**Implementation of MLOps using Azure ML**

*Submitting an Experiment using Mlflow using local compute*  
  
One way to implement MLOps is by submitting an experiment job to Azure ML. This can be achieved in two ways: on Azure ML compute, or on local compute by connecting the local workspace with Azure ML. The process is almost similar in both cases.  
  
i) The following code connects to an Azure Machine Learning workspace using the Azure ML SDK for Python. It uses the subscription ID, resource group, and workspace name to authenticate and create a connection to the specified workspace. Once the connection is established, the workspace object is stored in the variable named "ws". This workspace object can be used to manage various resources within the workspace, such as experiments, datasets, compute targets, and models.

#connecting workspace

from azureml.core import Workspace

ws = Workspace(subscription\_id='<>', resource\_group='<>', workspace\_name='<>')

ii) The following code creates a new experiment named 'mlops\_spam' and sets it as the active experiment in a single line of code using the following syntax:

mlflow.create\_experiment('mlops\_spam')

mlflow.set\_experiment('mlops\_spam')

iii) Register Dataset: You can register a dataset using this syntax:

dataset = dataset.register(workspace=ws, name=dataset\_name)

Registering a dataset in Azure Machine Learning is an important step in the machine learning workflow for several reasons:

1. Reproducibility: By registering a dataset, you create a reference to the data that is independent of the data's physical location. This makes it easier to reproduce experiments and ensures that your models are trained on the same data every time.
2. Data versioning: When you register a dataset, you can track changes to the data over time. This allows you to version your data and keep track of different versions of the dataset.
3. Sharing and collaboration: Registered datasets can be shared and used by other members of your team. This makes it easier to collaborate on projects and reuse datasets across different experiments and models.
4. Efficient data access: By registering a dataset in Azure Machine Learning, you can take advantage of the platform's data management features, such as caching and data streaming. This makes it faster and more efficient to access large datasets during training and inference.

iv) Load registered dataset: You can load a registered dataset using this syntax:

#loading binary dataset from azure

from azureml.core.dataset import Dataset

dataset = Dataset.get\_by\_name(ws, 'cleaned\_dataset')

You can convert the loaded dataset into different forms such as JSON, pandas dataframe, etc.

v) MLflow and MLOps: Training a Spam Classifier Model with LightGBM

import lightgbm as lgbm

from sklearn.metrics import accuracy\_score, recall\_score, precision\_score, f1\_score

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

import numpy as np

from azureml.core.model import Model

def log\_metrics(clf, X\_test, y\_test, run):

    preds = np.round(clf.predict(X\_test))

    run.log("accuracy (test)", accuracy\_score(y\_test, preds))

    run.log("precision (test)", precision\_score(y\_test, preds))

    run.log("recall (test)", recall\_score(y\_test, preds))

    run.log("f1 (test)", f1\_score(y\_test, preds))

def train\_model():

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(tfidf\_features, cleaned\_binary\_data['topic'], test\_size=0.15, random\_state=42, stratify=cleaned\_binary\_data['topic'])

    train\_data = lgbm.Dataset(data=X\_train, label=y\_train)

    test\_data = lgbm.Dataset(data=X\_test, label=y\_test)

    full\_train = lgbm.Dataset(data=tfidf\_features, label=cleaned\_binary\_data['topic'])

    mlflow.lightgbm.autolog()

    with mlflow.start\_run() as my\_run:

        lgbm\_params = {

            'colsample\_bytree': 0.4955555555555555,

            'learning\_rate': 0.09473736842105263,

            'max\_bin': 140,

            'max\_depth': 6,

            'min\_split\_gain': 0.10526315789473684,

            'n\_estimators': 25,

            'num\_leaves': 164,

            'reg\_alpha': 0.3157894736842105,

            'reg\_lambda': 0.3157894736842105

        }

        evaluation\_results = {}

        clf = lgbm.train(

            train\_set=train\_data,

            params=lgbm\_params,

            valid\_sets=[train\_data, test\_data],

            valid\_names=['train', 'val'],

            evals\_result=evaluation\_results,

            num\_boost\_round=500

        )

