“Formula 1 Podium Classifier: a PySpark application”

by

Fabio Campus

Università degli studi di Cagliari - facoltà di Scienze Economiche, Giuridiche e Politiche
corso di Laurea Magistrale in Data Science, Business Analytics e Innovazione

Abstract

The project aims to perform a classification task on Formula 1 data about drivers, circuits, constructors, races, qualifications and results, in order to predict to finish on podium.
To gather data for the study, different Kaggle csv files were merged together by using their foreign keys – ID_features – and then three machine learning models were used: Logistic Regression, Random Forest Classifier, Linear Support Vector Machine.
Among them, the Logistic Regression was chosen as the favorite model, thanks to its performances in prediction and inference (as good as others models’ performances) and its computational time (1/2 respect to SVC, 1/4 respect to Random Forest Classifier).

CONTENTS

1. Dataset Selection

For the project, a dataset from *Kaggle.com* containing Formula 1 race data from 1994 to 2023 was utilized. Since the dataset was divided into smaller ones – circuits.csv, constructors.csv, drivers.csv, qualifying.csv, races.csv, results.csv – some merge operations were needed, in order to get the full dataset. Thanks to this process, it was possible to easily access a wealth of information, organized into 9806 rows and 56 features. In particular: raceId, year, round, circuitId, race_name, date, race_url, fp1_date, fp1_time, fp2_date, fp2_time, fp3_date, fp3_time, quali_date, quali_time, sprint_date, sprint_time, circuitRef, circuit_name, location, country, lat, lng, alt, circuit_url, resultId, driverId, constructorId, grid, race_position_order, points, laps, fastestLap, rank, fastestLapTime, fastestLapSpeed, statusId, driverRef, driver_number, code, forename, surname, dob, driver_nationality, driver_url, status, constructorRef, constructor_name, constructor_nationality, constructor_url, qualifyId, number, quali_position, q1, q2, q3. All of this was saved in the *f1_updated.csv* file.

2. Data Cleaning

2.1. (preliminary) Feature selection

After uploading *f1_updated.csv* with the *read* function of PySpark, it was considered to drop some features – thanks to the *drop* function. The reasons behind such decision were different: redundant information (*lat & lng, location, constructorRef, code, driverRef, circuitRef*), features entirely composed by unique values (ID features), useless information (URL features, driver's date of birthday *dob*, *fp1_date, fp2_date, fp3_date, quali_date, sprint_date*), race information that are considered posterior to the object of our analysis (*points*, *status, rank, fastestLapSpeed, fastestLapTime, fastestLap*), features that have one-to-one correspondence with other features (*constructor_name, circuit_name, race_name, forename, surname*). However, after exploring the dataset, it was discovered some features contained many "\N" values. It was decided to manage such values by dropping features that presented more than 30% of their values under that format, while replacing it with the median value (relative to that column) for features where the percentage was below 30. In order to do that, PySpark functions such as *col* and *when* were used.

2.2. Managing feature type

Another needed check was about feature type. The feature *q1* – representing the driver time in the first segment of the qualifying session – presented a quite incomprehensible format (min:sec.millsec), so it was decided to convert such values to milliseconds (*int* type) – the function *split* was very useful for this specific task, as well as *col* and *when* – PySpark framework. Also the feature *alt* (circuit altitude) needed to be converted to type *int.* Feature *date* (race date) was of type *string* and we decided to keep only the month information. Lastly, the response variable (*race_position_order*), representing the final driver position, was used to generate a new column (*top_3*) composed by only two values: *yes* and *no*, depending on whether the driver finishes in the top 3 or not. This last operation, enabled the task to be a binary classification one.

2.3. Feature dictionary

The final used features, and their respective meaning, will be presented below.

- *Year*: race year;
- *Country*: circuit country;
- *Alt*: circuit altitude;

- *Grid*: starting race position;
- *Laps*: circuit laps;
- *Driver_nationality*: nationality of the driver;
- *Constructor_nationality*: nationality of the constructor (since there was one-to-one correspondence with *constructor_name* and *constructorRef,* it was chosen to keep the nationality because of the ongoing information it contains, rather than the name or reference that might finish out of the game with years – e.g. Benetton);
- *Number*: driver number;
- *Q1_millsec*: q1 time in milliseconds;
- *Race_month*: month of the race
- *Top_3* (response variable): finished on podium or not.

