Alessio Bonatesta

08313A

Project of Algorithms for Massive Datasets and Statistichal Methods for Machine Learning

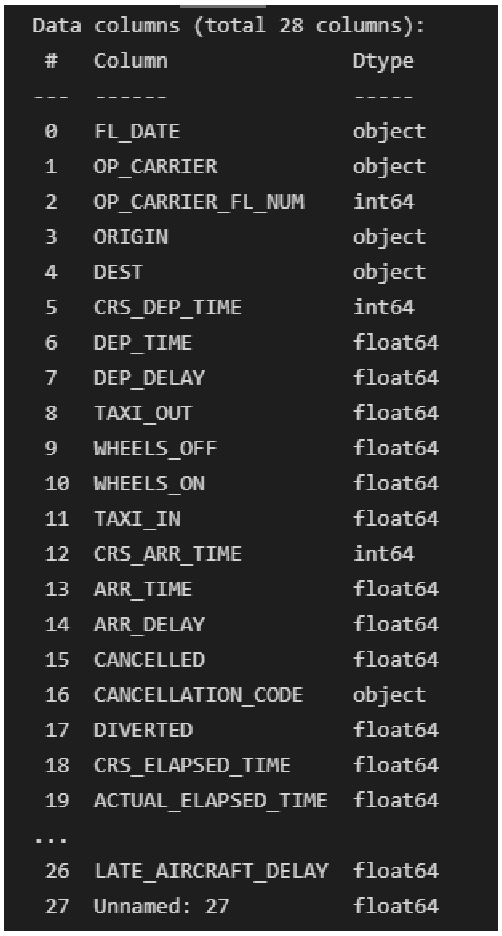
HOW TO BUILD A DISTRIBUTED LOGISTIC REGRESSION MODEL FROM SCRATCH TO PREDICT FLIGHTS CANCELLATION USING SPARK

1 PROBLEM DESCRIPTION

The aim of this project is to implement a logistic regression model from scratch whose training phase can be distributed among multiple clusters. The goal is therefore to create a classifier using the Apache Spark engine capable of predicting whether a flight will be canceled or not, only on the basis of information available at flight departure.

2 THE DATASET

The dataset considered for the construction of the model is the «Airline Delay and Cancellation Data, 2009 - 2018» dataset published on Kaggle. Initially, only the set of data referring to the year 2018 was taken into consideration, which was loaded using the Pandas library: Through the read_csv() function, as a matter of fact, it was possible to create a DataFrame containing 7213446 entries. The figure 1 shows the columns of the dataset as it was imported.



```
Data columns (total 28 columns):
 #   Column               Dtype
---  ------               -----
 0   FL_DATE              object
 1   OP_CARRIER           object
 2   OP_CARRIER_FL_NUM    int64
 3   ORIGIN               object
 4   DEST                 object
 5   CRS_DEP_TIME         int64
 6   DEP_TIME             float64
 7   DEP_DELAY            float64
 8   TAXI_OUT             float64
 9   WHEELS_OFF           float64
 10  WHEELS_ON            float64
 11  TAXI_IN              float64
 12  CRS_ARR_TIME         int64
 13  ARR_TIME             float64
 14  ARR_DELAY            float64
 15  CANCELLED            float64
 16  CANCELLATION_CODE    object
 17  DIVERTED             float64
 18  CRS_ELAPSED_TIME     float64
 19  ACTUAL_ELAPSED_TIME  float64
...
 26  LATE_AIRCRAFT_DELAY  float64
 27  Unnamed: 27          float64
```

Figure 1: Dataset columns .

Thanks to the Pandas' functions it was possible to carry out an initial analysis on the dataset. Underlining that a 0/1 label is associated with each entry, where 0 means that the flight has not been canceled and 1 means that the flight has been canceled, first it was found that the dataset was very unbalanced: the number of entries with label 1 was much lower than those with label 0 (only 116.584 flight canceled against the 7.096.862 not canceled). For this reason, a number of entries with label 0 approximately equal to the one with label 1 were randomly sampled. After that, it was decided to proceed by expanding the dataset including data from the set that refers to the year 2017.; by following the same steps described above on the new data and adding the extracted entries, it was possible to create a balanced dataset composed by 453.053 entries on which to train the model. The figure 2 shows the percentage of data with label 0 and label 1 before and after the balancing phase.

