A common challenge when developing machine learning models is to prepare for production scenarios. When you write code to process data and train models, you want the code to be scalable, repeatable, and ready for automation.

Though notebooks are ideal for experimentation and development, scripts are a better fit for production workloads. In Azure Machine Learning, you can run a script as a **command job**. When you submit a command job, you can configure various parameters like the input data and the compute environment. Azure Machine Learning also helps you track your work when working with command jobs to make it easier to compare workloads.

**Converting a Notebook to a Script for Production**

When transitioning from experimentation and development in notebooks to production-ready scripts, follow these steps:

1. **Remove Nonessential Code**
2. **Refactor Code into Functions**
3. **Test the Script in a Terminal**

**1. Remove Nonessential Code**

* Remove exploratory code such as print() statements and data inspection methods (e.g., df.describe()) that are used for understanding the data during development but are not needed in production.

**2. Refactor Code into Functions**

* Refactor the code into smaller, manageable functions to improve readability and maintainability. This makes it easier to test parts of your code individually.

**Example: Refactored Notebook Code**

**Original Notebook Code:**

# 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)

**Refactored Script Code:**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

def main(csv\_file):

df = get\_data(csv\_file)

X\_train, X\_test, y\_train, y\_test = split\_data(df)

# You can add more steps here (e.g., model training, evaluation)

def get\_data(path):

df = pd.read\_csv(path)

return df

def split\_data(df):

X = df[['Pregnancies','PlasmaGlucose','DiastolicBloodPressure','TricepsThickness',

'SerumInsulin','BMI','DiabetesPedigree','Age']].values

y = 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

if \_\_name\_\_ == "\_\_main\_\_":

main('diabetes.csv')

**3. Test Your Script**

* **Run the Script in a Terminal**: Before deploying the script in a production environment, test it by running it in a terminal to ensure it works as expected.
* **Using Azure Machine Learning Workspace**: You can run scripts directly from the terminal of a compute instance in the Azure Machine Learning workspace.

**Running the Script in Terminal:**

1. **Open Terminal**: Open the terminal of the compute instance.
2. **Run the Script**:

python train.py

1. **Check Outputs and Errors**: Verify the outputs of any remaining print statements and check for errors in the terminal.

**Running a Script as a Command Job in Azure Machine Learning**

To run a script as a command job in Azure Machine Learning, follow these steps to configure and submit the job using the Python SDK (v2).

**1. Configure the Command Job**

To configure a command job, use the command function and specify the necessary parameters:

* **code**: The directory containing the script.
* **command**: The command to execute the script.
* **environment**: The environment with the required packages.
* **compute**: The compute resource to use.
* **display\_name**: The name of the job.
* **experiment\_name**: The name of the experiment to which the job belongs.

**Example Code:**

from azure.ai.ml import command

# Configure the job

job = command(

code="./src", # Path to the folder containing the script

command="python train.py", # Command to run the script

environment="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest", # Preconfigured environment

compute="aml-cluster", # Compute cluster to run the job

display\_name="train-model", # Name of the job

experiment\_name="train-classification-model" # Name of the experiment

)

**2. Submit the Command Job**

Once the job is configured, submit it to run the script on the specified compute resource.

**Example Code:**

# Submit the job

returned\_job = ml\_client.create\_or\_update(job)

**3. Monitor and Review the Job**

You can monitor the progress and review the details of the job in the Azure Machine Learning studio:

* **Experiments**: Jobs with the same experiment name are grouped together.
* **Individual Job**: Locate the job using the specified display name.

In the Azure Machine Learning studio, you can view:

* **Inputs and Outputs**: Track the inputs and outputs of the command job.
* **Command**: Review the command that was executed.
* **Compute**: Check the compute resource used.
* **Environment**: Verify the environment used for running the script.

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.

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

**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

**MLflow** is an open-source platform that helps you to track model metrics and artifacts across platforms and is integrated with Azure Machine Learning.

When you use MLflow together with Azure Machine Learning, you can run training scripts locally or in the cloud.

When you train a model with a script, you can include MLflow in the scripts to track any parameters, metrics, and artifacts. When you run the script as a job in Azure Machine Learning, you're able to review all input parameters and outputs for each run.

MLflow is an open-source platform, designed to manage the complete machine learning lifecycle. As it's open source, it can be used when training models on different platforms. Here, we explore how we can integrate MLflow with Azure Machine Learning jobs.

