D602 – Deployment

Performance Assessment #2 – Data Production Pipeline

Bernard Connelly

Master of Science, Data Analytics, Western Governors University

Dr. Sherin Aly

February 09, 2025

D602 – Task 2: Data Production Pipeline

The below details outline the steps taken in the mock scenario of completing a peer's code to submit a flight delay analysis for the Miami Airport from August 2024. This project highlights the various steps of creating and implementing an automated MLFlow pipeline to generate the analysis and track the changes via version control so the code can be easily reproducible for other peers.

Part 1: Creating & Modifying the Code

The code was completed in Jupyter Lab, saved in pieces, and then converted to a .py file for implementation in the pipeline. Finally, all portions, steps, data files, and outputs were saved to GitLab for version control.

B. Import, Format, & Apply DVC to Data

After navigating to the Bureau of Transportation Statistics website and downloading a dataset, the subsequent steps to import the data and perform initial formatting were straightforward. First, the data was imported:

```
# B: Writing a script to import and format the data file
# Importing the Raw Data File
file_path = "C:/Users/bconn/OneDrive/Documents/WGUCoursework/Data/Miami_Flight_Data.csv"
data = pd.read_csv(file_path)
df = data.copy()
```

```
# Initial profiling of data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 91857 entries, 0 to 91856
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   YEAR               91857 non-null  int64
 1   MONTH              91857 non-null  int64
 2   DAY_OF_MONTH       91857 non-null  int64
 3   DAY_OF_WEEK        91857 non-null  int64
 4   ORIGIN_AIRPORT_ID  91857 non-null  int64
 5   DEST_AIRPORT_ID    91857 non-null  int64
 6   CRS_DEP_TIME       91857 non-null  int64
 7   DEP_TIME           88688 non-null  float64
 8   DEP_DELAY          88686 non-null  float64
 9   CRS_ARR_TIME       91857 non-null  int64
 10  ARR_TIME           88571 non-null  float64
 11  ARR_DELAY          88165 non-null  float64
 12  ARR_DELAY_NEW      88165 non-null  float64
dtypes: float64(5), int64(8)
memory usage: 9.1 MB
```

To simplify the coding process later, the easiest route was to modify the exported data to match what is expected in the Machine Learning model created by the peer who left the company. To do so, unnecessary columns were trimmed from the dataset, and the remaining columns were renamed to match the naming convention. This convention was described as follows:

```
| YEAR | MONTH | DAY | DAY_OF_WEEK | ORG_AIRPORT | DEST_AIRPORT | SCHEDULED_DEPARTURE | DEPARTURE_TIME | DEPARTURE_DELAY | SCHEDULED_ARRIVAL | ARRIVAL_TIME | ARRIVAL_DELAY |
|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|:--------:|
| integer | integer | integer | integer | string | string | integer | integer | integer | integer | integer | integer |
```

The Python code to complete these steps was straightforward. First, the data frame was trimmed:

```
# Trimming down to required columns
df_trimmed = df[['YEAR', 'MONTH', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'ORIGIN_AIRPORT_ID', 'DEST_AIRPORT_ID', 'CRS_DEP_TIME',
                 'DEP_TIME', 'DEP_DELAY', 'CRS_ARR_TIME', 'ARR_TIME', 'ARR_DELAY']]
print(df_trimmed.info())
```

Then, the columns were renamed:

```
# Renaming the columns to the required format for MLFlow
renamed_headers = ['YEAR', 'MONTH', 'DAY', 'DAY_OF_WEEK', 'ORG_AIRPORT', 'DEST_AIRPORT', 'SCHEDULED_DEPARTURE', 'DEPARTURE_TIME',
                   'DEPARTURE_DELAY', 'SCHEDULED_ARRIVAL', 'ARRIVAL_TIME', 'ARRIVAL_DELAY']
df_trimmed.columns = renamed_headers
print(df_trimmed.info())
```

