Airline Departure
Data Analysis and Regression

Lucchi Manuele & Tricella Davide

February 5, 2023

Instructors: Professor Cesa-Bianchi & Professor Malchiodi

We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.

Abstract

The purpose of this paper is to evaluate the usage of a Logistic Regression model on a airlines dataset to predict flight cancellation or diversion, in a scalable and time/space efficient implementation.

Contents

1 Definitions

Dataset The sample of data used to train the Model

Label The expected outcome of the prediction

Model The group of algorithms that tries to solve the problem

Overfitting When the model is too sensible to changes compared to the dataset

Vanishing Gradient When the gradient values becomes progressively smaller until they are insignificant for the process

2 Introduction

The following text details an analysis on the performances of a regression model on a flights dataset, both in a time/resources usage perspective and in the outcome precision one.
The computation will make use of Pyspark, a famous Python library for data manipulation on distributed environments, to test various distributed computation techniques and compare it to a serial approach. The text will instead compare the Logistic Regression model proposed with some state of the art implementations and even different approaches.
The document is structured in a describing part, where the operations on the dataset and how the model is built are detailed and an experimental part where a series of tests are performed and the results are stated.

Also, the document covers two different objectives, the classification of the dataset using the model to predict both the **canceled** and the **diverted** flights.
The two objectives are virtually the same, since the subset of the dataset used is identical and the model stays the same, so we could expect similar results.

3 Dataset

The initial dataset, "Airline Delay and Cancellation Data" [10] is made of 9 years of airlines flights data, composed by 10 files (one for each year from 2009 to 2018) with around 6 milions records each. The files presents 28 columns, of which only the 9 more relevant were took

FL_DATE The flight date.

OP_CARRIER The carrier code.

ORIGIN The departure airport.

DEST The destination airport.

CRS_DEP_TIME The planned departure time.

CRS_ARR_TIME The planned arrival time.

CANCELLED If the flight has been canceled.

DIVERTED If the flight has been diverted.

CRS_ELAPSED_TIME The planned total time of the flight, taxi time included.

DISTANCE The distance the flight has to cover.

The majority of columns have been excluded because contained information not available at departure time, like the ones regarding actual departure, flight and arrival time, which are at disposal only after the aircraft landed. Other columns also contained informations which do not have any correlation with the objective of the experiments, like the flight number assigned by the carrier.

In the case the prediction is about the cancellation, the `DIVERTED` column will be ignored, while if the prediction is on if the flight would be diverted or not, the `CANCELLED` column will be ignored.

The carrier code is a two characters alphanumeric code, the origin and destination places are a three characters alphanumeric code.
Flight date, departure time and arrival time are dates, while the elapsed time and the distance are real numbers.
Cancelled and diverted are either 0 or 1.

100,000 records sampled with an uniform distributed were took from each year file to perform the preprocessing. This number is due to the limited memory possessed by the machine on which the tests have been performed.

4 Preprocessing Techniques

Multiple preprocessing techniques were used.

4.1 Null values removal

The first operation consisted in the removal of the rows with null values, after counting them the conclusion was that the null values are really rare in the dataset, so it was possible to remove the rows entirely, without any significant loss of informations.

4.2 Data Analysis

After the null rows removal, some analysis have been performed using the charts of the MatPlotLib library, especially the bar charts.
The main objective of this analysis was to find an eventual relationship between the various features used and the probability that a flight would be cancelled or diverted.

None of the features showed significative patterns nor tendencies of the problematic flights to increase in relation to a particular value of a feature or a combination of features both the analysis of the dataset and the actual model costruction, probably because the dataset doesn't include informations on events that usually leads to this outcomes.

4.2.1 Problematic flights per carrier

The first chart drawn tried to correlate the carriers with an abnormal number of the cancelled flights, to make the chart more readable, every carrier with a percentage lower than the mean of all carriers was aggregated into a single column, to show only the highest percentages.

The result was a chart showing some carriers with a slightly-above-average number of cancelled flights, and a peak of 4% of the flights cancelled, but considering the huge number of distinct carriers in the dataset, the impact of this feature will probably be very low.

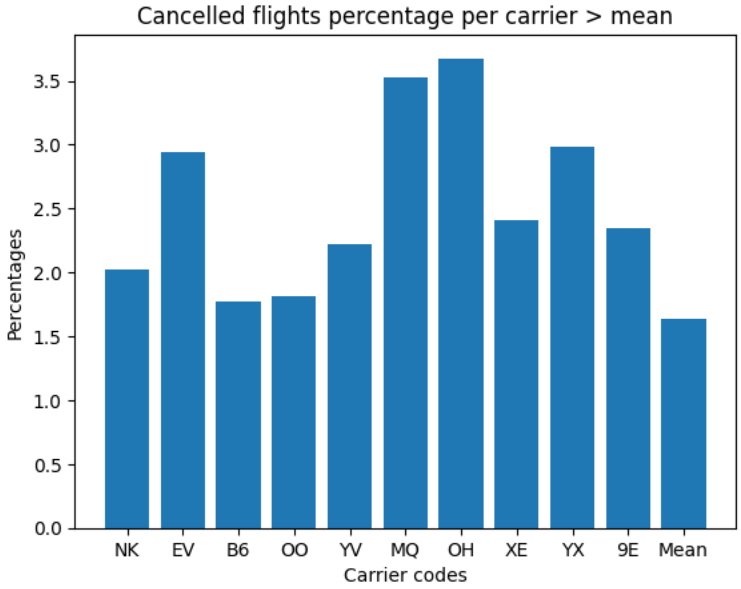

Figure 1

4.2.2 Problematics flights per origin

The second analysis was performed on the origin feature. The first chart plotted was a box plot, because due to the high number of airports, the bar was overcrowded and diffcult to read.

The result can be considered an improvement regarding the correlation between a feature and the result, because thre are some origins that posses a significant percentage of problematic flights, with a peak of over 25% of cancelled flights. This cannot be taken as a decisive factor though, because while the probability of those particular airports increase notably, the vast majority of the origins (at least two quartiles), show no evident correlation with this feature.

