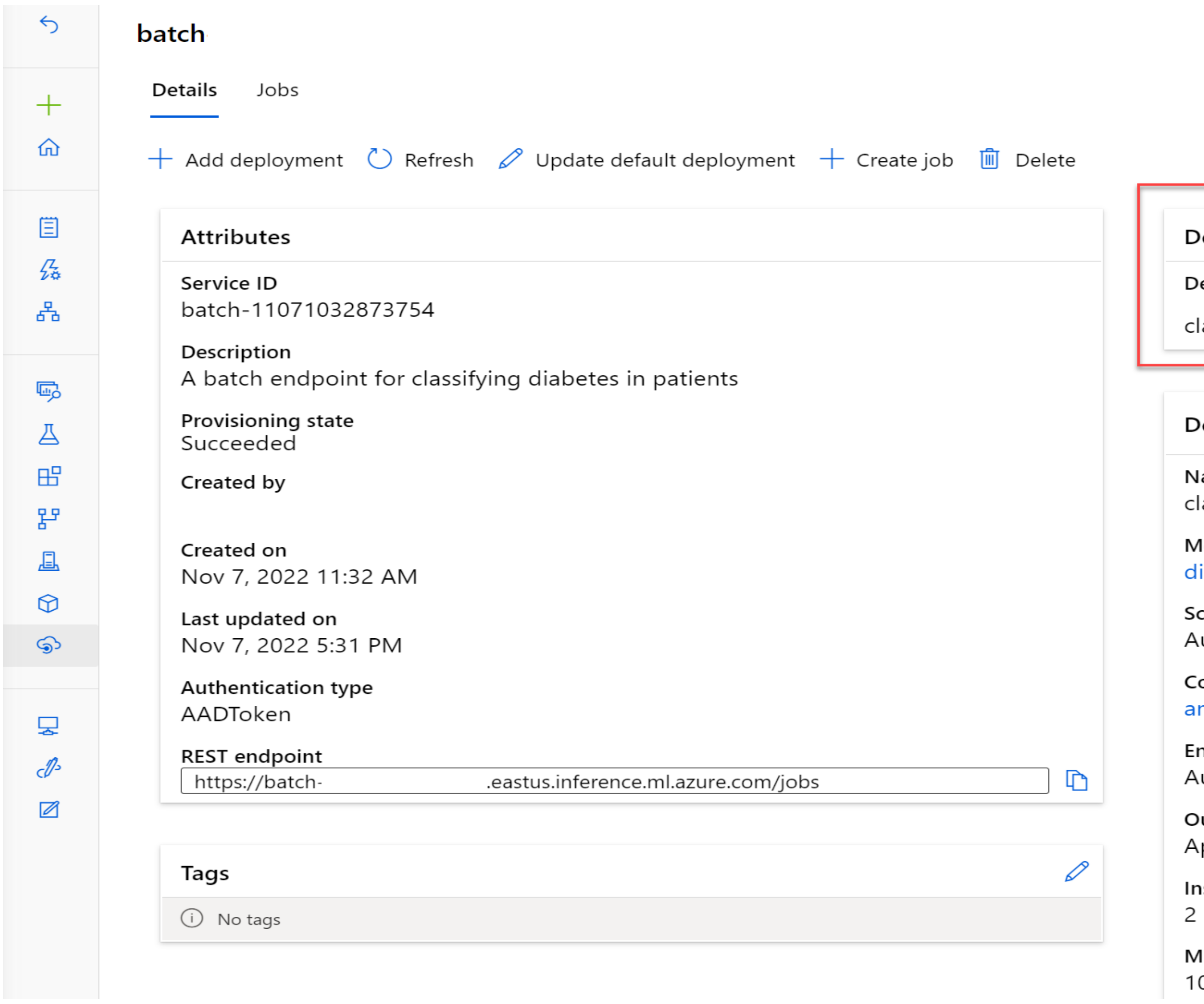
```python
# create a batch endpoint
endpoint = BatchEndpoint(
    name="endpoint-example",
    description="A batch endpoint",
)

ml_client.batch_endpoints.begin_create_or_update(endpoint)
```

Tip

Explore the reference documentation to **create a batch endpoint with the Python SDK v2**.

Deploy a model to a batch endpoint

You can deploy multiple models to a batch endpoint. Whenever you call the batch endpoint, which triggers a batch scoring job, the **default deployment** will be used unless specified otherwise.


batch

Attributes

Service ID
batch-11071032873754

Description
A batch endpoint for classifying diabetes in patients

Provisioning state
Succeeded

Created by

Created on
Nov 7, 2022 11:32 AM

Last updated on
Nov 7, 2022 5:31 PM

Authentication type
AADToken

REST endpoint
https://batch- .eastus.inference.ml.azure.com/jobs

Tags

No tags

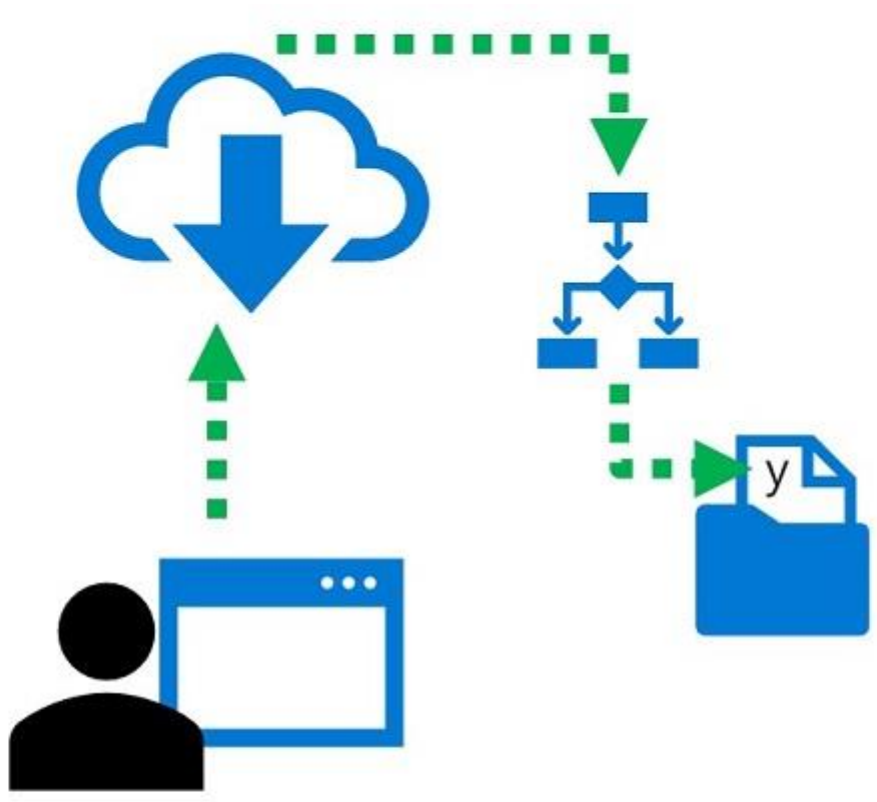
Use compute clusters for batch deployments

The ideal compute to use for batch deployments is the Azure Machine Learning compute cluster. If you want the batch scoring job to process the new data in parallel batches, you need to provision a compute cluster with more than one maximum instances.

To create a compute cluster, you can use the `AMLCompute` class.

PythonCopy

```python
from azure.ai.ml.entities import AmlCompute

cpu_cluster = AmlCompute(
    name="aml-cluster",
    type="amlcompute",
    size="STANDARD_DS11_V2",
    min_instances=0,
    max_instances=4,
    idle_time_before_scale_down=120,
    tier="Dedicated",
)

cpu_cluster = ml_client.compute.begin_create_or_update(cpu_cluster)
```



In Azure Machine Learning, you can implement batch inferencing solutions by deploying a model to a batch endpoint.

Learning objectives

In this module, you'll learn how to:

- Create a batch endpoint.
- Deploy your MLflow model to a batch endpoint.
- Deploy a custom model to a batch endpoint.
- Invoke batch endpoints.

Deploy your MLflow model to a batch endpoint

Completed100 XP

- 7 minutes

An easy way to deploy a model to a batch endpoint is to use an **MLflow** model. Azure Machine Learning will automatically generate the scoring script and environment for MLflow models.

To deploy an MLflow model, you need to have created an endpoint. Then you can deploy the model to the endpoint.

Register an MLflow model

To avoid needed a scoring script and environment, an MLflow model needs to be registered in the Azure Machine Learning workspace before you can deploy it to a batch endpoint.

To register an MLflow model, you'll use the `Model` class, while specifying the model type to be `MLFLOW_MODEL`. To register the model with the Python SDK, you can use the following code:

PythonCopy

```python
from azure.ai.ml.entities import Model
from azure.ai.ml.constants import AssetTypes

model_name = 'mlflow-model'
model = ml_client.models.create_or_update(
    Model(name=model_name, path='./model', type=AssetTypes.MLFLOW_MODEL)
)
```

In this example, we're taking the model files from a local path. The files are all stored in a local folder called `model`. The folder must include the `MLmodel` file, which describes how the model can be loaded and used.

Tip

Learn more about **the MLmodel format**.

Deploy an MLflow model to an endpoint

To deploy an MLflow model to a batch endpoint, you'll use the `BatchDeployment` class.

When you deploy a model, you'll need to specify how you want the batch scoring job to behave. The advantage of using a compute cluster to run the scoring script (which is automatically generated by Azure Machine Learning), is that you can run the scoring script on separate instances in parallel.

When you configure the model deployment, you can specify:

- `instance_count`: Count of compute nodes to use for generating predictions.
- `max_concurrency_per_instance`: Maximum number of parallel scoring script runs per compute node.
- `mini_batch_size`: Number of files passed per scoring script run.
- `output_action`: What to do with the predictions: `summary_only` or `append_row`.
- `output_file_name`: File to which predictions will be appended, if you choose `append_row` for `output_action`.

Tip

Explore the reference documentation to **create a batch deployment with the Python SDK v2**.

To deploy an MLflow model to a batch endpoint, you can use the following code:

PythonCopy

```python
from azure.ai.ml.entities import BatchDeployment, BatchRetrySettings
from azure.ai.ml.constants import BatchDeploymentOutputAction

deployment = BatchDeployment(
    name="forecast-mlflow",
    description="A sales forecaster",
    endpoint_name=endpoint.name,
    model=model,
    compute="aml-cluster",
    instance_count=2,
    max_concurrency_per_instance=2,
    mini_batch_size=2,
    output_action=BatchDeploymentOutputAction.APPEND_ROW,
    output_file_name="predictions.csv",
    retry_settings=BatchRetrySettings(max_retries=3, timeout=300),
    logging_level="info",
)
ml_client.batch_deployments.begin_create_or_update(deployment)
```

Deploy a custom model to a batch endpoint

Completed100 XP

- 9 minutes

If you want to deploy a model to a batch endpoint without using the MLflow model format, you need to create the scoring script and environment.

To deploy a model, you must have already created an endpoint. Then you can deploy the model to the endpoint.

Create the scoring script

The scoring script is a file that reads the new data, loads the model, and performs the scoring.

The scoring script must include two functions:

- `init()`: Called once at the beginning of the process, so use for any costly or common preparation like loading the model.
- `run()`: Called for each mini batch to perform the scoring.

The `run()` method should return a pandas DataFrame or an array/list.

A scoring script may look as follows:

PythonCopy

```python
import os
import mlflow
import pandas as pd


def init():
    global model

    # get the path to the registered model file and load it
    model_path = os.path.join(os.environ["AZUREML_MODEL_DIR"], "model")
    model = mlflow.pyfunc.load(model_path)


def run(mini_batch):
    print(f"run method start: {__file__}, run({len(mini_batch)} files)")
    resultList = []

    for file_path in mini_batch:
        data = pd.read_csv(file_path)
        pred = model.predict(data)

        df = pd.DataFrame(pred, columns=["predictions"])
        df["file"] = os.path.basename(file_path)
        resultList.extend(df.values)

    return resultList
```

There are some things to note from the example script:

- `AZUREML_MODEL_DIR` is an environment variable that you can use to locate the files associated with the model.
- Use `global` variable to make any assets available that are needed to score the new data, like the loaded model.
- The size of the `mini_batch` is defined in the deployment configuration. If the files in the mini batch are too large to be processed, you need to split the files into smaller files.
- By default, the predictions will be written to one single file.

Tip

Learn more about how to **author scoring scripts for batch deployments**.

Create an environment

Your deployment requires an execution environment in which to run the scoring script. Any dependency your code requires should be included in the environment.

You can create an environment with a Docker image with Conda dependencies, or with a Dockerfile.

You'll also need to add the library `azureml-core` as it is required for batch deployments to work.

To create an environment using a base Docker image, you can define the Conda dependencies in a `conda.yaml` file:

ymlCopy

```yml
name: basic-env-cpu
channels:
  - conda-forge
dependencies:
  - python=3.8
  - pandas
  - pip
  - pip:
      - azureml-core
      - mlflow
```

Then, to create the environment, run the following code:

PythonCopy

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

env = Environment(
    image="mcr.microsoft.com/azureml/openmpi3.1.2-ubuntu18.04",
    conda_file="./src/conda-env.yml",
    name="deployment-environment",
    description="Environment created from a Docker image plus Conda environment.",
)
```

```
ml_client.environments.create_or_update(env)
```

Configure and create the deployment

Finally, you can configure and create the deployment with the `BatchDeployment` class.

PythonCopy

```python
from azure.ai.ml.entities import BatchDeployment, BatchRetrySettings
from azure.ai.ml.constants import BatchDeploymentOutputAction

deployment = BatchDeployment(
    name="forecast-mlflow",
    description="A sales forecaster",
    endpoint_name=endpoint.name,
    model=model,
    compute="aml-cluster",
    code_path="./code",
    scoring_script="score.py",
    environment=env,
    instance_count=2,
    max_concurrency_per_instance=2,
    mini_batch_size=2,
    output_action=BatchDeploymentOutputAction.APPEND_ROW,
    output_file_name="predictions.csv",
    retry_settings=BatchRetrySettings(max_retries=3, timeout=300),
    logging_level="info",
)
ml_client.batch_deployments.begin_create_or_update(deployment)
```

Invoke and troubleshoot batch endpoints

Completed100 XP

- 5 minutes

When you invoke a batch endpoint, you trigger an Azure Machine Learning **pipeline job**. The job will expect an input parameter pointing to the data set you want to score.

Trigger the batch scoring job

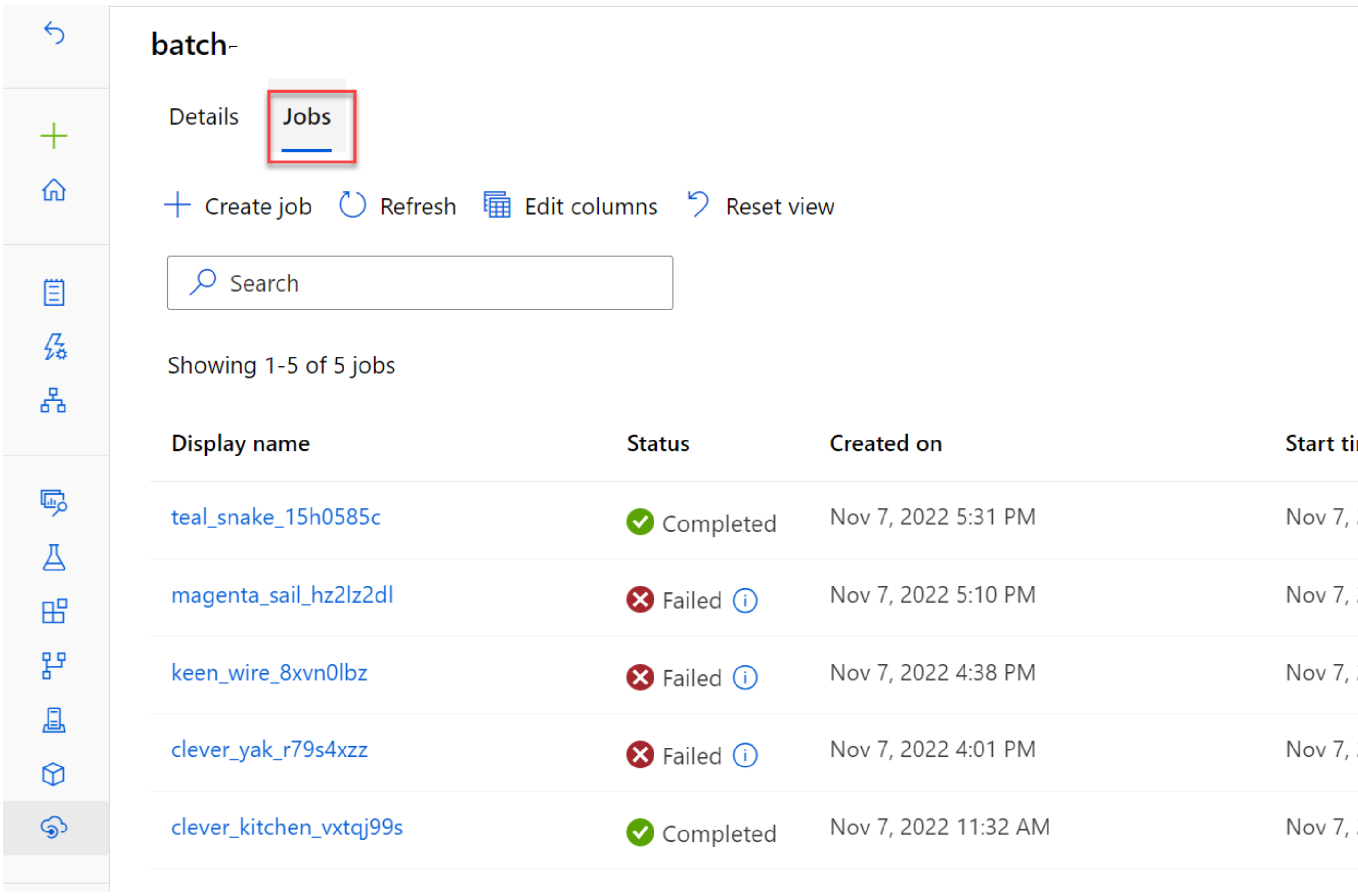
To prepare data for batch predictions, you can register a folder as a data asset in the Azure Machine Learning workspace.

You can then use the registered data asset as input when invoking the batch endpoint with the Python SDK:

PythonCopy