The datatypes needed to be updated as well to match what was required in the data training file:

```python
# Changing the datatypes to the correct types based upon poly_regressor instructions
df_filtered = df_filtered.astype({"YEAR": "int64",
            "MONTH": "int64",
            "DAY": "int64",
            "DAY_OF_WEEK": "int64",
            "ORG_AIRPORT": "string",
            "DEST_AIRPORT": "string",
            "SCHEDULED_DEPARTURE": "int64",
            "DEPARTURE_TIME": "int64",
            "DEPARTURE_DELAY": "int64",
            "SCHEDULED_ARRIVAL": "int64",
            "ARRIVAL_TIME": "int64",
            "ARRIVAL_DELAY": "int64"})
print(df_filtered.info())
```

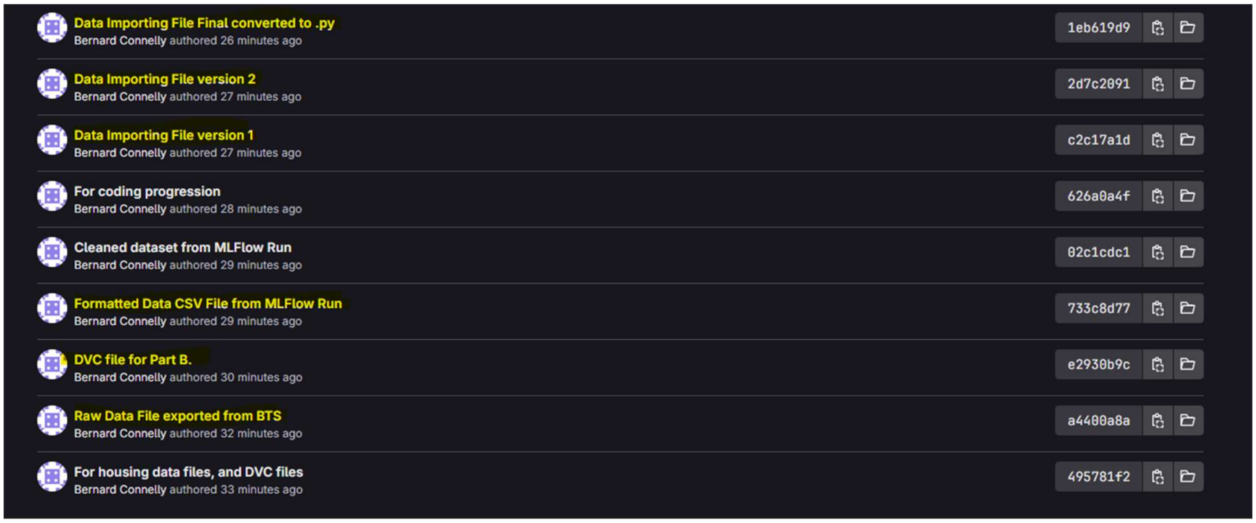
The final step in this file was to set up DVC to track updates or changes to the data. Initial attempts ended up throwing errors since re-initializing the DVC in an environment can cause issues, so I created a loop to check if the environment needed to be set up and then created the DVC data file.

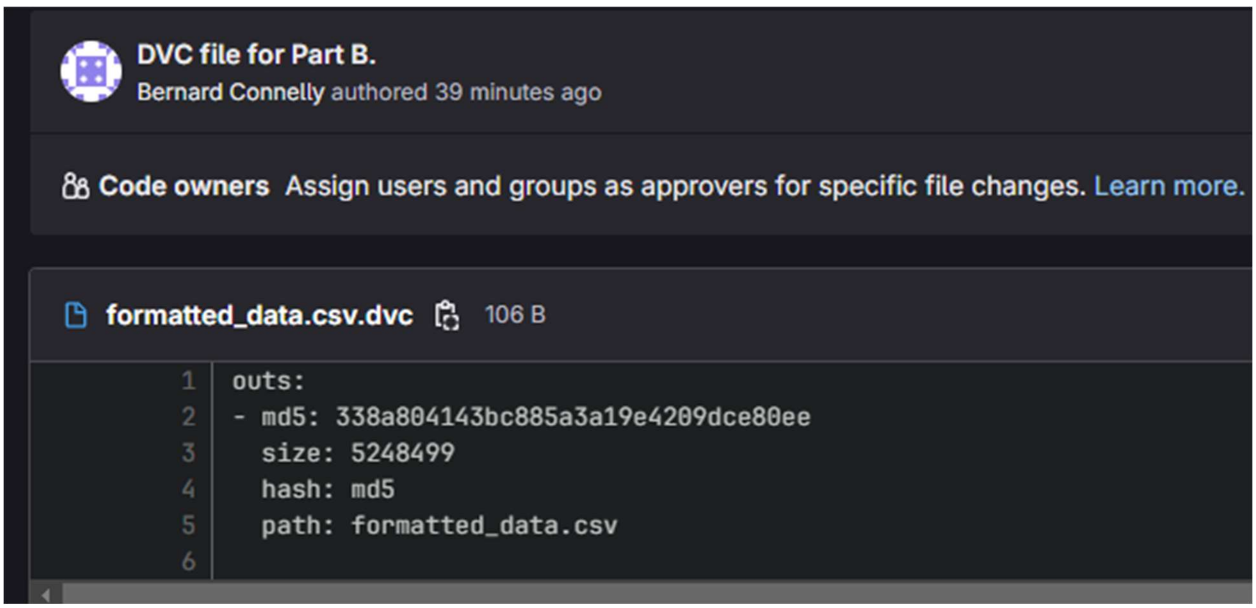
```python
# Setting up DVC

# Creating a loop to check if dvc is already initialized
if not os.path.exists(".dvc"):
    os.system("dvc init --no-scm")
else:
    print ("DVC already initialized")
```

```
DVC already initialized
```

```python
# Creating DVC data file
os.system("dvc add formatted_data.csv")
```