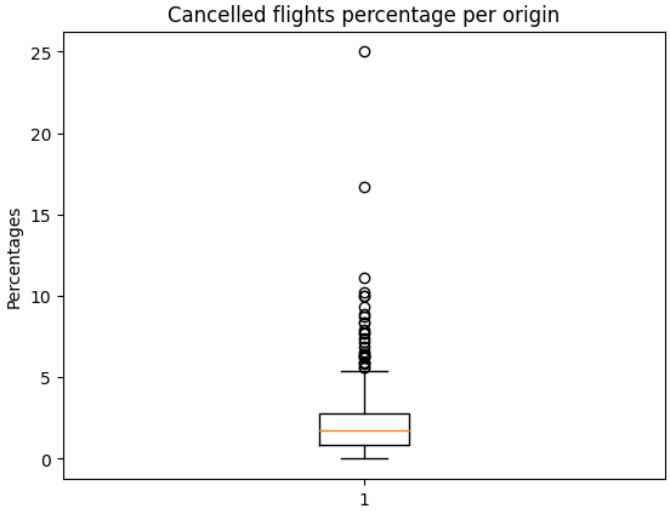

Figure 2

To show a readable bar chart, a similar technique as before has been applied, the chart shows only the airports where there is a rate of cancelled flights above 5%. This time it was not enough to remove the bars below the mean of the values, so a fixed number was used to filter the showing bars, which is switched based on the problem to solve, because the diverted problem possesses lower percentages of problematic flights.

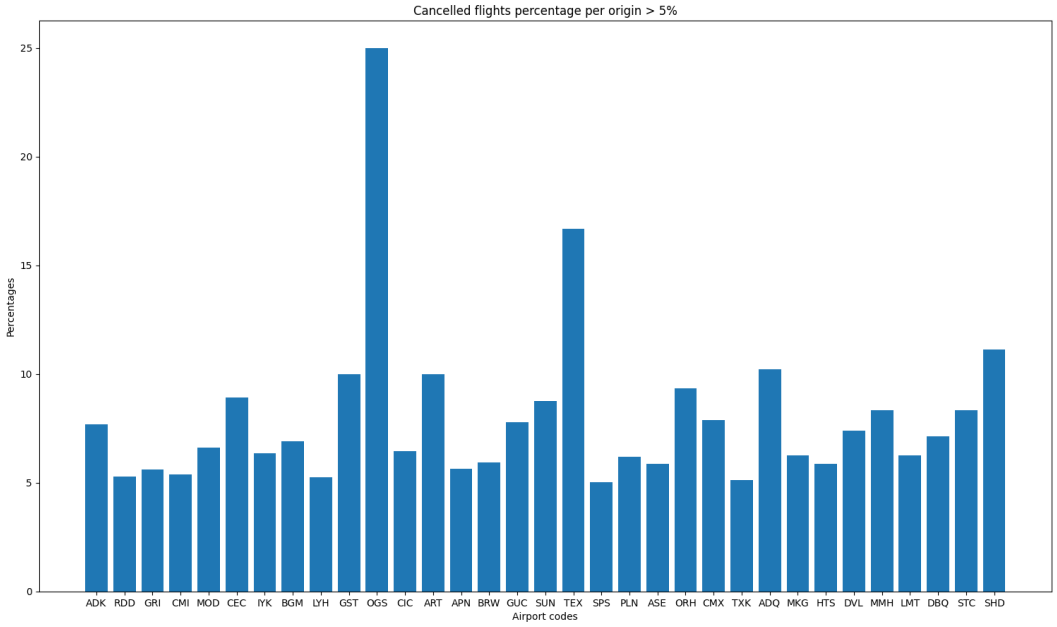
Cancelled flights percentage per origin > 5%
Percentages
0
5
10
15
20
25
ADK RDD GRI CMI MOD CEC IYK BGM LYH GST OGS CIC ART APN BRW GUC SUN TEX SPS PLN ASE ORH CMX TXK ADQ MKG HTS DVL MMH LMT DBQ STC SHD
Airport codes

Figure 3

4.2.3 Problematics flights per month

Trying to find a correlation between the year period and the problematic flights, a chart based on the month of the flight has been plotted. The result was expectedly skewed towards the winter months, where the bad weather conditions can influence the flight departures, but even in this case, the difference was not so dramatic, always remaining in a few percentage points.

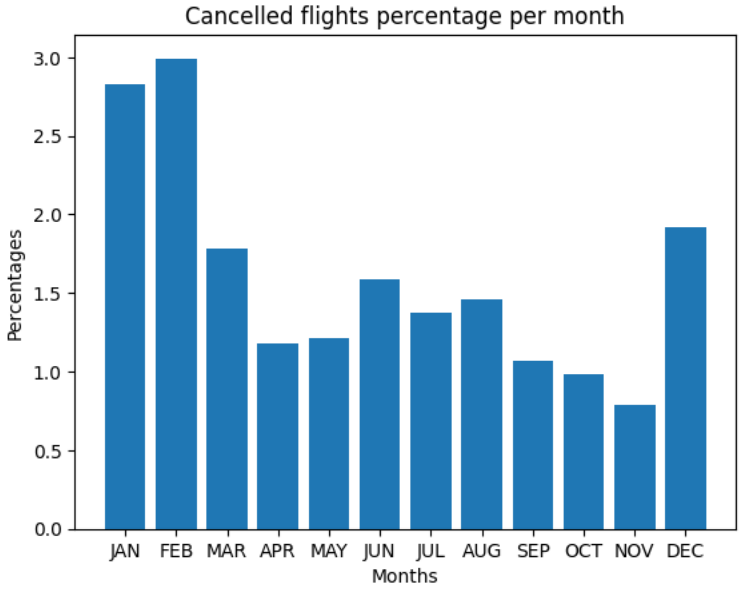

Figure 4

4.2.4 Problematics flights per weekday

The last chart was based on the weekday of the flight, trying to see if there was a relation between particular days and the rise of problematic flights.