```python
from azure.ai.ml import Input
from azure.ai.ml.constants import AssetTypes

input = Input(type=AssetTypes.URI_FOLDER, path="azureml:new-data:1")

job = ml_client.batch_endpoints.invoke(
    endpoint_name=endpoint.name,
    input=input)
```

You can monitor the run of the pipeline job in the Azure Machine Learning studio. All jobs that are triggered by invoking the batch endpoint will show in the **Jobs** tab of the batch endpoint.



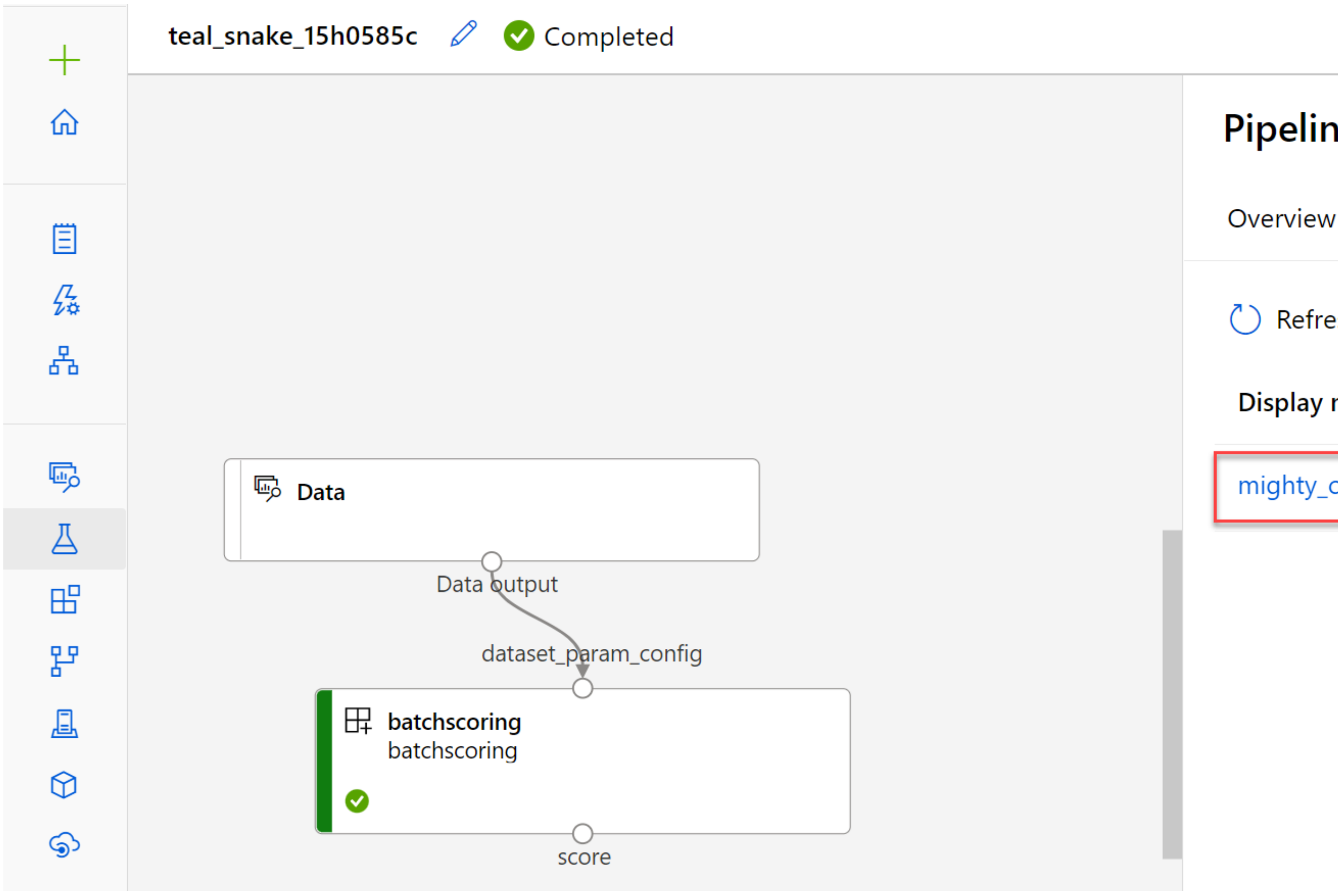
Showing 1-5 of 5 jobs

Display name	Status	Created on	Start ti
teal_snake_15h0585c	Completed	Nov 7, 2022 5:31 PM	Nov 7,
magenta_sail_hz2lz2dl	Failed	Nov 7, 2022 5:10 PM	Nov 7,
keen_wire_8xvn0lbz	Failed	Nov 7, 2022 4:38 PM	Nov 7,
clever_yak_r79s4xzz	Failed	Nov 7, 2022 4:01 PM	Nov 7,
clever_kitchen_vxtqj99s	Completed	Nov 7, 2022 11:32 AM	Nov 7,

The predictions will be stored in the default datastore.

Troubleshoot a batch scoring job

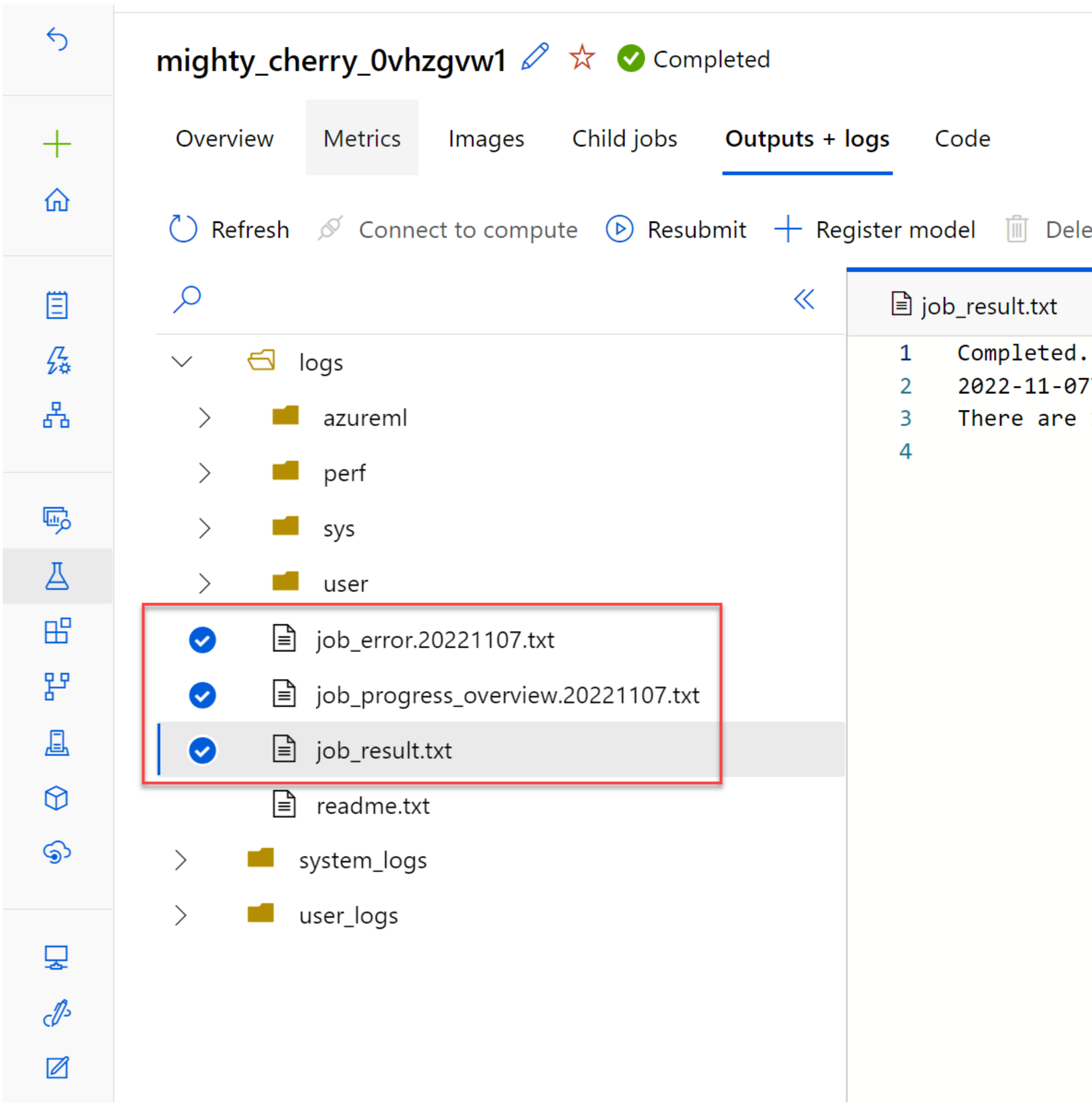
The batch scoring job runs as a *pipeline job*. If you want to troubleshoot the pipeline job, you can review its details and the outputs and logs of the pipeline job itself.



If you want to troubleshoot the scoring script, you can select the child job and review its outputs and logs.

Navigate to the **Outputs + logs** tab. The **logs/user/** folder contains three files that will help you troubleshoot:

- `job_error.txt`: Summarize the errors in your script.
- `job_progress_overview.txt`: Provides high-level information about the number of mini-batches processed so far.
- `job_result.txt`: Shows errors in calling the `init()` and `run()` function in the scoring script.



```
Completed.
2022-11-07
There are
```

1.

You are creating a batch endpoint that you want to use to predict new values for a large volume of data files. You want the pipeline to run the scoring script on multiple nodes and collate the results. What output action should you choose for the deployment?

Correct. You should use `append_row` **to append each prediction to one output file.**

2.

You have multiple models deployed to a batch endpoint. You invoke the endpoint without indicating which model you want to use. Which deployed model will do the actual batch scoring?

Correct. The default deployment will be used to do the actual batch scoring when the endpoint is invoked.

Understand Responsible AI

Completed100 XP

- 4 minutes

As a data scientist, you may train a machine learning model to predict whether someone is able to pay back a loan, or whether a candidate is suitable for a job vacancy. As models are often used when making decisions, it's important that the models are unbiased and transparent.

Whatever you use a model for, you should consider the **Responsible Artificial Intelligence** (**Responsible AI**) principles. Depending on the use case, you may focus on specific principles. Nevertheless, it's a best practice to consider all principles to ensure you're addressing any issues the model may have.

Microsoft has listed five Responsible AI principles:

- **Fairness and inclusiveness**: Models should treat everyone fairly and avoid different treatment for similar groups.
- **Reliability and safety**: Models should be reliable, safe, and consistent. You want a model to operate as intended, handle unexpected situations well, and resist harmful manipulation.
- **Privacy and security**: Be transparent about data collection, use, and storage, to empower individuals with control over their data. Treat data with care to ensure an individual's privacy.
- **Transparency**: When models influence important decisions that affect people's lives, people need to understand how those decisions were made and how the model works.
- **Accountability**: Take accountability for decisions that models may influence and maintain human control.

Create the Responsible AI dashboard

Completed100 XP

- 9 minutes

To help you implement the **Responsible Artificial Intelligence** (**Responsible AI**) principles in Azure Machine Learning, you can create the **Responsible AI dashboard**.

The Responsible AI dashboard allows you to pick and choose insights you need, to evaluate whether your model is safe, trustworthy, and ethical.

Azure Machine Learning has built-in **components** that can generate Responsible AI insights for you. The insights are then gathered in an interactive dashboard for you to explore. You can also generate a scorecard as PDF to easily share the insights with your colleagues to evaluate your models.

Create a Responsible AI dashboard

To create a Responsible AI (RAI) dashboard, you need to create a **pipeline** by using the built-in components. The pipeline should:

1. Start with the `RAI Insights dashboard constructor`.
2. Include one of the **RAI tool components**.
3. End with `Gather RAI Insights dashboard` to collect all insights into one dashboard.
4. *Optionally* you can also add the `Gather RAI Insights score card` at the end of your pipeline.

Explore the Responsible AI components

The available tool components and the insights you can use are:

- `Add Explanation to RAI Insights dashboard`: Interpret models by generating explanations. Explanations show how much features influence the prediction.
- `Add Causal to RAI Insights dashboard`: Use historical data to view the causal effects of features on outcomes.
- `Add Counterfactuals to RAI Insights dashboard`: Explore how a change in input would change the model's output.
- `Add Error Analysis to RAI Insights dashboard`: Explore the distribution of your data and identify erroneous subgroups of data.

Build and run the pipeline to create the Responsible AI dashboard

To create the Responsible AI dashboard, you build a pipeline with the components you selected. When you run the pipeline, a Responsible dashboard (and scorecard) is generated and associated with your model.

After you've trained and registered a model in the Azure Machine Learning workspace, you can create the Responsible AI dashboard in three ways:

- Using the Command Line Interface (CLI) extension for Azure Machine Learning.
- Using the Python Software Development Kit (SDK).
- Using the Azure Machine Learning studio for a no-code experience.

Using the Python SDK to build and run the pipeline

To generate a Responsible AI dashboard, you need to:

- Register the training and test datasets as MLtable data assets.
- Register the model.
- Retrieve the built-in components you want to use.
- Build the pipeline.
- Run the pipeline.

If you want to build the pipeline using the Python SDK, you first have to retrieve the components you want to use.

You should start the pipeline with the `RAI Insights dashboard constructor` component:

PythonCopy

```python
rai_constructor_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_insight_constructor", label="latest"
)
```

Then, you can add any of the available insights, like the explanations, by retrieving the `Add Explanation to RAI Insights dashboard component`:

PythonCopy

```python
rai_explanation_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_explanation", label="latest"
)
```

Note

The parameters and expected inputs vary across components. **Explore the component for the specific insights** you want to add to your dashboard to find which inputs you need to specify.

And finally, your pipeline should end with a `Gather RAI Insights dashboard` component:

PythonCopy

