Predicting Flights Cancellation using Logistic Regression

Vincenzo Conversano
Joint project for the courses:
«Algorithms for Massive Datasets»
«Statistical methods for Machine Learning»
University of Milan, Milan, Italy

1 INTRODUCTION

This report belongs to Scalable Machine Learning field and presents a from scratch implementation of the Logistic Regression algorithm in Apache Spark, aiming to predict Flights Cancellations.

During the last decade, the flight industry has grown a lot. Unfortunately, such a growth produced an increase in flight delays and cancellation rates. Delays and Cancellations not only influence passengers' plans but also waste flight companies' resources. This results in passengers losing their trust in famous Airline Companies and preferring other types of transportation, especially small-scale ones.

Thus, the need for technologies capable to predict whether a flight will be cancelled or not and its delay has arisen. Such technologies can help both airports and citizens have a better organization.

Many works have been published in the literature [1] [2] [3] [4], however, most of them 1) do not consider the enormous quantity of flights data that is produced every-day and apply standard methods to deal with them; and 2) are focused on predicting flight delays using regressors, ignoring that a flight cancellation is worse.

Motivated by the above, this work a) implements a Spark-based solution capable to deal with billion rows datasets; b) focuses on US flights data from 2009 to 2018; c) is based exclusively on reliable and publicly available data ensuring reproducibility; d) proposes a from scratch implementation of the Logistic Regression algorithm combining its benefits of explainability and capability to output a confidence level for the prediction; and e) optimizes the proposed model using the random search approach.

2 CLASSICAL LOGISTIC REGRESSION

The Logistic Regression as a classification model was introduced in 1944 by Joseph Berkson [5], but its history is dated back to 19th century, when Verhulot developed the logistic function for modeling population growth.

Despite its name, the Logistic Regression is a classification algorithm. Specifically, the term "regression" is used because we can find a vector $w \in \mathbb{R}^d$ very similarly to linear regression and compute a prediction for a datapoint x_t as:

$$\hat{y}_t = sign(w^T \cdot x_t) \tag{1}$$

In this way the model would output a crisp decision (either 1 or -1) but the main plus of Logistic Regression is the fact that it is able to return a confidence level for the predicted class. This is usually achieved by considering a prediction set $\mathcal{Z} = [0, 1]$ that is different by the label set $\mathcal{Y} = \{-1, +1\}$. Specifically, the above mentioned confidence level is just $P(Y = 1 \mid X = x)$ where X and Y are two random variables drawn from the joint data distribution $\mathcal{D}$ that describes our learning problem. In order to obtain such a quantity, the sigmoid function $\sigma : \mathbb{R} \to [0, 1]$ is used:

$$\sigma(u) = \frac{1}{1 + e^{-u}} \tag{2}$$

The logistic function is able to map the unbounded values we obtain from the dot product $w \cdot x_t$ in the interval $[0, 1]$.

The loss function that we used in this work, and that adapts perfectly to the setting in which we are, is the logistic loss, which has many interesting properties: 1) it is differentiable in all points; 2) it punishes more prediction that differ too much from the ground truth label; 3) it is a convex upper bound to the classical zero-one loss; and 4) it is Bayes consistent, thus, we can directly minimize the expected risk.

Specifically, it takes the product $z = y\hat{y}$ as argument and is defined as:

$$\ell_{log}(z) = \log(1 + e^{-z}) \tag{3}$$

The basic idea behind the Logistic Regression learning algorithm is to find the vector w^* that minimize the cumulative loss function plus a regularization term:

$$w^* = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \frac{1}{m} \sum_{t=1}^{m} \ell_{log}(y_t w \cdot x_t) + \frac{\alpha}{2}\|w\|^2 \tag{4}$$

Regularization is an essential technique when it comes to control overfitting (i.e. avoiding the algorithm to react to small changes in the training set). In case of linear predictors, as Logistic Regression, the regularization term allow the Empirical Risk Minimization algorithm to become stable. Intuitively, we can think to regularization as ball in the weights' space that controls the minimization procedure (avoiding the empirical risk to decrease too much and learn the noise present in the data). Moreover, α is just a strictly positive scaling factor for the regularization term that will be considered as an hyper parameter of the learning algorithm.

