Exploring, analyzing and processing the data

The dataset used for training the model was 1987.csv. The dataset initially containing 29 variables, was reduced to 19 due to the forbidden ones **not to be used**, because they contain unknown information useless for the model:

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

The rest, were described as follows:

- Year - the year for which the data was collected from all commercial flights on major carriers within the USA, 1987, of type big integer.
- Month - the respective month of the flight, big int type ranging from 10 (October) to 12 (December).
- DayofMonth - the day of the flight, with big int values from 1 to 31.
- DayOfWeek - the day of the week from 1 to 7, where 1 is Monday, 7 is Sunday, big int data type
- DepTime - time of actual departure, data type double, containing values from 1 to 2400 (following the hhmm format)
- CRSDepTime - scheduled departure time, big int type, with values from 1 to 2359, with the format hhmm, for example 23:59 being the highest value.
- CRSArrTime - the arrival time that was schedules, big int, minimum value 1, maximum 2400
- UniqueCarrier - a unique carrier code, type string
- FlightNum - the flight number ranging from1 to 6282
- TailNum - the plane's tail number
- CRSElapsedTime - the scheduled elapsed time, which represents the planned duration of the flight, expressed in minutes. The data type is big int with values from -40 to 1555
- ArrDelay - the arrival delay in minutes of type double, minimum value being -1302 (negative value) and maximum 1033
- DepDelay - of type double, represents the departure delay expressed in minutes with a negative minimum value of -1345.0 and max of 1439

- Origin - the origin IATA airport code, which is a 3 letter unique code to identify each airport. Represented by string data type, some examples are ABE (Lehigh Valley International Airport) and YUM (Yuma Airport)
- Dest - the corresponding destination IATA airport code, with the same data type as Origin, string
- Distance - the flight distance in miles, data type double, from 0 to 4983
- TaxiOut - the taxi out time (double data type), which means the number of minutes the plane will spend moving on the ground between leaving the gate and take-off.
- Cancelled - if the flight was cancelled or not, with values 1 and 0
- CancellationCode - representing the reason for cancellation (A = carrier, B = weather, C = NAS, D = security) of type double

After this analysis, there were multiple encounters of **"NA" values** for minimum or maximum values, which is an issue related to **missing values** or potential conflicting data types. The attributes that can be **dropped** are **TailNum**, since it has only NA (missing) values. The same step was implemented for **TaxiOut** and **CancellationCode** for the same reason. Additionally, **Year** attribute was **removed**, because it does not provide meaningful information to the model.

For the rest of the attributes containing NA values, these will be dropped. This is because the initial dataset containing over 1 million observations provides a **sufficient sample size**. In case of removing missing values, this will ensure a better data quality that is more reliable for the model as well. The number of **missing values** is as follows:

- DepTime = 19685
- ArrDelay = 23500
- DepDelay = 19685
- Distance = 1015

Next, **negative value** cases were analyzed. The first example is for the **CRSElapsedTime** attribute and since it has only 2 negative values. The next cases where negative values represent an issue were **ArrDelay** and **DepDelay**. This was solved by filtering only non-negative values to use for the rest of the preprocessing.

After more analysis, it was decided that the variables Month, DayofMonth, DayOfWeek are being kept since they may provide useful information for identifying a potential pattern in delays on certain days of the week. For example, delays may be more likely to happen in December around holidays or on the weekends. On the other hand, because the attribute **Year** has only one value for all the observations that the model was training on, it is **removed**.

Moreover, DepTime and CRSDepTime have a significant impact on the delay since they illustrate the scheduled and actual departure time. For this reason they are kept. CRSArrTime is also being kept, because it provides a reference point for the model to be able to compare the scheduled time with the arrival delay. In this way the model can learn the deviation between scheduled and actual arrival time.

The unique carrier code was not removed, since it also might be useful in indicating whether a specific airline is prone to delays or not. This will be demonstrated when working on feature engineering at a later stage. Therefore, because it may provide meaningful information it is kept. The importance, nonetheless, will be confirmed when applying the multivariate feature selection filter.