Unfortunately this was the most useless analysys performed, with every column of the chart being almost identical to each other, basically this information doesn't tell anything about the probability of a flight of being problematic or regular.

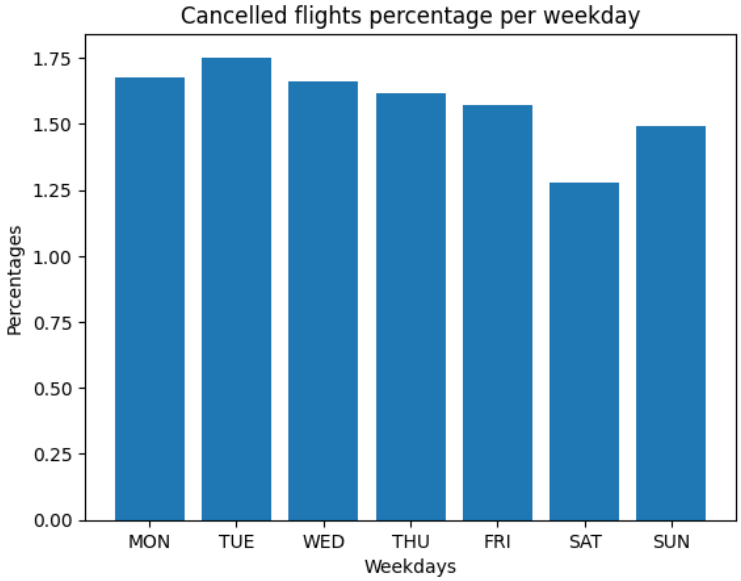

Figure 5

4.3 Data conversions

All the data not already represented as real numbers have been converted; airports and carriers identifiers, that were alphanumeric codes, had a number assigned based on the code. Dates were splitted between the year and the rest, the former has been discarded, while the latter has been hashed.

4.3.1 Hashing

Particularly, to convert the identifiers, the checksum function **crc32** [2] has been used, then the result has been put in END with the bytes representing the number -1, to ensure to get an unsigned integer. Finally, the value is normalized dividing it for the max integer value. This function has been chosen because is one of the fastest way to hash short strings, which is what is needed here, compared to other alghoritims like the SHA or MD5.

4.3.2 Normalization

To normalize dates, the day of the year has been extracted from every date, then divided by 365. A similar strategy has been used for the departure and arrival times, exctracting the minutes of the day and then dividing by 1140.

The distance has been normalized by dividing it for the maximum value found in the dataset, rounded by excess to 4970.

At this point, the dataset has been balanced in regard of the evaluated property, be it being canceled or diverted, so that there are an equal number of uniformly drawn positives and negatives.

The converted values were also limited between 0 and 1, to avoid exploding values during the training of the model.

4.3.3 Balancing

This was necessary since the diverted or cancelled flights are a really small percentage of the overall flights, this in the first tests has been proved to be a problem, because the trained model always responded that no flight would have been cancelled or diverted, since it has been trained on a dataset with basically zero problematic flights.

To solve the problem, the **Undersampling** technique has been used, matching the number of normal flights and problematic flights, limiting the amount of normal flights that has been kept.

Since less than 0.1% of the dataset are positive cases (for both the tasks) after the oversampling, the records remained are just some tens of thousands, instead of hundreds of thousands, half with positive outcomes and half with negative ones.

The following charts show the `REGULAR` and `CANCELLED` flights before and after balancing:

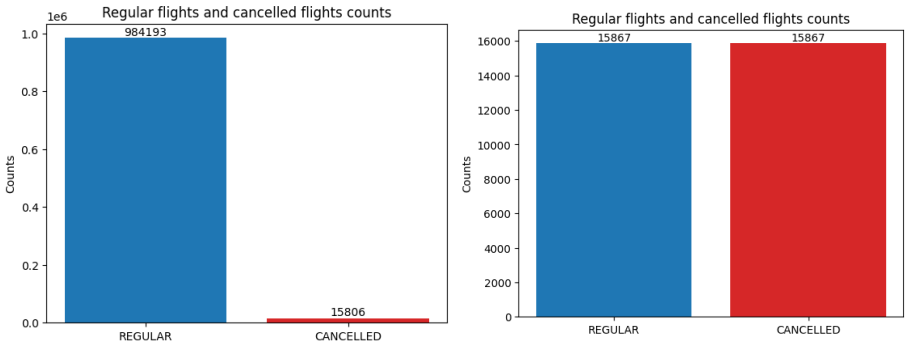

Figure 6

4.3.4 Z-Score Normalization

Before the actual computation, the dataset has been normalized again by subtracting the **mean** and dividing by the **standard deviation**. This is called **Standard Score** [16] or Z-Score Normalization.

The purpose of this normalization is to present to the model data that possesses less noise, improving the training performance. Particularly in this scenario, where the model proposed is simple and not sophisticated enough to work well with data that are not normalized.

4.3.5 K Fold Cross Validation

After the dataset has been processed, the resulting data must be splitted into a **training** part and a **testing** part. To avoid bias towards specific parts of the dataset, the K-Fold Cross Validation [3] was used.

This technique consists on splitting the dataset into K equally distributed and sized parts called folds, and while K-1 folds will be used for the training, the other one will be used for testing. This process will be repeated for K time, using each time a differend fold for the testing (and consequently excluding it from the training)

During the split, the percentages of positives and negatives cases have been kept costant for every partition, it has been ensured that every fold has a different set of rows to avoid overlapping of any kind.

$$x' = \frac{x - \mu}{\sigma}$$

4.4 Parallelization

4.4.1 PySpark

Keeping in mind that the implementation has to be space and time efficient and **scale up to large datasets**, the preprocessing part has been carried out using the library PySpark. PySpark is a wrapper for Python of the library Apache Spark [11], originally written in Java.