3 LOGISTIC REGRESSION AT SCALE

Data can scale up in two ways: in the number of dimensions and in the number of examples.

In case of linear regression, if we have only few dimensions to consider, we can think of resorting on a closed-form solution. But things are not that easy in the case of Logistic Regression since the non linearity of the sigmoid function makes it impossible to find a closed-form solution.

Thus, we need to resort to a local optimization procedure that is carried out by the gradient descent algorithm. However, the number of features that we will consider is not prohibitively large, thus we do not need to use dimensionality reduction techniques to deal with curse of dimensionality.

On the other hand, we also have to deal with the high number of examples: we can not think of storing the dataset in the memory of a single computer and processing it in a sequential fashion. Thus, we will distribute the computation exploiting Apache Spark[1].

Let J be our objective function (empirical risk) that we want to minimize through gradient descent.

$$J(w) = \frac{1}{m} \sum_{t=1}^{m} \ell_{log}(y_t w \cdot x_t) + \frac{\alpha}{2}\|w\|^2 \tag{5}$$

If we compute its generic i-th gradient's term, we can easily notice that there is an outer sum over all the datapoints. While the various terms needed to compute each summand can be easily accessed by different nodes in a Spark cluster.

$$\frac{\partial J(w)}{\partial w_i} = \frac{1}{m} \sum_{t=1}^{m} \left[-(1 - \frac{1}{1 + e^{-y_t w x_t}}) y_t x_t^i \right] + w_i \tag{6}$$

Hence, in terms of Map Reduce, the outer sum can be thought as a Reduce operation [2], while each Node computes, through a Map operation, its gradient summands over the datapoints it holds. More importantly, the weights vector $w \in \mathbb{R}^d$ is broadcasted [3] to every node in the cluster at each gradient descent iteration. The pseudo code describing the parallel gradient descent algorithm is bellow, while a scheme describing the Map Reduce job can be found in Figure 1.

3.1 Complexity Analysis of the proposed solution

In contrast to sequential computation, when it comes to distributed computation, the actual bottleneck is the network. The more data travel through the network, the more complex is our computation.

Interestingly, the gradient descent algorithm can be easily parallelized with a minimum amount of data flow

[1] https://spark.apache.org/

[2] In this context, we are referring to Spark `reduce()` implementation, that is a classical Reduce (in terms of Map Reduce) using the same key for all the (key,value) pairs.

[3] https://spark.apache.org/docs/latest/rdd-programming-guide.htmlbroadcast-variables

Algorithm 1 Distributed Gradient Descent

$m \leftarrow$ `rdd.count()`
$w \leftarrow (0, .., 0)$
for $k \leq no\_iter$ **do** ▷ Executed on the driver
  $\nabla J \leftarrow \frac{1}{m} \sum_{t=1}^{m} -\left[(1 - \frac{1}{1+e^{-y_t w x_t}}) y_t x_t\right] + w_k$ ▷ The computation of summands is parallelized over nodes
  $w_{k+1} \leftarrow w_k - \eta \nabla J$
  broadcast w to all nodes in the cluster
end for

