Run scripts as a pipeline job

A pipeline allows you to group multiple steps into one workflow. You can build a pipeline with components. Each component reflects a Python script to run. A component is defined in a YAML file which specifies the script and how to run it.

Before you start

You'll need the latest version of the azureml-ai-ml package to run the code in this notebook. Run the cell below to verify that it is installed.

> **Note: If the azure-ai-ml package is not installed, run `pip install azure-ai-ml` to install it.**

In [ ]:

```
pip show azure-ai-ml
```

Connect to your workspace

With the required SDK packages installed, now you're ready to connect to your workspace.

To connect to a workspace, we need identifier parameters - a subscription ID, resource group name, and workspace name. Since you're working with a compute instance, managed by Azure Machine Learning, you can use the default values to connect to the workspace.

In [ ]:

```
from azure.identity import DefaultAzureCredential, InteractiveBrowserCredential
from azure.ai.ml import MLClient

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

In [ ]:

```
# Get a handle to workspace
ml_client = MLClient.from_config(credential=credential)
```

Create the scripts

You'll build a pipeline with two steps:

1. **Prepare the data: Fix missing data and normalize the data.**
2. **Train the model: Trains a decision tree classification model.**

Run the following cells to create the src folder and the two scripts.

In [ ]:

```
import os

# create a folder for the script files
script_folder = 'src'
os.makedirs(script_folder, exist_ok=True)
```

```
print(script_folder, 'folder created')
```

In [ ]:

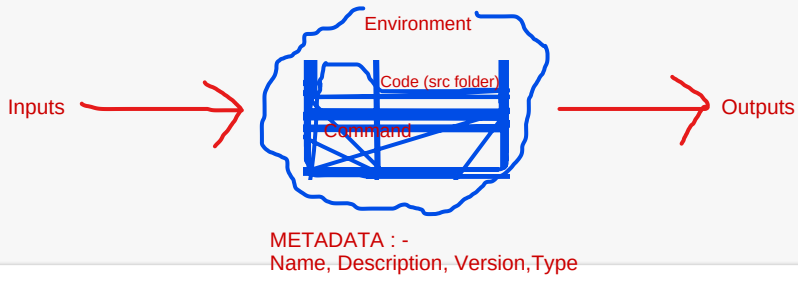
```
%%writefile $script_folder/prep-data.py
# import libraries
import argparse
import pandas as pd
import numpy as np
from pathlib import Path
from sklearn.preprocessing import MinMaxScaler

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

    cleaned_data = clean_data(df)

    normalized_data = normalize_data(cleaned_data)

    output_df = normalized_data.to_csv((Path(args.output_data) / "diabetes.csv"), index
= False)

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

    # Count the rows and print the result
    row_count = (len(df))
    print('Preparing {} rows of data'.format(row_count))

    return df

# function that removes missing values
def clean_data(df):
    df = df.dropna()

    return df

# function that normalizes the data
def normalize_data(df):
    scaler = MinMaxScaler()
    num_cols = ['Pregnancies','PlasmaGlucose','DiastolicBloodPressure','TricepsThickness'
,'SerumInsulin','BMI','DiabetesPedigree']
    df[num_cols] = scaler.fit_transform(df[num_cols])

    return df

def parse_args():
    # 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()

    # return args
    return args

# run script
if __name__ == "__main__":
    # add space in logs
    print("\n\n")
    print("*" * 60)

    # parse args
```

```
    args = parse_args()

    # run main function
    main(args)

    # add space in logs
    print("*" * 60)
    print("\n\n")
```

In [ ]:

```
%%writefile $script_folder/train-model.py
# import libraries
import mlflow
import glob
import argparse
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
import matplotlib.pyplot as plt

def main(args):
    # enable autologging
    mlflow.autolog()

    # read data
    df = get_data(args.training_data)

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

    # train model
    model = train_model(args.reg_rate, X_train, X_test, y_train, y_test)

    eval_model(model, X_test, y_test)

# function that reads the data
def get_data(data_path):

    all_files = glob.glob(data_path + "/*.csv")
    df = pd.concat((pd.read_csv(f) for f in all_files), sort=False)

    return df

# function that splits the data
def split_data(df):
    print("Splitting data...")
    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_sta
te=0)

    return X_train, X_test, y_train, y_test

# function that trains the model
def train_model(reg_rate, X_train, X_test, y_train, y_test):
    mlflow.log_param("Regularization rate", reg_rate)
    print("Training model...")
    model = LogisticRegression(C=1/reg_rate, solver="liblinear").fit(X_train, y_train)

    mlflow.sklearn.save_model(model, args.model_output)

    return model

# function that evaluates the model
def eval_model(model, X_test, y_test):
```

```
    # calculate accuracy
    y_hat = model.predict(X_test)
    acc = np.average(y_hat == y_test)
    print('Accuracy:', acc)