```
rai_gather_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_insight_gather", label="latest"
)
```

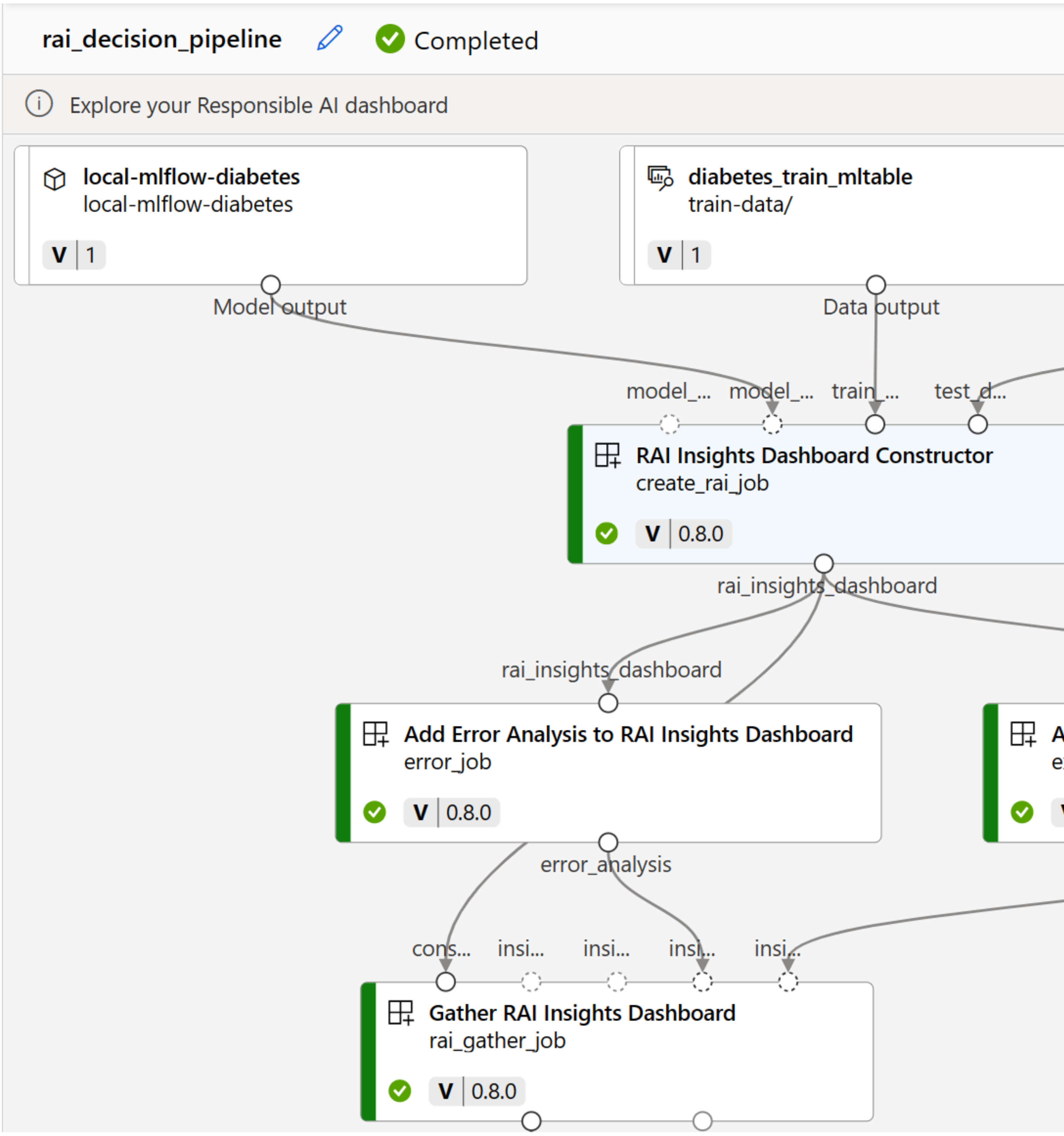
Once you have the components, you can build the pipeline:

PythonCopy

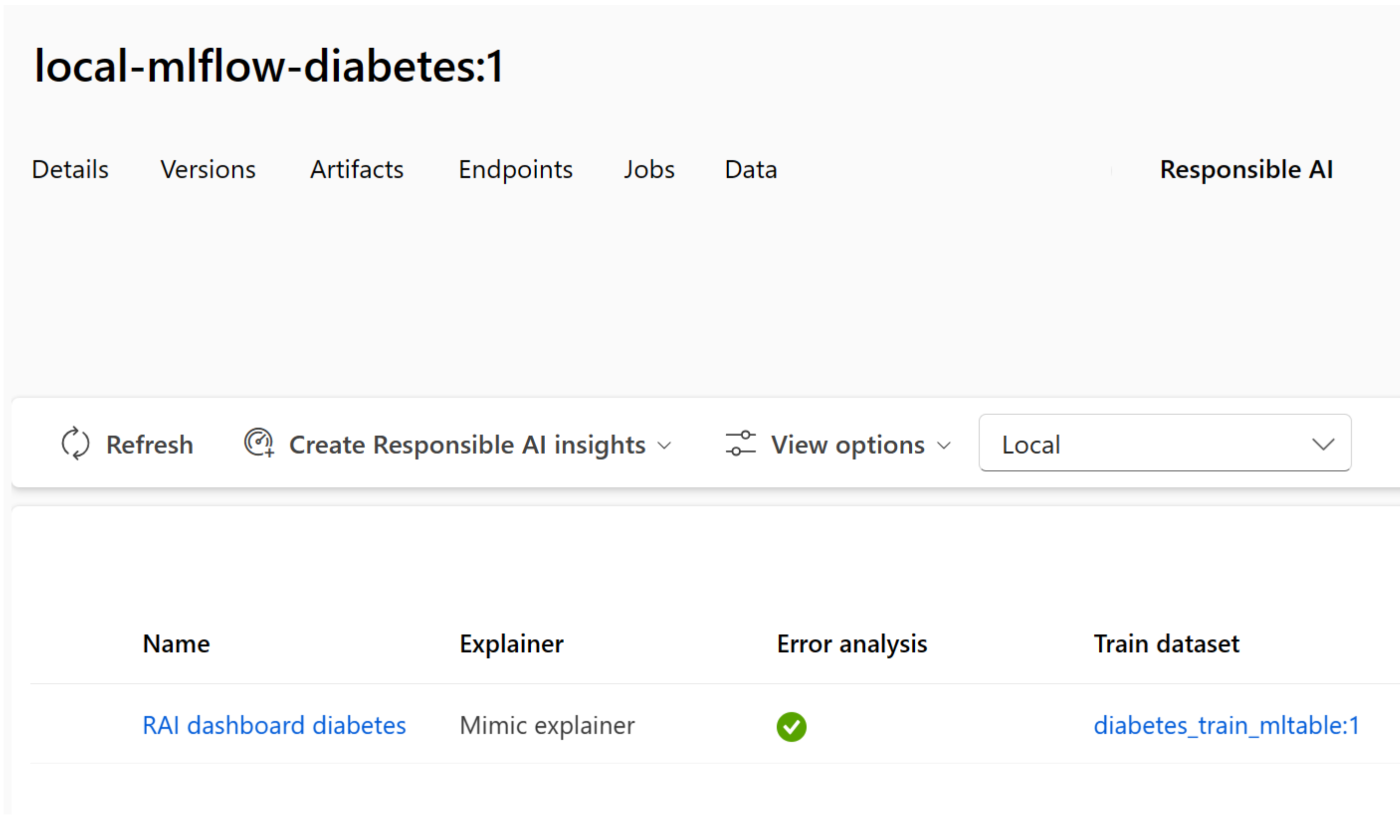
```
from azure.ai.ml import Input, dsl
from azure.ai.ml.constants import AssetTypes

@dsl.pipeline(
    compute="aml-cluster",
    experiment_name="Create RAI Dashboard",
)
def rai_decision_pipeline(
    target_column_name, train_data, test_data
):
    # Initiate the RAIInsights
    create_rai_job = rai_constructor_component(
        title="RAI dashboard diabetes",
        task_type="classification",
        model_info=expected_model_id,
        model_input=Input(type=AssetTypes.MLFLOW_MODEL, path=azureml_model_id),
        train_dataset=train_data,
        test_dataset=test_data,
        target_column_name="Predictions",
    )
    create_rai_job.set_limits(timeout=30)

    # Add explanations
    explanation_job = rai_explanation_component(
        rai_insights_dashboard=create_rai_job.outputs.rai_insights_dashboard,
        comment="add explanation",
    )
    explanation_job.set_limits(timeout=10)

    # Combine everything
    rai_gather_job = rai_gather_component(
        constructor=create_rai_job.outputs.rai_insights_dashboard,
        insight=explanation_job.outputs.explanation,
    )
    rai_gather_job.set_limits(timeout=10)

    rai_gather_job.outputs.dashboard.mode = "upload"