Additionally, the steps for the above files were uploaded to GitLab for version control, including several steps for importing and formatting the DVC file itself.





This file was exported as imported_data.csv to be used in the next step of the pipeline.

C. Filter & Clean Data

Data filtering included narrowing the dataset down to the Miami Airport, which was identified with airport code 13303 per the BTS website.

```
# Confirming filtering worked
df_filtered.info()
print('\n') #adding a space
print(df_filtered['ORG_AIRPORT'].value_counts())
## Confirmed only 13303 (Miami Airport Code) records remain
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8081 entries, 2030 to 91139
Data columns (total 12 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   YEAR                8081 non-null   int64
 1   MONTH               8081 non-null   int64
 2   DAY                 8081 non-null   int64
 3   DAY_OF_WEEK         8081 non-null   int64
 4   ORG_AIRPORT         8081 non-null   string
 5   DEST_AIRPORT        8081 non-null   string
 6   SCHEDULED_DEPARTURE 8081 non-null   int64
 7   DEPARTURE_TIME      8081 non-null   int64
 8   DEPARTURE_DELAY     8081 non-null   int64
 9   SCHEDULED_ARRIVAL   8081 non-null   int64
 10  ARRIVAL_TIME        8081 non-null   int64
 11  ARRIVAL_DELAY       8081 non-null   int64
dtypes: int64(10), string(2)
memory usage: 820.7 KB


ORG_AIRPORT
13303    8081
Name: count, dtype: Int64
```

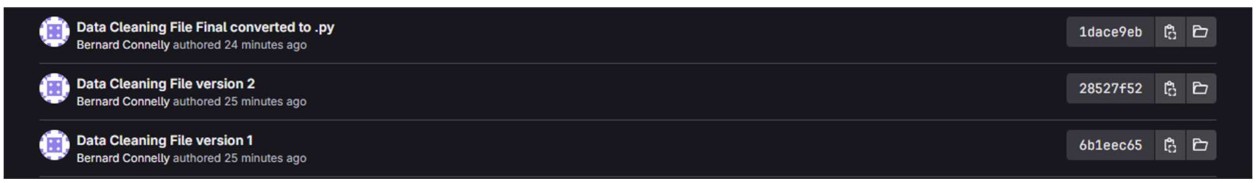
Subsequent steps were to clean the data to ensure accuracy. The two main methods employed were dropping flights whose values were null under the assumption that they did not occur and ensuring there were no leading or trailing spaces in the strings. This was coded using the dropna() and strip functions respectively.

```
# Dropping rows with missing values under the assumption the flights did not occur
df_filtered.dropna(subset=['DEPARTURE_TIME', 'DEPARTURE_DELAY', 'ARRIVAL_TIME', 'ARRIVAL_DELAY'], inplace=True)
print(df_filtered.isnull().sum())
## Confirming no more null values
```

```
YEAR                   0
MONTH                  0
DAY                    0
DAY_OF_WEEK            0
ORG_AIRPORT            0
DEST_AIRPORT           0
SCHEDULED_DEPARTURE    0
DEPARTURE_TIME         0
DEPARTURE_DELAY        0
SCHEDULED_ARRIVAL      0
ARRIVAL_TIME           0
ARRIVAL_DELAY          0
dtype: int64
```

```
# Checking string columns for leading or trailing spaces
string_columns = ["ORG_AIRPORT", "DEST_AIRPORT"]
df_trim_check = df_filtered[string_columns].map(lambda x: x.strip() != x)
print(df_trim_check.any())
## No leading or trailing spaces identified in the strings

ORG_AIRPORT     False
DEST_AIRPORT    False
dtype: bool
```