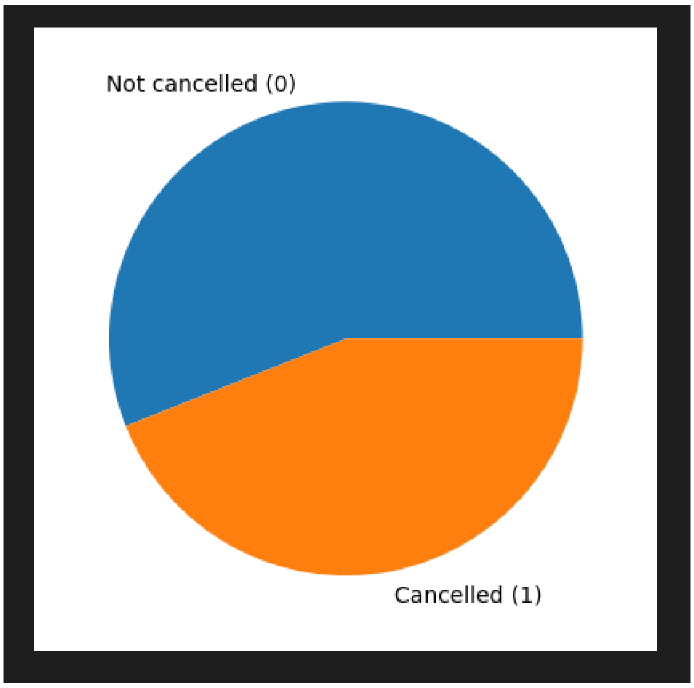
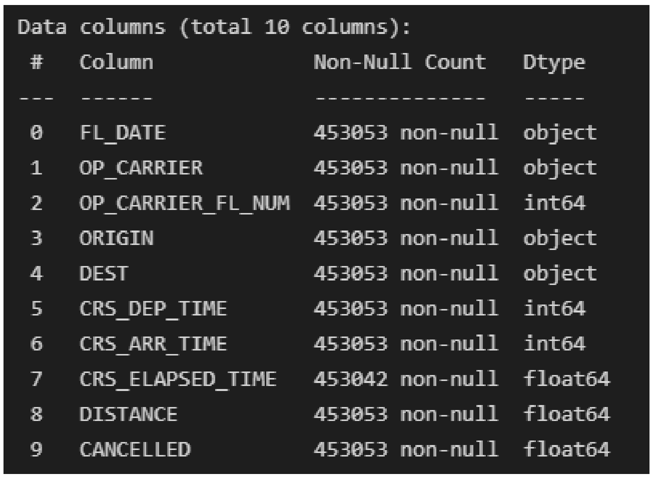


Figure 2: Points with label 0: 56.02% - Points with label 1: 43.98% .

I chose to only consider a small part of the entire dataset available for performance reasons of the computer on which the classifier has been trained. As we will see, however, the algorithm also lends itself well to processing much larger amounts of data since the data preprocessing and the learning algorithm have been implemented using Apache Spark, an engine capable of carrying out a large number of operations in parallel, distributing them among the available clusters. This means that having more processors available and changing some Spark configuration settings within this project (for example the number of RDD partitions), the model will remain stable and efficient even in front of huge amounts of data. As the model must be trained on the information available before the flight departure, the next step was to drop the columns of the table containing irrelevant information (data that refer to informations after flight's departure). With the help of the Pandas' drop() function, the dataset informations have been reduced as shown in figure 3.

