Spark Practical Project Report

Sven Ligensa Lige Zhao Sebastian Mendoza

Winter Term 2024/25

1 Introduction

The objective of this project is to predict the arrival delay of flights based on the dataset downloaded from this website by making use of the Apache Spark Python API.

The code is made available on the following GitHub repository. The three main files and their purpose is the following:

- `Preparation.ipynb` (Section 2): Notebook for exploring the dataset and preprocessing it to be useful for the models.
- `Models.ipynb` (Section 3): Notebook for training and evaluating different ML models.
- `app.py` (Section 4): Spark application to perform data preprocessing and model training and testing.

2 Preprocessing

The first step is to inspect the dataset, for which we loaded the flight data for the year 2007. In accordance with the task description, we dropped the forbidden variables, which are not available during takeoff. We further dropped the following variables, as they should not have an impact on the arrival delay: `FlightNum`, `TailNum`, `TaxiOut`, `CancellationCode`. We then remove all rows of cancelled flights `Cancelled == 1`, as it does not make sense to predict the arrival delay of a plan which will not reach its destination. We further remove the rows which have an `NA` value for `ArrDelay`, as we cannot use them for training.

The next step was to create histograms for the variables to get a better understanding of their distributions. Figure 1 shows that the number of flights are relatively evenly distributed throughout the year. Figure 2 shows a peculiar pattern, indicating that the values are in format HHMM, which we convert to "minutes after midnight" via a user-defined function. Figure 3 shows that `CRSElapsedTime`, `ArrDelay`, and `DepDelay` roughtly follow a Poisson distribution. The datatype of these variables is further cast to Integer, previously they were of datatype String.

We further looked at the pairwise correlations between the variables, as shown in Figure 4. From the correlation matrix, we can draw some conclusions:

1. There is a very high correlation (0.93) between the departure delay and the arrival delay (the variable we want to predict), making linear regression a good first model.
2. There is also a very high correlation (0.97) between the time planned for the flight (`CRSElapsedTime`) and the distance, which makes intuitive sense. To remove redundancy in our features, we remove the distance feature.

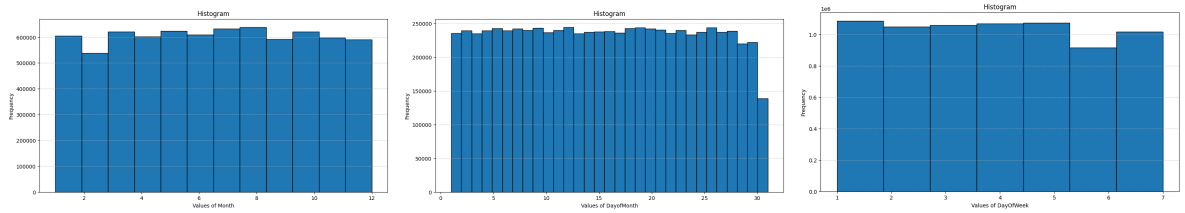

Figure 1: The histograms show that the number of flights are evenly distributed over the months, days of the month, and the days of the week.