    return {
        "dashboard": rai_gather_job.outputs.dashboard,
    }
```

Exploring the Responsible AI dashboard

After building the pipeline, you need to run it to generate the Responsible AI dashboard. When the pipeline successfully completed, you can select to **view** the Responsible AI dashboard from the pipeline overview.



Alternatively, you can find the Responsible AI dashboard in the **Responsible AI** tab of the registered model.


local-mlflow-diabetes:1

Details Versions Artifacts Endpoints Jobs Data Responsible AI

Refresh | Create Responsible AI insights | View options | Local

Name	Explainer	Error analysis	Train dataset
RAI dashboard diabetes	Mimic explainer	✓	diabetes_train_mltable:1

Evaluate the Responsible AI dashboard

Completed100 XP

- 6 minutes

When your Responsible AI dashboard is generated, you can explore its contents in the Azure Machine Learning studio to evaluate your model.

When you open the Responsible AI dashboard, the studio tries to automatically connect it to a compute instance. The compute instance provides the necessary compute for interactive exploration within the dashboard.

The output of each component you added to the pipeline is reflected in the dashboard. Depending on the components you selected, you can find the following insights in your Responsible AI dashboard:

- Error analysis
- Explanations
- Counterfactuals
- Causal analysis

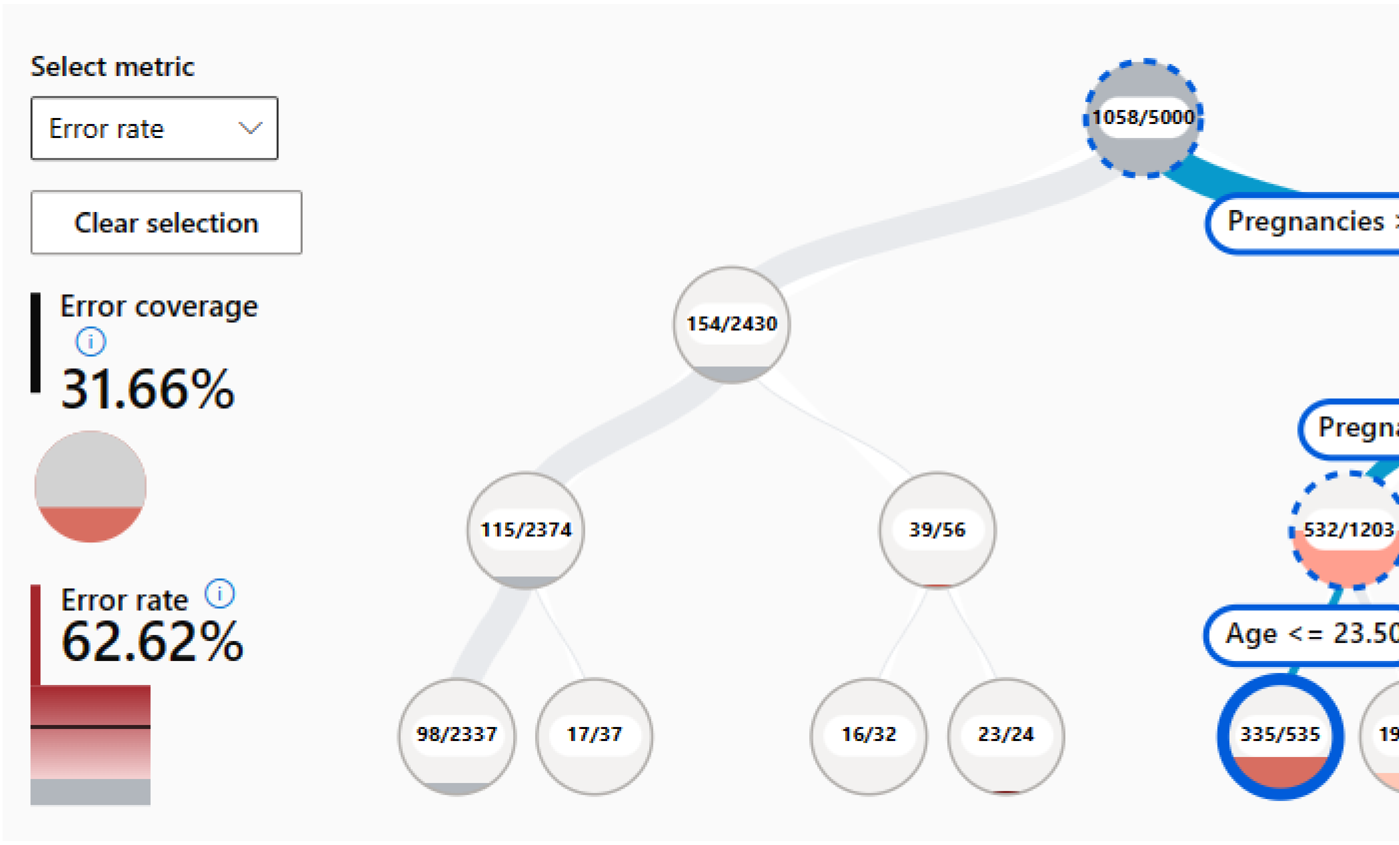
Let's explore what we can review for each of these insights.

Explore error analysis

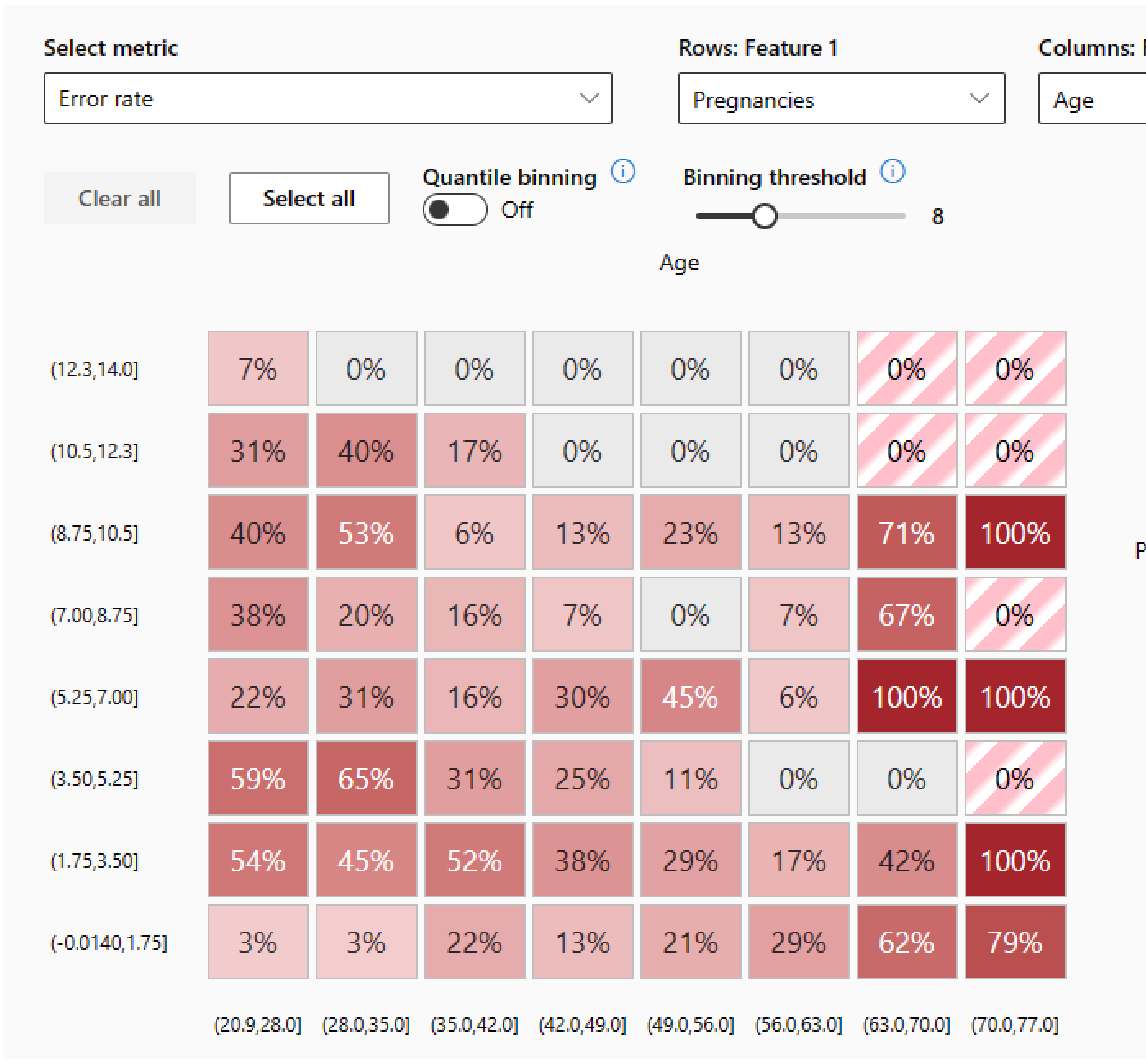
A model is expected to make false predictions, or errors. With the error analysis feature in the Responsible AI dashboard, you can review and understand how errors are distributed in your dataset. For example, are there specific subgroups, or cohorts, in your dataset for which the model makes more false predictions?

When you include error analysis, there are two types of visuals you can explore in the Responsible AI dashboard:

- **Error tree map**: Allows you to explore which combination of subgroups results in the model making more false predictions.



- **Error heat map**: Presents a grid overview of a model's errors over the scale of one or two features.



Select metric: Error rate

Rows: Feature 1: Pregnancies

Columns: Age

Quantile binning: Off

Binning threshold: 8

Age

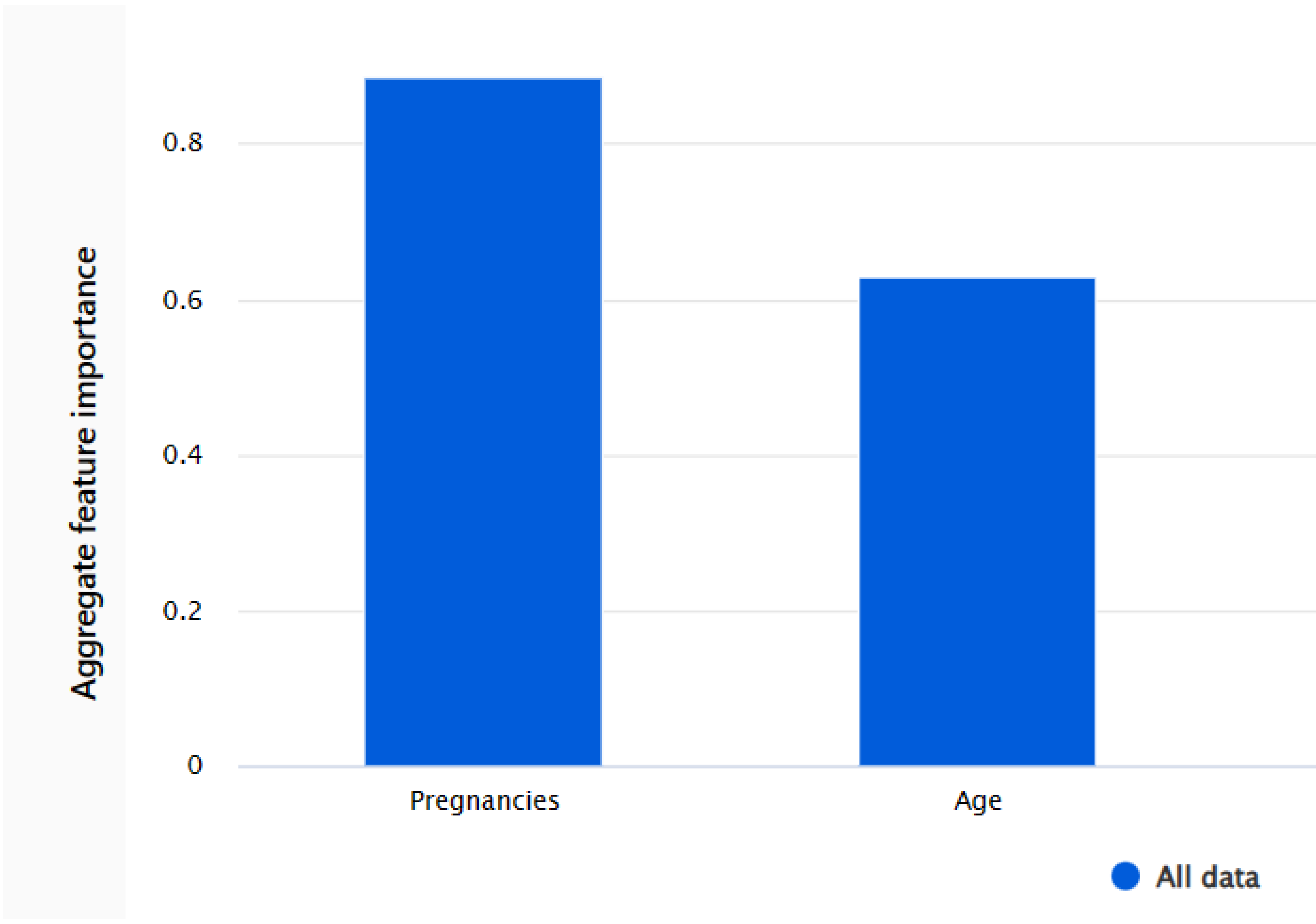
	(20.9,28.0]	(28.0,35.0]	(35.0,42.0]	(42.0,49.0]	(49.0,56.0]	(56.0,63.0]	(63.0,70.0]	(70.0,77.0]
(12.3,14.0]	7%	0%	0%	0%	0%	0%	0%	0%
(10.5,12.3]	31%	40%	17%	0%	0%	0%	0%	0%
(8.75,10.5]	40%	53%	6%	13%	23%	13%	71%	100%
(7.00,8.75]	38%	20%	16%	7%	0%	7%	67%	0%
(5.25,7.00]	22%	31%	16%	30%	45%	6%	100%	100%
(3.50,5.25]	59%	65%	31%	25%	11%	0%	0%	0%
(1.75,3.50]	54%	45%	52%	38%	29%	17%	42%	100%
(-0.0140,1.75]	3%	3%	22%	13%	21%	29%	62%	79%

Explore explanations

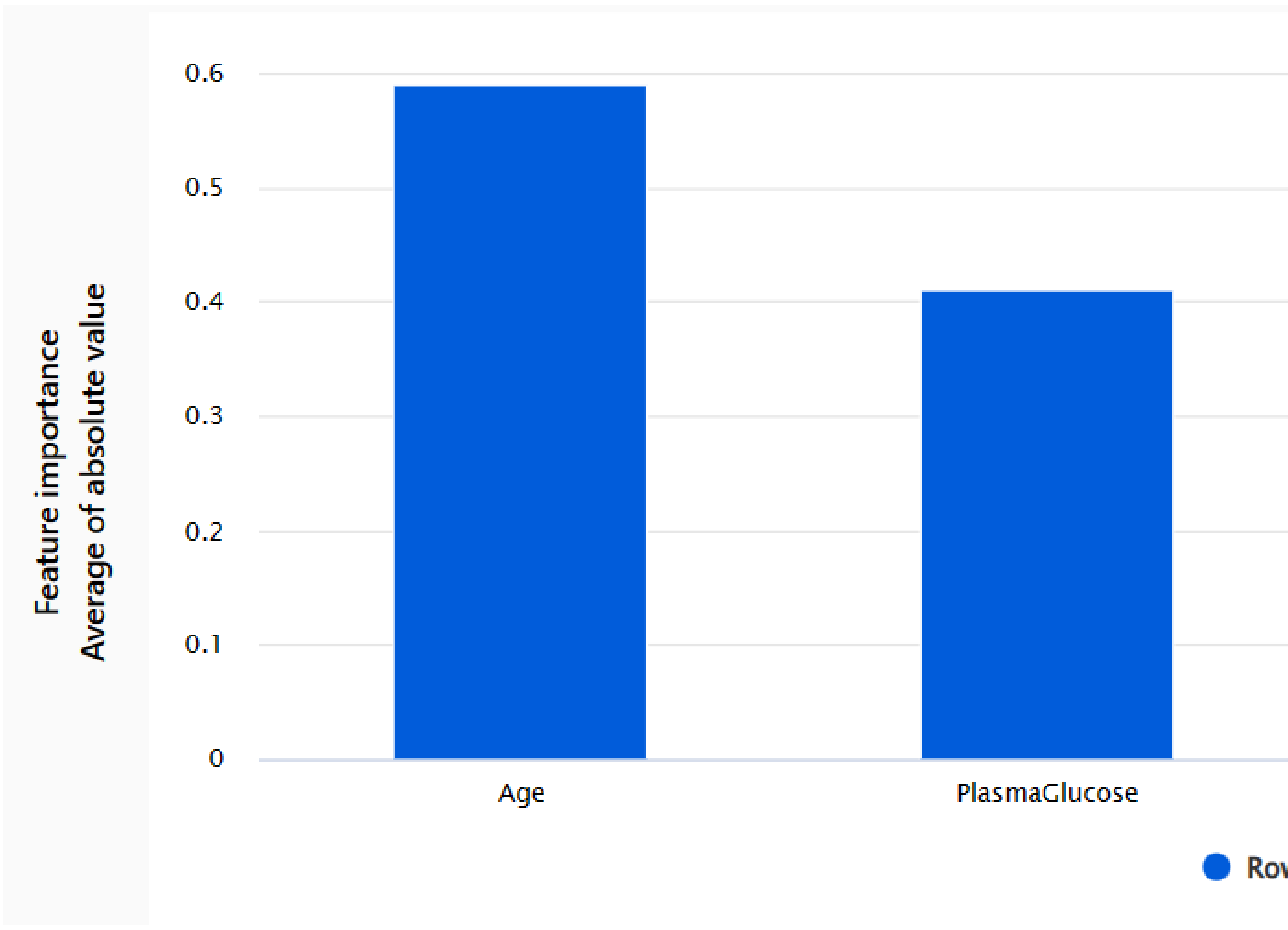
Whenever you use a model for decision-making, you want to understand how a model reaches a certain prediction. Whenever you've trained a model that is too complex to understand, you can run *model explainers* to calculate the **feature importance**. In other words, you want to understand how each of the input features influences the model's prediction.

There are various statistical techniques you can use as model explainers. Most commonly, the **mimic** explainer trains a simple interpretable model on the same data and task. As a result, you can explore two types of feature importance:

- **Aggregate feature importance**: Shows how each feature in the test data influences the model's predictions *overall*.



- **Individual feature importance**: Shows how each feature impacts an *individual* prediction.



Explore counterfactuals

Explanations can give you insights into the relative importance of features on the model's predictions. Sometimes, you may want to take it a step further and understand whether the model's predictions would change if the input would be different. To explore how the model's output would change based on a change in the input, you can use **counterfactuals**.

You can choose to explore counterfactuals *what-if* examples by selecting a data point and the desired model's prediction for that point. When you create a what-if counterfactual, the dashboard opens a panel to help you understand which input would result in the desired prediction.

Explore causal analysis

Explanations and counterfactuals help you to understand the model's predictions and the effects of features on the predictions. Though model interpretability may already be a goal by itself, you may also need more information to help you improve decision-making.

Causal analysis uses statistical techniques to estimate the average effect of a feature on a desired prediction. It analyzes how certain interventions or treatments may result in a better outcome, across a population or for a specific individual.

There are three available tabs in the Responsible AI dashboard when including causal analysis:

- **Aggregate causal effects**: Shows the average causal effects for predefined treatment features (the features you want to change to optimize the model's predictions).
- **Individual causal effects**: Shows individual data points and allows you to change the treatment features to explore their influence on the prediction.
- **Treatment policy**: Shows which parts of your data points benefit most from a treatment.

1.

A data scientist wants to investigate for which subgroups the model has relatively more false predictions, which Responsible AI component should be added to the pipeline to create the Responsible AI dashboard?

Explanations.

Counterfactuals.

Error analysis.

Correct. Error analysis provides an overview of the number of false predictions for specific subgroups in your dataset.

2.

What should be the first component in a pipeline to create a Responsible AI Dashboard?

`RAI Insights dashboard constructor`

Correct. The dashboard constructor should be the first component in the pipeline.

`Gather RAI Insights dashboard`

○

`Gather RAI Insights score card`

3.

A data scientist has trained a model, and wants to quantify the influence of each feature on a specific prediction. What kind of feature importance should the data scientist examine?

○

Aggregate feature importance.

○

Global feature importance.

○

Individual feature importance.

Correct. The individual feature importance shows how each feature influences an individual prediction.

Define a search space - Hyperparametres

Completed100 XP

- 5 minutes

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.

Discrete hyperparameters

Some hyperparameters require *discrete* values - in other words, you must select the value from a particular *finite* 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(values=[10,20,30])`), a **range** (`Choice(values=range(1,10))`), or an arbitrary set of comma-separated values (`Choice(values=(30,50,100))`)

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

- `QUniform(min_value, max_value, q)`: Returns a value like round(Uniform(min_value, max_value) / q) * q
- `QLogUniform(min_value, max_value, q)`: Returns a value like round(exp(Uniform(min_value, max_value)) / q) * q
- `QNormal(mu, sigma, q)`: Returns a value like round(Normal(mu, sigma) / q) * q
- `QLogNormal(mu, sigma, q)`: Returns a value like round(exp(Normal(mu, sigma)) / q) * q

Continuous hyperparameters

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

- `Uniform(min_value, max_value)`: Returns a value uniformly distributed between min_value and max_value
- `LogUniform(min_value, max_value)`: Returns a value drawn according to exp(Uniform(min_value, max_value)) so that the logarithm of the return value is uniformly distributed
- `Normal(mu, sigma)`: Returns a real value that's normally distributed with mean mu and standard deviation sigma
- `LogNormal(mu, sigma)`: Returns a value drawn according to exp(Normal(mu, sigma)) so that the logarithm of the return value is normally distributed

Defining a search space

To define a search space for hyperparameter tuning, create a dictionary with the appropriate parameter expression for each named hyperparameter.

For example, the following search space indicates that
the `batch_size` hyperparameter can have the value 16, 32, or 64, and
the `learning_rate` hyperparameter can have any value from a normal distribution with a mean of 10 and a standard deviation of 3.

PythonCopy

```python
from azure.ai.ml.sweep import Choice, Normal

command_job_for_sweep = job(
    batch_size=Choice(values=[16, 32, 64]),
    learning_rate=Normal(mu=10, sigma=3),
)
```

Configure a sampling method

Completed100 XP

- 8 minutes

The specific values used in a hyperparameter tuning run, or **sweep job**, depend on the type of **sampling** used.

There are three main sampling methods available in Azure Machine Learning:

- **Grid sampling**: Tries every possible combination.
- **Random sampling**: Randomly chooses values from the search space.
  - **Sobol**: Adds a seed to random sampling to make the results reproducible.
- **Bayesian sampling**: Chooses new values based on previous results.

Note

Sobol is a variation of random sampling.

Grid sampling

Grid sampling can only be applied when all hyperparameters are discrete, and is used to try every possible combination of parameters in the search space.

For example, in the following code example, grid sampling is used to try every possible combination of discrete *batch_size* and *learning_rate* value:

PythonCopy

```python
from azure.ai.ml.sweep import Choice

command_job_for_sweep = command_job(
    batch_size=Choice(values=[16, 32, 64]),
    learning_rate=Choice(values=[0.01, 0.1, 1.0]),
)

sweep_job = command_job_for_sweep.sweep(
    sampling_algorithm = "grid",
    ...
)
```

Random sampling

Random sampling is used to randomly select a value for each hyperparameter, which can be a mix of discrete and continuous values as shown in the following code example:

PythonCopy

```python
from azure.ai.ml.sweep import Normal, Uniform

command_job_for_sweep = command_job(
    batch_size=Choice(values=[16, 32, 64]),
    learning_rate=Normal(mu=10, sigma=3),
)

sweep_job = command_job_for_sweep.sweep(
    sampling_algorithm = "random",
    ...
)
```

Sobol

You may want to be able to reproduce a random sampling sweep job. If you expect that you do, you can use Sobol instead. Sobol is a type of random sampling that allows you to use a seed. When you add a seed, the sweep job can be reproduced, and the search space distribution is spread more evenly.

The following code example shows how to use Sobol by adding a seed and a rule, and using the `RandomParameterSampling` class:

PythonCopy

```python
from azure.ai.ml.sweep import RandomParameterSampling

sweep_job = command_job_for_sweep.sweep(
    sampling_algorithm = RandomParameterSampling(seed=123, rule="sobol"),
    ...
)
```

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. The following code example shows how to configure Bayesian sampling:

PythonCopy

```python
from azure.ai.ml.sweep import Uniform, Choice

command_job_for_sweep = job(
    batch_size=Choice(values=[16, 32, 64]),
    learning_rate=Uniform(min_value=0.05, max_value=0.1),
)