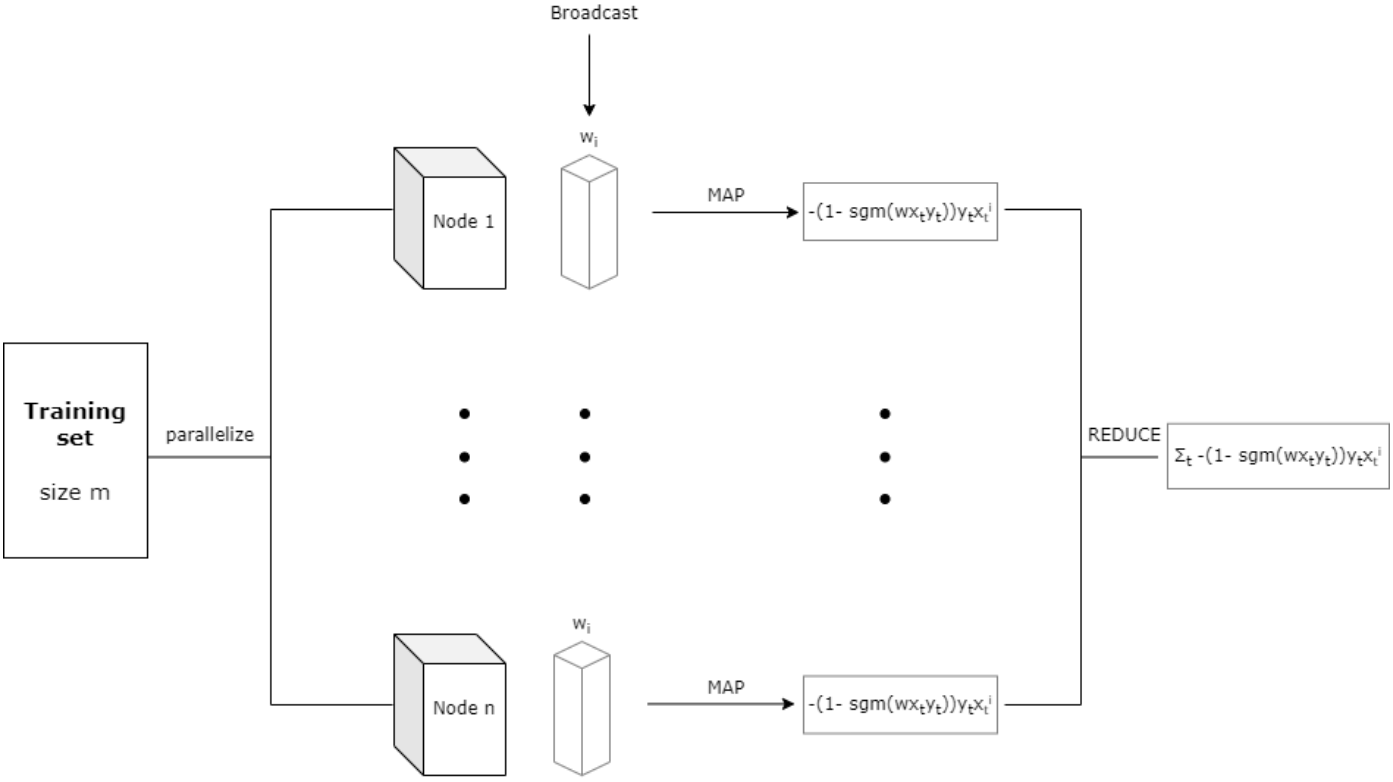

Fig. 1: Block diagram describing one GD iteration.

through network. The only step when there is network flow is when we broadcast the vector w to all nodes, but such flow involves only the driver that sends the computed vector of weights to all cluster's nodes. The real costly flows are node to node ones and there are no such operations here.

To formally measure the complexity of Map Reduce jobs, it is common to use the communication cost model. The model arranges the computation in a graph, in which each node represent a function (in this case a Map or a Reduce). Then the cost of each node is measured and they are aggregated together to provide the overall cost. The cost of one node is usually the total size of its inputs that can be measured as tuplets or bytes.

In our context, we can measure the cost of one gradient descent iteration by looking at the size of the inputs to the Map step and to the Reduce one, that are both the size of the training set. Thus, the complexity will be $\mathcal{O}(|S_{train}|)$.

3.2 Model hyper-parameters

The vector of weights is learnt by the learning algorithm directly from data, exploiting gradient descent algorithm, aiming to minimize the cumulative loss function. In addition to these parameters, Logistic Regression have other parameters (i.e. hyper-parameters) that are not learnt from the data and one should properly tune.

The hyper-parameters that we can usually found in the Logistic Regression learning algorithm are:

- Learning rate: it is a scalar that determines how much we are adjusting the weights' vector at each step. When starting the optimization procedure usually a larger learning rate is used, while it is lowered down when we are towards the minimum. Hence, we employed a decayment technique: at each iteration the learning rate is decayed according to the formula: $lr = \frac{initiallr}{\sqrt{i}}$.
- Scaling factor α: it is a scalar that regulates the influence of the regularization term.
- No. of iterations: it is the number of iterations to do during the gradient descent procedure.

Details about the parameterization approaches we used are described in the subsection 4.4.

4 EXPERIMENTAL SETUP

This section describes a) the considered task, b) how is the dataset composed, c) the data preprocessing steps, d) the model parameterization, and e) how results have been analyzed.

