Flights Arrival Delay Prediction

Big Data: Spark Practical Work

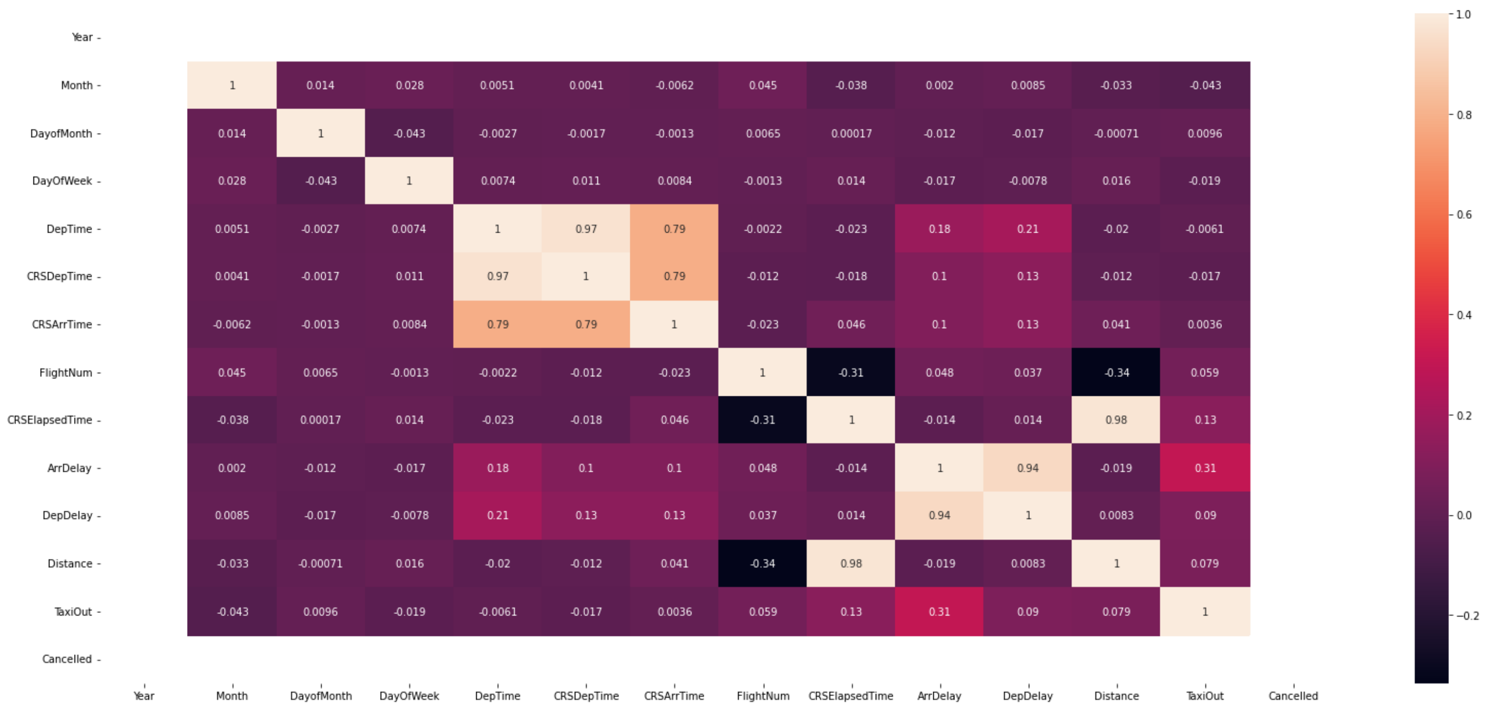
Alberto Megina Gonzalo
Jaime Guerrero Carrasco
Andrés Ramos Colombo

INDEX

1. Variables selected for the model

For selecting the variables to use we decided to use the correlation matrix to perform a bivariate analysis where we mainly focused on studying the correlation of each variable with "ArrDelay" but we also had to take care of possible multicollinearity effects that might exist between the independent variables.