        preds = np.round(clf.predict(X\_test))

        acc = accuracy\_score(y\_test, preds)

        pr = precision\_score(y\_test, preds)

        rec = recall\_score(y\_test, preds)

        f1 = f1\_score(y\_test, preds)

        mlflow.log\_metric('accuracy', acc)

        mlflow.log\_metric('precision', pr)

        mlflow.log\_metric('recall', rec)

        mlflow.log\_metric('f1 score', f1)

        model\_spam = lgbm.train(

            train\_set=full\_train,

            params=lgbm\_params,

            num\_boost\_round=500

        )

    return my\_run

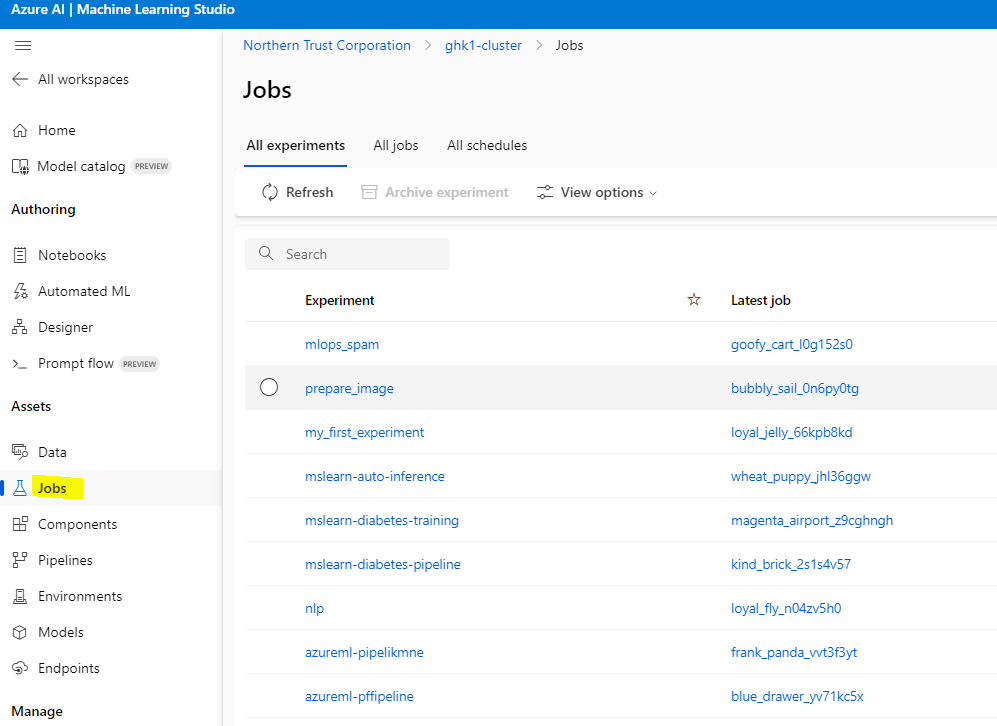
This code trains a spam classifier model using LightGBM and logs the evaluation metrics using MLflow.

The log\_metrics function takes in the trained model (clf), test data (X\_test and y\_test), and MLflow run object (run) as inputs. It uses the trained model to make predictions on the test data and calculates evaluation metrics such as accuracy, precision, recall, and F1 score. These metrics are then logged using the MLflow run.log function.