The cleaned data frame was exported as cleaned_data.csv to use in the poly_regressor file. Additionally, the steps for the above code were submitted to the GitLab repository.



D. Train Data & Modify Code Template

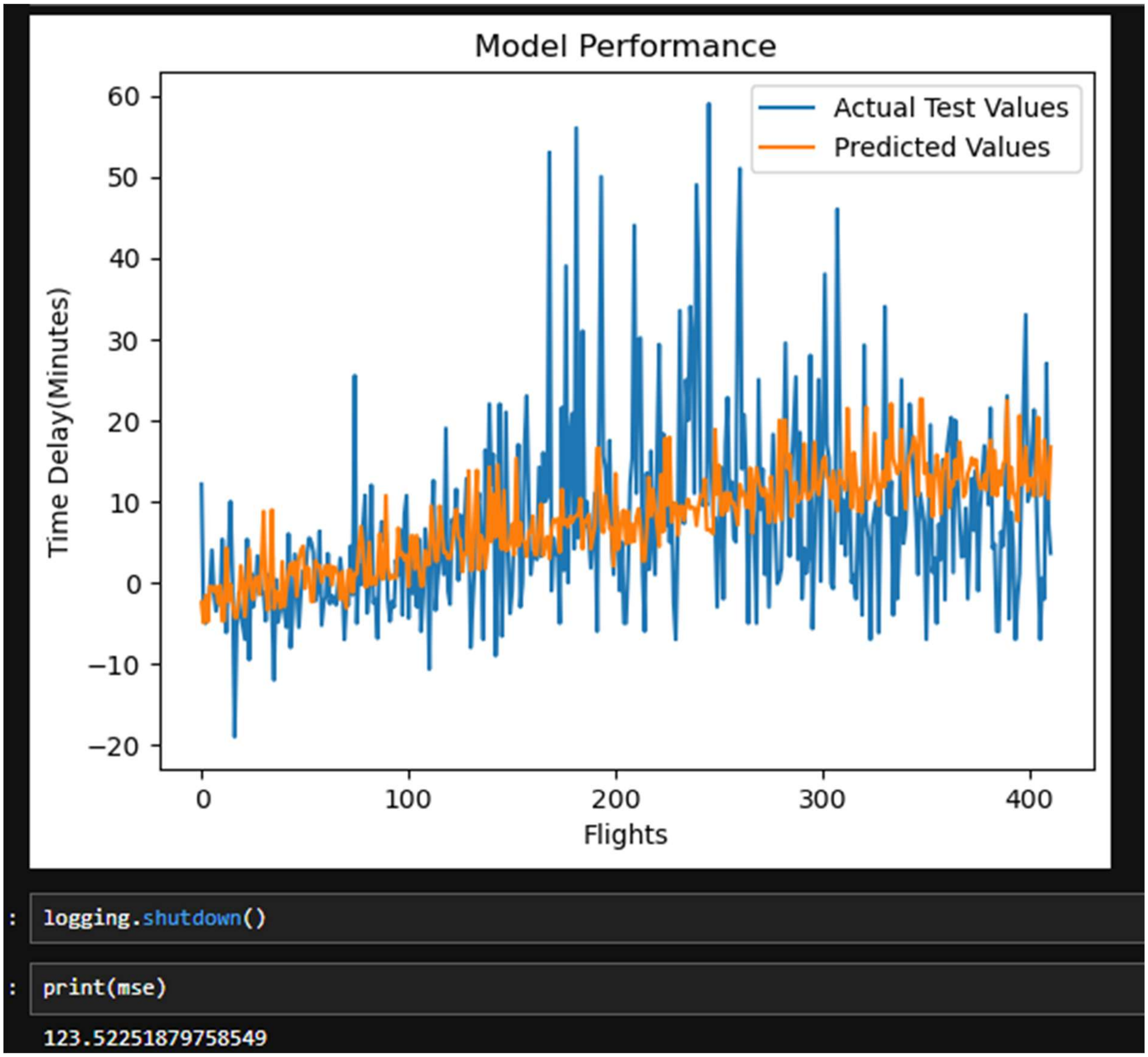
This portion of the project posed some of the most significant challenges, as taking on someone else's code without fully understanding their intentions can be daunting. In debugging the code, a few changes had to be made. The first was commenting on several variables in the argument parser section. In addition, the code for this section was written to only be utilized in an MLFlow environment. This made the project difficult as debugging through MLFlow was far more time-consuming than running directly in Python. To rectify this, I updated the argument parser section to add a loop to either search for a command line prompt if given for MLFlow or default to specific values for debugging purposes.

```
# Set up the argument parser
order = 1
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Parse the parameters for the polynomial regression')
    parser.add_argument('--num_alphas', metavar='N', type=int, default=20, help='Number of Lasso penalty increments')

    args, unknown = parser.parse_known_args()

    num_alpha_increments = args.num_alphas
    num_alphas = args.num_alphas
else:
    num_alpha_increments = 20
    num_alphas = 20
```