To obtain the correlation matrix we first tried to make the application itself in the CLI during execution, however we found difficulties to display the matrix on the screen in a visually appropriate way. Therefore, the solution has been create a python notebook (.ipynb) independent to the Spark project (available in the folder “/selectingVariables” of the project) where making use of libraries such as pandas or seaborn we managed to print the following correlation matrix:



	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	CRSArrTime	FlightNum	CRSElapsedTime	ArrDelay	DepDelay	Distance	TaxiOut
Month	1	0.014	0.028	0.0051	0.0041	-0.0062	0.045	-0.038	0.002	0.0085	-0.033	-0.043
DayofMonth	0.014	1	-0.043	-0.0027	-0.0017	-0.0013	0.0065	0.00017	-0.012	-0.017	-0.00071	0.0096
DayOfWeek	0.028	-0.043	1	0.0074	0.011	0.0084	-0.0013	0.014	-0.017	-0.0078	0.016	-0.019
DepTime	0.0051	-0.0027	0.0074	1	0.97	0.79	-0.0022	-0.023	0.18	0.21	-0.02	-0.0061
CRSDepTime	0.0041	-0.0017	0.011	0.97	1	0.79	-0.012	-0.018	0.1	0.13	-0.012	-0.017
CRSArrTime	-0.0062	-0.0013	0.0084	0.79	0.79	1	-0.023	0.046	0.1	0.13	0.041	0.0036
FlightNum	0.045	0.0065	-0.0013	-0.0022	-0.012	-0.023	1	-0.31	0.048	0.037	-0.34	0.059
CRSElapsedTime	-0.038	0.00017	0.014	-0.023	-0.018	0.046	-0.31	1	-0.014	0.014	0.98	0.13
ArrDelay	0.002	-0.012	-0.017	0.18	0.1	0.1	0.048	-0.014	1	0.94	-0.019	0.31
DepDelay	0.0085	-0.017	-0.0078	0.21	0.13	0.13	0.037	0.014	0.94	1	0.0083	0.09
Distance	-0.033	-0.00071	0.016	-0.02	-0.012	0.041	-0.34	0.98	-0.019	0.0083	1	0.079
TaxiOut	-0.043	0.0096	-0.019	-0.0061	-0.017	0.0036	0.059	0.13	0.31	0.09	0.079	1

Focusing on the `"ArrDelay"` column, these are the variables we have selected:

- **`DepDelay`:** is the variable with the highest correlation with 0.94. This makes a lot of sense because if we think about it, the delay in the departure of the flight will affect the delay in the arrival more than any other cause such as the flight time that could be estimated beforehand as the distance between speeds.

- **`DepTime`:** having the `"DepDelay"` time and one of these two variables [`"DepTime"` or `"CRSDepTime"`], with one of these two and `"DepDelay"` you can calculate the other and vice versa, so we could eliminate one of these two and keep the other one in the model. Both have a similar correlation value but we are going to keep "`DepTime`" simply because its value is a little higher than "`CRSDepTime`".

- **`TaxiOut`**: is the second variable with the highest correlation with 0.31. Therefore, this should also be a fixed variable in the model. The problem with this variable is that, mostly in datasets of older years, we do not have data about "`TaxiOut`". So, we can expect better accuracy values in those models that use datasets of more modern years.

- **`FlightNum`**: with a value of 0.048 we can not expect for this variable to be the most relevant one in the model but it can help to improve the accuracy of the model.

- **`CRSArrTime`**: has a value of 0.1 so it can be safely included in the model.

- **`Origin`**, **`Dest`**: are categorical variables that we have thought may influence delay in the way that perhaps flights departing from X airport are more likely to be delayed or flights arriving at X airport are more likely to be delayed. This would help airports to know whether they should refine the way they organize arrivals and departures.

- **`CRSElapsedTime`**: does not have a high correlation value, but the scheduled elapsed time we believe is a factor that can affect arrival delay.

- **`Distance`**: does not have a high correlation value, but the distance we believe is a factor that can affect arrival delay.

- **`ArrDelay`**: we have to keep it because we have to pass it to the model to indicate that it is the variable we want to predict in the model.

2. Variables excluded

In addition to the explanation corresponding to each excluded variable, it is worth mentioning that all of them also have a very small correlation value according to the correlation matrix in the previous section.