sweep_job = command_job_for_sweep.sweep(
    sampling_algorithm = "bayesian",
    ...
)
```

You can only use Bayesian sampling with **choice**, **uniform**, and **quniform** parameter expressions.

Configure early termination

Completed100 XP

- 8 minutes

Hyperparameter tuning helps you fine-tune your model and select the hyperparameter values that will make your model perform best.

For you to find the best model, however, can be a never-ending conquest. You always have to consider whether it's worth the time and expense of testing new hyperparameter values to find a model that may perform better.

Each trial in a sweep job, a new model is trained with a new combination of hyperparameter values. If training a new model doesn't result in a significantly better model, you may want to stop the sweep job and use the model that performed best so far.

When you configure a sweep job in Azure Machine Learning, you can also set a maximum number of trials. A more sophisticated approach may be to stop a sweep job when newer models don't produce significantly better results. To stop a sweep job based on the performance of the models, you can use an **early termination policy**.

When to use an early termination policy

Whether you want to use an early termination policy may depend on the search space and sampling method you're working with.

For example, you may choose to use a *grid sampling* method over a *discrete* search space that results in a maximum of six trials. With six trials, a maximum of six models will be trained and an early termination policy may be unnecessary.

An early termination policy can be especially beneficial when working with continuous hyperparameters in your search space. Continuous hyperparameters present an unlimited number of possible values to choose from. You'll most likely want to use an early termination policy when working with continuous hyperparameters and a random or Bayesian sampling method.

Configure an early termination policy

There are two main parameters when you choose to use an early termination policy:

- `evaluation_interval`: Specifies at which interval you want the policy to be evaluated. Every time the primary metric is logged for a trial counts as an interval.
- `delay_evaluation`: Specifies when to start evaluating the policy. This parameter allows for at least a minimum of trials to complete without an early termination policy affecting them.

New models may continue to perform only slightly better than previous models. To determine the extent to which a model should perform better than previous trials, there are three options for early termination:

- **Bandit policy**: Uses a `slack_factor` (relative) or `slack_amount`(absolute). Any new model must perform within the slack range of the best performing model.
- **Median stopping policy**: Uses the median of the averages of the primary metric. Any new model must perform better than the median.
- **Truncation selection policy**: Uses a `truncation_percentage`, which is the percentage of lowest performing trials. Any new model must perform better than the lowest performing trials.

Bandit policy

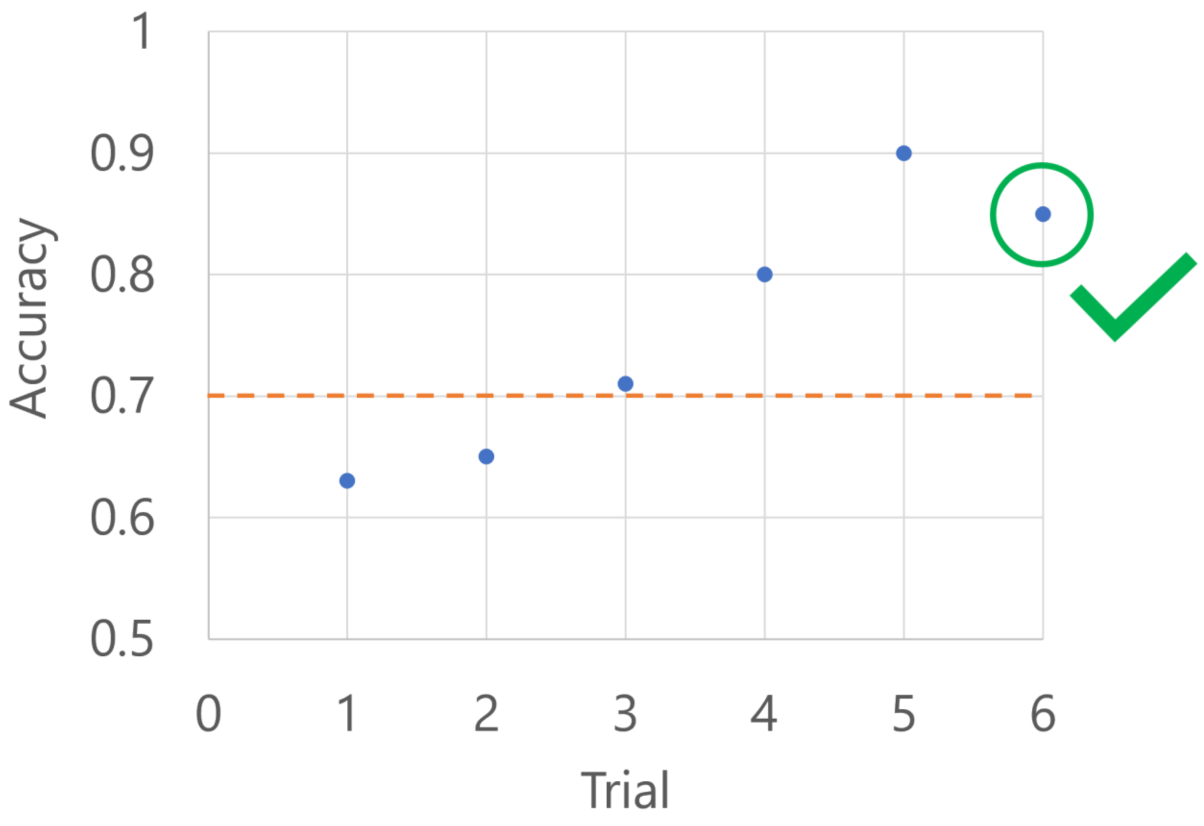
You can use a bandit policy to stop a trial if the target performance metric underperforms the best trial so far by a specified margin.

For example, the following code applies a bandit policy with a delay of five trials, evaluates the policy at every interval, and allows an absolute slack amount of 0.2.

PythonCopy

```python
from azure.ai.ml.sweep import BanditPolicy

sweep_job.early_termination = BanditPolicy(
    slack_amount = 0.2,
    delay_evaluation = 5,
    evaluation_interval = 1
)
```

Imagine the primary metric is the accuracy of the model. When after the first five trials, the best performing model has an accuracy of 0.9, any new model needs to perform better than (0.9-0.2) or 0.7. If the new model's accuracy is higher than 0.7, the sweep job will continue. If the new model has an accuracy score lower than 0.7, the policy will terminate the sweep job.



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

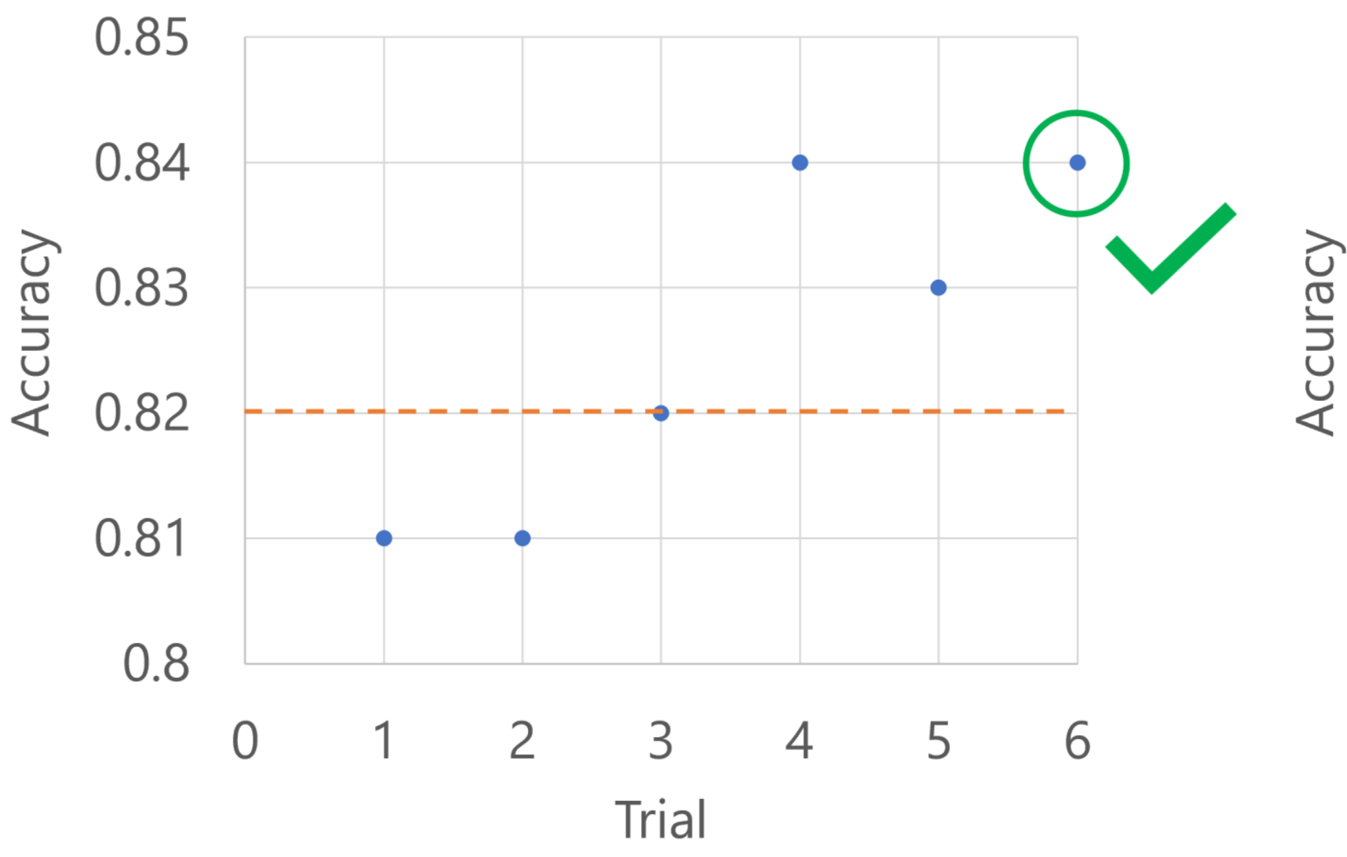
A median stopping policy abandons trials where the target performance metric is worse than the median of the running averages for all trials.

For example, the following code applies a median stopping policy with a delay of five trials and evaluates the policy at every interval.

PythonCopy

```python
from azure.ai.ml.sweep import MedianStoppingPolicy

sweep_job.early_termination = MedianStoppingPolicy(
    delay_evaluation = 5,
    evaluation_interval = 1
)
```

Imagine the primary metric is the accuracy of the model. When the accuracy is logged for the sixth trial, the metric needs to be higher than the median of the accuracy scores so far. Suppose the median of the accuracy scores so far is 0.82. If the new model's accuracy is higher than 0.82, the sweep job will continue. If the new model has an accuracy score lower than 0.82, the policy will stop the sweep job, and no new models will be trained.



Truncation selection policy

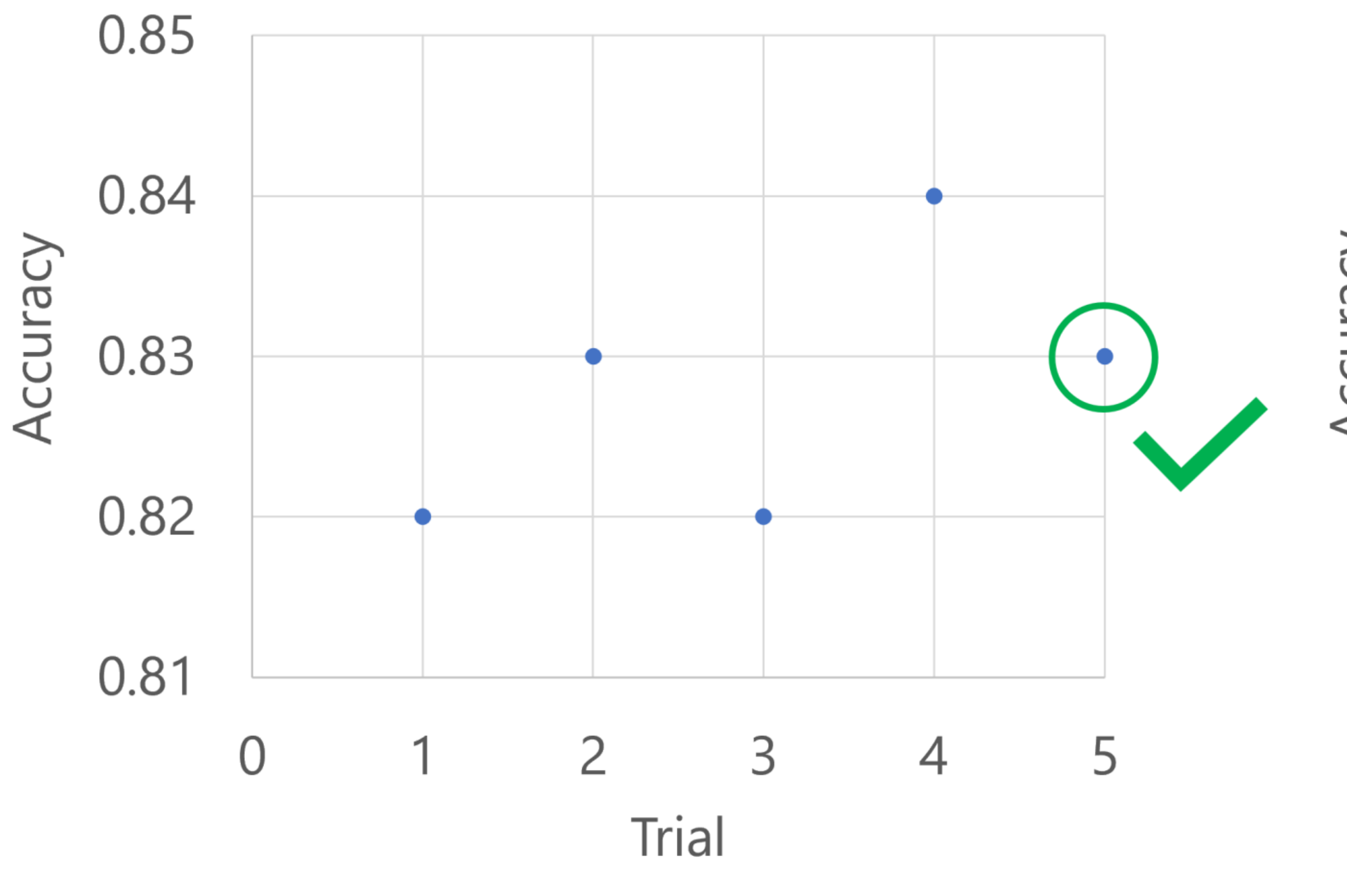
A truncation selection policy cancels the lowest performing *X*% of trials at each evaluation interval based on the *truncation_percentage* value you specify for *X*.

For example, the following code applies a truncation selection policy with a delay of four trials, evaluates the policy at every interval, and uses a truncation percentage of 20%.

PythonCopy

```python
from azure.ai.ml.sweep import TruncationSelectionPolicy

sweep_job.early_termination = TruncationSelectionPolicy(
    evaluation_interval=1,
    truncation_percentage=20,
    delay_evaluation=4
)
```

Imagine the primary metric is the accuracy of the model. When the accuracy is logged for the fifth trial, the metric should **not** be in the worst 20% of the trials so far. In this case, 20% translates to one trial. In other words, if the fifth trial is **not** the worst performing model so far, the sweep job will continue. If the fifth trial has the lowest accuracy score of all trials so far, the sweep job will stop.



Use a sweep job for hyperparameter tuning

Completed 100 XP

- 8 minutes

In Azure Machine Learning, you can tune hyperparameters by running a **sweep job**.

Create a training script for hyperparameter tuning

To run a sweep job, you need to create a training script just the way you would do for any other training job, except that your script ***must***:

- Include an argument for each hyperparameter you want to vary.
- Log the target performance metric with **MLflow**. A logged metric enables the sweep job to evaluate the performance of the trials it initiates, and identify the one that produces the best performing model.

Note

Learn how to **track machine learning experiments and models with MLflow within Azure Machine Learning**.

For example, the following example script trains a logistic regression model using a --regularization argument to set the *regularization rate* hyperparameter, and logs the *accuracy* metric with the name `Accuracy`:

PythonCopy

```python
import argparse
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import mlflow

# 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

# load the training dataset
data = pd.read_csv("data.csv")

# separate features and labels, and split for training/validatiom
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)
mlflow.log_metric("Accuracy", acc)
```

Configure and run a sweep job

To prepare the sweep job, you must first create a base **command job** that specifies which script to run and defines the parameters used by the script:

PythonCopy

```python
from azure.ai.ml import command

# configure command job as base
job = command(
    code="./src",
    command="python train.py --regularization ${{inputs.reg_rate}}",
    inputs={
        "reg_rate": 0.01,
    },
    environment="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest",
    compute="aml-cluster",
    )
```

You can then override your input parameters with your search space:

PythonCopy

```python
from azure.ai.ml.sweep import Choice

command_job_for_sweep = job(
    reg_rate=Choice(values=[0.01, 0.1, 1]),
)
```

Finally, call `sweep()` on your command job to sweep over your search space:

PythonCopy

```python
from azure.ai.ml import MLClient

# apply the sweep parameter to obtain the sweep_job
sweep_job = command_job_for_sweep.sweep(
    compute="aml-cluster",
    sampling_algorithm="grid",
    primary_metric="Accuracy",
    goal="Maximize",
)

# set the name of the sweep job experiment
sweep_job.experiment_name="sweep-example"

# define the limits for this sweep
sweep_job.set_limits(max_total_trials=4, max_concurrent_trials=2, timeout=7200)

# submit the sweep
returned_sweep_job = ml_client.create_or_update(sweep_job)
```

Monitor and review sweep jobs

You can monitor sweep jobs in Azure Machine Learning studio. The sweep job will initiate trials for each hyperparameter combination to be tried. For each trial, you can review all logged metrics.

Additionally, you can evaluate and compare models by visualizing the trials in the studio. You can adjust each chart to show and compare the hyperparameter values and metrics for each trial.

Tip

Learn more about how to **visualize hyperparameter tuning jobs**.

No termination policy (default)

If no policy is specified, the hyperparameter tuning service will let all training jobs execute to completion.

PythonCopy

```python
sweep_job.early_termination = None
```

Picking an early termination policy

- For a conservative policy that provides savings without terminating promising jobs, consider a Median Stopping Policy with `evaluation_interval` 1 and `delay_evaluation` 5. These are conservative settings that can provide approximately 25%-35% savings with no loss on primary metric (based on our evaluation data).
- For more aggressive savings, use Bandit Policy with a smaller allowable slack or Truncation Selection Policy with a larger truncation percentage.

Set limits for your sweep job

Control your resource budget by setting limits for your sweep job.

- `max_total_trials`: Maximum number of trial jobs. Must be an integer between 1 and 1000.
- `max_concurrent_trials`: (optional) Maximum number of trial jobs that can run concurrently. If not specified, max_total_trials number of jobs launch in parallel. If specified, must be an integer between 1 and 1000.
- `timeout`: Maximum time in seconds the entire sweep job is allowed to run. Once this limit is reached the system will cancel the sweep job, including all its trials.

- `trial_timeout`: Maximum time in seconds each trial job is allowed to run. Once this limit is reached the system will cancel the trial.

Note

If both max_total_trials and timeout are specified, the hyperparameter tuning experiment terminates when the first of these two thresholds is reached.

Note

The number of concurrent trial jobs is gated on the resources available in the specified compute target. Ensure that the compute target has the available resources for the desired concurrency.

PythonCopy

```
sweep_job.set_limits(max_total_trials=20, max_concurrent_trials=4, timeout=1200)
```

This code configures the hyperparameter tuning experiment to use a maximum of 20 total trial jobs, running four trial jobs at a time with a timeout of 1200 seconds for the entire sweep job.

Configure hyperparameter tuning experiment

To configure your hyperparameter tuning experiment, provide the following:

- The defined hyperparameter search space
- Your sampling algorithm
- Your early termination policy
- Your objective
- Resource limits
- CommandJob or CommandComponent
- SweepJob

SweepJob can run a hyperparameter sweep on the Command or Command Component.

Note

The compute target used in `sweep_job` must have enough resources to satisfy your concurrency level. For more information on compute targets, see **Compute targets**.

Configure your hyperparameter tuning experiment:

PythonCopy

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

```python
from azure.ai.ml.sweep import Choice, Uniform, MedianStoppingPolicy
from azure.identity import DefaultAzureCredential

# Create your base command job
command_job = command(
    code="./src",
    command="python main.py --iris-csv ${{inputs.iris_csv}} --learning-rate
${{inputs.learning_rate}} --boosting ${{inputs.boosting}}",
    environment="AzureML-lightgbm-3.2-ubuntu18.04-py37-cpu@latest",
    inputs={
        "iris_csv": Input(
            type="uri_file",

path="https://azuremlexamples.blob.core.windows.net/datasets/iris.csv",
        ),
        "learning_rate": 0.9,
        "boosting": "gbdt",
    },
    compute="cpu-cluster",
)

# Override your inputs with parameter expressions
command_job_for_sweep = command_job(
    learning_rate=Uniform(min_value=0.01, max_value=0.9),
    boosting=Choice(values=["gbdt", "dart"]),
)

# Call sweep() on your command job to sweep over your parameter expressions
sweep_job = command_job_for_sweep.sweep(
    compute="cpu-cluster",
    sampling_algorithm="random",
    primary_metric="test-multi_logloss",
    goal="Minimize",
)

# Specify your experiment details
sweep_job.display_name = "lightgbm-iris-sweep-example"
sweep_job.experiment_name = "lightgbm-iris-sweep-example"
sweep_job.description = "Run a hyperparameter sweep job for LightGBM on Iris
dataset."