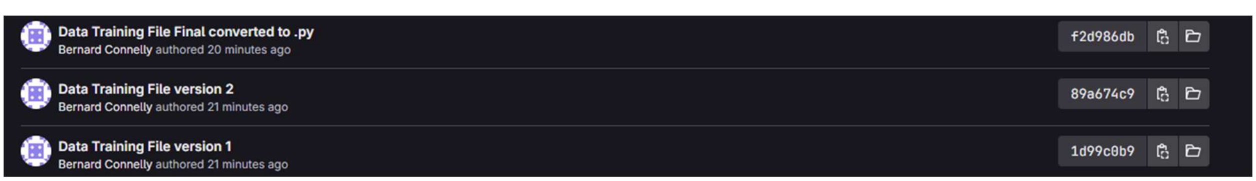
With that portion rectified, most of the code was left as is. Each export was reviewed for naming conventions to ensure no incorrect exporting or importing occurred. Still, no significant changes were made until the final steps, which the previous coder had left incomplete. These details were worked through as follows below:

```
# TO DO: create an MLFlow run within the current experiment that logs the following as artifacts, parameters,
# or metrics, as appropriate, within the experiment:

with mlflow.start_run(experiment_id = experiment.experiment_id, run_name = "Final Model - Test Data"):
    # 1.  The informational log files generated from the import_data and clean_data scripts
    mlflow.log_artifact("polynomial_regression.txt")

    # 2.  the input parameters (alpha and order) to the final regression against the test data
    mlflow.log_param("alpha", num_alphas)
    mlflow.log_param("order", order)

    # 3.  the performance plot
    # Labeling Variables
    X_poly = poly.transform(X_test)
    predict = ridgereg.predict(X_poly)

    # Creating performance plot
    plt.figure()
    plt.title ("Model Performance")
    plt.plot(Y_test, label="Actual Test Values")
    plt.plot(predict, label="Predicted Values")
    plt.xlabel('Flights')
    plt.ylabel('Time Delay(Minutes)')
    plt.legend()
    plt.savefig("performance_plot.png")
    mlflow.log_artifact("performance_plot.png")

    # 4.  the model performance metrics (mean squared error and the average delay in minutes)
    mse = mean_squared_error(Y_test, predict)
    average_delay_minutes = predict.mean()

    mlflow.log_metric("mean_squared_error", mse)
    mlflow.log_metric("average_delay_minutes", average_delay_minutes)

mlflow.end_run()
```

Each of the above steps functioned properly and yielded the outputs required.



All the above files were initially run in Jupyter Notebook as .ipynb files to ensure no further debugging was required. Afterward, they were converted to .py files using Jupyter's console. Each of these files was also saved to the GitLab Repository.