- **`Cancelled`**, **`CancellationCode`**: although it is true that a cancellation could be considered a delay, in this project we are trying to focus on the delays of all those flights that take off under ‘normal’ conditions.

- **`CRSDepTime`**: from the explanation given in “`DepTime`”, we had to choose between one or the other and we chose “`DepTime`” because it had a slightly higher correlation value.

- **`UniqueCarrier`, `TailNum`:** we thought they are not relevant factors that could cause delays in the arrival.
- **`Year, Month, DayofMonth, DayOfWeek`:** for the variables related with dates, at first we thought they might be interesting as there could be more delays at Christmas, on holidays, in the summer or on weekends, etc. But we thought about it carefully and came to the conclusion that all these factors "have already been taken into account" in the numerical value of `"DepDelay"`. Because of this and the fact that they have a very low correlation value, we decided to exclude them.

3. Variable transformations performed

When loading the dataframe all the variables appear as *string*, therefore we must transform all of them to numeric type in order to be able to use them in the prediction models.

Categorical variables will be transformed to *integer* type and quantitative variables will be transformed to *double* type to differentiate them.

- Categorical:
    - **`FlightNum`:** transformed into *integer*.
    - **`Origin`, `Dest`:** we can not introduce non-numerical variables in the model so we must transform them first. The problem is they are letter characters only so they cannot be transformed in the conventional way. It will be necessary to apply an *indexer* that will be explained in the following section, because two new variables come out of this transformation.
- Quantitative:
    - **`DepTime, CRSArrTime, CRSElapsedTime, DepDelay, Distance, TaxiOut:`** transformed into *double*.
    - **`ArrDelay`:** apart from being transformed into a *double* type like the others, the name of this variable will be replaced, so the variable `"ArrDelay"` will no longer exist and we will have `"label"` instead. The reasons are explained in the following section.

In both cases, missing values will be filled with 0's.

4. New variables created

- **`Origin`, `Dest`:** we have made an *indexer* that receives as input the columns `"Origin"` and `"Dest"` and returns `"OriginIdx"` and `"DestIdx"` in the output. To avoid missing values, the *indexer* will skip the null values. Finally, the *indexer* will be applied later in the pipeline, not at the time it is defined.

- **`label`:** the target column to predict must be named "label" to make the cross validation work, so we transformed “`ArrDelay`” into ”`label`”.

After all transformations and creation of new variables we have the following schema:

Name	Type	Nullable (has null values?)	Considered categorical?
DepTime	*double*	false	no
CRSArrTime	*double*	false	no
FlightNum	*integer*	false	yes
CRSElapsedTime	*double*	false	no
DepDelay	*double*	false	no
OriginIdx	*integer*	false	yes
DestIdx	*integer*	false	yes
Distance	*double*	false	no
TaxiOut	*double*	false	no
label	*double*	false	no

5. Machine learning techniques selected

To solve this problem we have decided to use three different machine learning models by testing different approaches and then comparing them.

The first model is a **Linear Regression**, that creates a linear relationship between all features and the ArrDelay target variable. The result model will have assigned coefficients to each of the independent variables representing the change in the response variable for each unit change in the predictor variable. We decided to select it because it is a simple and interpretable model that could help us acquire a better understanding of the problem at the beginning.

The main parameters of this model are:

- **`regParam`**: controls the complexity of the model by adding a penalty term to the loss function
- **`elasticNetParam`**: determines the balance between Lasso and Ridge regularization
- **`fitIntercept`**: is a boolean parameter that determines whether the model should use the intercept term or not.
- **`maxIter`**: maximum number of iterations that the optimization algorithm should run

The second model selected is a **Decision Tree Regressor**. In this case, a set of rules is learned from the model to split the data in each step based on the features values. Now we no longer have a linear relationship and more importantly, categorical variables are considered. These are the main reasons to try out this model as we can test the validity of the categorical features 'FlightNum', 'OriginIdx' and 'DestIdx'.

Main parameters of the model:

- **`maxBins`**: the maximum number of bins that the continuous features should be divided into when finding the splits
- **`minInstancesPerNode`**: the minimum number of observations that must be present at a node in order for the node to be split
- **`maxDepth`**: the maximum depth of the tree, which determines the complexity of the model.