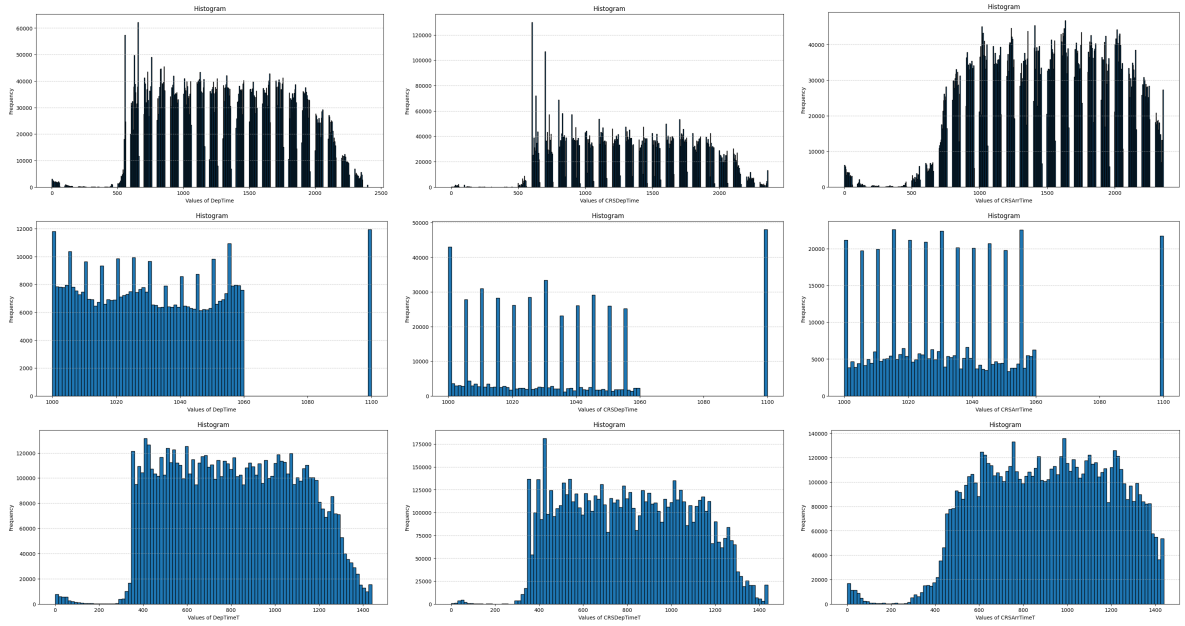

Figure 2: The first row of histograms depicts the total distribution of flights over the whole day, and the second row zoomed into the interval between 1000 and 1100. We see that the values of variables representing times are in format HHMM. For better interpretability by our models, we convert them into the format "minutes after midnight".

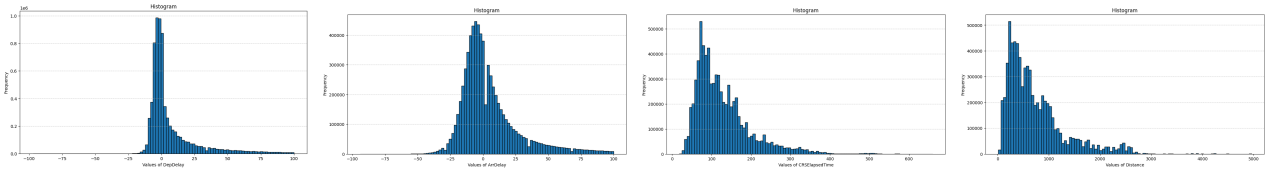

Figure 3: The distributions roughly follow a Poisson distribution.