On the other hand, the flight number (**FlightNum**), which is unique every time, has no correlation to delays and **is removed**. Next, **CRSElapsedTime** is kept, because it may also provide a pattern, for example for shorter flights there can be a bigger delay. **DepDelay** is also highly correlated to the target class ArrDelay, as can be observed in the correlation heatmap (Figure 1) so its removal is avoided.

Origin and **Dest**, may also capture important information for a potential pattern, such as specific airports that are more likely to have flight delays (due to traffic or weather for instance). And for the motive that these are used to create new variables, they will not be removed. **Distance**, in a similar way as **CRSElapsedTime**, can indicate the dependency between flight duration/distance and delay occurrence. Therefore, it is kept as well.

Finally, the **Cancelled** attribute becomes useless to the arrival delay, since there is no arrival and the flight is not completed. For this reason, **it is removed**.

After feature engineering, some new variables will be created and some will be dropped that replace the newly created ones.

The first step done is, based on the UniqueCarrier, creating the **Carrier_Avrg_Delay attribute**. This new variable calculates the average delay for each carrier. So instead of having the unique code of the carrier, the Carrier_Avrg_Delay incorporates a potential pattern to be observed. Therefore, the **UniqueCarrie**r is **dropped** after.

In a similar way, with the aim to create observable patterns in delays, based on the Origin and Dest attributes, were **created Origin_Avrg_Delay and Dest_Avrg_Delay**. So, instead of having only the IATA code, now there is shown the average delay for each airport, which contains more meaningful information. As a result, the old **Origin** and **Dest** attributes are **removed**.

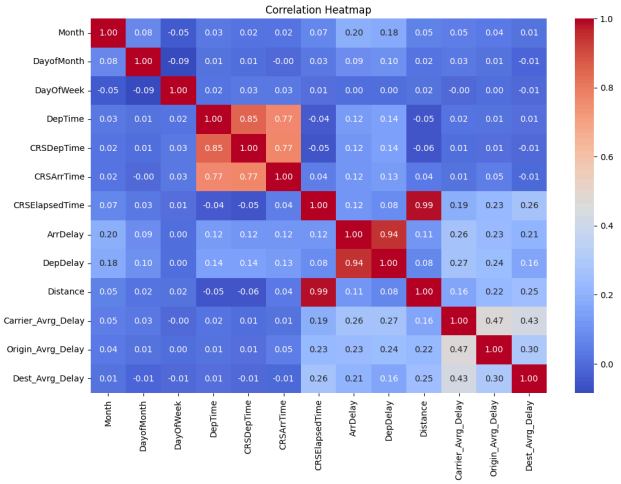

Figure 1 - Correlation Heatmap

And in the end, all the attributes (all of them are numerical) are converted to float, because this ensures consistency and it is a better option for the rest of the preprocessing steps (e.g. scaling).

Moreover, because the original dataset contained over 1 million samples it required a lot of computational resources. Therefore, this is why it was decided to apply undersampling to reduce it to around 10 500 samples. The best option was considered **stratified undersampling** over random sampling, because it would preserve the target class distribution. The **distribution** was considered as following:

1. Small delays - under 30 minutes;
2. Medium delays - between 30 and 60 minutes;
3. High delays - over 60 minutes.

It creates a new temporary attribute called 'DelayCategory' containing these 3 distinct categories and assigns each observation to its corresponding category based on the arrival delay value (ArrDelay). The goal of this step is to achieve a relatively balanced dataset based on the target value. In this way each of the 3 categories had the aim to contain 3500 samples each. And in order to achieve this goal, it was not possible to implement random undersampling, this is why stratified undersampling method was favorable.

Additionally, **normalization** was applied using the **min-max scaling** method. This normalized the features in a range from 0 to 1. This transformation preserves the original shape of the data distribution while bringing the values to the specific range. Many regression models, such as linear regression, SVM, are susceptible to the scale in features, this is why normalization will contribute to an improved performance in this case.