All the experiments were performed on Google Colab[4] and on a local machine equiped with i7-1065G7 CPU.

4.1 The considered task

This work aims to analyze the «Airline Delay and Cancellation Data, 2009 - 2018» dataset[5]. Specifically, the task is to implement from scratch a classifier based on logistic regression and to train it to predict whether or not a flight will be canceled, only on the basis of information available at flight departure.

4.2 The dataset

The dataset «Airline Delay and Cancellation Data, 2009 - 2018» contains data of domestic flights operated by large United States' air carriers. The included features supply a lot of information, as Table I shows. However, not all of them can be used because of their direct relation with the labels.

[4]https://colab.research.google.com/

[5]https://www.kaggle.com/datasets/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018

In order to retrieve data from the source we used the `kaggle API`[6].

4.3 Data Pre-processing

The data pre-processing step is one of the most important step in a common Machine Learning pipeline. In this work we considered the following sub steps.

4.3.1 Data Organization: The raw dataset is organized in `csv` files, one file per year. Since we are dealing with a time series dataset, we can not think of arbitrarily shuffling the datapoints and then do a train-test split. Thus we organize splits as follows:

- Data from 2009 to 2016 for the training set S_{train}.
- Data of 2017 for the validation set S_{dev}, that will be used for hyperparameters tuning.
- Data of 2018 for the test set S_{test}, that will be used for risk estimation.

As previously mentioned, some features can not be used because they are directly related to the label. For instance, if we consider the feature `ARR_TIME`, that is referring to the actual arrival time of the plane, it would be easy for our classifier to solve the task just by looking if its value is missing.

In the end, we considered only the features available before the departure, plus two features that were created ad hoc (as it will be described in the subsection 4.3.6).

It should be mentioned that all the datasets were stored in Resilient Distributed Datasets (`RDDs`).

An `RDD` is a fault-tolerant collection of data elements partitioned across the nodes of a cluster that can be operated on in parallel. The choice of using `RDDs` instead of Spark `Dataframes` or `Datasets` was driven by the fact that it is easier to implement Map Reduce jobs using them, and we are not constrained by a predefined schema.

4.3.2 Missing Values: The raw dataset is full of Missing Values, but such missingness only involves features that are only available at the arrival time. For such features, it is obvious that all the **cancelled** flights

[6]https://github.com/Kaggle/kaggle-api

Fl. Date	Airline Id	Origin	Destination	Planned Dep. Time	Planned Arr. Time	Dep. Time	Arr. Time	. . .
2009-01-01	XE	DCA	EWR	11:00	12:02	10:58	12:06	. . .
2009-01-01	XE	EWR	IAD	15:10	16:32	15:09	16:24	. . .
2009-01-01	XE	EWR	DCA	11:00	10:59	12:10	12:01	. . .

TABLE I: Overview of the Raw Dataset.

will have missing values, so it is important to correctly discard these features.

Not surprisingly, after dropping the non-usable features the number of rows with missing values drastically decreased, ending up with 61 rows. Thus, we can assume that the mechanism behind the missingness is completely at random (MCAR) and proceed with discarding these rows.

Dataset	#rows with NaNs
S_{train}	44
S_{dev}	7
S_{test}	10

TABLE II: Missing values distribution.

4.3.3 Dataset Balancing: When organizing the data we took into account the classes imbalance. As expected, the vast majority of flights was not cancelled, thus we are in imbalanced setting. To understand the severity of the imbalance, we plotted a simple barplot showing the classes' absolute frequencies (Figure 2) and calculated the imbalance ratio ($\frac{\#regular}{\#cancelled}$ = 61.88). Class imbalance can cause problems in the learning procedure, pushing the models to learn better the most representative class. Thus we decided to undersample the most representative class, obtaining a new training set composed of 1550619 datapoints.

This choice was driven by set balancing reason only (as discussed in Section 2 this solution can scale up to billion rows dataset without any problem).

Moreover, in this work, we avoid considering oversampling approaches such as SMOTE [6] because we consider the procedure of data generation (based on the feature set we are considering) complex to be mimicked.

Regarding validation set, we applied the same undersampling approach used on the training set. This was done because validation set will also be used for retraining after having optimized parameters.