    # calculate AUC
    y_scores = model.predict_proba(X_test)
    auc = roc_auc_score(y_test,y_scores[:,1])
    print('AUC: ' + str(auc))

    # plot ROC curve
    fpr, tpr, thresholds = roc_curve(y_test, y_scores[:,1])
    fig = plt.figure(figsize=(6, 4))
    # Plot the diagonal 50% line
    plt.plot([0, 1], [0, 1], 'k--')
    # Plot the FPR and TPR achieved by our model
    plt.plot(fpr, tpr)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.savefig("ROC-Curve.png")

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

    # add arguments
    parser.add_argument("--training_data", dest='training_data',
                        type=str)
    parser.add_argument("--reg_rate", dest='reg_rate',
                        type=float, default=0.01)
    parser.add_argument("--model_output", dest='model_output',
                        type=str)

    # parse args
    args = parser.parse_args()

    # return args
    return args

# run script
if __name__ == "__main__":
    # add space in logs
    print("\n\n")
    print("*" * 60)

    # parse args
    args = parse_args()

    # run main function
    main(args)

    # add space in logs
    print("*" * 60)
    print("\n\n")
```



Define the components

To define the component you need to specify:

- **Metadata:** ***name***, ***display name***, ***version***, ***description***, ***type*** **etc. The metadata helps to describe and manage the component.**
- **Interface:** ***inputs*** **and** ***outputs*****. For example, a model training component will take training data and the regularization rate as input, and generate a trained model file as output.**
- **Command, code & environment: the** *command*, *code* **and** *environment* **to run the component. Command is the shell command to execute the component. Code usually refers to a source code directory. Environment could be an AzureML environment (curated or custom created), docker image or conda environment.**

Run the following cells to create a YAML for each component you want to run as a pipeline step.

In [ ]:

```
%%writefile prep-data.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_folder
code: ./src
environment: azureml:AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest
command: >-
  python prep-data.py
  --input_data ${{inputs.input_data}}
  --output_data ${{outputs.output_data}}
```

In [ ]:

```
%%writefile train-model.yml
$schema: https://azuremlschemas.azureedge.net/latest/commandComponent.schema.json
name: train_model
display_name: Train a decision tree classifier model
version: 1
type: command
inputs:
  training_data:
    type: uri_folder
  reg_rate:
    type: number
    default: 0.01
outputs:
  model_output:
    type: mlflow_model
code: ./src
environment: azureml:AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest
command: >-
  python train-model.py
  --training_data ${{inputs.training_data}}
  --reg_rate ${{inputs.reg_rate}}
  --model_output ${{outputs.model_output}}
```

Load the components

Now that you have defined each component, you can load the components by referring to the YAML files.

In [ ]:

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

prep_data = load_component(source=parent_dir + "./prep-data.yml")
train_decision_tree = load_component(source=parent_dir + "./train-model.yml")
```

Build the pipeline

After creating and loading the components, you can build the pipeline. You'll compose the two components into a pipeline. First, you'll want the `prep_data` component to run. The output of the first component should be the input of the second component `train_decision_tree`, which will train the model.

The `diabetes_classification` function represents the complete pipeline. The function expects one input variable: `pipeline_job_input`. A data asset was created during setup. You'll use the registered data asset as

the pipeline input.

In [ ]:

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

@pipeline()
def diabetes_classification(pipeline_job_input):
    clean_data = prep_data(input_data=pipeline_job_input)
    train_model = train_decision_tree(training_data=clean_data.outputs.output_data)

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

pipeline_job = diabetes_classification(Input(type=AssetTypes.URI_FILE, path="azureml:diab
etes-data:1"))
```

You can retrieve the configuration of the pipeline job by printing the `pipeline_job` object:

In [ ]:

```
print(pipeline_job)
```

You can change any parameter of the pipeline job configuration by referring to the parameter and specifying the new value:

In [ ]:

```
# change the output mode
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"

# print the pipeline job again to review the changes
print(pipeline_job)
```

Submit the pipeline job

Finally, when you've built the pipeline and configured the pipeline job to run as required, you can submit the pipeline job:

In [ ]:

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