There are two options to track machine learning jobs with MLflow:

* Enable autologging using mlflow.autolog()
* Use logging functions to track custom metrics using mlflow.log\_\*

Before you can use either of these options, you need to set up the environment to use MLflow.

To use MLflow during training job, the mlflow and azureml-mlflow pip packages need to be installed on the compute executing the script. Therefore, you need to include these two packages in the environment. You can create an environment by referring to a YAML file that describes the Conda environment. As part of the Conda environment, you can include these two packages.

**Enable Autologging (Optional but Recommended):**

* MLflow supports autologging for popular machine learning libraries. This automatically logs parameters, metrics, and artifacts without explicit calls in your script.
* Example to enable autologging for scikit-learn:

import mlflow

mlflow.sklearn.autolog()

* Autologging is supported for various libraries like TensorFlow, PyTorch, XGBoost, etc. Use the specific mlflow.\*.autolog() function based on the library you're using.

In your training script, you can decide whatever custom metric you want to log with MLflow.

Depending on the type of value you want to log, use the MLflow command to store the metric with the experiment run:

* mlflow.log\_param(): Log single key-value parameter. Use this function for an input parameter you want to log.
* mlflow.log\_metric(): Log single key-value metric. Value must be a number. Use this function for any output you want to store with the run.
* mlflow.log\_artifact(): Log a file. Use this function for any plot you want to log, save as image file first.

To add MLflow to an existing training script, you can add the following code:

import mlflow

reg\_rate = 0.1

mlflow.log\_param("Regularization rate", reg\_rate)

When your job is completed, you can review the logged parameters, metrics, and artifacts in the Azure Machine Learning studio.

When you review job runs in the Azure Machine Learning studio, you'll explore a job run's metrics, which is part of an experiment.

To view the metrics through an intuitive user interface, you can:

1. Open the Studio by navigating to [https://ml.azure.com](https://ml.azure.com/).
2. Find your experiment run and open it to view its details.
3. In the **Details** tab, all logged parameters are shown under **Params**.
4. Select the **Metrics** tab and select the metric you want to explore.
5. Any plots that are logged as artifacts can be found under **Images**.
6. The model assets that can be used to register and deploy the model are stored in the **models** folder under **Outputs + logs**.

**Search all the experiments**

You can get all the active experiments in the workspace using MLFlow:

experiments = mlflow.search\_experiments(max\_results=2)

for exp in experiments:

print(exp.name)

If you want to retrieve archived experiments too, then include the option ViewType.ALL:

from mlflow.entities import ViewType

experiments = mlflow.search\_experiments(view\_type=ViewType.ALL)

for exp in experiments:

print(exp.name)

To retrieve a specific experiment, you can run:

exp = mlflow.get\_experiment\_by\_name(experiment\_name)

print(exp)

**Retrieve runs**

MLflow allows you to search for runs inside of any experiment. You need either the experiment ID or the experiment name.

For example, when you want to retrieve the metrics of a specific run:

mlflow.search\_runs(exp.experiment\_id)

You can use search\_all\_experiments=True if you want to search across all the experiments in the workspace.

By default, experiments are ordered descending by start\_time, which is the time the experiment was queued in Azure Machine Learning. However, you can change this default by using the parameter order\_by.

For example, if you want to sort by start time and only show the last two results:

mlflow.search\_runs(exp.experiment\_id, order\_by=["start\_time DESC"], max\_results=2)

You can also look for a run with a specific combination in the hyperparameters:

mlflow.search\_runs(

exp.experiment\_id, filter\_string="params.num\_boost\_round='100'", max\_results=2

)

In machine learning, models are trained to predict unknown labels for new data based on correlations between known labels and features found in the training data. Depending on the algorithm used, you may need to specify **hyperparameters** to configure how the model is trained.

For example, the *logistic regression* algorithm uses a *regularization rate* hyperparameter to counteract overfitting; and deep learning techniques for convolutional neural networks (CNNs) use hyperparameters like *learning rate* to control how weights are adjusted during training, and *batch size* to determine how many data items are included in each training batch.

 Data scientists refer to the values determined from the training features as *parameters*, so a different term is required for values that are used to configure training behavior but which are ***not*** derived from the training data - hence the term *hyperparameter*.

The choice of hyperparameter values can significantly affect the resulting model, making it important to select the best possible values for your particular data and predictive performance goals.