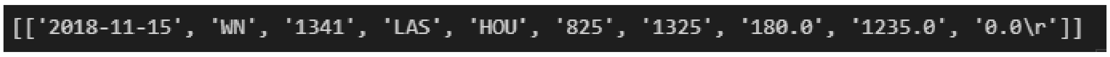


```
Data columns (total 10 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   FL_DATE            453053 non-null  object
 1   OP_CARRIER         453053 non-null  object
 2   OP_CARRIER_FL_NUM  453053 non-null  int64
 3   ORIGIN             453053 non-null  object
 4   DEST               453053 non-null  object
 5   CRS_DEP_TIME       453053 non-null  int64
 6   CRS_ARR_TIME       453053 non-null  int64
 7   CRS_ELAPSED_TIME   453042 non-null  float64
 8   DISTANCE           453053 non-null  float64
 9   CANCELLED          453053 non-null  float64
```

Figure 3: Dataset info after the dropping phase.

3 PREPROCESSING

Once that the dataset was more balanced and free of unnecessary information, it was created the Spark RDD and it was possible to work with the datas.
First of all, with a random split the dataset was splitted in a training test and a test set: from now on we will work only on the training set, putting aside the test set until the moment in which we will evaluate the trained and tuned model.
Initially every single point appears as a single string containing a series of values separated by a comma. For this reason, first the elements of the RDD were split on the comma in order to manipulate the single values, performing a mapping of the type $[string]->[list\ of\ strings]$ (as shown in figure 4).



```
[['2018-11-15', 'WN', '1341', 'LAS', 'HOU', '825', '1325', '180.0', '1235.0', '0.0\r']]
```

Figure 4: A point after the split operation.

3.1 Adding One Hot Encoding

The preprocessing phase continued by performing the encoding of the strings values (like "WN", "LAS","HOU") using one hot encoding. To do so, two functions were defined in order to create a dictionary that map each feature with a unique integer value. The first function maps each possible value of the OP_CARRIER column and the second one maps each pair (origin,destination); origin and destination are mapped together to make the model learn the importance of the couple and not of the individual airports taken separately. The two functions collect all existing distinct values of their respective columns and apply the function zipWithIndex() which executes a map of the type element -> (element,index); the collectAsMap() function transform the list of pairs in a dictionary. After defining these two functions, the addOHE() function took care of applying the OHE representation (adding into each point the list of One Hot Encoded feature computed using the dictionaries).

3.1.1 Other operations

The next step was creating a function in order to handle the values in the data column. This function maps any day into its index in the year (for example, 1st January will be transformed in 1, 2nd Jan in 2, and so on).The year has not been specified anymore in order to generally highlight the importance of the time of the year: we don't care if some flights have been cancelled in a specific year in the past, we only want to learn the relevance of the days in a year. The 'datetime' library has been used to create the dateMap() function passed as a parameter to the map operation. After that, since the remaining unmanipulated values still appeared as strings, the stringToFLoat() mapped these values to a float representation. As regards the label (the last dimension of each point), the choice was to substitute each 0 value with the number -1 for a mathematical purpose (we needed this to apply the gradient descent rule further on).

3.1.2 Sparse representation

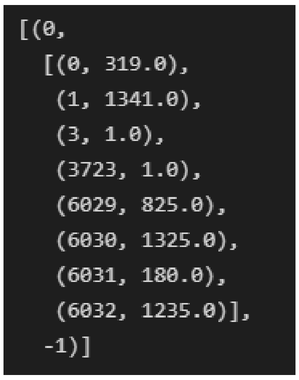
Since the One Hot Encoding part largely increased the number of feature of the vectors, each point has been mapped from a list of float values to a list of pair (index,value). Specifically, each point was zipped with its identifying index and then transformed into the form (index, X, y), where:

- X : list of pairs (index,values)
- y : its label

In order to do so the $sparsePoint()$ function has been defined, which takes a list of values and transform this list in a list of pairs (index, value), where the index is the position of the value in the list (in our case the "feature id "), mapping only the values different from 0. This function was applied to map each point in the RDD within its new representation.
Thus, it was possible to reduce the number of features of each point from more than 6000 to only 8. Using this representation method allows the model to possibly handle much larger amounts of data, where the One Hot Encoding representation would greatly increase the number of

features due to the increase of possible values to encode.