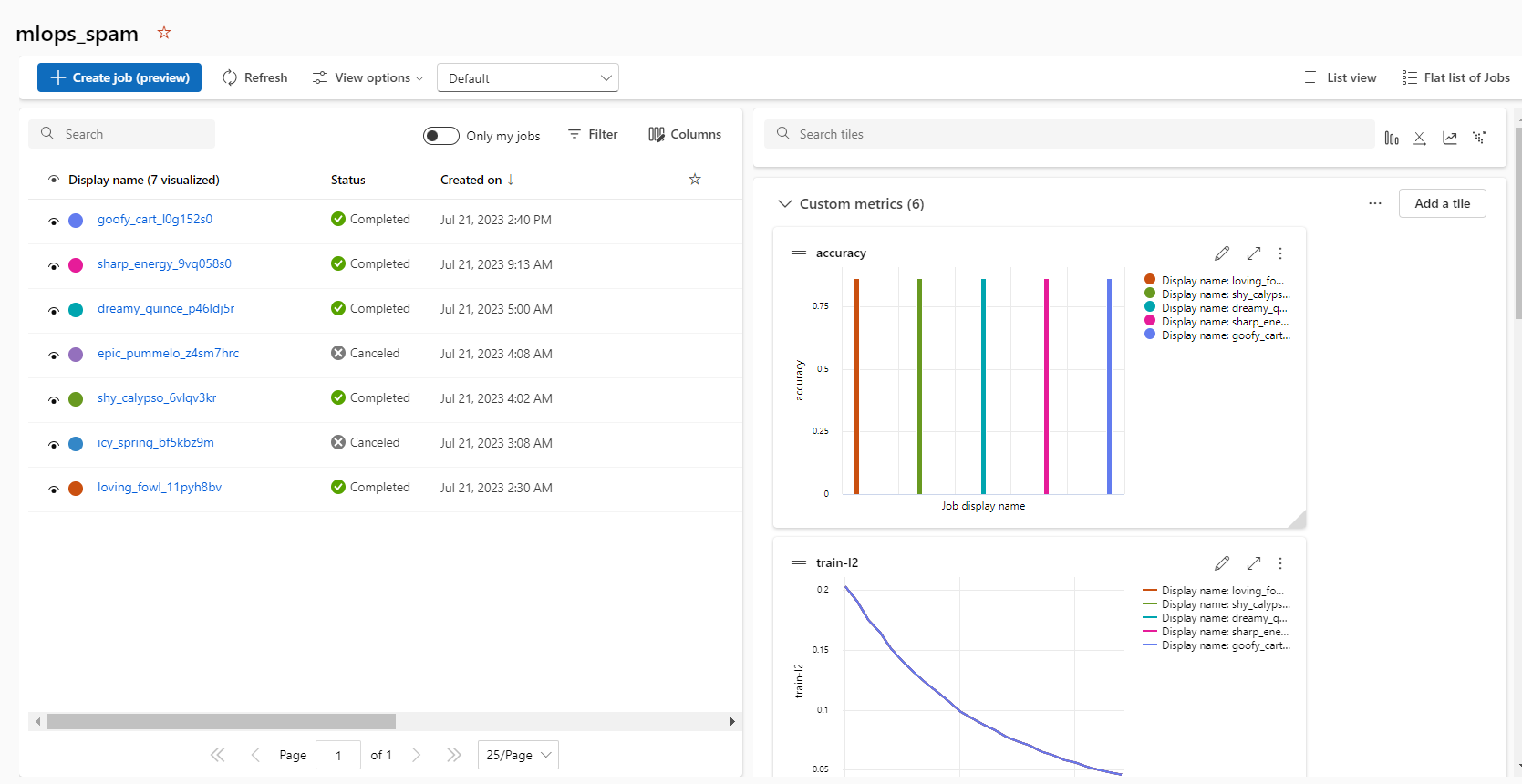
MLflow's LightGBM autologging feature is enabled using mlflow.lightgbm.autolog(). This automatically logs the LightGBM parameters, metrics, and model artifacts.  
  
Within an MLflow run context (with mlflow.start\_run() as my\_run), the LightGBM model is trained using train\_set, params, and valid\_sets. Evaluation results are stored in evaluation\_results. The model is then used to make predictions on the test data, and the evaluation metrics are calculated and logged using mlflow.log\_metric.  
  