Part 2: Running the Pipeline

After the code was imported, filtered, cleaned, and trained, the next step was to create the automation files to run an MLFlow pipeline. The .ipynb files for the previous steps were converted to .py files, a main.py file was designed to simplify and run the code in tandem, and an MLProject file was created to guide the system in executing an automated pipeline.

E. MLProject File & Environment .yaml File

To simplify running the pipeline, I created a Python file that combined all the separate steps into one command and named this main.py. Details of this file were as follows:

```python
import os
import mlflow

def main():
    with mlflow.start_run(run_name="Full_Pipeline_Run"):
        # Step 1: Import data
        print("Running Import Data Script...")
        os.system("python data_importing_final.py")

        # Step 2: Clean data
        print("Running Clean Data Script...")
        os.system("python data_cleaning_final.py")

        # Step 3: Train Model
        print("Running Train Model Script...")
        os.system("python poly_regressor_Python_1.0.2_Final.py")

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

To initiate the MLProject portion, I had to update the pipeline_env.yaml file that was in the base epository in GitLab. Additional packages needed to be installed, and the versions also required to be resolved.

```yaml
name: pipeline_env
channels:
  - conda-forge
dependencies:
  - python=3.12.3
  - pandas=2.2.3
  - numpy=1.26.4
  - seaborn=0.13.2
  - matplotlib=3.8.4
  - scikit-learn=1.4.2.*
  - mlflow=2.20.1
```

Finally, an MLFlow file was coded to execute the code and kick off the MLFlow project.