**Hyperparameter tuning** is accomplished by training the multiple models, using the same algorithm and training data but different hyperparameter values. The resulting model from each training run is then evaluated to determine the performance metric for which you want to optimize (for example, *accuracy*), and the best-performing model is selected.

To define a search space for hyperparameter tuning, you can specify the possible values or distributions for each hyperparameter using Python objects provided by Azure Machine Learning's sweep capabilities. Here's a summary of how you can define a search space:

**Discrete Hyperparameters**

* **Choice**: Specifies a discrete set of values.
  + Example: Choice(values=[10, 20, 30])
  + Example: Choice(values=range(1, 10))
  + Example: Choice(values=(30, 50, 100))
* **QUniform**: Quantized uniform distribution within a range.
  + Example: QUniform(min\_value, max\_value, q)
* **QLogUniform**: Quantized log-uniform distribution within a range.
  + Example: QLogUniform(min\_value, max\_value, q)
* **QNormal**: Quantized normal distribution.
  + Example: QNormal(mu, sigma, q)
* **QLogNormal**: Quantized log-normal distribution.
  + Example: QLogNormal(mu, sigma, q)

**Continuous Hyperparameters**

* **Uniform**: Uniform distribution within a range.
  + Example: Uniform(min\_value, max\_value)
* **LogUniform**: Log-uniform distribution within a range.
  + Example: LogUniform(min\_value, max\_value)
* **Normal**: Normal (Gaussian) distribution.
  + Example: Normal(mu, sigma)
* **LogNormal**: Log-normal distribution.
  + Example: LogNormal(mu, sigma)

**Example of Defining a Search Space**

Here's an example of defining a search space where batch\_size can take discrete values [16, 32, 64] and learning\_rate follows a normal distribution with mean 10 and standard deviation 3:

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),

)

In this setup:

* batch\_size is a discrete choice between 16, 32, and 64.
* learning\_rate is sampled from a normal distribution with mean 10 and standard deviation 3.

To configure early termination policies for hyperparameter tuning in Azure Machine Learning, you have several options to control when a sweep job should stop based on the performance of the models being trained. Here’s a summary of the available early termination policies and their configurations:

**Bandit Policy**

* **Description**: Stops a trial if the target performance metric underperforms the best trial so far by a specified margin (slack amount or slack factor).
* **Parameters**:
  + slack\_amount: Absolute amount by which a trial's metric can underperform the best trial.
  + slack\_factor: Ratio by which a trial's metric can underperform the best trial.
  + evaluation\_interval: Interval (in terms of trials) at which to evaluate the policy.
  + delay\_evaluation: Number of trials to wait before starting to evaluate the policy.
* **Example Configuration**:

from azure.ai.ml.sweep import BanditPolicy

sweep\_job.early\_termination = BanditPolicy(

slack\_amount=0.2,

delay\_evaluation=5,

evaluation\_interval=1

)

**Median Stopping Policy**

* **Description**: Stops a trial if the trial's performance metric is worse than the median of the running averages of all trials.
* **Parameters**:
  + evaluation\_interval: Interval (in terms of trials) at which to evaluate the policy.
  + delay\_evaluation: Number of trials to wait before starting to evaluate the policy.
* **Example Configuration**:

from azure.ai.ml.sweep import MedianStoppingPolicy

sweep\_job.early\_termination = MedianStoppingPolicy(

delay\_evaluation=5,

evaluation\_interval=1

)

**Truncation Selection Policy**

* **Description**: Cancels the lowest performing X% of trials at each evaluation interval, where X is specified by truncation\_percentage.
* **Parameters**:
  + truncation\_percentage: Percentage of trials to cancel at each evaluation interval.
  + evaluation\_interval: Interval (in terms of trials) at which to evaluate the policy.
  + delay\_evaluation: Number of trials to wait before starting to evaluate the policy.
* **Example Configuration**:

from azure.ai.ml.sweep import TruncationSelectionPolicy

sweep\_job.early\_termination = TruncationSelectionPolicy(

evaluation\_interval=1,

truncation\_percentage=20,

delay\_evaluation=4

)

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.

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

In Azure Machine Learning, you can experiment in notebooks and train (and retrain) machine learning models by running scripts as jobs.

In an enterprise data science process, you'll want to separate the overall process into individual tasks. You can group tasks together as **pipelines**. Pipelines are key to implementing an effective **Machine Learning Operations** (**MLOps**) solution in Azure.

You'll learn how to create **components** of individual tasks, making it easier to reuse and share code. You'll then combine components into an Azure Machine Learning pipeline, which you'll run as a **pipeline job**.

