UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

FLIGHT DELAY PREDICTION REPORT

Victor Ramirez Castaño

Mohammed Ahajjam Ziggaf Kanjaa

Younes Labchiri Boukhalef

INDEX

1. Introduction

The aim of this project is to develop a machine learning model to predict the **arrival delay (`ArrDelay`)** of commercial flights using the data from the U.S. Department of Transportation.

This project requires developing a Spark application capable of loading data, training a model using **MLlib in a Jupyter Notebook**, validating it, and creating an application to test the model on unseen data.

2. Dataset Used

The dataset contains information on U.S. domestic flights. As per the requirements, we must predict the delay using only information known **at the time of takeoff**.

2.1. Forbidden Variables

The following variables were excluded from the analysis because they contain information that is unknown at the moment the plane takes off :

- `ArrTime`
- `ActualElapsedTime`
- `AirTime`
- `TaxiIn`
- `Diverted`
- `CarrierDelay`
- `WeatherDelay`
- `NASDelay`
- `SecurityDelay`
- `LateAircraftDelay`

Additionally, we filtered out flights where `Cancelled == 1` because cancelled flights do not have an arrival time or delay, making them irrelevant for this task.

2.2. Selected Variables

After analyzing the dataset, we selected the following variables for our model:

- **`DepDelay`:** The delay at departure. This is the strongest predictor for arrival delay.
- **`TaxiOut`:** The time spent rolling from the gate to the runway. This is known once the flight takes off and significantly impacts the total elapsed time.
- **`Distance`:** The flight distance in miles. Longer flights allow for more potential time delays.
- **`CRSElapsedTime`:** The scheduled flight duration.
- **`UniqueCarrier`:** The airline code. Different carriers have different efficiencies.
- **`Origin and Dest`:** Airport codes. These capture location and other specific factors.
- **`Year, Month, DayOfWeek and DayofMonth`:** Temporal variables to capture seasonal patterns.

- **`CRSDepTime and CRSArrTime`:** Scheduled departure and arrival times.
- **DepTime:** Real departure time of the plane.

2.3. Other excluded variables

Other variables that were excluded from the model were **`FlightNum`** and **`TailNum`**, since those are just identifiers and don't provide any type of information to whether or not the flight is going to delay itself. Also **`Cancelled`** and **`CancellationCode`** are not taken into account, since a canceled flight doesn't have a delay because it hasn't even taken off.

3. Data Processing (Spark Pipeline)

WARNING!!!: At the beginning of the notebook when creating the Spark Session, a specific amount of RAM is assigned to the JVM which suits one of our machines. You may want to change that to save some memory.

To ensure the process is reproducible, we stored all transformations in a Spark ML Pipeline.

3.1. Data Cleaning and Feature Engineering

Before training, we made several transformations to the data:

- **Filtered instances:** First of all, every cancelled flight is dropped from the dataframe**.** Then, if any instance of the dataset contains a null value in any of the following variables, it is dropped from the dataframe, since the amount of null values in this variables after dropping cancelled flights is low enough (compared to the total size of the dataset) to be able of not worrying too much about losing information:
    - `CRSArrTime`
    - `ArrDelay`
    - `Year`
    - `Month`
    - `DayofMonth`
    - `DayOfWeek`
    - `UniqueCarrier`
    - `CRSElapsedTime`
    - `DepDelay`
    - `Origin`
    - `Dest`
    - `Distance`
- **SQL Transformations:** We used a SQLTransformer to create derived variables and fix data format issues:
    - **`TaxiOut` Fix**: Since some derived features depend on the value of taxiOut, we ensure it always has a value using the median of the feature. If the median can't be computed, 0 is used as its value.
    - **`Time` Fix:** We implemented logic to handle `CRSDepTime` = 2400, converting it correctly to 0000 of the following day to prevent errors.
    - **`TakeOffTime:`** It is calculated as `CRSDepTime + DepDelay + TaxiOut`. This represents the exact minute the plane leaves the ground.

