Airline Departure
Data Analysis and Regression

Lucchi Manuele & Tricella Davide

November 19, 2022

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 paper is splitted 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.

Both the cases represent real world problems that are still not completely solved, since there aren't algorithms that predicts these situations with accuracy yet, so a simple Logistic Regression model is probably not powerful enough to accomplish this tasks, but it's still worth analyzing its behavior in different circumstances.

3 Dataset

The initial dataset, "Airline Delay and Cancellation Data" [9] 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.

500,000 records sampled with an uniform distributed were took from each year file to perform the preprocessing.

4 Preprocessing Techniques

4.1 Data processing

Multiple preprocessing techniques were used.

Initially the rows without valid key values are removed, while the ones with invalid data inside the columns for which is possible to assign a default value are eventually corrected.

Then the data not already represented as real numbers has 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.

4.1.1 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.

4.1.2 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 dataset.

4.1.3 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 an **Oversampling** has been applied, matching the number of normal flights and problematic flights.

Since less than 0.1% of the dataset are positive cases (for both the tasks) after the oversampling, just 20000 of records could be used for the experiments, half with positive outcomes and half with negative ones. As a note, this last operation could have been performed before the others to greatly boost the overall performances of the preprocessing, but then the first part of the objective, to measure the performances of a system that scales with massive datasets, would not have been really useful.
To speed up test runs a flag has been added to decide if perform the balancinbg before the preprocessing or not.

4.1.4 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** [15] or Z-Score Normalization.

4.1.5 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 (https://rss.onlinelibrary.wiley.com/doi/10.1111/j.2517-6161.1974.tb00994.x) [**crossvalidation**] 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)

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

4.2 Parallelization

4.2.1 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 [10], 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.

4.2.2 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 [14].

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.

To evaluate the differences between the PySpark and Pandas performances, the preprocessing part has been implemented using both methods, with the possibility to choose which library to use at the start of the computation using a flag.

4.2.3 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. The data saved on the intermediate CSV is valid for both the tasks that are the goal of the paper.

4.2.4 Integration

The preprocessing part of the solution is therefore computed in a distributed manner, using the PySpark dataframe as main data structure to perform the various calculations on the columns of the dataset. At the end of this section of the solution, the distributed dataframes are merged into one using the `collect` method. This operation can create memory problems if the dataset is extremely large, but the model training and computing can't be efficiently distributed as easily as the preprocessing, so the main part of the solution will be computed using standard Python data structures.

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** [13] 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 [3] 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 [5] 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** [7] 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 [6] 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 [8] 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** [11].

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

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

5.3 Differences with Sklearn implementation

In the following chapters the presented model performances will be compared with the Sklearn implementation [12], that has quite some differences.
First, the Sklearn version doesn't use the SGD solver, it uses **L-BFGS-B - Software**

for Large-scale Bound-constrained Optimization [4] instead, by default. For this reason, the solver doesn't need any form of Learning Rate.
Also, with this implementation, the L2 Regularization is enabled by default as well.

6 Experiments

As stated in the previous sections, before starting the experiments the dataset is composed of about 20000 records, whose 15000 are for the training set and 5000 for the test set.
They are sampled from each file following a uniform distribution and contain equally positives and negatives outcomes.

6.1 Preprocessing Performance

The table below summarizes the time required by the **Colab enviroment** to complete various parts of the preprocessing section.

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 Single Node	PySpark Multiple Nodes
Data reading	168.81801	58.69491	41.72719
Null values removal	3.52260	116.81526	112.14040
Names conversion	14.52482	115.43204	117.07912
Dates conversion	53.43010	110.92972	109.54266
Times conversion	12.46091	111.87970	109.56157
Distance conversion	1.28371	112.53075	110.38996
Strings conversion	-	120.45830	110.59919
Column removal	0.73976	111.37853	110.63221
Balancing	0.73683	265.81415	256.77912
Splitting	0.04397	1190.70186	1145.30147
Total	255.56071	2314.63522	2223.75289

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.

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.