The cleaned dataset was saved in the *f1_cleaned.csv* file – using *Pandas*, since *PySpark* returned the following error:

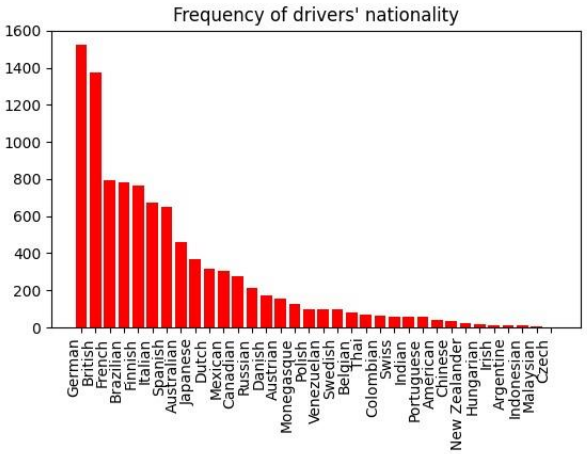
```
ERROR FileFormatWriter: Aborting job 064112a8-cbf1-4d9f-8011-26e2d5dce865.
```

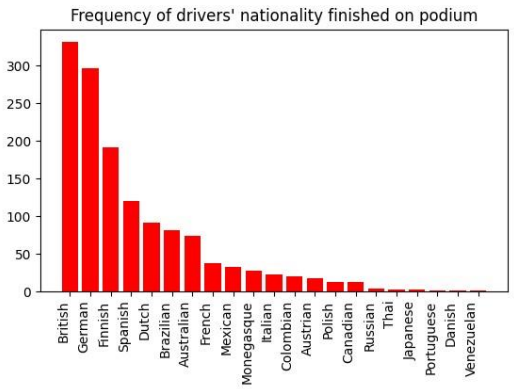
All the above-mentioned operations are present in the *data_cleaning.py* file.

3. Data Visualization

The code for graphical representations is located in the *data_visualization.py* file, where *f1_cleaned.csv* was uploaded before getting insights. Thanks to such graphics, ideas about classes distribution and preliminary in-depth insights into the importance for the response variable can be made. For this purpose, 4 barplots were generated using Matplotlib.

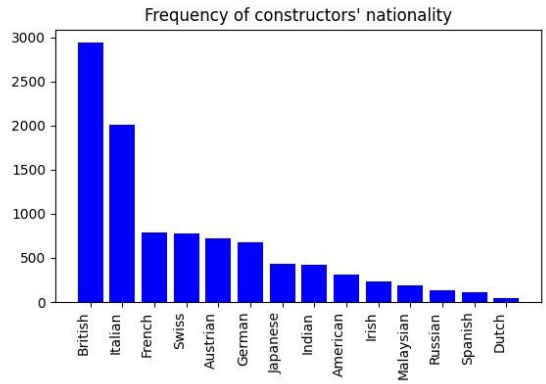
The first 2 barplots explore pattern about the feature *driver_nationality*, in particular:

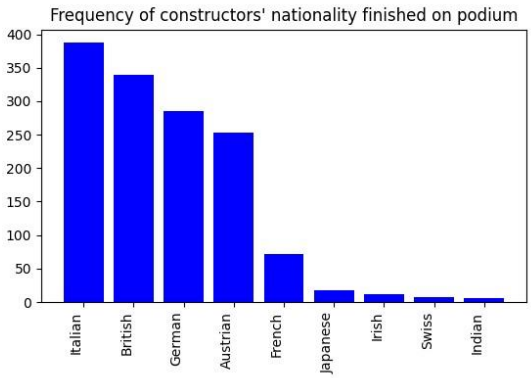




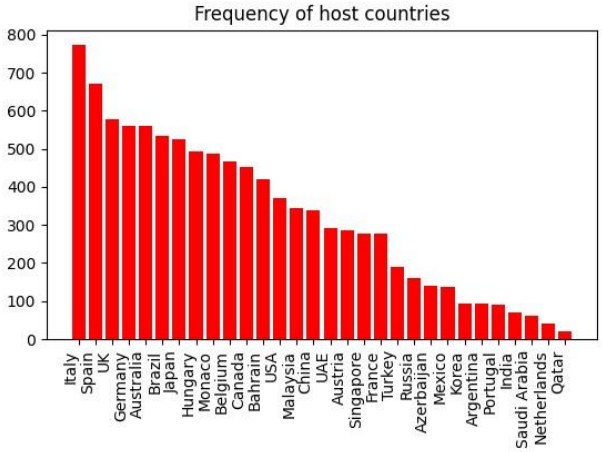
It is shown that *German* and *British* are the two most frequent drivers' nationalities, both in races where they took part and in races where they finished on podium. About this last pattern, it is possible to say that, since the two mentioned classes are much more frequent than others, they could play an important role for the machine learning classifier.