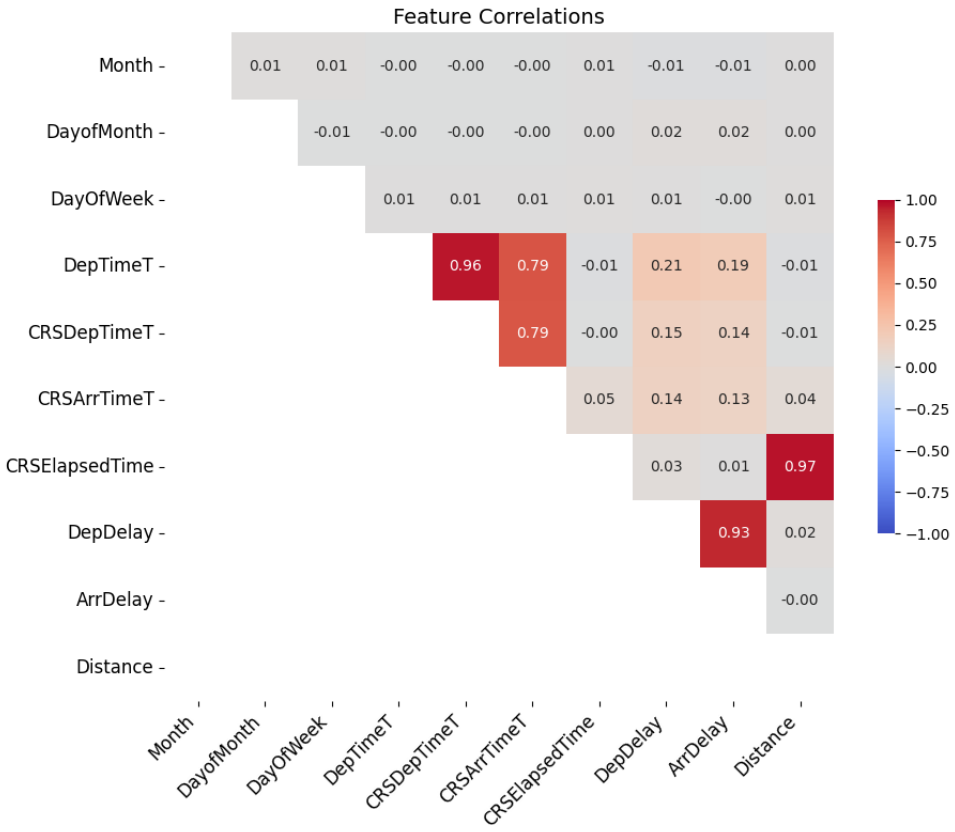

Figure 4:

Lastly, we took a look at the categorical variables by using kernel density estimation (KDE). We decided for a one-hot encoding for the carrier. As can be seen in Figure 5, two carriers stand out as being more punctual than the rest (the red and blue lines which are on top near an arrival delay of 0 minutes). We encode those as `PunctualCarrier` and the rest as `AverageCarrier`.

We decided against encoding the origin and destination airport, as there are 304 different possibilities, which would require advanced clustering to derive a reasonable one-hot encoding.

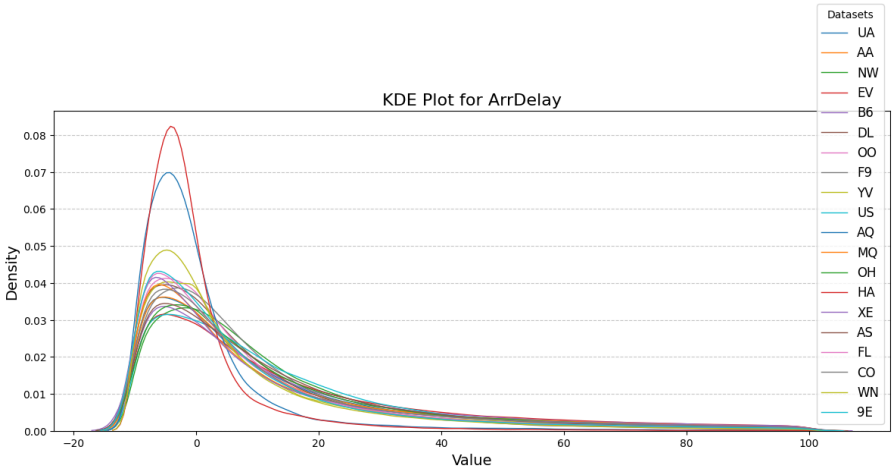

Figure 5:

The preprocessing leaves us with the following variables (all Integers), which we can use as inputs for our models: `Month`, `DayofMonth`, `CRSElapsedTime`, `ArrDelay`, `DepDelay`, `DepTimeT`, `CRSDepTimeT`, `CRSArrTimeT`, `PunctualCarrier`, and `AverageCarrier`.

3 Machine Learning Models

To perform the regression, we wanted to try out four well-known methods: Linear Regression, Decision Tree, Random Forest, and Gradient-Boosted Trees. All of these methods are independent of the scaling of the features.

Linear Regression predicts the dependent variable (`ArrDelay`) as a linear combination of the independent variables. We did not tune any hyperparameters.

A **Decision Tree (DT)** determines (nearly) optimal partitions of the data, represented as a feature and a threshold, in all the inner nodes. The leaf nodes determine the prediction. The hyperparameters we tuned are the maximum depth of the tree and the maximum number of leaves the DT may have.

The prediction of a **Random Forest (RF)** is a combination of multiple, less powerful DTs, combined via the method of bagging, which ensures diversity among the trees. The hyperparameters we tuned are the number of trees, as well as the maximum depth and the maximum number of leaves of each DT.

Gradient-Boosted Trees (GBT) iteratively train weak learners (DTs) on the errors of the previous learners. Sadly, in our experiments the grid search took more than 24 hours, which is why we aborted it and did not consider it further.