In terms of MLOps, this code demonstrates the use of MLflow to track and log the metrics, parameters, and models during the training process. This provides reproducibility and versioning of models, allowing easy comparison of different runs and hyperparameters. The logged metrics can be used to monitor model performance and make informed decisions. The trained model can be deployed and served as an endpoint for real-time inference or used in batch scoring pipelines, making it a valuable tool for productionizing machine learning models.

vi) Outputs, Artifacts, Logs

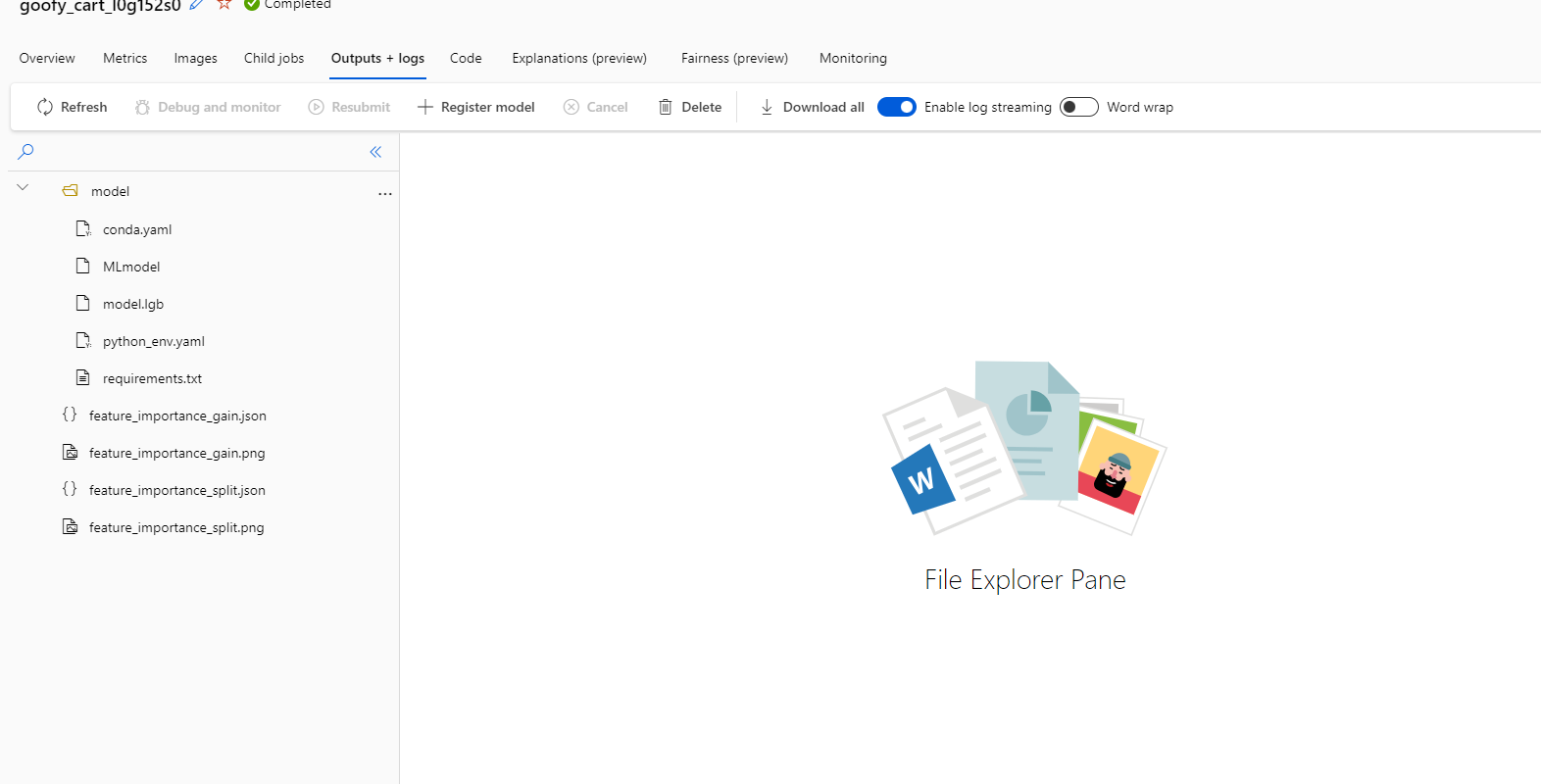
To see artifacts, such as the trained model, the requirements.txt file, the conda.yaml file, as well as metrics and hyperparameters, you can navigate to Azure ML and select 'Jobs' from the menu below.