This difference reaches **a factor of 1000**, which is the factor describing the overall difference of time required to complete the entire preprocessing. This could be explained by the big overhead the distributed enviroment creates that cannot be transformed into an advantage given the inadequate hardware.

Another proof to support this thesys, is the comparison between the PySpark experiment forced into one node and the one that lets the system choose how many worker create. The first case was just slightly slower than the second, proving that both cases didn't fully use the parallelization the framework is specialized to. 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.**

For completition, the following tables shows the same experiment with the balancing phase executed before the rest of the preprocessing, that greatly decreases the number of records, as stated in 4.1.3

Section	Pandas	PySpark Single Node	PySpark Multiple Nodes
Data reading	179.81432	24.72598	40.64999
Early balancing	0.63052	55.55881	60.29146
Null values removal	0.04448	111.82450	112.65933
Names conversion	0.07760	109.92273	110.72936
Dates conversion	0.22399	110.83473	109.68199
Times conversion	0.04854	116.25847	108.61510
Distance conversion	0.00560	110.05368	107.94065
Strings conversion	-	109.58877	108.42743
Column removal	0.00507	110.72890	110.85607
Balancing	0.00384	224.42019	218.66665
Splitting	0.02637	501.79631	492.85677
Total	180,85660	1585,71307	1581,3748

Table 2: Performance comparison between Pandas preprocessing and distributed PySpark preprocessing, with just 20000 records

According to the tests performed using an early balancing before all the preprocessing procedures, the Pandas library performances have improved accordingly with

the reduction of the input dimensions. This was not the case for the PySpark performances, the vast majority of the sections require a time interval similar to the previous table, with the exception of the splitting section where the time required went down, but not proportionally to the input reduction.

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 Canceled Flights

The following sections represents the results about a series of experiments for the tuning of the hyperparameters of the model.

6.2.1 Changing the Learning Rate

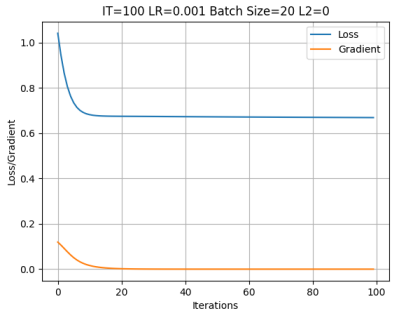
The following experiment consists on the training and evaluation of the model with different values of the Learning Rate. The training is fixed at 100 iterations.

LR	Train Loss	Test Loss
0.1	0.68296	0.69123
0.01	1.12164	1.02453
0.001	1.27019	1.21357
0.0001	1.11576	1.30633

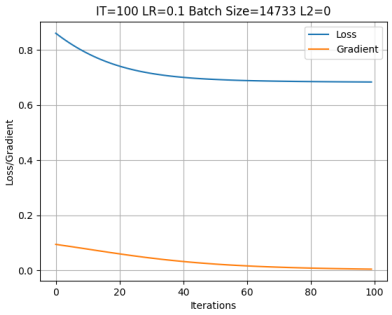
Table 3: Training and Test Loss for different values of the Learning Rate

As the results shows, an higher learning rate seems to perform better, therefore a learning rate of 1 has been tried showing similar results.
This is an unexpected behavior, since usually the gradient descent performs better with a lower learning rate. Given the complexity of the problem for a simple model like the Logistic Regression, another experiment has been made to choose the best Learning Rate.

Using the Mini-Batch technique for $\mu = 0.001$ and $BatchSize = 20$ (whose experiment will be detailed in the section below) a difference with this test can be found. As the charts shows, while both the hyperparameters configurations converge to the same (local) minimum, the one that uses mini batch and a lower learning rate converges at a much faster speed. In the following experiments, this configuration will be the one used.



(a) $\mu = 0.001$ and $BatchSize = 20$



(b) $\mu = 0.1$ and $BatchSize = Max$

6.2.2 Implementing Mini-Batch

As anticipated before, the mini-batch technique has been used. Being able to train the model with a fraction of the dataset at time resulted in a great reduction of the loss.