**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.

**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.

 **Pipeline Definition**: A pipeline in Azure ML is defined using the @pipeline() function, where each task or operation is encapsulated within a component. Components can execute sequentially or in parallel, enabling complex workflows.

 **Component Execution**: Components within the pipeline can be executed on specific compute targets, allowing for flexibility in combining different processing types to achieve overall machine learning goals.

 **Pipeline Job Execution**: The entire pipeline is executed as a pipeline job. Each component within the pipeline is treated as a child job, with dependencies managed automatically by Azure ML.

 **YAML Configuration**: Azure ML pipelines are configured using a YAML file. This file includes the pipeline job's name, inputs, outputs, and any additional settings required for execution.

 **Example Pipeline Function**:

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,

}

* This example defines a pipeline function named pipeline\_function\_name that takes pipeline\_job\_input as input.
* It sequentially executes loaded\_component\_prep to prepare data and then loaded\_component\_train to train a model using the prepared data.
* Outputs are defined as pipeline\_job\_transformed\_data and pipeline\_job\_trained\_model, capturing the outputs from each component.

 **Pipeline Outputs**: Outputs from the pipeline are specified by returning variables from the pipeline function. Each output corresponds to the output of a specific component within the pipeline.

 **Visualization**: After defining the pipeline using @pipeline(), you can visualize and inspect the YAML configuration by printing the pipeline\_job object. This YAML file captures the entire structure and configuration of the pipeline, including component dependencies and settings.

 **Flexibility**: Azure ML pipelines allow for flexibility in integrating various machine learning operations, from data preparation to model training and deployment, into cohesive workflows managed and orchestrated within the Azure ML ecosystem.

**Configure a Pipeline Job**

1. **Modify Pipeline Configuration**:
   * After defining your pipeline using the @pipeline() function or a YAML file, you can modify its settings, such as output modes, default compute targets, or datastores.

# Change output modes

pipeline\_job.outputs.pipeline\_job\_transformed\_data.mode = "upload"

pipeline\_job.outputs.pipeline\_job\_trained\_model.mode = "upload"

# Set pipeline level compute

pipeline\_job.settings.default\_compute = "aml-cluster"

# Set pipeline level datastore

pipeline\_job.settings.default\_datastore = "workspaceblobstore"

1. **Review Pipeline Configuration**:
   * Print the pipeline job object to review its YAML configuration:

print(pipeline\_job)

**Submit a Pipeline Job**

1. **Submit the Pipeline Job**:
   * Use the Azure Machine Learning client (ml\_client) to submit the pipeline job to your workspace:

pipeline\_job = ml\_client.jobs.create\_or\_update(

pipeline\_job, experiment\_name="pipeline\_job"

)

1. **Monitor Execution**:
   * Once submitted, Azure Machine Learning creates a new job in your workspace. This job includes child jobs that represent the execution of individual components within the pipeline.
   * Use the Azure Machine Learning studio to monitor and explore the pipeline's progress, parameters, outputs, and child jobs.
2. **Troubleshoot Issues**:
   * If the pipeline or any component fails:
     + Check the outputs and logs of the pipeline job and its child jobs.
     + Logs will provide detailed information about configuration issues or runtime errors encountered during execution.

**Schedule a Pipeline Job**

1. **Automate Pipeline Execution** (Optional):
   * You can schedule a pipeline job to automate its execution using a time-based trigger.
   * Define a recurrence trigger using the RecurrenceTrigger class:

from azure.ai.ml.entities import RecurrenceTrigger

recurrence\_trigger = RecurrenceTrigger(

frequency="minute",

interval=1, # Runs every minute

)

1. **Create a Job Schedule**:
   * Associate the recurrence trigger with your pipeline job using the JobSchedule class:

from azure.ai.ml.entities import JobSchedule

job\_schedule = JobSchedule(

name="run\_every\_minute",

trigger=recurrence\_trigger,

create\_job=pipeline\_job,

)

job\_schedule = ml\_client.schedules.begin\_create\_or\_update(

schedule=job\_schedule

).result()

1. **Manage Scheduled Jobs**:
   * View scheduled jobs prefixed with the schedule name in the Azure Machine Learning studio.
   * To delete a schedule, first disable it and then delete it:

ml\_client.schedules.begin\_disable(name="run\_every\_minute").result()

ml\_client.schedules.begin\_delete(name="run\_every\_minute").result()