Under Jobs, in the ‘All Experiments’ tab, you can view all the experiments that you have created, including our 'mlops\_spam' experiment. If you click on 'mlops\_spam', you can see all the experiments run within this experiment. On the right-hand side, you can create custom graphs to compare each experiment in terms of hyperparameters, metrics, and other relevant information. This allows for easy visualization and comparison of different experiment runs, making it easier to identify the best performing models and fine-tune hyperparameters.



We can get detailed overview of each of the experiments by going in there. We can look the metrics, Image produced, and can download the artifacts of our model. We can also register the model as well from here, but we would implement mlflow to register the model.



vii) Model Registry and Creating Real time End Point

We can use this syntax to register the mlflow model:

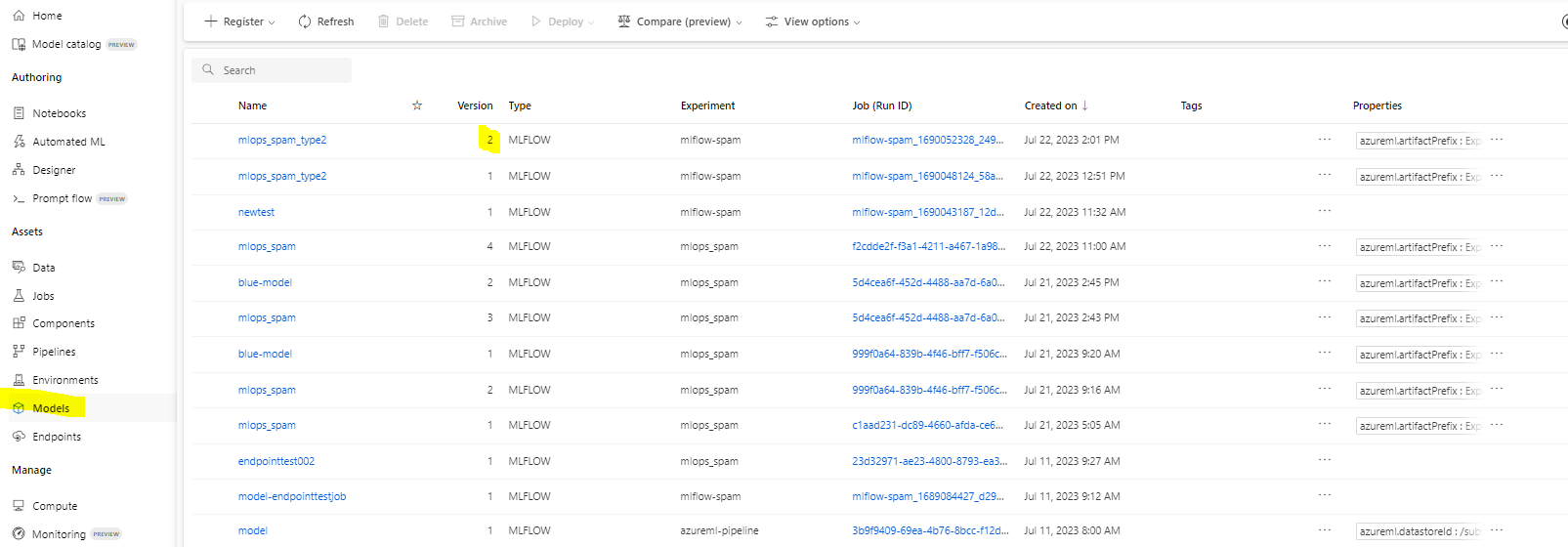
model\_uri = f"runs:/{run\_id}/model"

registered\_model = mlflow.register\_model(model\_uri=model\_uri,name='mlops\_spam')