Batch Size	Train Loss	Test Loss
1	0.37784	1.02391
20	0.60918	0.70032
1000	1.23453	0.92476
Max	1.02630	1.01142

Table 4: Training and Test Loss for different size of batches

In the first case, the model has been updated for each sample. This caused a degradation on performances and while the training loss is low, the test loss is three times higher, which lead to think the model is overfitted.

In the second case, the test loss is much lower and similar to the train loss, while on the last two cases it's just a worse result in general. Thus, for the next experiments, a **Batch Size of 20** has been chosen.

6.2.3 Adding the L2 regularization

As described before, to avoid overfitting, a regularization term has been added, in this case the L2.

L2	Train Loss	Test Loss
0	0.60740	0.70037
0.1	0.61829	0.69714
0.01	0.61010	0.69960
0.001	0.60819	0.70023

Table 5: Training and Test Loss for a different value of the L2 term

Since the model didn't appear to be overfitting from the start, adding the regularization doesn't show any significant benefit.
The following experiments will use $\lambda = 0.1$

6.2.4 More iterations

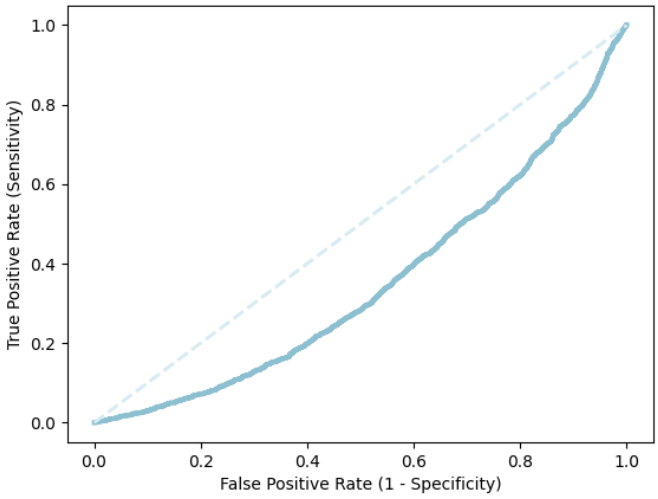
Iterations	Train Loss	Test Loss
100	0.60978	0.69974
200	0.60978	0.69969
1000	0.60962	0.69971

Table 6: Training and Test Loss for different number of iterations

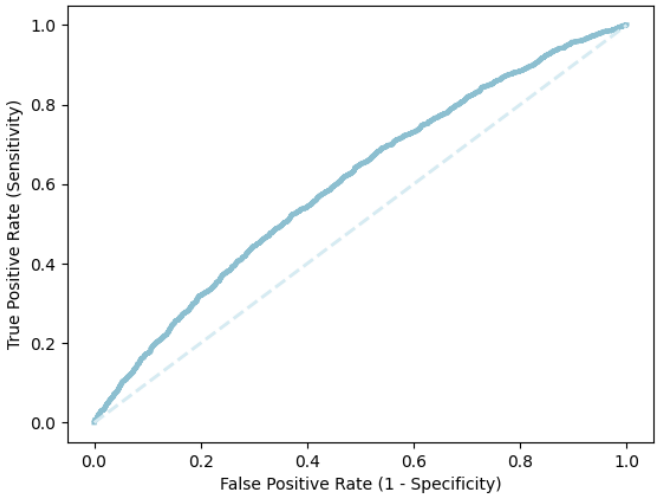
Increasing the number of iterations didn't lead to any noticeable improvement.

6.2.5 ROC Curve

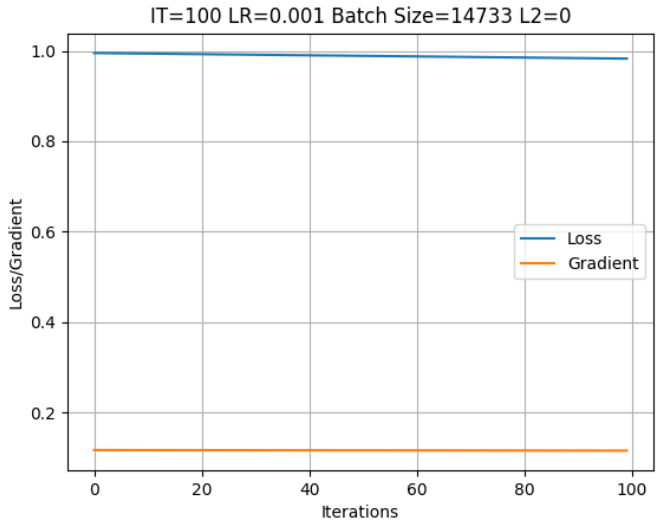
The apparently better performing configuration of the hyperparameters has the following **ROC curve**