# Define the limits for this sweep
sweep_job.set_limits(max_total_trials=20, max_concurrent_trials=10, timeout=7200)

# Set early stopping on this one
sweep_job.early_termination = MedianStoppingPolicy(
    delay_evaluation=5, evaluation_interval=2
)
```

The `command_job` is called as a function so we can apply the parameter expressions to the sweep inputs. The `sweep` function is then configured with `trial`, `sampling-algorithm`, `objective`, `limits`, and `compute`. The above code snippet is taken from the sample notebook Run hyperparameter sweep on a Command or CommandComponent. In this sample, the `learning_rate` and `boosting` parameters will be tuned. Early stopping of jobs will be determined by a `MedianStoppingPolicy`, which stops a job whose primary metric value is worse than the median of the averages across all training jobs.(see MedianStoppingPolicy class reference).

To see how the parameter values are received, parsed, and passed to the training script to be tuned, refer to this code sample

Important

Every hyperparameter sweep job restarts the training from scratch, including rebuilding the model and *all the data loaders*. You can minimize this cost by using an Azure Machine Learning pipeline or manual process to do as much data preparation as possible prior to your training jobs.

Submit hyperparameter tuning experiment

After you define your hyperparameter tuning configuration, submit the job:

PythonCopy

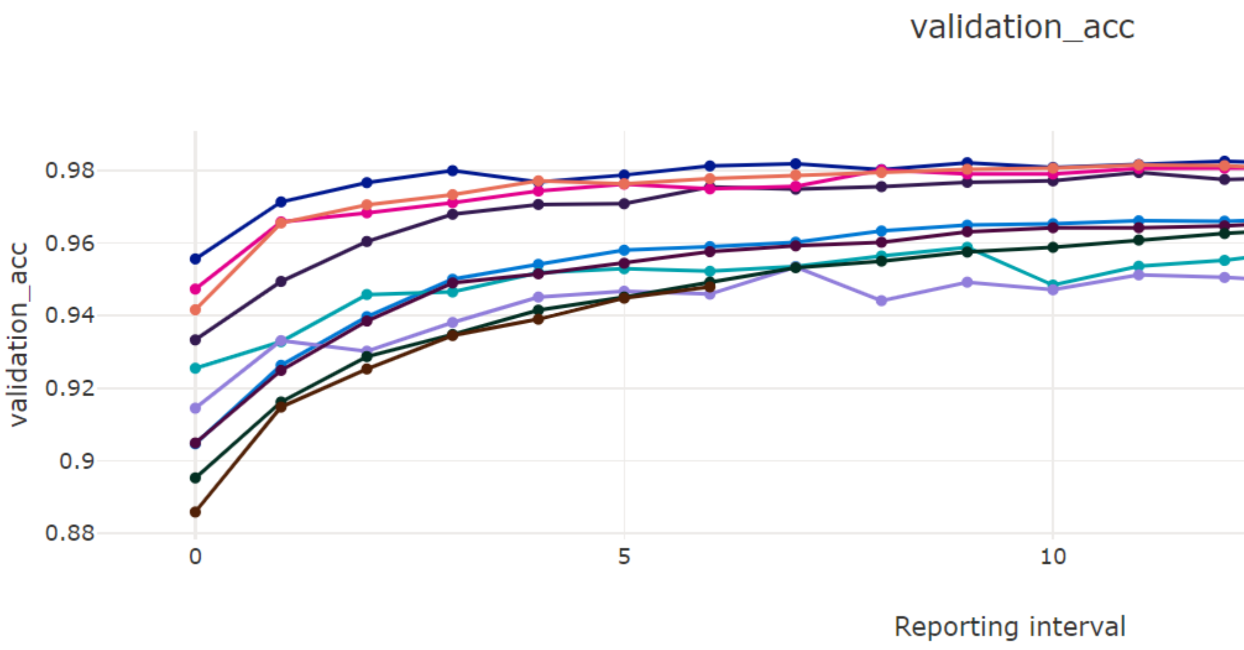
```python
# submit the sweep
returned_sweep_job = ml_client.create_or_update(sweep_job)
# get a URL for the status of the job
returned_sweep_job.services["Studio"].endpoint
```

Visualize hyperparameter tuning jobs

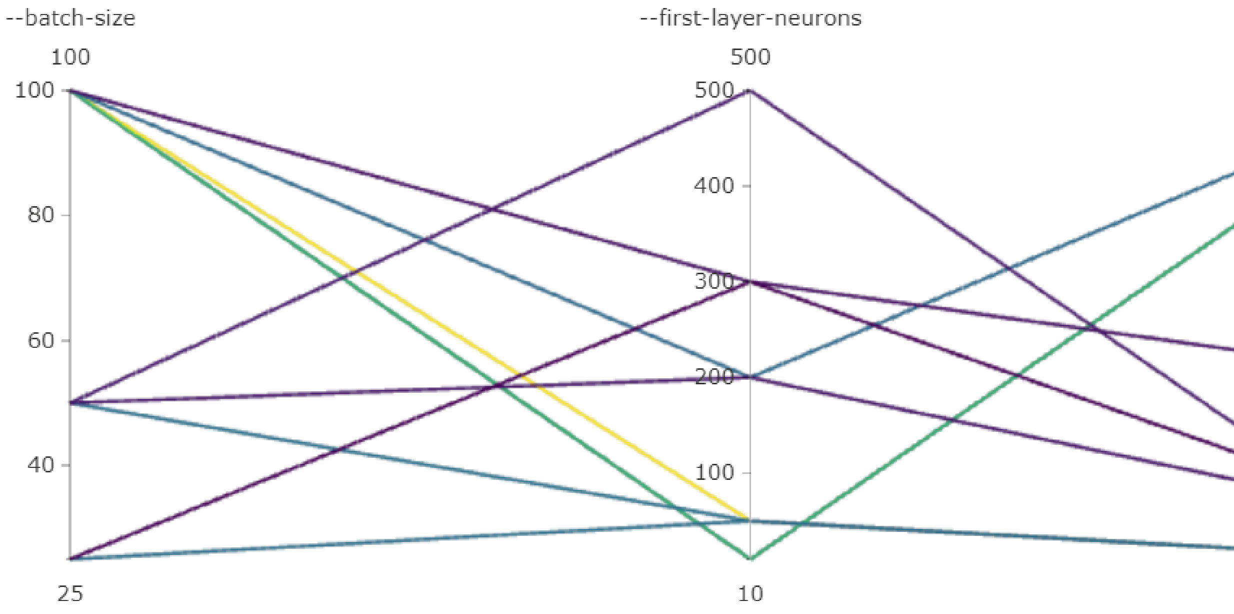
You can visualize all of your hyperparameter tuning jobs in the Azure Machine Learning studio. For more information on how to view an experiment in the portal, see View job records in the studio.

- **Metrics chart**: This visualization tracks the metrics logged for each hyperdrive child job over the duration of hyperparameter tuning. Each line represents a child job, and each point measures the primary metric value at that iteration of runtime.

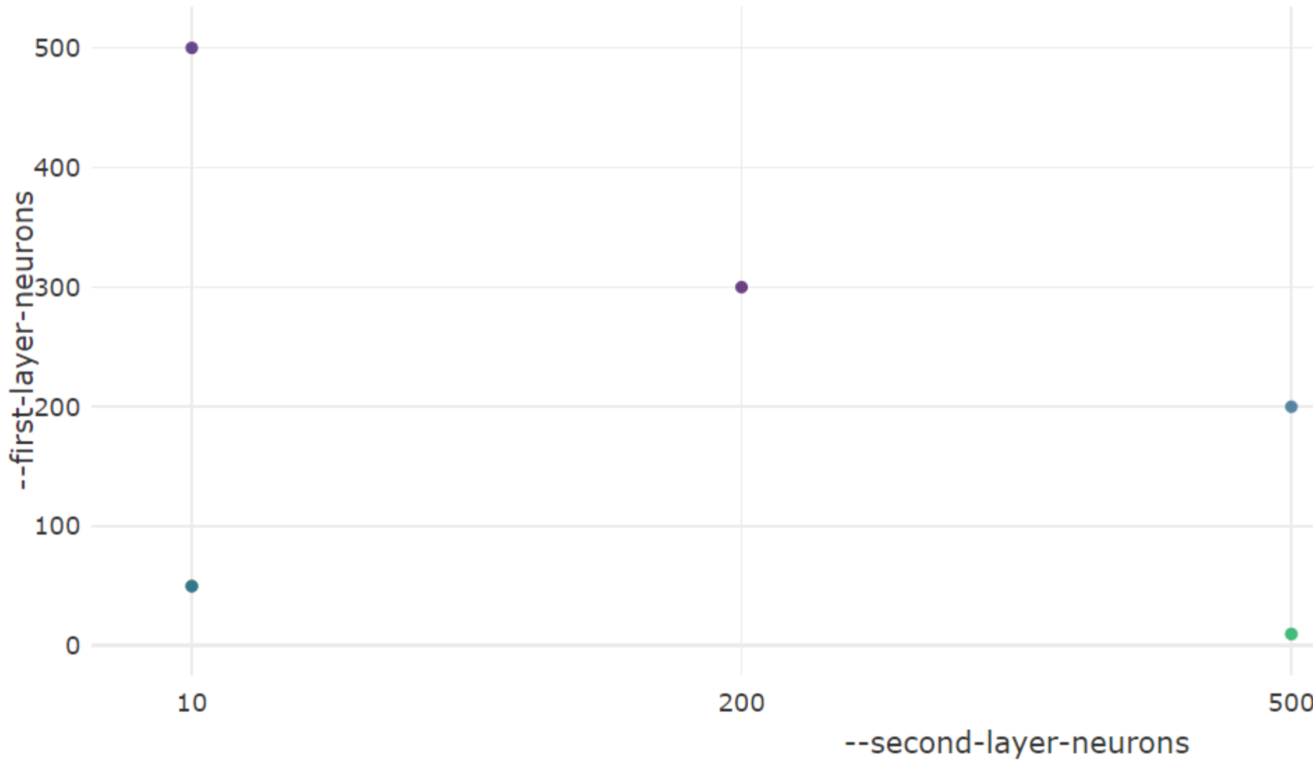
Child runs ⓘ



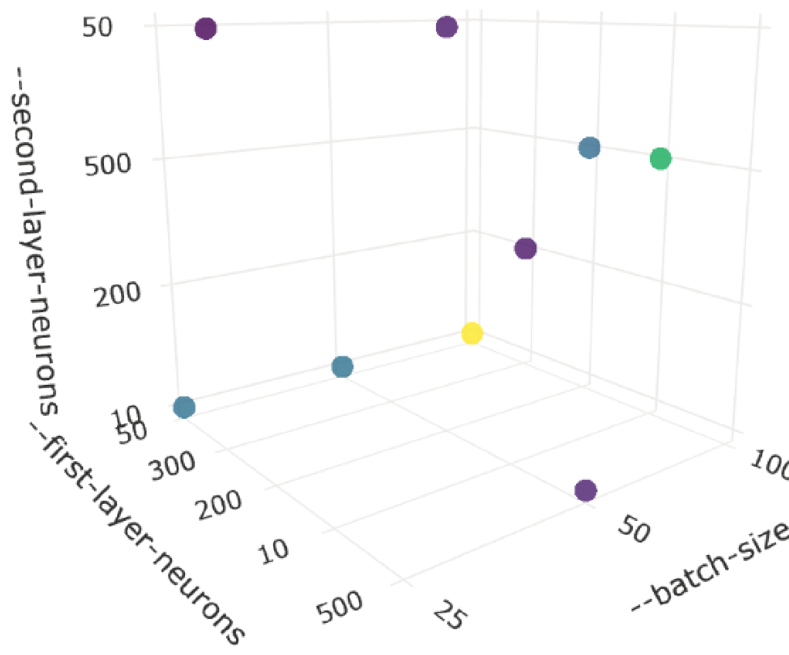
- **Parallel Coordinates Chart**: This visualization shows the correlation between primary metric performance and individual hyperparameter values. The chart is interactive via movement of axes (click and drag by the axis label), and by highlighting values across a single axis (click and drag vertically along a single axis to highlight a range of desired values). The parallel coordinates chart includes an axis on the rightmost portion of the chart that plots the best metric value corresponding to the hyperparameters set for that job instance. This axis is provided in order to project the chart gradient legend onto the data in a more readable fashion.



- **2-Dimensional Scatter Chart**: This visualization shows the correlation between any two individual hyperparameters along with their associated primary metric value.



- **3-Dimensional Scatter Chart**: This visualization is the same as 2D but allows for three hyperparameter dimensions of correlation with the primary metric value. You can also click and drag to reorient the chart to view different correlations in 3D space.



Find the best trial job

Once all of the hyperparameter tuning jobs have completed, retrieve your best trial outputs:

PythonCopy

```python
# Download best trial model output
ml_client.jobs.download(returned_sweep_job.name, output_name="model")
```

You can use the CLI to download all default and named outputs of the best trial job and logs of the sweep job.

Copy

```
az ml job download --name <sweep-job> --all
```

Optionally, to solely download the best trial output

Copy