Registering models has these benefits:

* Versioning and tracking of models
* Reproducibility of models
* Governance and compliance enforcement
* Collaboration and teamwork among data scientists
* Seamless integration with deployment pipelines and monitoring tools

You can check model versions using sdk as well as using ml studio UI.



ix) Model Deployment and Endpoint

To deploy the trained model and create a real-time endpoint on Azure ML, you have two options: using the Azure ML UI or using the Azure ML Python SDK.

Using the Azure ML UI, you can navigate to the 'Endpoints' section of the Azure ML workspace and select 'Create Endpoint'. From there, you can select the model you want to deploy and configure the endpoint settings, such as the compute target and deployment environment.

To deploy the trained model and create a real-time endpoint on Azure ML, you can use the Azure ML Python SDK and the deployment\_client.create\_deployment() function.  
  
The provided code creates a new deployment named 'blue\_deployment' for the trained model at the location specified by model\_uri. The deployment is associated with an existing endpoint called 'endpoint\_name', and the deployment configuration is specified in a separate file located at deployment\_config\_path.  
  
Once the deployment is created, you can use the endpoint to make real-time predictions on new data or in batch scoring pipelines. The Azure ML platform provides monitoring and logging capabilities to track the performance of the endpoint and troubleshoot any issues that may arise.

blue\_deployment = deployment\_client.create\_deployment(

    name=blue\_deployment\_name,

    endpoint=endpoint\_name,

    model\_uri=model\_uri,

    config={"deploy-config-file": deployment\_config\_path},

)

The whole code is available here: [MLOps-using-Azure-ML/mlops\_ using Mlflow using local compute.ipynb at main · AA706\_ntrs/MLOps-using-Azure-ML (github.com)](https://github.com/AA706_ntrs/MLOps-using-Azure-ML/blob/main/mlops_%20using%20Mlflow%20using%20local%20compute.ipynb)

*Submitting an Experiment using Mlflow using Azure ML compute*

We can also leverage the Azure ML compute to train our models. It has some benefits.

* Provides a scalable and elastic infrastructure for training models
* Supports a wide range of machine learning frameworks
* Can distribute training across multiple nodes to accelerate the process
* Provides a consistent and reproducible environment for training
* Ensures that results are reproducible and consistent
* Reduces the need for managing infrastructure, allowing you to focus on model development
* Improves the efficiency and quality of the training process.

There are few things to be done differently from the above way:

* 1. Create or retrieve Azure ML compute Cluster

from azureml.core.compute import ComputeTarget, AmlCompute

from azureml.core.compute\_target import ComputeTargetException

def get\_aml\_cluster(ws, cluster\_name, vm\_size='STANDARD\_D2\_V2', max\_nodes=4):

    try:

        cluster = ComputeTarget(workspace=ws, name=cluster\_name)

    except ComputeTargetException:

        config = AmlCompute.provisioning\_configuration(vm\_size=vm\_size, max\_nodes=max\_nodes)

        cluster = ComputeTarget.create(ws, cluster\_name, config)

    return cluster

This code defines a function called get\_aml\_cluster that creates or retrieves an Azure Machine Learning compute cluster. The function takes three parameters:

* ws: The Azure Machine Learning workspace object.
* cluster\_name: The name of the compute cluster to retrieve or create.
* vm\_size: The size of the virtual machines to use in the cluster (default is STANDARD\_D2\_V2).
* max\_nodes: The maximum number of nodes to use in the cluster (default is 4).  
    