One important point is that the maxBins parameter must be at least as large as the number of distinct values in a categorical variable so we compute this.

The last tested model is the **Random Forest Regressor**. Once we had the Decision Tree Regressor we decided to try this model to ensure we were not overfitting the training dataset and also because this type of model tends to be more accurate and stable in the results as it combines the output results of several trees.

The most important parameters are the same as in the previous model, however the numTrees parameter is added which controls the total number of trees that will be created.

6. Validation process

All the models have been trained and validated using a cross validation approach. In order to do so, each model was included into a pipeline together with the indexer and assembler. The latter is needed to make input columns data selected to be in the form of a single feature vector. Also a parameter grid was provided to try different combinations of hyperparameters to find the optimal one.

Regarding the use of the dataset, initially it is divided in two subsets: train and test. While training the models, it is used the train dataset that will be divided into 3 folds to proceed with the cross validation method (using 2 of them to actually train and the other to validate). Finally, once we have the best model fitted, we evaluate it with the test dataset extracting two accuracy metrics that are R^2 and RMSE.

While the first two models will test combinations of their most important parameters as stated, the Random Forest Regressor will use the optimal parameters of the Decision Tree Regressor for each of its individual trees as we thought that repeating the search for these parameters was not really useful .

7. Final evaluation of the prediction model

Once the models are validated we obtain the R^2 and RMSE metrics so we can compare their performance among them. In our test done with a subset of the total dataset, both the Decision Tree and Random Forest seem to deliver really similar results. The Linear Regression returns the best values for both metrics so this model will be the one selected in the end to solve this regression problem.

8. Instructions on how to compile and execute the application

The first requirement to compile and execute the application is to have sbt installed and to be able to use the "sbt" command in the Command Line Interface. Here you can find a tutorial of how to install sbt: sbt_instalation_guide, although you can use any other available on the Internet.

Once sbt is installed, the next step is to open the terminal and move to the project's main folder called “BigDataFlights”. At this point we should be able to see an architecture like: “/src/main/scala/Main.scala”.

Now we can compile the application executing the command: `sbt compile`

When the compilation is done, we are ready to run the application. At this point we have to choose between executing with or without arguments.

- **Executing with arguments:**
  - **Command:** `sbt "run path_to_the_directory"`
  - **Explanation:** The application will search for the csv files in the directory that we have indicated in the arguments. `path_to_the_directory` must be replaced with your real path.
  - NOTICE that the quotation marks (“ ”) are important for the correct execution of the command.
- **Executing without arguments:**
  - **Command:** `sbt run`
  - **Explanation:** The application will search for the csv files in the current directory where we are running the application.

Alternatively, we can execute these commands from within the sbt console. To do this the first step would be to execute:

```
sbt
```

Now we would be in the sbt console and we can directly execute:

1. `build`
2. `run` **or** `run path_to_the_directory`

9. Instructions on where to place the input data

All input data must be included in a single folder, containing all the .csv files that form the dataset. Also its path would be provided to the application by an argument.

However, these files can also be placed in the main folder of the project “BigDataFlights” and the application will gather the files from this location if no argument is passed.

In both types, we handle the process of reading covering us against failures. To do so, we check if the path of the data directory is valid and it is not empty. Additionally, only the csv files will be selected. If some of these requirements are not met we avoid creating an empty dataframe and an error will be prompted to indicate that it was not possible to correctly read the files.

In the case of execution without input arguments, a warning will appear on the screen warning that the current directory from where the application is being executed will be used to search for the files. And also, regardless of whether it is executed with or without input arguments, the application displays all the files that will be used to build the dataframe.

10. Final conclusions and personal remarks

In conclusion, this project has demonstrated the importance of careful and effective planning while dealing with large datasets to solve a real regression problem. By utilizing Spark and its various modules, we were able to effectively handle this data and make use of machine learning models to reach a solution.

As a personal remark, working on this project has allowed us to deepen our understanding of the power and potential of BigData technologies, particularly in how to handle a real Data Science project. It has also given us the opportunity to further develop our skills in using Spark and Scala, which without doubt will be valuable in our future. Overall, this has been a highly educational and rewarding experience.