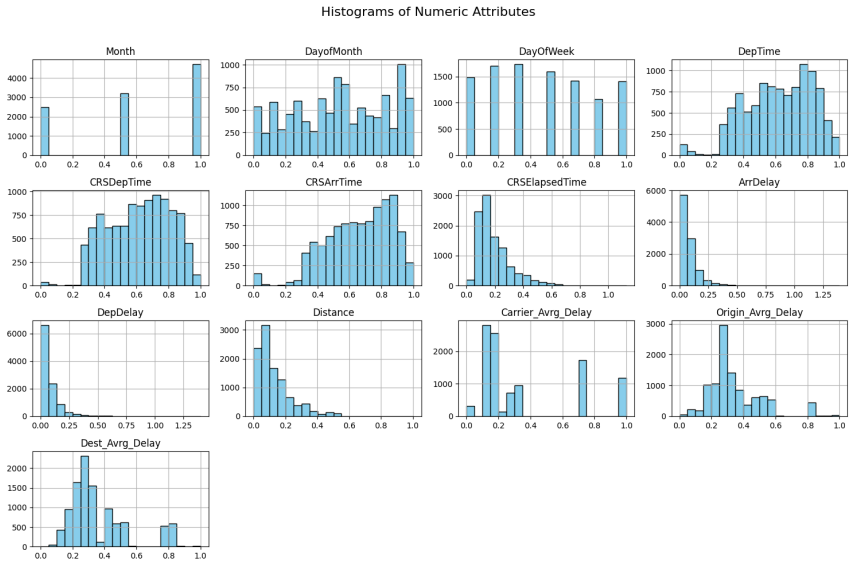

Figure 2 - Histograms of numeric attributes after preprocessing

Moreover, **multivariate feature selection** was applied. It considers the relationship between all the features and aims to select the most relevant ones. For the current application, the steps in this process are:

1. Combination of features into a single vector and correlation analysis between each feature and the target variable;
2. The Linear Regression model is trained on the data to asses the features' importance;
3. Features are ranked in a list of 7, the result being: 'Month', 'DepDelay', 'Dest_Avrg_Delay', 'Carrier_Avrg_Delay', 'Origin_Avrg_Delay', 'DayOfWeek', 'DayofMonth'.

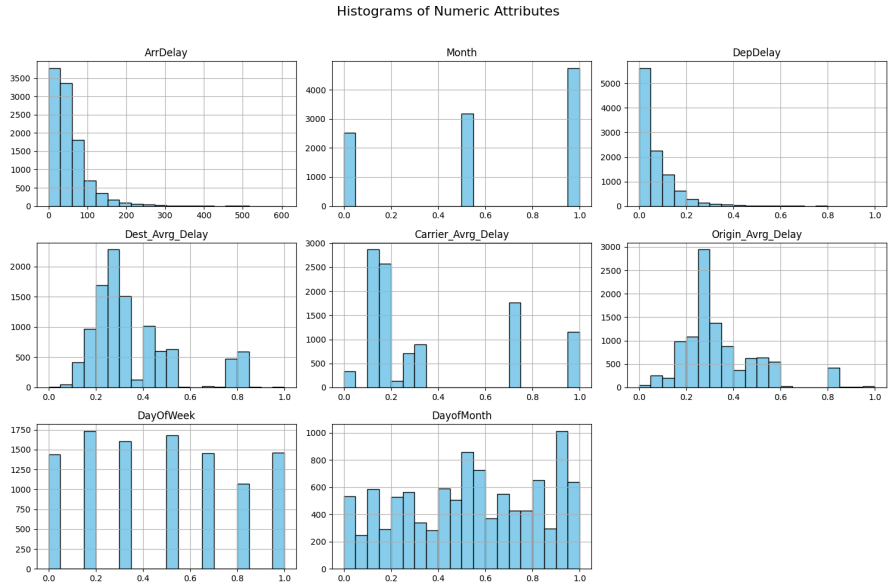

Figure 3 - Histogram of attributes after multivariate feature selection

As a conclusion, it can be assumed that temporal features like Month, DayOfWeek, and DayofMonth are significant, suggesting patterns in delays related to seasonality and weekly traffic variations. However, the most important one was selected the month, suggesting the strong influence on arrival delay. Through the key factors it can also be identified the departure delay which nevertheless impacts directly the arrival delay. The new attributes that reflect the average delay of the carrier and airports also suggest a pattern when it comes to specific cities and their airports and airline companies.