  The function first tries to retrieve an existing compute cluster with the given cluster\_name using the ComputeTarget class. If the cluster does not exist, it creates a new cluster using the AmlCompute class and the provided vm\_size and max\_nodes values.  
    
  Finally, the function returns the compute cluster object, which can be used to submit jobs for training machine learning models on the cluster.  
    
  Overall, this code is a convenient way to create or retrieve an Azure Machine Learning compute cluster for use in training machine learning models.

ii) Creating/ Retrieving Environment container in which Training script will run

from azureml.core.runconfig import RunConfiguration

from azureml.core.conda\_dependencies import CondaDependencies

from azureml.core import Environment

aml\_run\_config = RunConfiguration()

aml\_run\_config.target = aml\_cluster

# Use a curated environment

if use\_curated\_env:

    curated\_env = Environment.get(workspace=ws, name="AzureML-Tutorial").clone("mlops\_env")

    curated\_env.python.conda\_dependencies.add\_conda\_package("sentence-transformers")

    curated\_env.register(workspace=ws)

    aml\_run\_config.environment = curated\_env

else:

    aml\_run\_config.environment.python.user\_managed\_dependencies = True

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

                                            pip\_packages=['azureml-sdk', 'nltk', 'string', 're'],

                                            channels=['conda-forge'])

    aml\_run\_config.environment.python.conda\_dependencies = dependencies

This code sets up a RunConfiguration object for a machine learning training script. It specifies the target compute, dependencies, and environment.

* The target attribute of the RunConfiguration object is set to an Azure Machine Learning compute cluster object called aml\_cluster.
* The script can use a curated environment or user-managed dependencies.
* If USE\_CURATED\_ENV is True, a curated environment called "AzureML-Tutorial" is cloned and modified to add the "sentence-transformers" package. The modified environment is registered and set as the environment for the RunConfiguration object.
* If USE\_CURATED\_ENV is False, the user\_managed\_dependencies attribute of the RunConfiguration object is set to True. The required packages are specified using CondaDependencies.create().
* The conda\_packages parameter specifies the packages to install via conda, while pip\_packages specifies the packages to install via pip. The channels parameter specifies additional conda channels to use.
* The conda\_dependencies attribute of the environment in the RunConfiguration object is set to the CondaDependencies object created.

This code creates a RunConfiguration object that can execute a machine learning training script on an Azure Machine Learning compute cluster, with the necessary dependencies and environment specified.

iii) Using ‘argparse’ library to create customizable training script

We can use argparse lib to create customizable parameters for data and hyperparameter.

This is the training script:

parser = argparse.ArgumentParser()

parser.add\_argument('--data', type=str) #Getting data

parser.add\_argument('--colsample-bytree', type=float)

parser.add\_argument('--learning-rate', type=float, default=0.001)

This is from notebook file that runs the training script:

script\_params = [

    '--data', dataset2.as\_named\_input('cleaned\_dataset'),

    '--colsample-bytree', '0.4955555555555555',

This code demonstrates how to use the argparse library to create customizable parameters for data and hyperparameters in a machine learning training script.

* The argparse module is imported and an ArgumentParser object is created.
* Three arguments are added to the parser using the add\_argument() method. --data is used to specify the data file, --colsample-bytree is used to specify a hyperparameter, and --learning-rate is used to specify a learning rate hyperparameter with a default value of 0.001.
* In the notebook file that runs the training script, a list of script parameters is created. The --data parameter is set to a NamedInput object that represents the cleaned dataset, and the --colsample-bytree parameter is set to a float value.
* When the training script is executed, the values of the script parameters are retrieved using the argparse library.  
    
  This approach makes the training script more customizable, as the values of the data file and hyperparameters can be changed without modifying the script itself.

iv) Submitting training script using ScriptRunConfig

This code from notebook:

from azureml.core import ScriptRunConfig

script = 'model\_spam(mlflow).py'

script\_folder = os.getcwd()

src = ScriptRunConfig(source\_directory=script\_folder,script=script,  run\_config=aml\_run\_config,

  arguments=script\_params)

 ScriptRunConfig object that defines how a machine learning training script will be executed.