The purpose of this library is the handling of **parallelized data processing**, particularly regarding the Distributed File System Hadoop, also created by the Apache Foundation. The library handles automatically the work distribution on the available nodes that the system provide, it can be composed of a single machine with multiple cores or a cluster of machines, this improves significantly the **scalability** of the solution, which can be run on competely different system without code modification.

4.4.2 Compatibility

The usage of this library created some compatibility issues, because the data structures used by PySpark were not compatible with various parts of the preprocessing section, which had been written initially using the data science library Pandas [15].

To solve these problems, it wasn't possible to simply use a conversion and leave the parts written in Pandas as they were, because the computation would have run on a single machine, without parallelization, making the use of PySpark completely pointless. The issue has been addressed using the **PySpark.SQL** functionality, which allow to execute queries on a distributed dataframe. For our purposes various UDF (User Defined Functions), have been created, which then have been applied to every column containing certain types of data. When it was possible, the conversions have been accomplished using a select method only, because the UDF functions are not as optimized as the native ones, making the computations with UDF significantly longer.

To evaluate the differences between the PySpark and Pandas performances, the preprocessing part has been implemented using both methods, with the possibility to test the Pandas implementation in its own code cell.

4.4.3 I/O

The library PySpark is also able to handle the csv file reading and writing, so it has been used to save the preprocessed data to speed up multiple runs on the dataset. To carry out the writing of the various distributed dataframes, various files are created, then at the time of reading, the data is distributed to the various nodes. There are two intermediate csv files, one for the 'cancelled' problem and one for the 'diverted' problem.

4.4.4 Data structures

The preprocessing part of the solution uses the PySpark **Dataframe** structure, the main reason for this choice is to take advantage of the column notation. This makes the code cleaner and easier to read, than the standard RDD (Resilient Distributed Dataset) implementation. The dataframe SQL functions are also really handy to convert efficienlty the various columns of the dataframes. However, after the preprocessing, entering the training section of the project, it looked like the RDD structure was better suited to accomplish the various calculations required by the model, so the folded dataframes were converted to RDDs before passing them to the next section.

4.4.5 Caching

The **cache** method has proven to be a fundamental piece to improve PySpark computation performances. This method saves in the main memory the reuslts of the

transormations applied to a dataframe at the moment when its called. Caching after the data loading brought a significant reduction in computation time. Before the using of this method, folding part was almost unviable due to the huge amount of time required by the procedure. This method was also used in the parallel training part, by caching the various batches to process.

5 Model

The proposed model is a simple Logistic Regression [1] algorithm that makes use of a few techniques to avoid overfitting (batching, L2 regularization) and uses the Gradient Descent as a solver.
The implementation uses **numpy** [14] as its primary library of mathematical functions

5.1 Parameters initialization

Parameters such as Weights and Bias are initialized using a **uniform distribution** between 0 and 1, with the first one having the same length as the number of columns and the second being a scalar value. The other hyperparameters were tuned throught various experiments that will be detailed later.

5.2 Algorithm

5.2.1 Estimate

The estimate is computed as follows

$$\hat{y} = \sigma(w^T x + b)$$

where σ is the Sigmoid function [4] and it's defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \in [0, 1]$$

and w and b are the model weights and bias and x is the input.
The code implementation uses `np.exp` for the exponential calculation.

5.2.2 Gradient

$$\nabla w = \frac{1}{m} x^T (\hat{y} - y)$$

$$\nabla b = \frac{1}{m} \sum (\hat{y} - y) b$$

In the code implementation, `np.dot` and `np.mean` were used.

5.2.3 Update

Gradient Descent [6] is a technique that allows to find maximus e minimum in a multi-variable function, like the model taken in consideration.
Once the gradients are calculated, the parameters (weights and bias in this case) will be updated with the gradient value properly mitigated with the **Learning Rate**

$$w' = w - \mu \nabla w$$
$$b' = b - \mu \nabla b$$

5.2.4 Loss

For the loss the **Binary Cross Entropy** [8] function, also called **Log Loss**, was used. It is defined as

$$loss(\hat{y}, y) = -\frac{1}{n}(ylog(\hat{y}) + (1-y)log(1-\hat{y}))$$

This particular function is used since, to perform the gradient descent, it can be derived and conducted to the weights update formula to minimize the loss in the same way as it's done for the MSE [7] in the Linear Regression.

For the MSE and Linear Regression

$$MSE(w) = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(w^T x - y)^2$$
$$\frac{\partial J}{\partial w} = (w^T x - y)x$$
$$w' = w - \mu(w^T x - y)x$$

For the Log Loss and Logistic Regression

$$LogLoss(\hat{y}, y) = -ylog\hat{y} - (1-y)log(1-\hat{y})$$
$$\hat{y} = sigmoid(w^T x) = \frac{1}{1 + e^{w^T x}}$$
$$\frac{\partial LogLoss}{\partial w} = (\hat{y} - y)x$$
$$w' = w - \mu(\hat{y} - y)x$$
$$w' = w - \mu(w^T x - y)x$$

5.2.5 Batching

There are multiple types of Gradient Descent.

Stochastic Gradient Descent updates the model after each sample and has a convergence rate that is non-linear $O(\frac{1}{k})$ where k is a fixed step size.

Batching Gradient Descent updates the model once per iteration using the whole dataset at once. It has a better convergence rate.

Mini Batch Gradient Descent [9] uses small chunks of samples, so it's a middle solution between the previous ones, but adds a new hyperparameter to tune, the **Batch Size**.
Its convergence rate is

$$O(\frac{1}{\sqrt{bk}} + \frac{1}{k})$$

In this project the last one was choosen after a dedicated experiment.

5.2.6 Regularization

Regularization is a technique used to prevent the overfittings. A regularization term is added to the optimization problem (i.e. the gradient calculation) to avoid overfitting. The used version is called **L2**, also known as **Ridge Regression** [12].