Creating the model

The machine learning models used were **LinearRegression, DecisionTreeRegressor, RandomForestRegressor and Gradient Boosted Trees Regressor**. Some the advantages of this models are:

- LinearRegression: Simple, Fast and Interpretable. Good when there are Linear relationships between the features and the target variable
- DecisionTreeRegressor: It can model non-linear relationships and it is interpretable.

- RandomForestRegressor: It's an ensemble method which might outperform the DecisionTree, but it lacks interpretability.
- GBTRegressor: It's a boosting technique which offers a high performance in many cases, but it also lacks interpretability.

The use of these different models assure a good combination of performance and interpretability.The models were generated with the **whole preprocessed dataset** and with the **reduced one by multivariate feature selection**. For each of the models, different values of their hyperparameters were used. Below a few advantages of each method with a short definition of each of the tuned hyperparameters will be given. The different values used are the ones between brackets:

- LinearRegression:
  - ElasticNet [0,0.2,0.8,1]: Allows to combine the regularization methods Lasso (L1) and Ridge (L2). With value 0 is pure L2 and with 1 is pure L1.
  - RegParam [0.05,0.1,0.2]: Controls the overall strength of the regularization. With value 0 no regularization is applied.
  - MaxIter [10,100,200]: Defines the maximum number of iteration.
- DecisionTreeRegressor, RandomForestRegressor and GBTRegressor (Gradient-Boosted Trees):
  - maxDepth [5,10,20,30]: Determines the maximum depth of the tree
  - minInfoGain[0,0.1]: Specified the minimum Information Gained to split a node
  - maxBins[32,64]: Maximum number of bins used to discretize the continuous features
- RandomForestRegressor extra parameter:
  - Bootstrap [True, False]: If Bootstrap is used for the training data of the trees.

The models generated by the combination of these hyperparameters were compared using a **5-fold cross-validation**, using as a metric the **Root Mean Squared Error**.

Validating the model

The original dataset was divided in two datasets. **70% of the dataset was used for the training,** and the **other 30% was used for the final evaluation** of the model.

For validating the performance of the models, the RegressionEvaluator method was used. This method calculates the Root Mean Squared Error. Then, each of the model generated with the different hyperparameters were trained with 5-fold cross validation and compared using the RMSE. Finally, the model with the best performance was validated against the test data calculating the RMSE.

Using cross-validation provides a **more reliable performance metric** than using directly the training split, as they are evaluated with different unseen test data and allows for a better hyperparameter selection. Finally, by using the test split to evaluate the final performance gives

us a real case scenario for how the performance will be on unseen data with the final model. The Root Mean Squared error is a **widely used metrics that offers an easy interpretation**, as it is in the same metric as the target variable, and it is suitable for hyperparameter tuning as it is sensitive for small changes in the performance of the model.

A summary table with the best models obtained for each algorithm is displayed:

Dataset	Model	Hyperparameters	RMSE
Full	LinearRegression	ElasticNetParam: 1 maxIter: 100 regParam: 0.1	18.48
Full	DecisionTreeRegressor	MaxBins: 64 MaxDepth: 5 minInfoGain: 0.0	20.55
Full	RandomForestRegressor	MaxBins: 64 MaxDepth: 10 minInfoGain: 0.0 Bootstrap: false	20.47
Full	GBTRegressor	MaxBins: 64 MaxDepth: 5 minInfoGain: 0.0	20.41
Multivariate	LinearRegression	ElasticNetParam: 0 maxIter: 10 regParam: 0.05	16.13
Multivariate	DecisionTreeRegressor	MaxBins: 64 MaxDepth: 5 minInfoGain: 0.0	20.41
Multivariate	RandomForestRegressor	MaxBins: 64 MaxDepth: 10 minInfoGain: 0.0 Bootstrap: True	18.90
Multivariate	GBTRegressor	MaxBins: 64 MaxDepth: 5 minInfoGain: 0.1	20.42