```
az ml job download --name <sweep-job> --output-name model
```

Create components for Azure ML Pipelines

Completed100 XP

- 8 minutes

Components allow you to create reusable scripts that can easily be shared across users within the same Azure Machine Learning workspace. You can also use components to build an Azure Machine Learning pipeline.

Use a component

There are two main reasons why you'd use components:

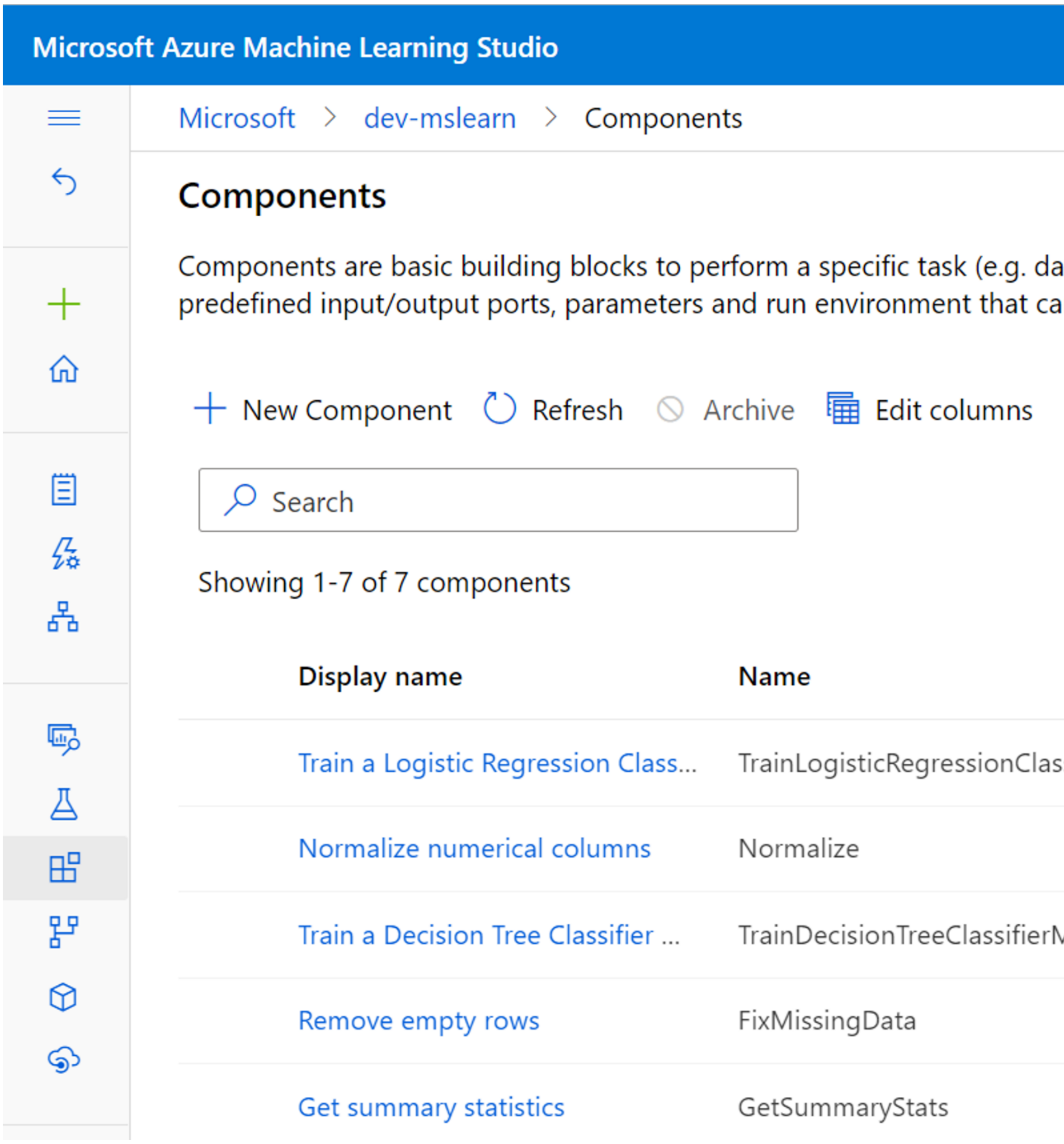
- To build a pipeline.
- To share ready-to-go code.

You'll want to create components when you're *preparing your code for scale*. When you're done with experimenting and developing, and ready to move your model to production.

Within Azure Machine Learning, you can create a component to store code (in your preferred language) within the workspace. Ideally, you design a component to perform a specific action that is relevant to your machine learning workflow.

For example, a component may consist of a Python script that normalizes your data, trains a machine learning model, or evaluates a model.

Components can be easily shared to other Azure Machine Learning users, who can reuse components in their own Azure Machine Learning pipelines.


Microsoft Azure Machine Learning Studio

Microsoft > dev-mslearn > Components

Components

Components are basic building blocks to perform a specific task (e.g. da
predefined input/output ports, parameters and run environment that ca

Showing 1-7 of 7 components

Display name	Name
Train a Logistic Regression Class...	TrainLogisticRegressionClas
Normalize numerical columns	Normalize
Train a Decision Tree Classifier ...	TrainDecisionTreeClassifierM
Remove empty rows	FixMissingData
Get summary statistics	GetSummaryStats

Create a component

A component consists of three parts:

- **Metadata**: Includes the component's name, version, etc.
- **Interface**: Includes the expected input parameters (like a dataset or hyperparameter) and expected output (like metrics and artifacts).

- **Command, code and environment**: Specifies how to run the code.

To create a component, you need two files:

- A script that contains the workflow you want to execute.
- A YAML file to define the metadata, interface, and command, code, and environment of the component.

You can create the YAML file, or use the `command_component()` function as a decorator to create the YAML file.

Tip

Here, we'll focus on creating a YAML file to create a component. Alternatively, learn more about **how to create components using command_component()**.

For example, you may have a Python script `prep.py` that prepares the data by removing missing values and normalizing the data:

PythonCopy

```python
# import libraries
import argparse
import pandas as pd
import numpy as np
from pathlib import Path
from sklearn.preprocessing import MinMaxScaler

# setup arg parser
parser = argparse.ArgumentParser()

# add arguments
parser.add_argument("--input_data", dest='input_data',
                    type=str)
parser.add_argument("--output_data", dest='output_data',
                    type=str)

# parse args
args = parser.parse_args()

# read the data
df = pd.read_csv(args.input_data)

# remove missing values
df = df.dropna()

# normalize the data
scaler = MinMaxScaler()
num_cols = ['feature1','feature2','feature3','feature4']
df[num_cols] = scaler.fit_transform(df[num_cols])

# save the data as a csv
output_df = df.to_csv(
    (Path(args.output_data) / "prepped-data.csv"),
    index = False
)
```

To create a component for the `prep.py` script, you'll need a YAML file `prep.yml`:

ymlCopy

```yml
$schema: https://azuremlschemas.azureedge.net/latest/commandComponent.schema.json
name: prep_data
display_name: Prepare training data
version: 1
type: command
inputs:
  input_data:
    type: uri_file
outputs:
  output_data:
    type: uri_file
code: ./src
environment: azureml:AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest
command: >-
  python prep.py
  --input_data ${{inputs.input_data}}
  --output_data ${{outputs.output_data}}
```

Notice that the YAML file refers to the `prep.py` script, which is stored in the `src` folder. You can load the component with the following code:

PythonCopy

```python
from azure.ai.ml import load_component
parent_dir = ""

loaded_component_prep = load_component(source=parent_dir + "./prep.yml")
```

When you've loaded the component, you can use it in a pipeline or register the component.

Register a component

To use components in a pipeline, you'll need the script and the YAML file. To make the components accessible to other users in the workspace, you can also register components to the Azure Machine Learning workspace.

You can register a component with the following code:

PythonCopy

```python
prep = ml_client.components.create_or_update(prepare_data_component)
```

Create a pipeline

Completed100 XP

- 6 minutes

In Azure Machine Learning, a **pipeline** is a workflow of machine learning tasks in which each task is defined as a **component**.

Components can be arranged sequentially or in parallel, enabling you to build sophisticated flow logic to orchestrate machine learning operations. Each component can be run on a specific compute target, making it possible to combine different types of processing as required to achieve an overall goal.

A pipeline can be executed as a process by running the pipeline as a **pipeline job**. Each component is executed as a **child job** as part of the overall pipeline job.

Build a pipeline

An Azure Machine Learning pipeline is defined in a YAML file. The YAML file includes the pipeline job name, inputs, outputs, and settings.

You can create the YAML file, or use the `@pipeline()` function to create the YAML file.

Tip

Review the **reference documentation for the `@pipeline()` function**.

For example, if you want to build a pipeline that first prepares the data, and then trains the model, you can use the following code:

PythonCopy

```python
from azure.ai.ml.dsl import pipeline

@pipeline()
def pipeline_function_name(pipeline_job_input):
    prep_data = loaded_component_prep(input_data=pipeline_job_input)
    train_model = 
loaded_component_train(training_data=prep_data.outputs.output_data)

    return {
        "pipeline_job_transformed_data": prep_data.outputs.output_data,
        "pipeline_job_trained_model": train_model.outputs.model_output,
    }
```

To pass a registered data asset as the pipeline job input, you can call the function you created with the data asset as input:

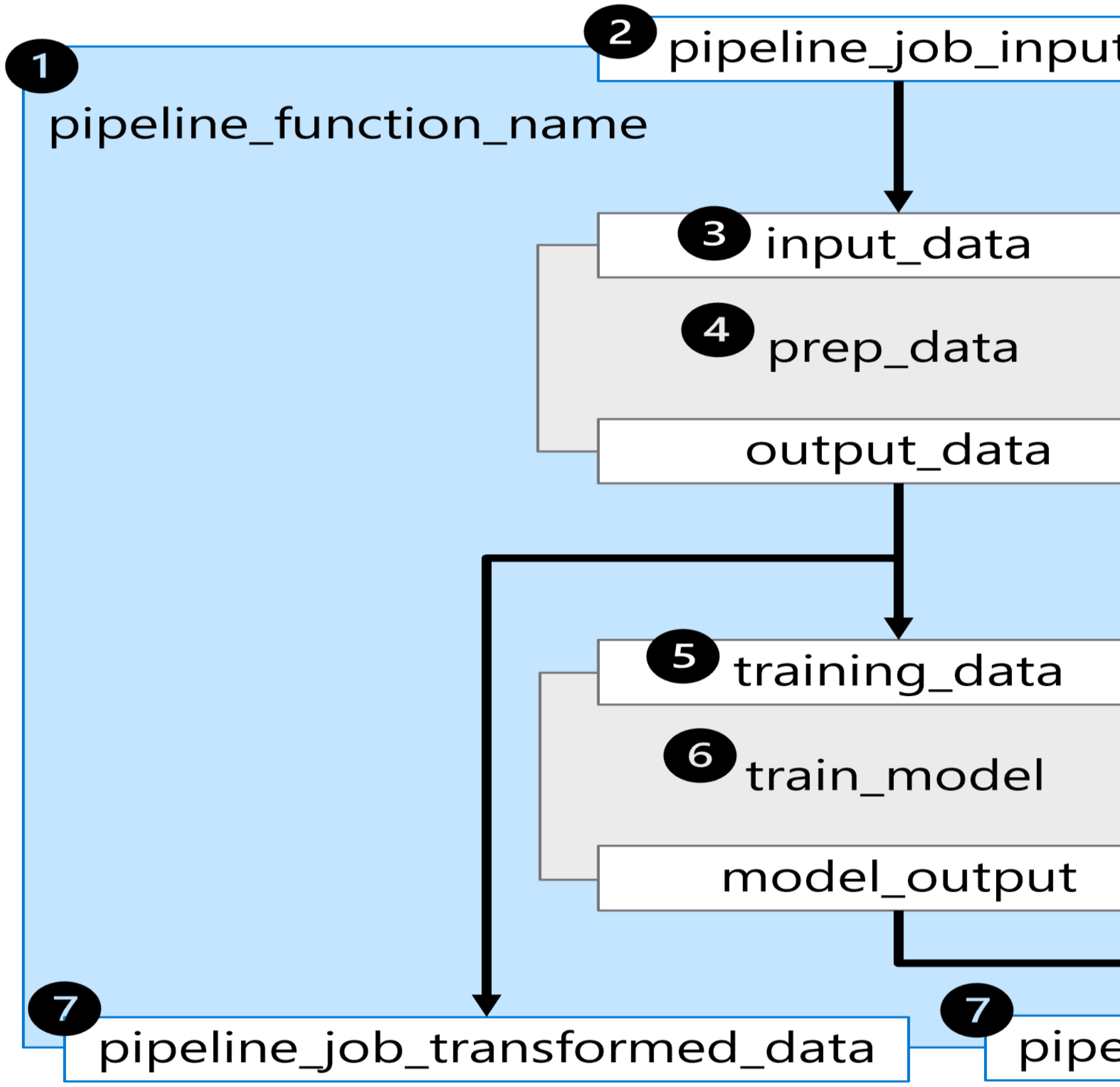
PythonCopy

```python
from azure.ai.ml import Input
from azure.ai.ml.constants import AssetTypes

pipeline_job = pipeline_function_name(
    Input(type=AssetTypes.URI_FILE,
    path="azureml:data:1"
))
```

The `@pipeline()` function builds a pipeline consisting of two sequential steps, represented by the two loaded components.

To understand the pipeline built in the example, let's explore it step by step:

1. The pipeline is built by defining the function `pipeline_function_name`.
2. The pipeline function expects `pipeline_job_input` as the overall pipeline input.
3. The first pipeline step requires a value for the input parameter `input_data`. The value for the input will be the value of `pipeline_job_input`.
4. The first pipeline step is defined by the loaded component for `prep_data`.
5. The value of the `output_data` of the first pipeline step is used for the expected input `training_data` of the second pipeline step.
6. The second pipeline step is defined by the loaded component for `train_model` and results in a trained model referred to by `model_output`.
7. Pipeline outputs are defined by returning variables from the pipeline function. There are two outputs:
   - `pipeline_job_transformed_data` with the value of `prep_data.outputs.output_data`
   - `pipeline_job_trained_model` with the value of `train_model.outputs.model_output`



The result of running the `@pipeline()` function is a YAML file that you can review by printing the `pipeline_job` object you created when calling the function:

PythonCopy

```
print(pipeline_job)
```

The output will be formatted as a YAML file, which includes the configuration of the pipeline and its components. Some parameters included in the YAML file are shown in the following example.

ymlCopy

```yaml
display_name: pipeline_function_name
type: pipeline
inputs:
  pipeline_job_input:
    type: uri_file
    path: azureml:data:1
outputs:
  pipeline_job_transformed_data: null
  pipeline_job_trained_model: null
jobs:
  prep_data:
    type: command
    inputs:
      input_data:
        path: ${{parent.inputs.pipeline_job_input}}
    outputs:
      output_data: ${{parent.outputs.pipeline_job_transformed_data}}
  train_model:
    type: command
    inputs:
      input_data:
        path: ${{parent.outputs.pipeline_job_transformed_data}}
    outputs:
      output_model: ${{parent.outputs.pipeline_job_trained_model}}
tags: {}
properties: {}
settings: {}
```

Run a pipeline job

Completed100 XP

- 9 minutes

When you've built a component-based pipeline in Azure Machine Learning, you can run the workflow as a **pipeline job**.

Configure a pipeline job

A pipeline is defined in a YAML file, which you can also create using the `@pipeline()` function. After you've used the function, you can edit the pipeline configurations by specifying which parameters you want to change and the new value.

For example, you may want to change the output mode for the pipeline job outputs:

PythonCopy

```python
# change the output mode
pipeline_job.outputs.pipeline_job_transformed_data.mode = "upload"
pipeline_job.outputs.pipeline_job_trained_model.mode = "upload"
```

Or, you may want to set the default pipeline compute. When a compute isn't specified for a component, it will use the default compute instead:

PythonCopy

```python
# set pipeline level compute
pipeline_job.settings.default_compute = "aml-cluster"
```

You may also want to change the default datastore to where all outputs will be stored:

PythonCopy

```python
# set pipeline level datastore
pipeline_job.settings.default_datastore = "workspaceblobstore"
```

To review your pipeline configuration, you can print the pipeline job object:

PythonCopy

```python
print(pipeline_job)
```

Run a pipeline job

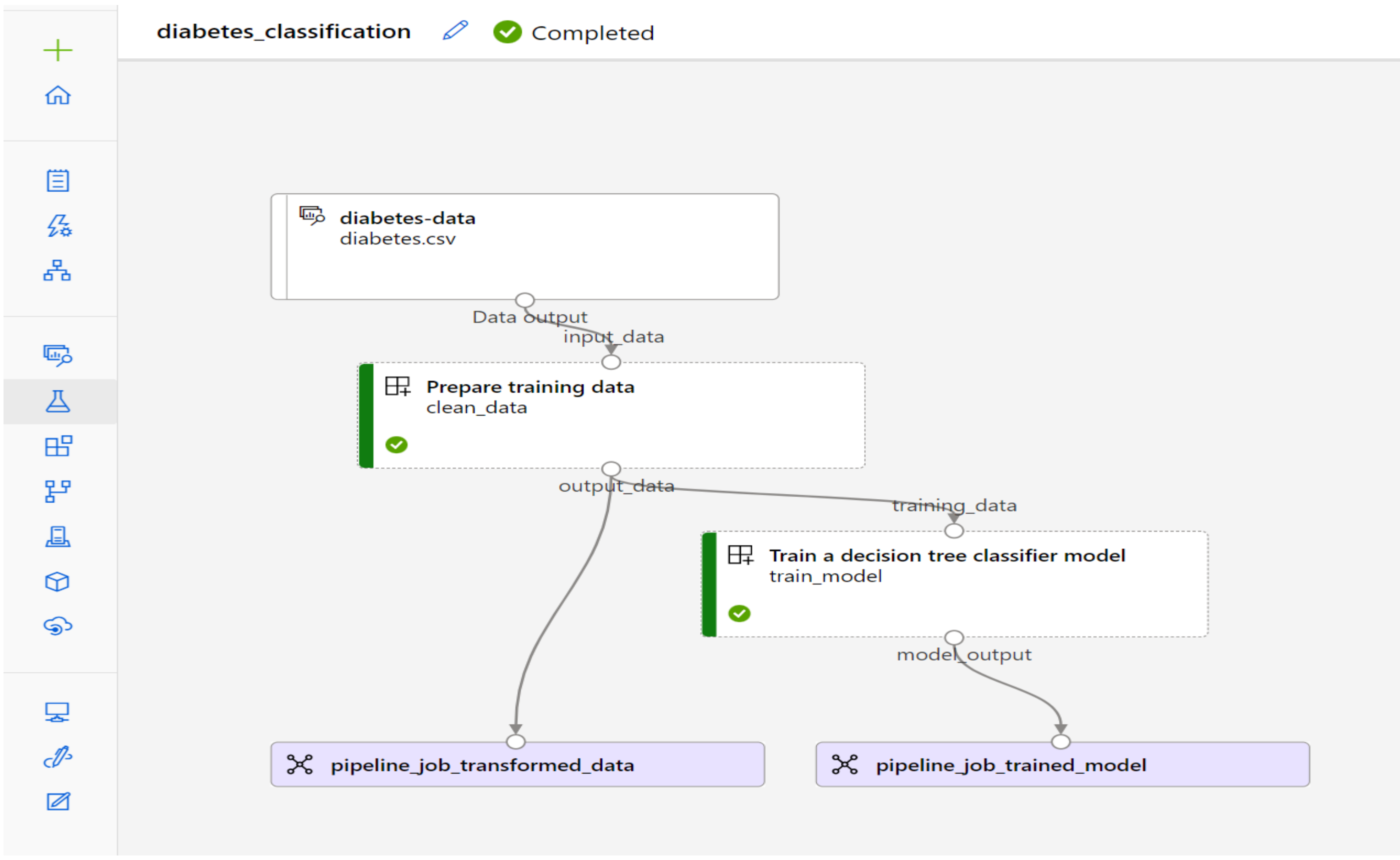
When you've configured the pipeline, you're ready to run the workflow as a pipeline job.

To submit the pipeline job, run the following code:

PythonCopy