The regularization factor for the loss is defined as

$$L2 = \frac{\lambda}{2}||w||^2$$

where $L2$ is calculated as

$$\frac{\lambda}{2}||w||^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

The loss then becomes

$$loss(\hat{y}, y) = -\frac{1}{n}(ylog(\hat{y}) + (1-y)log(1-\hat{y})) + \frac{\lambda}{2}\boldsymbol{||w||^2}$$

While the weights gradients formula becomes

$$\nabla w = \frac{1}{m}x^T(\hat{y} - y)\boldsymbol{+\lambda w}$$

5.2.7 Hyperparameters

To train a model, some parameters complementary to the dataset that influence the training should be individuated and their values tuned to try to find the global minimum of the function the user is trying to reproduce.
In this case, there are 3 parameters:

a. The number of iterations

b. The Learning Rate

c. The constant for the L2 Regulation

5.2.8 Model Implementation

As stated before, the parallel version of the model has been implemented using RDDs and the MapReduce paradigm. This is since SQL syntax is not suited for mathematical operations and at this stage of the execution, all the data has been converted to float, so there is no need for the DataFrame schema anymore.
The MapReduce paradigm is the perfect way to parallelize bitwise operations followed by a reduction, like the dot product. And since most of the operations in training a machine learning model are matrix multiplications, that are a sequence of dot products, there should be a gain in performances.

5.3 Differences with Pyspark ML and Sklearn implementation

In the following chapters the presented model performances will be compared with the Sklearn implementation [13], that has quite some differences.
First, both the Pyspark and the Sklearn version doesn't use the SGD solver, where the first one doesn't appear to specify what approach it uses and the second one uses **L-BFGS-B - Software for Large-scale Bound-constrained Optimization** [5], by default. In both case the solver doesn't need any form of Learning Rate.
Also, with these implementations, the L2 Regularization is enabled by default as well.

5.4 Differences with other algorithms: Decision Tree

The Decision tree is a predictive model where each node represents a variable, each arc is a possible value for that property and each leaf is the predicted value for that variable based on the values of the other properties.
To compare the previous results to a Decision Tree Classifier, the Pyspark ML implementation has been used. This version supports both binary and multiclass classification, as well as both continuous and categorical features.
Due to the structure of the problem, only the binary classification is needed. Also, since the dataset is already converted from categorical to continuous data, there's no need for other conversions.
Similarly to the Logistic Regression library from Pyspark, the model accepts directly a parallel DataFrame.

6 Experiments

The first wave of experiments follows the performances and results of the techniques described and implemented in this research.

6.1 Test Hardware

To perform the experiments, two environments were setupped.

a. A powerful and multicore execution context, represented by device with a last generation 8 core and 16 thread Ryzen CPU with 16GB of RAM and an high speed SSD, which the following results are based on.

b. A more accessible cloud environment represented by a Google Colab base instance, with a dual core virtual CPU and 12 GB of RAM

6.2 Preprocessing Performance

6.2.1 Time

The table below summarizes the time required to complete various parts of the preprocessing section.
All this tests have been performed on the same machine to avoid differences due to hardware performances.
In order to collect the data of PySpark, at the end of every section the method count() was launched, forcing the library to execute immediately the operation requested.

Without these requests, the lazy behaviour of PySpark data structures makes any type of benchmarking impossible.This probably makes the times of execution longer, due to the execution of the count operations, and the **loss of some optimization** thath could be done if the library evaluates all the operations in one run. This delay must be taken into consideration, but should not alter the overall results of the tests.

The numbers represents seconds of execution.

Section	Pandas	PySpark
Data reading	95.57988	17.09711
Null values removal	0.33791	60.71444
Balancing	0.09456	25.36964
Column conversions	0.42862	1.42960
Normalization	0.17330	64.45151
Folding	0.18040	181.06777
Total	96.79467	350.13007

Table 1: Performance comparison between Pandas preprocessing and distributed PySpark preprocessing

Due to the limitations of the hardware used during the experiments, both locally and on Google Colab, there weren't significant improvements over preprocessing speed using PySpark, because of the lack of a high number of worker nodes.
It's worth noticing though that Pyspark used correctly every core of the host, almost at maximum power, so the problem does not reside in the handling of the underlying hardware, the performance decrease must be caused by other factors.

The only section where there is a noticeable improvement is the data reading section, in the remaining ones the performances gets worse, even significantly worse, especially in the splitting section, where PySpark takes an enormous amount of time, to perform an operation completed in less than one second by Pandas.

The differences found are not stable between the various phases of preprocessing, this seems to suggest that some operations, like the normalization or the null dropping, take a higher toll over the distributed system of PySpark, forcing it to waste more time than others to organize the parallel computation, even if they appear as simpler processes. This could be explained by the big overhead the distributed enviroment creates that cannot be transformed into an advantage given the inadequate hardware.

It's also worth noticing that the Google Colab enviroment was really unstable, with the same sequence of operations taking different times in the order of **30-40%** of difference. **As a result, this data does not really provide any valuable conclusion regarding the preprocessing phase analysis.**

This suggests that the PySpark processing procedure would gain advantage over Pandas only over a really large dataset, where the time used to put up the processing infrastructure is justified by the huge amount of data to be processed. In this case instead, it seems that most of the time required by PySpark is wasted organizing the computation, without gaining any advantage from this organization, over a library that procedes straight up to the calculation of the results.

It si also possible that the time required would go down if the execution took place on a more parallelized machine than the Colab enviroment, giving PySpark the opportunity to take advantage of work distribution on a higher number of nodes.

6.2.2 Memory

Another interesting comparison to make is the memory usage. This analysis would be more significative if Pysaprk could use more than one machine to distribute the workload over multiple nodes, but it will be reported anyway.