```yaml
name: bconnelly_D602_Task2_Pipeline

name: airport_delay_pipeline

conda_env: pipeline_env.yaml

entry_points:
  main:
    command: "python main.py"
```

With the setup complete, the command line was utilized to kick off the MLFlow project, which correctly moved through the data importing, cleaning, and training steps and logged the appropriate artifacts, metrics, and parameters to the MLFlow instance. The relevant updated files were also committed to GitLab.

Updated yaml file with correct package versions Bernard Connelly authored 14 minutes ago	7ef5d0a4
Main Python file converted to .py Bernard Connelly authored 15 minutes ago	85d64002
Main Python file to guide and order the steps Bernard Connelly authored 16 minutes ago	6d9ebb7b
MLProject file to guide pipeline Bernard Connelly authored 17 minutes ago	d3943f88

Part 3: Process & Challenges

F. Describe writing the code, challenges faced, and how they were addressed

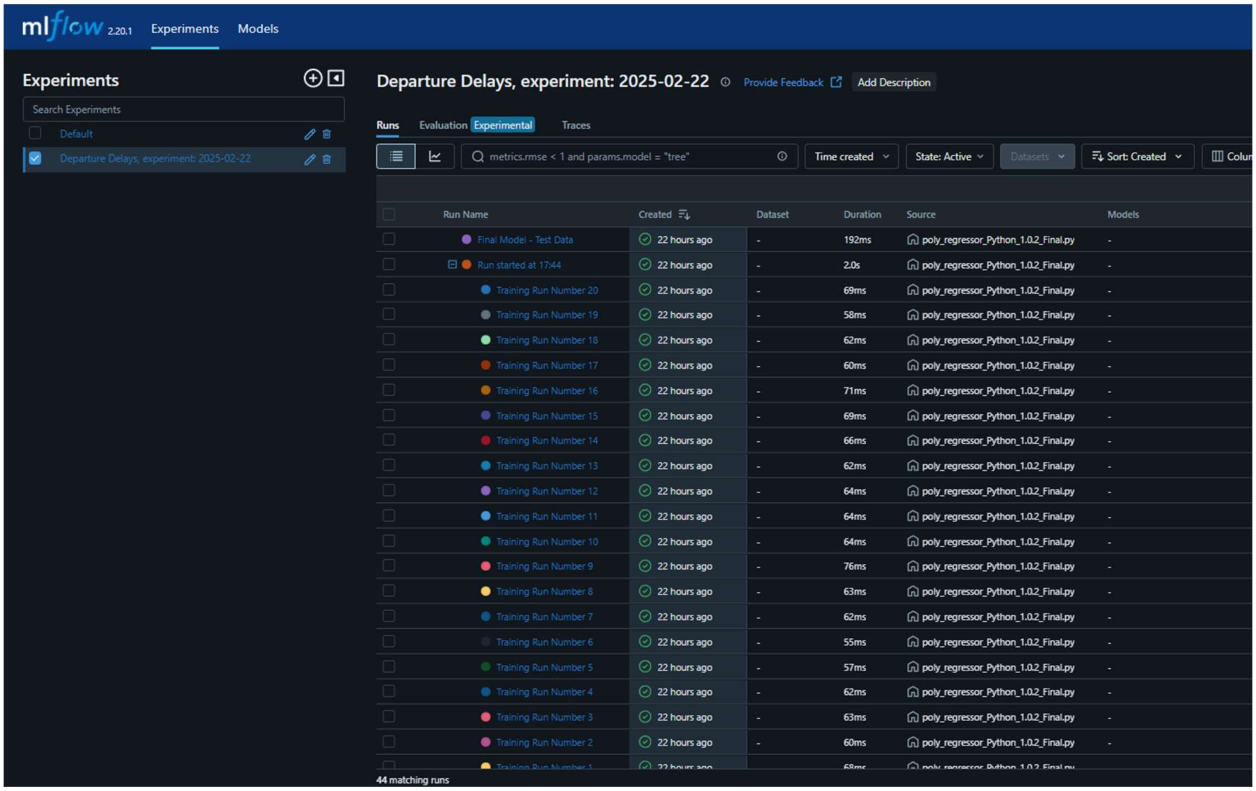
This project was challenging as I did not know how MLOps functioned and had zero knowledge of MLFlow before starting. Completing the individual importing and cleaning steps presented a little problem, but the specific details for MLFlow presented additional challenges. The first issue came with creating the DVC file, where initializing the environment as a part of the importing process threw an error because the environment existed. This was resolved via an if/else loop to skip the creation of the environment if it already existed. With the DVC in order, the remainder of the issues came from fixing the code in the poly_regressor file. Initially, several lines needed to be uncommented to clear errors in the initial coding, and a variable needed to be updated to match with other portions of the pre-written code. Additionally, to debug the code, I needed it to run on a local machine first, which would not work with the argument parser portion of the script. This was rewritten to ensure the file would run locally and in an MLFlow project.

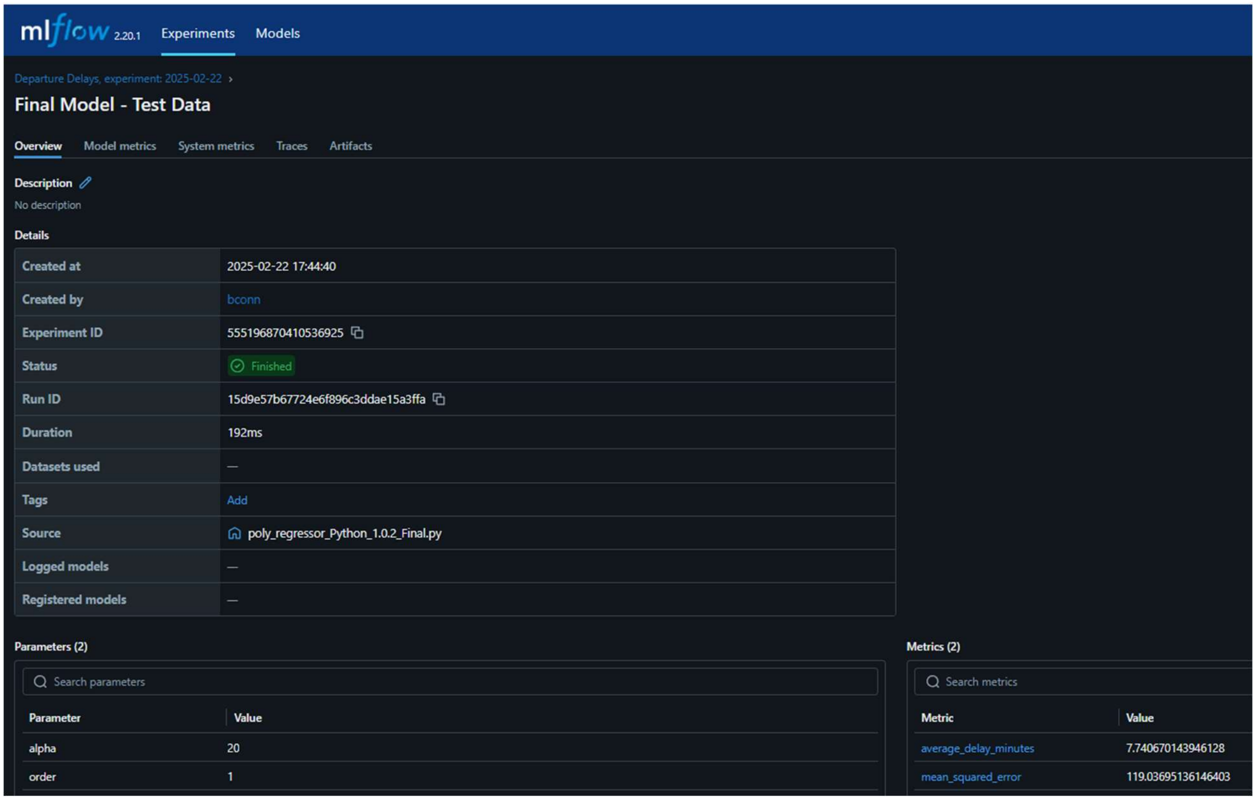
Attempts to run the pipeline failed because the pipeline_env file was housing versions of packages I was not running on my machine, and some packages were not included in the code. I resolved this by checking versions within the poly_regressor script as the packages were imported and updating the yaml file with the appropriate values. This was completed using the .__version__ function, as seen below.