The figure 5 shows as an example the first element of the RDD after the operations performed so far.



```
[(0,
  [(0, 319.0),
   (1, 1341.0),
   (3, 1.0),
   (3723, 1.0),
   (6029, 825.0),
   (6030, 1325.0),
   (6031, 180.0),
   (6032, 1235.0)],
  -1)]
```

Figure 5: A point with its sparse representation

3.1.3 Data normalization

Finally, since the values of the features cover a very wide range, the data have been normalised to obtain better model results. The normalization method chosen is the z-score normalization, which means that after the normalization the mean of all the values in the same column will be 0 and the standard deviation 1. In order to do this, the average and standard deviation of each column have initially been calculated. As for the average, these are the operations to calculate it along each column of the dataset:

- map each point (index, X, y) to only X
- flatMap(): all the pairs (index, value) of X become single elements of the RDD
- reduceByKey(): sum up all the values with the same index associated (the index represent the column)
- map each point (index,sum) to (index, sum / N), where N is the number of points in the dataset
- collect the results and store them in a list 'meansVector', where where 'meansVector[i]' is the mean for the column at the index i

For what concerns the standard deviation:

- map each point (index, X, y) to only X
- flatMap(): all the pairs (index, value) of X become single elements of the RDD
- map each element (index, value) to (index, $(value - meansVector[index])^2$)
- reduceByKey(): sum up all the values with the same index associated (the index represent the column)
- map each point (index,sum) to (index, $\sqrt[2]{sum}/N$), where N is the number of points in the dataset
- collect the results and store them in a list 'deviatiosnVector',where deviationsVector[i] is the standard deviation for the column at the index i

The function zNormalize() take as arguments a sparse point, a list of means and a list standard deviations and gives as a result the point with its values normalized. Note from the code in the project that the normalization didn't affect the OHE values, which remained 1.

4 THE MODEL

Once the data has been prepared, it was possible to define the LogisticRegression class. The loss function used to train the model is the logarithmic loss:

$$l(y, \hat{y}) = I(y = +1) \log \frac{1}{\hat{y}} + I(y = -1) \log \frac{1}{1 - \hat{y}}$$

This function can be reduced to the logistic loss function:

$$\log(1 + e^{y\hat{y}})$$

where $\hat{y}$ is the predictor $g(x)$ trained and used in the sigmoid function. In our case $g(x)$ is a linear model $g(x) = w^{\intercal}x$ and so, after the addition of a regularization term and after calculating the derivative, the update rule for the vector w can be written as:

$$w_{t+1} = w_t + \eta_t(\sigma(-y_t w^{\intercal} x) y_t x_t - \lambda w)$$

In order to distribute the computation of the update terms for each point, the batch learning method was applied: this means that the term $\sigma(-y_t w^{\intercal} x) y_t x_t - \lambda w$ is computed for each point and all the results were summed up and divided by the number of points.

4.1 Model Initialization

Fisrt of all, the *init* function of the class initialize the parameter of the model:

- 'lr': learning rate for the update rule;
- 'maxIter' : maximum number of iterations if the model doesn't converge before to a local minimum;
- 'decayRate' : decay rate of the learning rate parameter
- 'regTerm' : regularization term added to the update rule for stability.