Regarding test set, the set balancing procedure was not done. We just need to be careful in using appropriate figures of merit (that do not suffer class imbalance) to evaluate the performances, an example is F1-Score.

Summarizing, a general overview of sets' sizes is given in Table III.

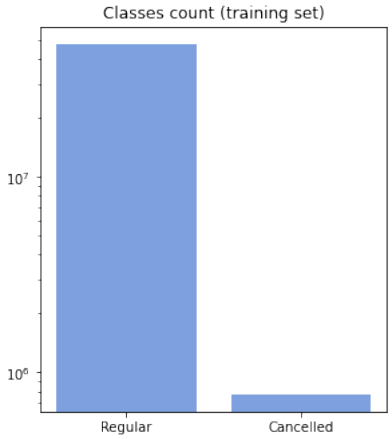

Fig. 2: Classes count.

Dataset	Size
S_{train}	1550619
S_{dev}	165025
S_{test}	7213436

TABLE III: Datasets' sizes.

4.3.4 Categorical Features: Dealing with categorical features has always been a problem in machine learning field because the vast majority of machine learning algorithms work with numerical data. Among the considered features, we have three categorical ones: a) `ORIGIN`: i.e. starting airport code; b) `DEST`: i.e. destination airport code; and c) `OP_CARRIER`: i.e. airline identifier.

Initially, we were thinking about applying feature hashing [7] [8] to `ORIGIN` and `DEST` features, because of their huge amount of categorical values that would be transformed into new features in case

of one-hot encoding. But then we discovered that the DISTANCE feature already embeds the information of the two airports, because there is a unique distance per airports combination. One could argue that we lose the information of which is the starting and the arrival airport but we assume this loss to be acceptable, considering that feature hashing produces non-interpretable features and the problems of the One Hot Encoding approach. On the other hand, the OP_CARRIER feature was one-hot encoded because it can take only 21 categorical values[7] over the training dataset.

4.3.5 Periodic Features: It is important to notice that the features CRS_DEP_TIME and CRS_ARR_TIME are periodic, and thus they need an appropriate encoding. For example, let's suppose to have a flight departing at 23:30 and another one departing at 00:30. Using the original encoding or other classical ones, such as the number of minutes passed since midnight, it is impossible for any machine learning model to figure out that there is just a difference of one hour between the two flights.

To cope with this problem, we exploit the periodicity of sine and cosine functions to create two coordinates, lying on the unit circle, of the considered time value t. These two coordinates will be our two new features and are defined as:

$$\begin{cases} x = \sin\left(\frac{2\pi t}{24}\right) \\ y = \cos\left(\frac{2\pi t}{24}\right) \end{cases} \tag{7}$$

4.3.6 Feature Engineering: Inspired by the work of Lambelho et al. [2] we decided to add two new features in our feature set to help the models in the classification task. Specifically, we thought that ORGIN airports that show an higher Average Delay and Taxiout could be more prone to cancellations. Thus, we computed for each ORIGIN airport these two values and embedded them in the features' set describing each flight.

4.3.7 Data Normalization: Performing data normalization is an essential step in a common Machine Learning pipeline. In this specific case, since we are using a really small model, we can not pretend that our weights can learn the various scales of the features, leading to infinity losses.

In this work, we opted to normalize data using Z-score normalization (8), whose computation is easy to distribute using Map Reduce. Indeed, we just need to compute the mean and standard deviations vectors on the training Resilient Distributed Dataset (train_rdd) and broadcast them on the various nodes of the cluster, so that they can normalize the datapoints they are keeping. The means and standard deviations are computed in a distributed fashion exploiting the Statistics class from PySpark [8]. In order to avoid any information leakage, the validation RDD and test RDD are transformed using training set's means and standard deviations.

$$x = \frac{x - \mu}{\sigma} \tag{8}$$

It should be pointed out that categorical features and periodic features are excluded from normalization since they already have a limited range.

4.3.8 Final Features' Set: At the end of the preprocessing steps we got a new feature set that will be fed to the learning algorithms. An overview of the new features is shown in Table IV.