* The ScriptRunConfig class is imported from the azureml.core module.
* The name of the training script is assigned to the script variable.
* The script folder is assigned to the script\_folder variable using the os.getcwd() method, which gets the current working directory.
* A ScriptRunConfig object is created using the ScriptRunConfig() constructor.
* The source\_directory parameter specifies the path to the directory that contains the training script.
* The script parameter specifies the name of the training script.
* The run\_config parameter specifies the RunConfiguration object that was created earlier, which specifies the target compute, dependencies, and environment for the script.
* The arguments parameter specifies the script parameters that were created earlier using the argparse library. These parameters can be customized by the user when running the script.
* The ScriptRunConfig object can be used to submit the training script to Azure Machine Learning for execution.

Apart from this, rest of functionalities are pretty similar from the ‘*Submitting an Experiment using Mlflow using local compute’*

You can view this code:

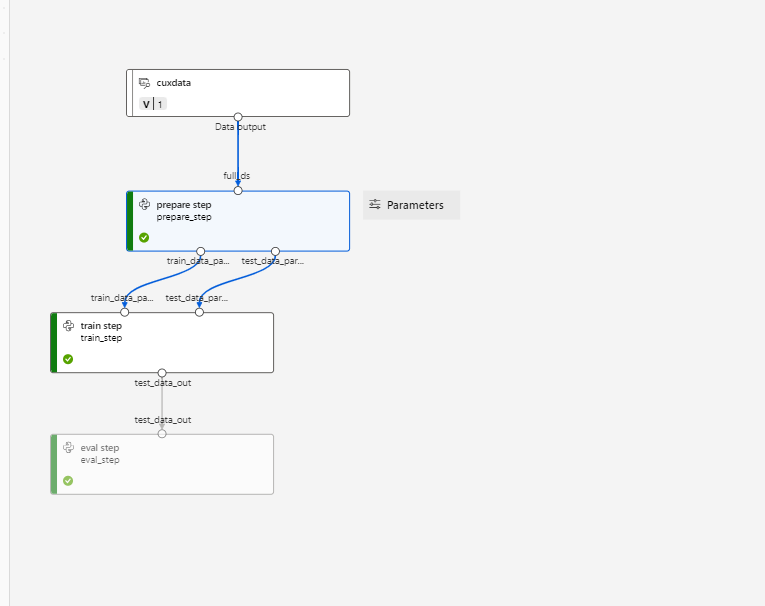
1. Notebook: <https://github.com/AA706_ntrs/MLOps-using-Azure-ML/blob/main/mlops_using%20Mlflow%20using%20Azure%20ML%20compute.ipynb>
2. Training Script: <https://github.com/AA706_ntrs/MLOps-using-Azure-ML/blob/main/model_spam(mlflow).py>

*Creating reusable pipelines*

You can also use reusable pipelines that have all the functionalities listed in 2 above plus they also come up with these benefits:

1. **End-to-end workflow orchestration:** Azure Machine Learning pipelines allow you to define and manage the entire machine learning workflow, from data preparation to model training and deployment. This helps streamline the process and reduces the time and effort required to manage the workflow.
2. **Reproducibility and traceability:** With pipelines, each step in the workflow is tracked and versioned, making it easy to reproduce and trace back to specific versions of the data, code, and models that were used at each stage of the workflow.
3. **Visualization:** Azure Machine Learning pipelines provide a visual representation of the entire machine learning workflow, including data preprocessing, model training, and deployment.

It looks like this:



You can see an code example:

1. Notebook: <https://github.com/AA706_ntrs/MLOps-using-Azure-ML/blob/main/mlops_pipeline.ipynb>
2. Training Script: <https://github.com/AA706_ntrs/MLOps-using-Azure-ML/blob/main/spam_articles.py>