2 barplots about *constructor_nationality* were generated as well.





It is shown that *Italian* and *British* are the two most frequent constructors' nationalities, both in races where they took part and in races where they finished on podium. About this last pattern, it is possible to say that, since the frequencies of the two nationalities finished on podium aren't much higher than others (*German* and Austrian), such two classes could have a less important role compared with that of the two *driver_nationality*'s most frequent classes.



The last barplot explores frequency of countries that hosted a race.

4. Data Transformation

In the file *scaler_encoder.py*, there are two classes: *RobustColumnScaler* and *OHencoder.*

4.1. Class RobustColumnScaler

In order to scale quantitative columns from the dataset, a class that performs *robust scaling* was created. This class enables to scale directly the concerned features, once created the instance of the class and called its method – *scale*. Such method works by calling other methods, in particular: *find_quantitative_columns* (enables to select only quantitative columns excluding the response

variable), *assemble_features* (assembles the selected features via *VectorAssembler* class from PySpark, since the PySpark's RobustScaler needs an assembled vector), *scale_columns* (performs the *RobustScaler* class on the assembled features), *select_features* (generates dataset with scaled columns, qualitative ones and response variable).

It was used the Robust Scaler because of its ability to process outliers, since underlying formula is:

$$Scaled\ Value = \frac{Original\ Value\ -\ Input's\ Median}{Input's\ IQR}$$

4.2. Class OHencoder

Since all the qualitative columns were nominal, in order to encode their values, One-Hot Encoder was used. Following the same scheme of the previous class, a new class called *OHencoder* that enables to encode directly qualitative columns – once created an instance and called its method – was created. Here, the methods called by the main *encode* method are: *find_string_columns* (enables to select only qualitative columns excluding the response variable), *encode_columns* (two stages: indexing the selected columns via *StringIndexer* class from PySpark, performing *OneHotEncoder* class from PySpark on such columns. These two stages were passed to a PySpark *Pipeline*), *select_columns_to_keep* (generates dataset with encoded columns, scaled ones and response variable).

5. Data Modeling

Before exploring the models used for predictions, a multicollinearity – i.e. a statistical phenomenon that occurs when two or more independent and quantitative variables are highly correlated with each other – analysis was performed. In order to understand which (scaled and encoded) features had the highest (absolute) value, a correlation matrix was generated, using the PySpark *Correlation* class. After sorting the values in descendent order, the 10 highest values were printed, understanding that there was no multicollinearity. Such operations were performed after uploading *f1_cleaned.csv*, and are present in *data_correlation.py.*

5.1. Logistic Regression

The first tested machine learning model was the Logistic Regression, in the *model_logit.py* file. After importing *f1_cleaned.csv*, scaler and encoder instances were created and their methods called – importing classes *RobustColumnScaler* and *OHencoder* from the py file *scaler_encoder* – in order to have scaled and encoded features. Before defining the model, three operations were needed: indexing the target variable (via PySpark's *StringIndexer*); assembling the independent variables (via *VectorAssembler* from PySpark); splitting the dataset into training set and test set (to do that, PySpark's method *randomSplit* was used, setting proportions as 0.7 and 0.3). By using *LogisticRegression* class of PySpark, a model named *logit* was created (passing the assembled features and the indexed target variable). It was decided to evaluate the model by appealing to the cross-validation, so two more elements were needed: evaluator and grid. The first one was created using to the *MulticlassClassificationEvaluator* class of PySpark, setting *metricName* to "accuracy". PySpark enabled to define a grid for the concerned parameters, simply using *ParamGridBuilder* class. In particular, two parameters were passed: *regParam* (it specifies the regularization parameter for logistic regression, helping to control overfitting by adding a penalty term to the loss function during model training) and *elasticNetParam* (it specifies the mixing parameter for Elastic Net Regularization, which is a combination of L1 – Lasso – and L2 – Ridge –