- **Pandas Memory Usage**
  In this case the RAM usage was stable around 1.2 GB, with some peaks.
  - Average: 1.2 GB
  - Peak: 1.6 GB
- **PySpark Memory Usage**
  PySpark doubled the space required, with every step adding the amount indicated below to the former one.
  - Dataset reading: 1 GB
  - Null rows value dropping: 1.1 GB
  - Dataframe balancing: 0.4 GB

- Columns conversion: 0.4 GB
- Z score normalization: 0.3 GB
- Splitting: 0.3 GB

6.3 Hyperparameter Tuning

To choose the hyperparameters tuple θ such that the loss of the model trained using said hyperparameters is minimized, the **search grid**. This technique consists on the choice of multiple different values for each parameter and the creation of a permutation that leaves a set of tuples with all possible combinations.
The model is then trained for each tuple and the losses compared, resulting on the best model the one with the lowest loss.

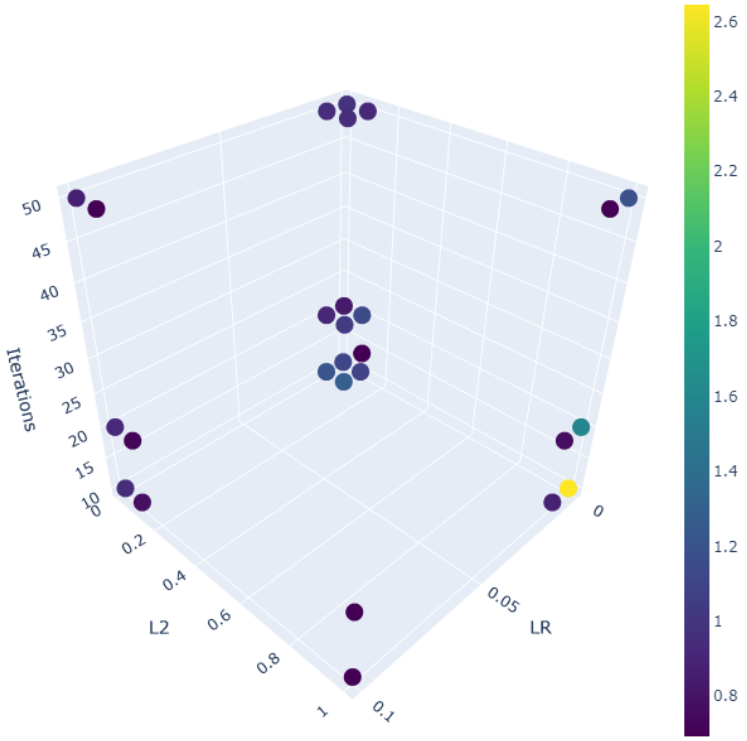

Figure 7: Parameters Space for the model

As can be seen in the 3D chart above, that represents the resulting losses for each different configuration of parameters (also called **hyperparameters space**), some values like a very small Learning Rate results in an high loss.
What follows are the values for the best configuration

Iterations	LR	L2	Loss
10	0.1	1	0.6931

Table 2: Parallel Logistic Regression best configuration

For the same configuration

TN	TP	FN	FP	Precision	Recall	F1	Accuracy	Specificity	AUROC
789	800	762	773	0.5085	0.5121	0.510367	0.5086	0.505122	0.5086

Table 3: Parallel Logistic Regression Metrics

And the following ROC curve

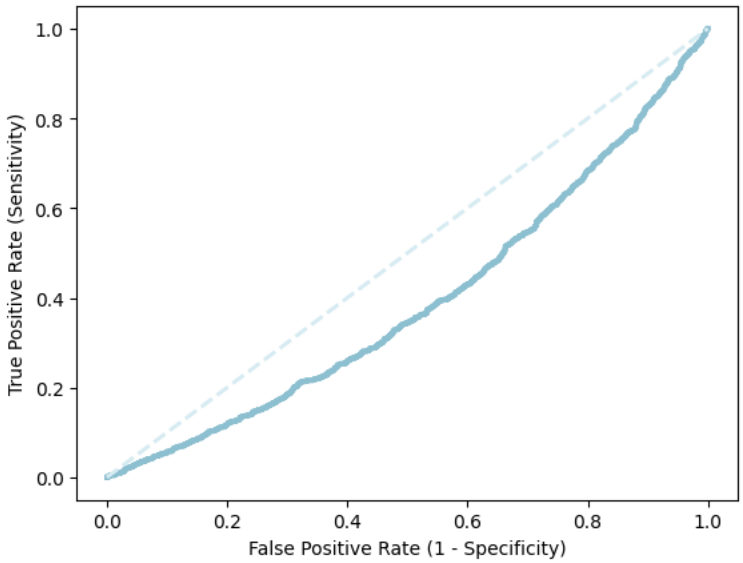

Figure 8: Parallel Logistic Regression ROC

It took **61 minutes** to compute all the 27 configurations, with the ones with 50 iterations taking most of the time.

6.4 Mini Batch

Another technique described before is the use of Mini Batches, chunk of the train datasets after each one the parameters are updated that should result in a better convergence rate.
However not only there wasn't any noticeable difference in the quality of the model, as can be seen in the following table

Loss	TN	TP	FN	FP	Precision	Recall	F1	Accuracy	Specificity	AUROC
0.6802	814	945	617	748	0.5581	0.6049	0.5806	0.5630	0.5211	0.5630

Table 4: Mini Batch version of the algorithm

but the performances of the parallel enviroment were greatly penalized by this different management of the datasets, resulting in processing time increased enough for it to not be completed by the available hardware.
This can be reconducted on the overhead added by the exchange of messages, that increases with the fragmentation of the memory and computations. The sequential implementation has been used and it just took 2.4 seconds.
For thse reasons, the Mini Batch technique doesn't seem an option worth for this specific experiment.

6.5 Sequential Implementation

Since the Pyspark implementation is full of options and various optimizations, it's difficult to compare it in term of speed with the proposed implementation.
Thus, a sequential version with the same algorithm has been implemented. The difference is not functional but just technical, since the computation is entirely done by numpy and not also by PySpark MapReduce.
By default, numpy uses a single thread and it's optimized for sequential computation. While using a single core instead of all the available ones is tipically a disadvantage, Pyspark is built for distributed computation and for this readon it has a certain overhead.
This overhead resulted in a staggering difference, since the sequential implementation took about **2 minutes and 12 seconds**.