4.2 Fitting the model to data

The $fit()$ function is the one that deals with the learning of the model. Receveid a training set as input, in fact, it performs a series of operations to update its weights through the update

rule.
Remembering that an element of the RDD is a point in the form (index, X, y), first of all it finds out the number of the features involved by taking the first element of the RDD, taking the second object of this item (the X, a series of index-value), then the last pair (index, value) and finally the index of this element indicate the number of features.
The function, then, initializes the weights of the model to a numpy array of zeros long as much as the number of features and, after calculating the total number of points in the training set (by applying the $sum()$ function of Spark), starts the regression phase.
The regression phase consist in a for loop in which at each cycle:

1. the learning rate is decreased by applying the learning rate decay technique;

2. the update vestor is calculated by mapping each point in the respective derivative of the loss function (using the computeDew() function), summing up the results, dividing the total for the number of points and multiplying the result for the learning rate;

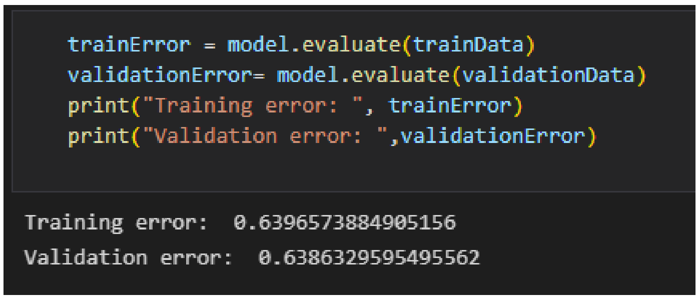
3. the for loop termination condition is checked: if the maximum value of the vector of the update terms is less than 0.0001 then the for terminates;

4. if the loop keeps going, the weights' vector is updated.

Note that the update term is a vector in which each component represents the updated value of the respective weight; in fact, $computeDex()$ is a function that returns as result a Numpy vector corresponding to the term $\sigma(-y_t w^\intercal x) y_t x_t - \lambda w$ (the signs are reversed because in the update rule the minus has been taken out of the parenthesis). Another important function defined is the *evaluate*() function, which takes as argument a set of points (index, X, y), computes the result of the loss function for each of them using the weights computed during the fitting phase and then returns the mean error.

4.3 First test

After defining these methods of the class, the model was trained on previously preprocessed data. The collection of points was split into two subsets: a training set (corresponding to the 80% of the points) and a validation set (the remaining 20%). The model was initially trained on the first set and evaluated on both; the figure 6 shows the results.

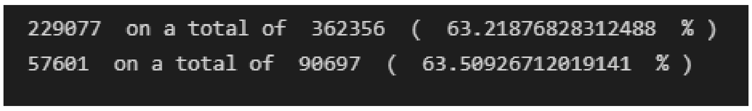


```
trainError = model.evaluate(trainData)
validationError= model.evaluate(validationData)
print("Training error: ", trainError)
print("Validation error: ",validationError)
```

```
Training error:  0.6396573884905156
Validation error:  0.6386329595495562
```

Figure 6: First test on the model

After that, the test checked how many predictions were correct on the total with the help of a function named *predict*(): this function computes the probability of a point belonging to one of the two classes and assigns to the point the label 1 if the calculated value is greater than the threshold, -1 otherwise; so, the mapping of this function is (X, y) -> (prediction, y). The figure 7 shows the results.



```
229077  on a total of  362356  (  63.21876828312488  % )
57601  on a total of  90697  (  63.50926712019141  % )
```

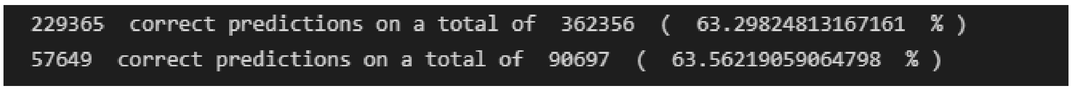
Figure 7: First test on the model. The first line shows the correct predictions made on the training set, the second line refers to the validation set.

As can be seen from the two images, the mean error and the mean of correct answers are about the same for the training set and the validation set.