```python
# submit job to workspace
pipeline_job = ml_client.jobs.create_or_update(
    pipeline_job, experiment_name="pipeline_job"
)
```

After you submit a pipeline job, a new job will be created in the Azure Machine Learning workspace. A pipeline job also contains child jobs, which represent the execution of the individual components. The Azure Machine Learning studio creates a graphical representation of your pipeline. You can expand the **Job overview** to explore the pipeline parameters, outputs, and child jobs:



To troubleshoot a failed pipeline, you can check the outputs and logs of the pipeline job and its child jobs.

- If there's an issue with the configuration of the pipeline itself, you'll find more information in the outputs and logs of the pipeline job.
- If there's an issue with the configuration of a component, you'll find more information in the outputs and logs of the child job of the failed component.

Schedule a pipeline job

A pipeline is ideal if you want to get your model ready for production. Pipelines are especially useful for automating the retraining of a machine learning model. To automate the retraining of a model, you can schedule a pipeline.

To schedule a pipeline job, you'll use the `JobSchedule` class to associate a schedule to a pipeline job.

There are various ways to create a schedule. A simple approach is to create a time-based schedule using the `RecurrenceTrigger` class with the following parameters:

- `frequency`: Unit of time to describe how often the schedule fires. Value can be either `minute`, `hour`, `day`, `week`, or `month`.

- `interval`: Number of frequency units to describe how often the schedule fires. Value needs to be an integer.

To create a schedule that fires every minute, run the following code:

PythonCopy

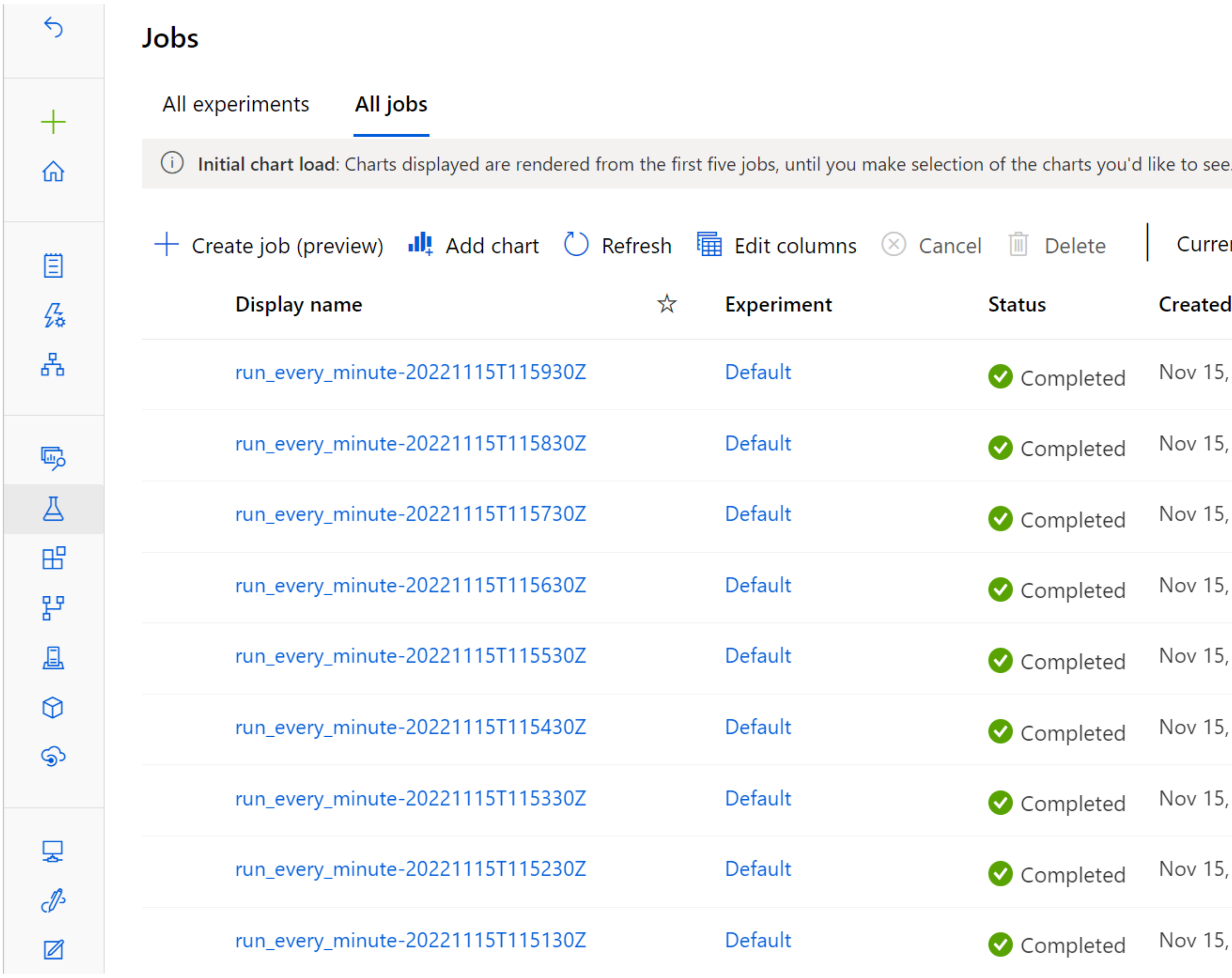
```python
from azure.ai.ml.entities import RecurrenceTrigger

schedule_name = "run_every_minute"

recurrence_trigger = RecurrenceTrigger(
    frequency="minute",
    interval=1,
)
```

To schedule a pipeline, you'll need `pipeline_job` to represent the pipeline you've built:

PythonCopy

```python
from azure.ai.ml.entities import JobSchedule

job_schedule = JobSchedule(
    name=schedule_name, trigger=recurrence_trigger, create_job=pipeline_job
)

job_schedule = ml_client.schedules.begin_create_or_update(
    schedule=job_schedule
).result()
```

The display names of the jobs triggered by the schedule will be prefixed with the name of your schedule. You can review the jobs in the Azure Machine Learning studio:



Initial chart load: Charts displayed are rendered from the first five jobs, until you make selection of the charts you'd like to see

Display name	Experiment	Status	Created
run_every_minute-20221115T115930Z	Default	Completed	Nov 15,
run_every_minute-20221115T115830Z	Default	Completed	Nov 15,
run_every_minute-20221115T115730Z	Default	Completed	Nov 15,
run_every_minute-20221115T115630Z	Default	Completed	Nov 15,
run_every_minute-20221115T115530Z	Default	Completed	Nov 15,
run_every_minute-20221115T115430Z	Default	Completed	Nov 15,
run_every_minute-20221115T115330Z	Default	Completed	Nov 15,
run_every_minute-20221115T115230Z	Default	Completed	Nov 15,
run_every_minute-20221115T115130Z	Default	Completed	Nov 15,

To delete a schedule, you first need to disable it:

PythonCopy

```python
ml_client.schedules.begin_disable(name=schedule_name).result()
ml_client.schedules.begin_delete(name=schedule_name).result()
```

YAML: hello pipeline

YAMLCopy

```yaml
$schema: https://azuremlschemas.azureedge.net/latest/pipelineJob.schema.json
type: pipeline
display_name: hello_pipeline
jobs:
  hello_job:
    command: echo "hello"
    environment: azureml:AzureML-sklearn-1.0-ubuntu20.04-py38-cpu@latest
    compute: azureml:cpu-cluster
  world_job:
    command: echo "world"
    environment: azureml:AzureML-sklearn-1.0-ubuntu20.04-py38-cpu@latest
    compute: azureml:cpu-cluster
```

YAML: input/output dependency

YAMLCopy

```yaml
$schema: https://azuremlschemas.azureedge.net/latest/pipelineJob.schema.json
type: pipeline
display_name: hello_pipeline_io
jobs:
  hello_job:
    command: echo "hello" && echo "world" > ${{outputs.world_output}}/world.txt
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
    compute: azureml:cpu-cluster
    outputs:
      world_output:
  world_job:
    command: cat ${{inputs.world_input}}/world.txt
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
    compute: azureml:cpu-cluster
    inputs:
      world_input: ${{parent.jobs.hello_job.outputs.world_output}}
```

YAML: common pipeline job settings

YAMLCopy

```yaml
$schema: https://azuremlschemas.azureedge.net/latest/pipelineJob.schema.json
type: pipeline
display_name: hello_pipeline_settings

settings:
  default_datastore: azureml:workspaceblobstore
  default_compute: azureml:cpu-cluster
jobs:
  hello_job:
    command: echo 202204190 & echo "hello"
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
  world_job:
    command: echo 202204190 & echo "hello"
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
```

YAML: top-level input and overriding common pipeline job settings

YAMLCopy

```yaml
$schema: https://azuremlschemas.azureedge.net/latest/pipelineJob.schema.json
type: pipeline
display_name: hello_pipeline_abc
settings:
    default_compute: azureml:cpu-cluster

inputs:
  hello_string_top_level_input: "hello world"
jobs:
  a:
    command: echo hello ${{inputs.hello_string}}
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
    inputs:
      hello_string: ${{parent.inputs.hello_string_top_level_input}}
  b:
    command: echo "world" >> ${{outputs.world_output}}/world.txt
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
    outputs:
      world_output:
  c:
    command: echo ${{inputs.world_input}}/world.txt
    environment: azureml://registries/azureml/environments/sklearn-
1.0/labels/latest
    inputs:
      world_input: ${{parent.jobs.b.outputs.world_output}}
```

1.

You're creating a pipeline that includes two steps. Step 1 prepares some data, and step 2 uses the preprocessed data to train a model. Which option should you use as input to the second step to train the model?

Correct. `prep_data.outputs.output_data` **is the output of the step that prepares the data.**

2.

You've built a pipeline that you want to run every week. You want to take a simple approach to creating a schedule. Which class can you use to create the schedule that runs once per week?

Correct. You need the RecurrenceTrigger class to create a schedule that runs at a regular interval

Convert a notebook to a script

Completed100 XP

- 9 minutes

When you've used notebooks for experimentation and development, you'll first need to convert a notebook to a script. Alternatively, you might choose to skip using notebooks and work only with scripts. Either way, there are some recommendations when creating scripts to have production-ready code.

Scripts are ideal for testing and automation in your production environment. To create a production-ready script, you'll need to:

- Remove nonessential code.
- Refactor your code into functions.
- Test your script in the terminal.

Remove all nonessential code

The main benefit of using notebooks is being able to quickly explore your data. For example, you can use `print()` and `df.describe()` statements to explore your data and variables. When you create a script that will be used for automation, you want to avoid including code written for exploratory purposes.

The first thing you therefore need to do to convert your code to production code is to remove the nonessential code. Especially when you'll run the code regularly, you want to avoid executing anything nonessential to reduce cost and compute time.

Refactor your code into functions

When using code in business processes, you want the code to be easy to read so that anyone can maintain it. One common approach to make code easier to read and test is to use functions.

For example, you might have used the following example code in a notebook to read and split the data:

PythonCopy

```python
# read and visualize the data
print("Reading data...")
df = pd.read_csv('diabetes.csv')
df.head()

# split data
print("Splitting data...")
X, y = 
df[['Pregnancies','PlasmaGlucose','DiastolicBloodPressure','TricepsThickness','SerumInsulin','BMI','DiabetesPedigree','Age']].values, df['Diabetic'].values

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, 
random_state=0)
```

As functions also allow you to test parts of your code, you might prefer to create *multiple smaller functions*, rather than one large function. If you want to test a part of your code, you can choose to only test a small part and avoid running more code than necessary.

You can refactor the code shown in the example into two functions:

- Read the data
- Split the data

An example of refactored code might be the following:

PythonCopy

```python
def main(csv_file):
    # read data
    df = get_data(csv_file)

    # split data
    X_train, X_test, y_train, y_test = split_data(df)

# function that reads the data
def get_data(path):
    df = pd.read_csv(path)

    return df

# function that splits the data
def split_data(df):
    X, y = 
df[['Pregnancies','PlasmaGlucose','DiastolicBloodPressure','TricepsThickness',
    'SerumInsulin','BMI','DiabetesPedigree','Age']].values, df['Diabetic'].values

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, 
random_state=0)

    return X_train, X_test, y_train, y_test
```

Note

You may have noticed that nonessential code was also omitted in the refactored code. You may choose to use `print` statements in production code if you'll review the script's output and you want to ensure all code ran as expected. However, when you know you're not going to review the output of a script in a terminal, it's best to remove any code that has no purpose.

Test your script

Before using scripts in production environments, for example by integrating them with automation pipelines, you'll want to test whether the scripts work as expected.

One simple way to test your script, is to run the script in a terminal. Within the Azure Machine Learning workspace, you can quickly run a script in the terminal of the compute instance.

When you open a script in the **Notebooks** page of the Azure Machine Learning studio, you can choose to **save and run the script in the terminal**.

Alternatively, you can navigate directly to the terminal of the compute instance. Navigate to the **Compute** page and select the **Terminal** of the compute instance you want to use. You can use the following command to run a Python script named `train.py`:

Copy

```
python train.py
```

Outputs of `print` statements will show in the terminal. Any possible errors will also appear in the terminal.

Run a script as a command job

Completed100 XP

- 5 minutes

When you have a script that train a machine learning model, you can run it as a command job in Azure Machine Learning.

Configure and submit a command job

To run a script as a command job, you'll need to configure and submit the job.

To configure a command job with the Python SDK (v2), you'll use the `command` function. To run a script, you'll need to specify values for the following parameters:

- `code`: The folder that includes the script to run.
- `command`: Specifies which file to run.
- `environment`: The necessary packages to be installed on the compute before running the command.
- `compute`: The compute to use to run the command.
- `display_name`: The name of the individual job.
- `experiment_name`: The name of the experiment the job belongs to.

Tip

Learn more about **the `command` function and all possible parameters** in the reference documentation for the Python SDK (v2).

You can configure a command job to run a file named `train.py`, on the compute cluster named `aml-cluster` with the following code:

PythonCopy

```python
from azure.ai.ml import command

# configure job
job = command(
    code="./src",
    command="python train.py",
    environment="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest",
    compute="aml-cluster",
    display_name="train-model",
    experiment_name="train-classification-model"
    )
```

When your job is configured, you can submit it, which will initiate the job and run the script:

PythonCopy

```python
# submit job
returned_job = ml_client.create_or_update(job)
```

You can monitor and review the job in the Azure Machine Learning studio. All jobs with the same experiment name will be grouped under the same experiment. You can find an individual job using the specified display name.

All inputs and outputs of a command job are tracked. You can review which command you specified, which compute was used, and which environment was used to run the script on the specified compute.

Use parameters in a command job

Completed100 XP

- 7 minutes

You can increase the flexibility of your scripts by using parameters. For example, you might have created a script that trains a machine learning model. You can use the same script to train a model on different datasets, or using various hyperparameter values.

Working with script arguments

To use parameters in a script, you must use a library such as `argparse` to read arguments passed to the script and assign them to variables.

For example, the following script reads an arguments named `training_data`, which specifies the path to the training data.

PythonCopy

```python
# import libraries
import argparse
import pandas as pd
from sklearn.linear_model import LogisticRegression

def main(args):
    # read data
    df = get_data(args.training_data)

# function that reads the data
def get_data(path):
    df = pd.read_csv(path)

    return df

def parse_args():
    # setup arg parser
    parser = argparse.ArgumentParser()

    # add arguments
    parser.add_argument("--training_data", dest='training_data',
                        type=str)

    # parse args
    args = parser.parse_args()

    # return args
    return args

# run script
if __name__ == "__main__":

    # parse args
    args = parse_args()

    # run main function
    main(args)
```

Any parameters you expect should be defined in the script. In the script, you can specify what type of value you expect for each parameter and whether you want to set a default value.

Passing arguments to a script

To pass parameter values to a script, you need to provide the argument value in the command.

For example, if you would pass a parameter value when running a script in a terminal, you would use the following command:

Copy

```
python train.py --training_data diabetes.csv
```

In the example, `diabetes.csv` is a local file. Alternatively, you could specify the path to a data asset created in the Azure Machine Learning workspace.

Similarly, when you want to pass a parameter value to a script you want to run as a command job, you'll specify the values in the command:

PythonCopy

```python
from azure.ai.ml import command

# configure job
job = command(
    code="./src",
    command="python train.py --training_data diabetes.csv",
    environment="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest",
    compute="aml-cluster",
    display_name="train-model",
    experiment_name="train-classification-model"
    )
```

After submitting a command job, you can review the input and output parameters you specified.