6.6 Pyspark ML Logistic Regression

As stated before, the proposed model has been compared to the Pyspark ML implementation of the Logistic Regression Model. Since there's no learning rate, the hyperparameters grid is smaller, leading to less configurations to train. Temporal wise, the computation took **37 seconds**, which is dozen of times faster than the proposed implementation even considering the fewest configurations.
The following table represents the obtained results

Loss	TN	TP	FN	FP	Precision	Recall	F1	Accuracy	Specificity	AUROC
0.6108	834	906	591	720	0.5571	0.6052	0.5802	0.5703	0.5366	0.5709

Table 5: Pyspark ML Logistic Regression Metrics

And the following ROC curve

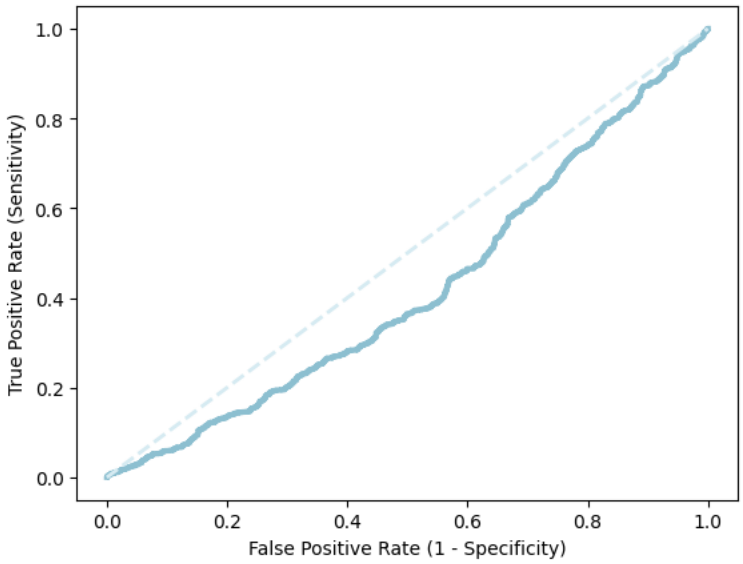

Figure 9: Pyspark ML Logistic Regression ROC

To compare the quality of the custom trained model and the Pyspark model, weights value can be compared. The following table shows the values of each element of the weights for each model and their absolute difference

Custom	Pyspark	Absolute difference
-0.0950	-0.0823	0.0127
-0.0378	0.0445	0.0823
0.0233	0.0343	0.0110
-0.2209	-0.2550	0.0359
0.1156	0.0956	0.02
0.0301	-0.0072	0.0373
-0.1946	-0.1940	0.0006

Table 6: Comparison between models weights

This comparison can also be easily done in the code using the **np.allclose**, that compares two vectors elementwise. As can be seen the elements are close by **0.08**, a very small value that implies similar expected results during the evaluation.

It's worth notice the weights absolute values are very small, meaning that for both the implementations, none of the corresponding features are impactful on the prediction.
It's then clear that the Logistic Regression, at least on how has been used in these experiments, **is not the correct choice for this complex problem.**

Besides, reaching this local minimum after just 10 iterations means that a further

number would not improve the situation in any case, creating a situation were repeating the tuning multiple times could produce a different best configuration.

6.7 Pyspark ML Decision Tree

As stated before, another algorithm has been used to compare the proposed solution to a completely different approach.
The Decision Tree ended up having better results than the Logistic Regression, with a bigger Area Under the Curve.
What follows are the complete metrics

Loss	TN	TP	FN	FP	Precision	Recall	F1	Accuracy	Specificity	AUROC
0.5250	1019	832	665	535	0.6086	0.5557	0.5810	0.6066	0.6557	0.6057

Table 7: Pyspark ML Decision Tree Metrics

And the ROC curve

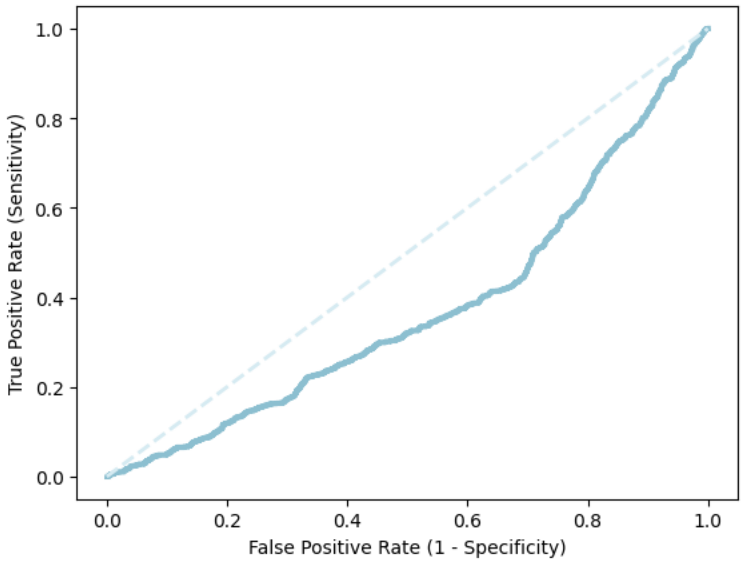

Figure 10: Pyspark ML Decision Tree ROC

The hyperparameters grid for this experiment is smaller, with just 2 hyperparameters leading to few different configurations and it just took **7.6 seconds**.

7 Diverted Flights

As specified at the start of the document, all the observations and implementations have been thought to be valid for both the initial problems and considering all the previous experiments, the overall results are similar with some differences.

7.1 Dataset & Preprocessing

The considerations on the dataset are the same, since both of the possible problems come up when there's too much delay before the start or bad climate conditions (often caused by the time of the year). In the same way, every tested aggregation didn't show any particular correlation between the data and the occourrence of the problem.
The feature taken and the way they are processed are therefore the same, with the only exception being the results column, changed from the cancelled one to the diverted one.