```python
## Printing out package versions for .yaml file
print("\nPackage versions:")
import sys #For local running and testing
import datetime
import argparse
import logging
import os
import pickle
import json
## Standard Python Packages do not have individual versions
print("Python:", sys.version)
print("Json, OS, Logging, ArgParse, DateTime, Pickle are all built-in Python modules")
import pandas as pd
print("Pandas:", pd.__version__)
import seaborn as sns
print("Seaborn:", sns.__version__)
import matplotlib
import matplotlib.pyplot as plt
print("Matplotlib:", matplotlib.__version__)
import numpy as np
print("Numpy:", np.__version__)
import sklearn
from sklearn.preprocessing import PolynomialFeatures, LabelEncoder, OneHotEncoder
print("Sklearn:", sklearn.__version__)
from sklearn import metrics, linear_model
from sklearn.metrics import mean_squared_error #For calculating MSE
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
import mlflow
import mlflow.sklearn
print("MLFlow:", mlflow.__version__)
```

```
Package versions:
Python: 3.12.3 | packaged by conda-forge | (main, Apr 15 2024, 18:20:11) [MSC v.1938 64 bit (AMD64)]
Json, OS, Logging, ArgParse, DateTime, Pickle are all built-in Python modules
Pandas: 2.2.3
Seaborn: 0.13.2
Matplotlib: 3.8.4
Numpy: 1.26.4
Sklearn: 1.4.2
MLFlow: 2.20.1
```

Working through the steps left as “To Do” at the bottom of the poly_regressor script was a trial and error process until the correct code would yield the artifacts, plots, and calculations required for the project. This was another reason running locally was a crucial step in the process – running the code in an MLFlow instance took significantly longer. Finally, linking the different components of the project together posed a problem, so this was resolved by creating a single .py file that combined all of the files and ran them seamlessly. This connected all components, resulting in the pipeline being run successfully and the appropriate metrics being logged in the MFlow UI.




mlflow 2.20.1
Experiments
Models
Departure Delays, experiment: 2025-02-22 ›
Final Model - Test Data
Overview
Model metrics
System metrics
Traces
Artifacts
Description
No description
Details
Created at
2025-02-22 17:44:40
Created by
bconn
Experiment ID
555196870410536925
Status
Finished
Run ID
15d9e57b67724e6f896c3ddae15a3ffa
Duration
192ms
Datasets used
—
Tags
Add
Source
poly_regressor_Python_1.0.2_Final.py
Logged models
—
Registered models
—
Parameters (2)
Search parameters
Parameter
Value
alpha
20
order
1
Metrics (2)
Search metrics
Metric
Value
average_delay_minutes
7.740670143946128
mean_squared_error
119.03695136146403

References

No other sources were used outside of the WGU course materials provided