regularization techniques). For *regParam*, three values were chosen: 0.01 that represents almost no regularization and carries overfitting risk (the logistic regression model would tend to be more flexible, potentially allowing it to capture complex relationships in the data); 0.1 that indicates a moderate level of regularization, striking a balance between bias and variance in the model; 1 that represents relatively strong regularization, imposing a significant penalty on the model's coefficients during training, and leading the logistic regression model to be more constrained and simpler. For *elasticNetParam*, three values were selected: 0.0 that corresponds to pure Ridge (it adds a penalty term to the loss function that is proportional to the square of the coefficients, shrinking them towards zero – not necessarily – thus reducing the model's complexity and variance); 1.0 that corresponds to pure Lasso (it adds a penalty term to the loss function that is proportional to the absolute value of the coefficients, eventually driving some of them to exactly zero – i.e. effectively performing feature selection); 0.5 that corresponds to a situation where both Ridge and Lasso penalties are applied to the model's coefficients, with equal weight given to each (it provides a balanced approach to regularization, suitable for situations where both feature selection and coefficient stability are desired). After that, an instance of the PySpark's *CrossValidator* was created, using as *estimator* the *logit* model previously created, as *estimatorParamMaps* the grid built before, as *evaluator* the accuracy-evaluator previously generated and setting *numFolds* equals to 5. It was finally possible to fit the model on the training set following the working of *CrossValidator* (it begins by splitting the dataset – *train_data* – into a set of 5 folds which are used as separate training and test datasets. To evaluate a particular *ParamMap*, it computes the average evaluation metric – *accuracy* – for the 5 models produced by fitting the *estimator* on the 5 different dataset pairs. After identifying the best *ParamMap*, it finally re-fits the *estimator* using the best *ParamMap* and the entire dataset) and to know the best combination of parameters – thanks to the *extractParamMap* method. By using the fitted model, predictions on the *test_data* were generated – via *transform* method – and saved in a variable called *predictions*. This variable was useful to evaluate the model using different metrics, such as: accuracy, precision, recall, F1-score (via *MulticlassClassificationEvaluator*) and ROC-AUC (via *BinaryClassificationEvaluator*), with their respective values showed in the table below.

MODEL	regParam	elasticNetParam	accuracy	precision	recall	F1-score	ROC-AUC
logit	0.01	0.5	0.912	0.905	0.912	0.904	0.929

Furthermore, by using the *coefficients* methods, it was possible to get an array of features' coefficients, and by recalling their names and sorting them in descendent order (absolute value), feature importance was performed. Logistic model found the features *q1_millsec* and *race_month* as the ones with greatest absolute values, with the second (positive) slightly higher than the first (negative). The important inverse relationship with *q1_millsec* comes from the fact that the lower the milliseconds the driver takes in q1, the higher the probability to qualify first, hence being first in grid and having higher probability to finish on podium.

5.2. Random Forest Classifier

The second tested machine learning model was the Random Forest Classifier, in the *model_rf.py* file. All the operations regarding feature preparation for the model were the exact ones performed for the Logistic Regression (previously indicated) – importing the dataset; calling *RobustColumnScaler* and *OHencoder* from the py file *scaler_encoder;* indexing the target variable; assembling the independent variables; splitting the dataset into training set and test set. After that, by using *RandomForestClassifier* class of PySpark, a model named *rf* was created (passing the assembled features and the indexed target variable). It was decided to evaluate the model by appealing to the cross-validation, so two more elements were needed: evaluator and grid. The first one was created using to the *MulticlassClassificationEvaluator* class of PySpark, setting *metricName* to "accuracy". PySpark enabled to define a grid for the concerned parameters, simply using *ParamGridBuilder* class.