The only difference in this section is that the `DIVERTED` flight number is quite inferior than the `CANCELLED` one, this play a significant role in reducing the preprocessing time required. As shown in the following balancing charts, the diverted flights are just some thousands:

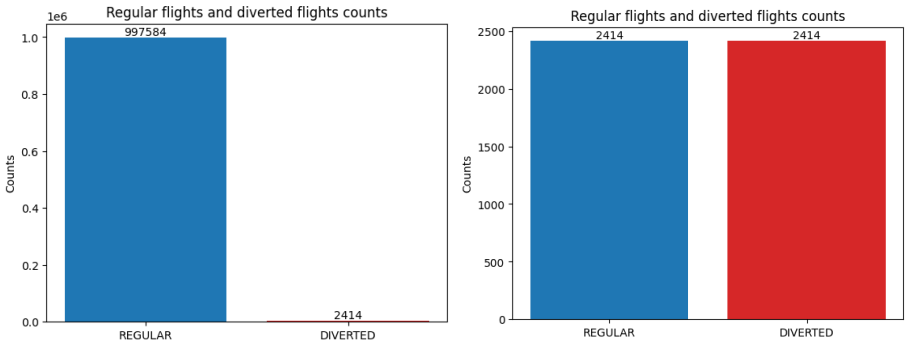

Figure 11

Section	Time in seconds
Data reading	18.5
Null values removal	67
Balancing	5.5
Column conversions	0.4
Normalization	18.3
Folding	9.9
Total	119.6

Table 8: Preprocessing performance using PySpark for `DIVERTED` flights

7.2 Model & Training

Since the data is still represented by the same conceptual columns represented in the same way, the model remained the same. The execution of the experiments,

starting with the hyperparameters search grid of the proposed implementation, resulted once again on a problem too complex to solve with a decent precision by a Logistic Regressiong model.
Even in this case the Decision Tree performed better, but not enough to be consistent.

Model	Time(s)	Loss
Custom	2538	0.6898
Pyspark	31.7	0.6139
Decision Tree	5.3	0.5673

Table 9: Times and loss values for every method used

8 Results and Conclusions

The objective of this experiments was to reproduce a complete flow to create and train a Logistic Regression model to find a relationship between flights data and flights being cancelled or diverted.
The initial dataset didn't show any particular tendency that could connect some of the columns to influence the outcome, but it was (of course) really unbalanced towards flights without issues, making it essential to balance the dataset.
The features were normalized and separated into stratified folds to enforce the K-Fold Cross Validation and the processing has been successfully parallelized using PySpark, but the lack of a proper distributed system to test on lead to worse performances compared to the pandas implementation.

The model was instead built using different techniques and implementations, starting with the Logistic Regression SGD version (both in pyspark and numpy and properly tuned), going throught the mini batch version (whose results did not bring any appreciable improvement) and ending with the Pyspark ML implementations of both the Logistic Regression and the Decision Tree. None of the models had decent performances, however the Decision Tree had the best results (on all the indicators) by quite a margin. The disappointing results are probably associated with the scarce correlation between the data and the labels, since most of the problematic flights are connected to unexpected events, like exceptional weather (that is not strongly connected to seasons) or some kind of security concerns. The computation took a considerable amount of time on the custom PySpark implementation, while much faster in both Pyspark ML and the sequential implementation, that doesn't have the overhead of the parallelization on an inadequate hardware and dataset size.

In conclusion, the experiment lead to considering the Logistic Regression and simple models in general to not be performant enough to be able to solve this complex real world problem and Pyspark a tool too much complex to be used in a non distributed environment, where it performs significantly worse than a sequential implementation.

References

[1] D.R Cox. *The Regression Analysis of Binary Sequences*. 1958. URL: https://www.jstor.org/stable/2983890.

[2] W. W. Peterson; D. T. Brown. *Cyclic Codes for Error Detection*. 1961. URL: https://ieeexplore.ieee.org/document/4066263/.

[3] M.STONE. *Cross-Validatory Choice and Assessment of Statistical Predictions*. 1974. URL: https://rss.onlinelibrary.wiley.com/doi/10.1111/j.2517-6161.1974.tb00994.x.

[4] Francisco Mira José; Sandoval. *From Natural to Artificial Neural Computation*. 1995. URL: https://archive.org/details/fromnaturaltoart1995inte/page/195.

[5] P. Lu R. H. Byrd and J. Nocedal. *A Limited Memory Algorithm for Bound Constrained Optimization*. 1995. URL: http://www.ece.northwestern.edu/~nocedal/PSfiles/limited.ps.gzl.

[6] Harold J Kushner. *Stochastic approximation and recursive algorithms and applications*. 2003.

[7] David M. Allen. *Mean Square Error of Prediction as a Criterion for Selecting Variables*. https://www.tandfonline.com/doi/abs/10.1080/00401706.1971.10488811. 2012.

[8] Pieter-Tjerk de Boer. *A Tutorial on the Cross-Entropy Method*. 2015. URL: https://research.utwente.nl/en/publications/a-tutorial-on-the-cross-entropy-method.

[9] ALEKSEY BILOGUR. *Full batch, mini-batch, and online learning*. 2018. URL: https://www.kaggle.com/code/residentmario/full-batch-mini-batch-and-online-learning.

[10] Yuanyu 'Wendy' Mu. *Airline Delay and Cancellation Data, 2009 - 2018*. 2018. URL: https://www.kaggle.com/datasets/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018.

[11] Apache. *Unified engine for large-scale data analytics*. 2022. URL: https://spark.apache.org/.

[12] *Estimating with MAP*. 2022. URL: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote08.html#map-estimate.

[13] *LogisticRegression*. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

[14] Numpy. *Numpy*. 2022. URL: https://pandas.pydata.org/.

[15] Pandas. *Pandas*. 2022. URL: https://numpy.org/.

[16] *Standard Score*. 2022. URL: https://en.wikipedia.org/wiki/Standard_score.