For the Linear Regression (LR), we used a Spark `Pipeline` to chain together all the operations necessary for model training. For the LR on the raw features, it is just the `VectorAssembler` and the `LinearRegression` model; for the LR on normalized features, we also include a `MinMaxScaler` for the uniformly distributed features and a `StandardScaler` for the features following a Poisson distribution.

Table 1 shows the optimized coefficients. Based on the LR using *raw features*, we can see how the prediction changes, if we change the feature by one unit. The coefficients indicate that a `DepDelay` of one minute corresponds to about one minute of `ArrDelay`. The rest of the features don't have a discernible impact. The coefficients of the Linear Regression with *normalized features* lets us judge the relative impact of the features. We see that `DepDelay` has by far the strongest influence on `ArrDelay`. (Which coincides with our findings from the correlation matrix, as both consider linear relationships.)

Table 1: Comparison of Raw and Normalized Features

Feature	**Raw Features**	**Normalized Features**
Intercept	-2.76	4453.68
Month	-0.03	-0.29
DayofMonth	0.01	0.20
DepTimeT	0.00	5.37
CRSDepTimeT	-0.01	-7.50
CRSArrTimeT	0.00	1.05
DepDelay	1.01	36.60
CRSElapsedTime	-0.01	-0.82
PunctualCarrier	3.85	-4442.56
AverageCarrier	3.47	-4442.94

For training the DTs and RFs, the code structure is exactly the same: We build a pipeline consisting of a `VectorAssembler`, the model (`DecisionTreeRegressor`/`RandomForestRegressor`), and the components needed for cross-validation (`ParamGridBuilder`, `RegressionEvaluator`, and `CrossValidator`). We set the number of folds to 3. After the whole training process, we

extract the best performing model and save it.

To evaluate our models, we compute three metrics: Root Mean Squared Error (**RMSE**), Mean Absolute Error (**MAE**), and the coefficient of determination ($\mathbf{R^2}$).

The RMSE is the *most common* evaluation metric of regression models, and it penalizes greater errors stronger. We use this metric for cross-validation, to compare the performances between our models and choose the best-performing one. The MAE is *more robust to outliers*. And the R^2 measures the *percentage of variance which can be explained* by the model.

Table 2 shows the results of our three models: LR, DT, and RF, as measured by the three metrics. Interestingly, the Linear Regression performs best. Our hypothesis is that the linear relationship between departure and arrival delay—which can be captured very well by LR, but less so by DTs and RFs—is so strong, that it cannot be made up by the additional complexity allowed by DTs and RFs.

Table 2: Performances of the models as measured by the metrics RMSE, MAE and R^2.

Metric	**Linear Regression**	**Decision Tree**	**Random Forest**
RMSE	14.2601	16.9432	14.4477
MAE	9.4471	10.5309	9.4782
R^2	0.8683	0.8141	0.8648

4 Spark Application

The Spark application `app.py` provides an easy and condensed way to process the data and obtain predictions.

It expects three command-line arguments: `$ spark.py <stages> <data_path> <model_dir>`.

- `stages`: `1` for training, `2` for evaluation and testing, or `12` for both.
- `data_path`: Path to train/test data file.
- `model_dir`: Directory where the models should be stored/loaded from. One subdirectory needed per model.

E.g. `$ spark-submit --master local[*] app.py 12 ./data/2007.csv.bz2 ./models` will train and evaluate a linear regression model on the data of `2007.csv.bz2`, with an 80/20 split.

It performs exactly the same preprocessing steps as given in `Preparation.ipynb`, only without producing graphics in between. In the training phase, the script will only train a linear regression model, as this was determined by our evaluation to be the strongest one. If the user has already trained models (e.g. with `Models.ipynb`), they can place the trained models in the `model_dir`, and the models will also be considered in the testing phase.

One peculiarity to be considered is that our script also requires the testing data to have a `ArrDelay` column, as it is expected later in the code. It should further have non-`NA` values, e.g. all zeros (as it would be dropped in the preprocessing steps otherwise).

5 Conclusion

The project deepened our understanding of how to prototype and develop robust applications with Pyspark. Some further features would have been intriguing to implement, but they were out of scope. These features include more sophisticated logging in the script, considering even more parameters in the grid search, training on a larger subset of the dataset, etc. The lack of computing resources on our local machines also played a role, as we otherwise could have experimented with larger grid searches and training Gradient-Boosted Trees.