**Task 2: Data Production Pipeline**

Joseph Emmanuel Cayetano

Western Governors University

D602: Deployment Task 2

December 8, 2024

1. **Create your subgroup and project in GitLab**

**GitLab repository URL:** https://gitlab.com/wgu-gitlab-environment/student-repos/jcayet5/d602-deployment-task-2/-/tree/Task2Branch?ref\_type=heads

**Repository branch history screenshot:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **Write a script in Python to import the data and format it according to the criteria required by the model script. Create a DVC metafile for the dataset.**

A screenshot of a computer

Description automatically generated

This is the script (imported\_formatted\_data.py) for part B.

A screenshot of a computer

Description automatically generated

cleaned\_data.csv.dvc is the metadata file generated by DVC (This was done after part C when the data was cleaned).

1. **Write a script in Python to filter data only departures from the chosen airport, then implement at least two other data cleaning steps.**

**A screenshot of a computer program

Description automatically generated**

This is the script (cleaned\_data.py) for part C.

1. **Implement an MLFlow experiment that captures the features listed in the comments within the poly\_regressor file for Python.**

A screenshot of a computer program

Description automatically generated

This code is part of the poly\_regressor script that has been modified.

1. **Write an MLProject file that links the two scripts you wrote in parts B and C with the modified poly\_regressor script from part D.**

A computer screen shot of a program

Description automatically generated

This is my MLProject file that links the two scripts (imported\_formatted\_data.py & cleaned\_data.py) I wrote in parts B and C with the modified poly\_regressor script from part D.

1. **Provide a written explanation of how you wrote the code and MLProject pipeline, including any challenges you encountered and how you addressed these challenges. Include a screenshot of your MLProject pipeline running successfully.**

For part B, I created a python script called imported\_formatted\_data.py. This script imports the raw flight data from the Bureau of Transportation Statistics website and converts the data into a pandas dataframe. Then, the script renames the columns to match those defined in the poly\_regressor script, ensuring its functionality. The script also replaces missing values with zeros and converts the datatype of the columns DEPARTURE\_TIME, DEPARTURE\_DELAY, ARRIVAL\_TIME, and ARRIVAL\_DELAY from float to an integer. The script saves the processed data to imported\_formatted\_data.csv, which is the CSV file the cleaned\_data.py script from part C will use to clean and prepare for regression analysis.

For part C, I created a python script called cleaned\_data.py. This script imports the formatted data that was created in part B and filters the data to only departures from LAX. The script also performs data cleaning methods like removing duplicate rows and dropping the unnecessary FL\_DATE column to reduce noise. The script then exports the cleaned dataset as cleaned\_data.csv, which is the data the poly\_regressor script from part D will use to train the polynomial regression model. The cleaned\_data.py script basically ensures that the data to be used for training the polynomial regression model is cleaned and filtered.

Before implementing an MLFlow experiment in the poly\_regressor script, I used Data Version Control (DVC) on cleaned\_data.csv to store the metafile of the dataset in my GitLab repository. This allows me to track changes in the data without storing it directly in GitLab where it can slow down the repository due to its large size. DVC also enables collaborators to retrieve the exact version of cleaned\_data.csv used in the pipeline without manually generating it. This “ensures everyone works with the same data versions, eliminating confusion and wasted time” (Tuychiev, 2024, par. 7).

For part D, I implemented MLFlow into the poly\_regressor script to log key details about the model training and testing process. I used the log\_param() function that logs the two input parameters, ‘alpha’ and ‘order’, to track the configuration used for the polynomial regression model. This allows me to compare different runs with different parameter values. For the metrics, I used the log\_metric() function to record two key performance metrics, which are mean squared error and average delay. This allows me to evaluate the regression model’s performance and compare it with other runs. For the artifacts, I used the log\_artifact() function to upload files that were generated during training and evaluation. The files that were created are model\_performance\_test.jpg and polynomial\_regression.txt. The .jpg file is a visualization that shows the relationship between predicted delays and actual mean delays for the regression model, while the .txt file details the steps and outcomes of the regression modeling process. Lastly, the log\_model() function from sklearn library saves the trained regression model as an MLFlow model artifact under the name “final model.”

For part E, I created an MLProject file to connect imported\_formatted\_data.py, cleaned\_data.py, and the modified poly\_regressor script into an MLflow pipeline. First, I set the project name to ‘flight\_delay\_prediction,’ and included the Conda environment file pipeline\_env.yaml that defines all dependencies to ensure reproducibility. Then, I wrote entry points to define the key steps of the pipeline. The entry points included import\_data to run imported\_formatted\_data.py, clean\_data to run cleaned\_data.py, and train\_model to run the modified poly\_regressor script. The last entry point, run\_pipeline, combines the three previous entry points into a single pipeline to automate the entire workflow. The command I wrote in Anaconda prompt to execute the MLFlow pipeline is: mlflow run . -e run\_pipeline --experiment-name "FlightDelays". This command basically runs the run\_pipeline entry point within the MLProject file, and ensures that all metrics, parameters, and artifacts from the pipeline are logged under “FlightDelays” experiment.

There are three challenges that I have encountered during this task. The first challenge is about the dependency conflict between the two Python packages, s3fs and fsspec, during environment creation. In my initial setup, s3fs was set to version 2023.10.0, whereas fsspec was set to version 2024.10.0. Since DVC relies on these packages, dependency conflicts between them could lead to unexpected behavior in DVC. To solve this issue, I simply upgraded s3fs to the latest version which is 2024.10.0.

The second challenge is about active run conflicts and run ID mismatches. When I ran the MLFlow pipeline for the first time, it failed because there was an active run ID already present in the environment, and MLFlow does not allow multiple runs in the same process. This happened because the poly\_regressor script encountered an error, which ended the script unexpectedly but left the run active, causing conflicts when starting a new run. To solve this issue, I added a chunk of code within the poly\_regressor script that closes any active runs before running the MLFlow experiment to avoid active run conflicts. Another issue arose, however, involving run ID mismatches. The MLFlow run failed to execute because there was a mismatch between the experiment ID used in the mlflow.start\_run() call within the poly\_regressor script, and the experiment set in the current environment. To fix this issue, I explicitly set the experiment name to “FlightDelays” in the script by using the set\_experiment() function to ensure that the MLFlow run is correctly associated with the intended experiment. All MLFlow runs are now guaranteed to log under the “FlightDelays” experiment.

The third challenge is about DVC not working due to cleaned\_data.csv already being tracked by Git. Before adding the dataset to DVC, I pushed the dataset into my GitLab repository not knowing it would create a conflict. Since the dataset was already under Git tracking, DVC could not manage it because DVC requires exclusive control over files it tracks. To solve this issue, I removed cleaned\_data.csv from Git tracking and re-added it under DVC management. DVC now manages the dataset, while Git only tracks the metafile of the dataset.

A black screen with white text

Description automatically generated

Screenshot of my MLProject pipeline running successfully.

**References**

Tuychiev, B. (2024, July 14). *The Complete Guide to data version control with DVC*. DataCamp. https://www.datacamp.com/tutorial/data-version-control-dvc