The model with the best performance with both datasets is the **LinearRegression** model, with the minimum RMSE obtained with the multivariate dataset. However, when used in the final application with the different datasets, the performance was **far from the one obtained in**

the test. This can be because the dataset is trained with a specific year flight and it has overfitted to that year. Using the linear Regression model with the full dataset obtained similar results with the one obtained in the test.

Testing the model with unseen data

The app first loads all csv files from the test data folder, then performs the necessary preprocessing to prepare the data for the model. This preprocessing includes dropping useless columns, dealing with missing and inappropriate values for the specific columns and scaling using MinMax technic. When the test data is preprocessed, it is used on a model and predictions and evaluations are made. There are multiple try-catch branches, ensuring that if the user provides the wrong format of data or model, or in case of any error, the application executes and provides info about the error. The final evaluation was done on 2008.csv and 1989.csv. The RMSE of **31.29** was obtained, which means that, on average, the model's predictions are off by around **31 minutes** from the actual arrival delay. This value is **higher than the RMSE during validation**, indicating a **significant performance drop when tested on new, unseen data**. The increase in RMSE suggests that the model struggles to generalize well to new datasets from different time periods (such as **2008.csv** and **1989.csv**). This performance gap likely stems from **overfitting to the training data from 1987**, meaning the model learned patterns that were too specific to that year and did not capture broader trends.

The MAE of **15.70** indicates that the model's predictions are off by an **average of 15.7 minutes** from the actual delay. Compared to the RMSE, this value is **lower**, which suggests that **most prediction errors are relatively small**, but **some large errors exist** that inflate the RMSE. The difference between RMSE and MAE shows that the model **occasionally makes large mistakes**, but most of the time, the errors are smaller. For instance, the model might predict a **20-minute delay** for a flight that ends up being delayed by **an hour or more**, which would significantly impact the RMSE but have a smaller impact on the MAE.

The **R^2 score** of **0.27** indicates that the model explains only **27% of the variance** in the arrival delay. In other words, the model captures **less than a third of the factors** affecting arrival delays, and the remaining **73% of the variation is unexplained** by the current features used in the model. This low R^2 value suggests that **important features affecting delays are missing from the dataset**, or that **external factors like weather, air traffic, and operational issues** play a significant role in determining arrival delays but are not captured in the training data.

The **Linear Regression model** remains the best choice due to its simplicity and solid performance during validation. However, its performance on unseen data highlights the need to account for **temporal variability** in future iterations. For future improvements, using a **more diverse training dataset spanning multiple years** could reduce overfitting and improve generalizability, also exploring **regularization techniques** further to mitigate overfitting.

The app is executed using the spark-submit command and no change of the code is necessary if best model or test data changes. The whole command is:

spark-submit --master local[*] app.py /path/to/test_data /path/to/best_model, where:

- part spark-submit --master local[*] is immutable and necessary
- app.py is the file where the code of the application is, also necessary while executing the application, if the file is renamed locally, changes have to be applied to the command
- /path/to/test_data is a path to the folder where test data is saved, application will read all csv files inside this folder, necessary
- /path/to/best_model is a path to the folder where best_model is saved, necessary

After executing this command, the application will provide the first 20 rows of predictions and actual values of target variable, to compare and then root mean squared error, mean absolute error and r-squared. Also, all the metrics and the predictions of the first 100 rows will be written in the file results.txt, that can be found in the folder results, which will be made in the working directory, if it doesn't already exist. The state of application can be monitored by opening https://localhost/4040 in the browser after executing the command.