- **`LandingEst`:** It is calculated as `TakeOffTime` + `CRSElapsedTime`. This is the estimated landing time based on the schedule.

3.2. Encoding and Scaling

The categorical variables have been encoded and the numerical variables scaled.

- **Categorical Encoding**: We applied `StringIndexer` followed by `OneHotEncoder` to variables like `UniqueCarrier`, `Origin`, `Dest`, `Month`, and `DayOfWeek`. This was also applied to some numerical variables like `DayofMonth` and `Year` because even though these variables are numerical, there isn't an importance hierarchy in them. This converts categories into binary vectors suitable for regression.
- **Numerical Scaling:** We used **`VectorAssembler`** to combine features and **`StandardScaler`** to normalize the numerical ones (mean 0 and standard deviation 1). This prevents variables with large values (like `Distance`) from dominating the model optimization.

4. Model Training

4.1. Algorithms Used

In this project, we evaluated three tree-based regression algorithms. The Decision Tree Regressor, the Random Forest Regressor, and the Gradient-Boosted Tree (GBT) Regressor. These models were selected to progressively increase model complexity while maintaining interpretability and robustness, allowing a fair comparison between the different algorithms from a simple to a more advanced method.

The Decision Tree Regressor was chosen as a baseline model. Its simplicity and interpretability allow for an initial assessment of the model performance and help identify the limitations such as overfitting and high variance, that are typical on this model.

The Random Forest Regressor was selected as a more robust extension of the decision tree. By combining the predictions of multiple trees, it reduces variance and overfitting, resulting in a more stable and reliable model for complex datasets than the previous one.

Finally, **the Gradient-Boosted Tree (GBT) Regressor** was included as a high-performance model that captures complex non-linear patterns by correcting the errors of previous trees in a sequential way. This usually improves predictive accuracy, although it requires some tuning to avoid overfitting.

4.2. Hyperparameter tunning

The code in the notebook has been prepared to be able to tune hyperparameters using `ParamGridBuilder`, but configured in a way it will not search for the best parameters since we lack the resources to compute the grid search.
The parameters set as default for each model training are:

- Decision Tree:
  - `maxDepth`: 5
  - `maxBins`: 32

- Random Forest:
  - `numTrees`: 20
  - `maxDepth`: 5

- Gradient-Boosting:
  - `maxIter`: 10
  - `maxDepth`: 3

4.3. Validation Process

To validate the model, we used the following strategy:

- **Split:** The data was split into **80% Training** and **20% Testing**.
- **Cross-Validation:** We used **3-Fold Cross-Validation** on the training set. This ensures the hyperparameters are tuned on a robust estimate of the model's performance.

4.4. Model Selection Logic

We implemented an evaluation loop in the Jupyter Notebook. For each model (DT, RF, GBT), we calculated:

- **RMSE (Root Mean Squared Error):** Penalizes large errors.
- **MAE (Mean Absolute Error):** The average error in minutes.

Hybrid Score: We used a conditional metric that we can see below to select the best model:

> *If RMSE > 2 * MAE (which indicate significant outliers), we select the model with the best MAE. Otherwise, we select the model based on RMSE.*

The best model found during this process was automatically saved to the `best_model` directory in the project root.

5. Testing the Model (Application)

We developed a spark application (app.py) to test the saved model on unseen data.

How to run the application:

The application can be run with the following command, receiving the test data as a parameter to the spark-submit script.

```
spark-submit app.py <path_to_test_file.csv>
```

Example:

```
spark-submit app.py ../data/2008.csv
```

The application loads the data, applies the pipeline (cleaning, encoding, scaling), generates predictions using the best_model, and prints the final performance metrics (RMSE, MAE, R^2) to an output.txt. Also, it stores the results in a parquet format.

6. Conclusion

In conclusion, this project compared three supervised machine learning models using the selected dataset. The use of the Spark Pipelines ensured that the same preprocessing steps were applied in a consistent way for both training and test data, reducing the risk of errors and ensuring a clear separation between the training and test data. Finally, the cross-validation and the use of RMSE and MAE helped to select a robust model out of the three used.