In particular, two parameters were passed: *maxDepth* (it specifies maximum depth of the decision tree models within the random forest ensemble. By restricting the depth, the trees are less likely to capture noise in the data and are more likely to learn the underlying patterns) and *numTrees* (it specifies the number of decision trees to be built in the ensemble. More trees generally lead to better performance, but the improvement may diminish as the number of trees increases, and the computational cost also rises). For *maxDepth*, three values were chosen: 5, 10, 15; while for *numTrees*: 20, 50 and 100 were chosen. Then, an instance of the PySpark's *CrossValidator* was created, using as *estimator* the *rf* model previously created, as *estimatorParamMaps* the grid built before, as *evaluator* the accuracy-evaluator previously generated and setting *numFolds* equals to 5. It was finally possible to fit the model on the training set following the same working of *CrossValidator* explained in the previous paragraph, and to know the best combination of parameters – thanks to the *getOrDefault* method, after specifying both parameters. By using the fitted model, predictions on the *test_data* were generated – via *transform* method – and saved in a variable called *predictions*. This variable was useful to evaluate the model using different metrics, such as: accuracy, precision, recall, F1-score (via *MulticlassClassificationEvaluator*) and ROC-AUC (via *BinaryClassificationEvaluator*), with their respective values showed in the table below.

MODEL	maxDepth	numTrees	accuracy	precision	recall	F1-score	ROC-AUC
Random Forest Classifier	15	100	0.915	0.908	0.915	0.91	0.937

Lastly, it was possible to compute feature importance by simply using the *featureImportances* method of *randomForestClassifier* class. After recalling feature names and sorting them in descendent order by their feature importance absolute value, it was possible to identify that every features had a very small value, making it hard to figure out which feature played a crucial role for the model.

5.3. Support Vector Classifier

The last tested machine learning model was the Linear Support Vector Classifier, in the *model_svc.py* file. Also in this case, feature preparation for the model was the same performed for Logistic Regression and Random Forest Classifier – refer to paragraph 5.1. for a detailed analysis. By using *LinearSVC* class of PySpark, a model named *svc* was created (passing the assembled features and the indexed target variable). Also for this model, it was decided to evaluate it by appealing to the cross-validation, so evaluator and grid were needed – created by following the same reasoning adopted for the other two models. In particular, for the grid, two parameters were passed: *regParam* (it is the same parameter passed to the grid as part of the Logistic Regression model, as well as the values passed – refer to paragraph 5.1. for a detailed analysis) and *maxIter* (it specifies the maximum number of iterations that the optimization algorithm is allowed to perform during the model training, in order to maximize the separation between classes) – for this one, values 10, 50 and 100 were chosen. An instance of the PySpark's *CrossValidator* was created, using as *estimator* the *svc* model previously created, as *estimatorParamMaps* the grid built before, as *evaluator* the accuracy-evaluator previously generated and setting *numFolds* equals to 5. After that, it was possible to fit the model on the training set following the usual working of *CrossValidator* and to know the best combination of parameters – thanks to the *getMaxIter* and *getRegParam* methods. By using the fitted model, predictions on the *test_data* were generated – via *transform* method – and saved in a variable called *predictions*, which was used to evaluate the model with the same metrics computed for the *logit* and the *random forest classifier* – see the table below.

MODEL	maxIter	regParam	accuracy	precision	recall	F1-score	ROC-AUC
Support Vector Classifier	50	0.01	0.91	0.907	0.91	0.909	0.93

Features' coefficients were showed as well, by using the *coefficients* method. It was chosen to print them in descendent order, in order to catch the greatest absolute values – following the same scheme of paragraph 5.3. As already noticed in the *logit* case, also the SVC indicated *q1_millsec* (the inverse relationship represents what was already said for the Logistic Regression model) and *race_month* (positive relationship), as the features with highest absolute values.

6. Conclusion

The metrics and the computational time found for each model can be observed in the table below.

MODEL	accuracy	precision	recall	F1-score	ROC-AUC	time
Logistic Regression	0.912	0.905	0.912	0.904	0.929	1 min
Random Forest Classifier	0.915	0.908	0.915	0.91	0.937	4 min
Support Vector Classifier	0.91	0.907	0.91	0.909	0.93	2 min

By looking at the values above, it is possible to notice how similar they are. For this reason, in order to pick the favorite model, it was chosen to use computational time as crucial factor. Random Forest Classifier took twice the time of the SVC, and four times the time of the Logistic Regression – such proportions might change with the number of observations – and for that reason it was considered the least convenient among them. SVC and Logistic Regression present almost same values, as well as the coefficients importance – *q1_millsec* and *race_month* as the highest coefficients absolute values in both models – meaning that, for the information held, the preference goes to the Logistic Regression, thanks to its ability to perform as good as SVC, but taking half of its time (for the analyzed sample).