It has a decent ability to predict, but as an inverse predictor, suggesting that the model found easier to predict that a flight would be canceled because most of the dataset had similar values.
A notable case is the experiment with the same learning rate but without the Mini Batch feature.



It has a similar (but worse) curve to the previous case, but this time it's not predicting the inverse. This is interesting since the loss of this configuration is really high and it suffers of vanishing gradient as can be seen from the following chart



It's safe to say that this model is not really reliable, and just underfitted.

6.2.6 Comparison with Sklearn

To conclude the experiments, the instance with the best hyperparameters from the previous experiments has been compared with the Sklearn implementation.

The Sklearn one uses the same number of iterations and uses L2 regularization, but doesn't use the SGD (and thus the Learning Rate) or the batching. Instead it uses **L-BFGS-B** [4], a solver for Large-scale Bound-constrained Optimization.

Model	Iterations	LR	Batch Size	L2	Train Loss	Test Loss
Custom	100	0.001	20	0.1	0.60978	0.69974
Sklearn	100	N/A	N/A	Unknown	N/A	0.66953

Table 7: Training and Test Loss comparison with Sklearn with the best hyperparameters values for both

The results are very close, with the Sklearn implementation being slightly better.

6.3 Diverted Flights

As stated before, the two problems (canceled and diverted flights) are virtually the same problem. They are both a binary classification and both uses the same columns, since they are influenced by the same factors and all are known until before the takeoff.

For this reason, the results obtained are equivalent, so they will just be reported in the following sections, with the comments for the previous section being valid for this one.

6.3.1 Changing the Learning Rate

LR	Train Loss	Test Loss
0.1	0.67999	0.69171
0.01	0.90016	0.91724
0.001	0.94824	1.16871
0.0001	1.03176	1.30609

Table 8: Training and Test Loss for different values of the Learning Rate

6.3.2 Implementing Mini-Batch

Batch Size	Train Loss	Test Loss
1	0.17018	0.99730
20	0.61886	0.69662
1000	1.24722	1.13225
Max	0.93345	1.03499

Table 9: Training and Test Loss for different size of batches

6.3.3 Adding the L2 regularization

L2	Train Loss	Test Loss
0	0.61563	0.69760
0.1	0.62842	0.69513
0.01	0.61832	0.69720
0.001	0.61989	0.69576

Table 10: Training and Test Loss for a different value of the L2 term

6.3.4 More iterations

Iterations	Train Loss	Test Loss
100	0.62105	0.69645
200	0.61847	0.69721
1000	0.61822	0.69749

Table 11: Training and Test Loss for different number of iterations

6.3.5 Comparison with Sklearn

Model	Iterations	LR	Batch Size	L2	Train Loss	Test Loss
Custom	100	0.001	20	0.1	0.62105	0.69645
Sklearn	100	N/A	N/A	Unknown	N/A	0.67672

Table 12: Training and Test Loss comparison with Sklearn with the best hyperparameters values for both

7 Results and Conclusions

The reason of this experiments was to reproduce a complete flow to create and train a Logistic Regression model to classify flights data.
The dataset preprocessing 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.
It's still important to notice that the parallelization overhead was strangely high even on a single worker configuration, leading to think part of it is by default in PySpark.

Instead, the model built using SGD, L2 Regularization, Normalization and Batching (properly tuned) resulted in an accuracy similar to the state-of-art Sklearn implementation.
Both of the models probably converged to a local minimum, that would explain the high loss, and this could be explained by the Logistic Regression being a weak model for a real world complex problem like the one taken in consideration.

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

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

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