4.4 Parameters tuning and second test

Following the first test, the parameter tuning activity was carried out in order to improve the performance of the model. This operation consisted of training 9 different predictors by combining 3 different values for the learning rate and 3 different values for the regularization term. This means that 2 nested for loop were defined in order to build a model each time with different parameters. After the training of these models, the one with best results was chosen. Although the nested cross validation algorithm was more suitable for tuning parameters since it returns more precise results, it was chosen to proceed as described above due to limited computational resources: it would be too computationally expensive for a personal computer with only 8 cores. However, as the results of the test show, a model with a learning rate equal to 120 and a regularization term equal to 0.0001 seems to be better then the others trained in terms of precision. Certainly an even better result could be obtained by looking for other parameters that optimize the model; the problem, as explained above, lies in the resources

available. With a greater amount of resources available, a more exhaustive search could be carried out by testing the behavior of the predictor with other parameters.
As you can see from figure 8 (which shows the percentage of correct predictions on the two sets), the trained model set with the parameters just found turned out to be more efficient (although by a very small percentage).



```
229365  correct predictions on a total of  362356  (  63.29824813167161  % )
57649  correct predictions on a total of  90697  (  63.56219059064798  % )
```

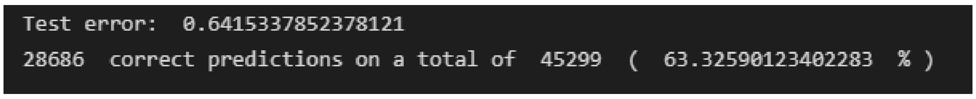
Figure 8: Test with the new parameters. The first line shows the correct predictions made on the training set, the second line refers to the validation set.

4.5 Tuning the treshold

A last simple test was carried out to try to improve the performance of the predictor by trying to change the threshold with which the *predict*() function assigns a class rather than the other. As the results of this test show, lowering the threshold from 0.5 to 0.48 we obtain a greater number of correct predictions.

5 EVALUATION ON THE TEST SET

After training the model and tuning the parameters and the prediction threshold, the model was evaluated on the test set initially extracted from the dataset. Obviously, before carrying out the evaluation, the test set was preprocessed following the same steps carried out for the training set: starting from the raw data, therefore, a normalized test set was obtained, represented in a sparse way.
After the preprocessing of this set, the last test was done: the *evaluate*() method and the *predict*() method of the model were executed on these data, producing the results in Figure 9. As can be seen, since the test error is really close to the training error, the phenomena of overfitting didn't occur.



```
Test error:  0.6415337852378121
28686  correct predictions on a total of  45299  (  63.32590123402283  % )
```

Figure 9: Results of the evaluation on the test set

6 COMMENTS

As we have seen, the development of this project has shown how it is possible to implement a logistic regression model from scratch using the Spark engine, thus allowing the learning algorithm to scale on much larger amounts of data than those treated during the development of this work. The results obtained are far from being acceptable for applications in real contexts. One of the hypothesized factors is precisely the amount of data taken into consideration. As already explained, since the project was developed on a personal computer equipped with a single processor with 8 cores, for obvious reasons of performance, the experiment was limited to considering a quantity of data evidently not sufficient to extract the features of interest in the learning phase. However, since the algorithm is designed to scale over larger data sets, it is thought that it is enough to change the initial settings of Spark and use a greater number of processors to make it possible to train on larger amounts of data: with more resources available, the partitions of the RDD could be increased to distribute the computation on more machines and train the model in an acceptable amount of time.
Another crucial factor deriving from the same problem was certainly the number of iterations that the model performs in the learning phase to update the values of the weights: also in this case the number of iterations was kept low for performance reasons, but as explained the problem would be solvable with more resources available and the results obtained would certainly be better.
Assuming another factor to the detriment of the results, probably being able to perform a more exhaustive tuning of the parameters (which, once again, was not possible due to the lack of adequate resources) would have allowed to obtain a more efficient model.
Problems aside, the importance of the work lies in having created a predictive model capable of scaling on large amounts of data thanks to the implementation of all its crucial phases (data preprocessing, learning and evaluation) through the use of the map-reduce paradigm and using the Spark engine to distribute and parallelize all the task.
Please note that all this talk about performance problems is even more significant if you run the project on Colab, where the available resources are even smaller than those on the computer on which the work was developed.