Feature	Type	Notes
Day	Numeric	-
Month	Numeric	-
Weekday	Numeric	-
Planned Elapsed Time	Numeric	Planned duration of the flight
Distance	Numeric	-
Avg. Delay	Numeric	Avg. Flights' Delay of Origin
Avg. Taxiout	Numeric	Avg. Taxiout of Origin
Planned Dep. Time x	Periodic	-
Planned Dep. Time y	Periodic	-
Planned Arr. Time x	Periodic	-
Planned Arr. Time y	Periodic	-
Airline ID	Categorical	One hot encoded

TABLE IV: Overview of the final Feature Set.

[7]Actually, during one hot encoding we added one more feature for encoding airports that were not present in the training set but present in the validation and test sets (Only two airports).

[8]https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.stat.Statistics.htmlpyspark.mllib.stat.Statistics.colStats

4.3.9 Features' Analysis: It is interesting noticing that some features are linked with the output label. For example Figure 3 shows that there are days during the week that are more prone to cancellations. While in Figure 4 we observe that June is the most unreliable month to flight.

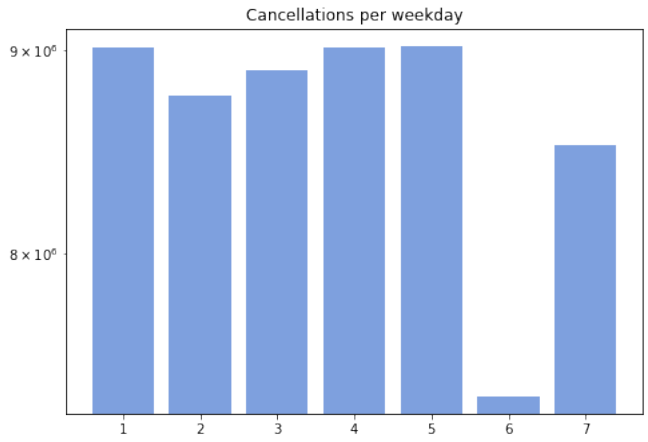

Fig. 3: No. of cancellations per weekday.

4.4 Models Parameterization

This section describes the parameterization process of the learning algorithm.

Fortunately for us, the Logistic Regression algorithm does not have many hyper-parameters to tune, in such situations we would usually define a grid $\Theta_0 \subset \Theta$ - where Θ is the set of all possible hyperparameters' values - and exhaustively search the best hyper-parameters' configuration $\theta \in \Theta_0$ in it. Indeed, this is what we would have done if this work was run in an actual cluster, but since everything is run locally we have to settle for a random search.

Moreover, the big data situation in which we are allow us to not consider cross validation because we assume that our validation set is well describing the distribution $\mathcal{D}$ behind the data. It should be pointed out that in this way we are estimating the risk of the classifier generated by the learning algorithm when we give it the specific training set S_{train}.

So in summary, we used two main approaches: 1) set the hyper-parameters using fixed values (first experiment); 2) optimizing hyper-parameters through a random search (second experiment).

4.4.1 Fixed Model: Firstly, we train our classifier configuring the hyper-parameters manually. After some trials, we found the following configuration.

Learning parameter	Value
no. iterations	100
learning rate	1.0
alpha	0.01

TABLE V: Fixed Model's configuration.

With this configuration we minimized the empirical risk reaching a value of 0.65, which is not much but, as described in section 5.1, outperformed the `MLLib` implementation of Logistic Regression.

4.4.2 Meta Model: Secondly, we tuned the model hyper-parameters on the validation set using a random search of 30 trials. Random Search tuning is a very effective way of tuning hyper-parameters when we do not have enough resources for doing a grid search. Instead of testing every combination of hyper-parameters, the idea is to randomly sample the defined grid Θ_0 and search over these combinations. To implement a random searcher, we used the python package `itertools`[9] which computes all possible combinations of hyper-parameters' values for the considered grid Θ_0. Then we shuffled them, selected the first n configurations (where n is the number of trials we want to do) and iterate on them. In each iteration we configured the model with the current parameters' values, ran the training and measured the reached cost on the validation data. At the end we retrain the model on the training and the validation sets and evaluate it on the test set for risk estimation.

4.5 Numeric Stability

The computation of logistic loss and of its gradient can easily overflow or underflow. This is due to the fact that double-precision floating-point numbers (i.e., 64-bit IEEE) only support a domain for $\exp(\alpha)$ of roughly $\alpha \in$

[9]https://docs.python.org/3/library/itertools.html

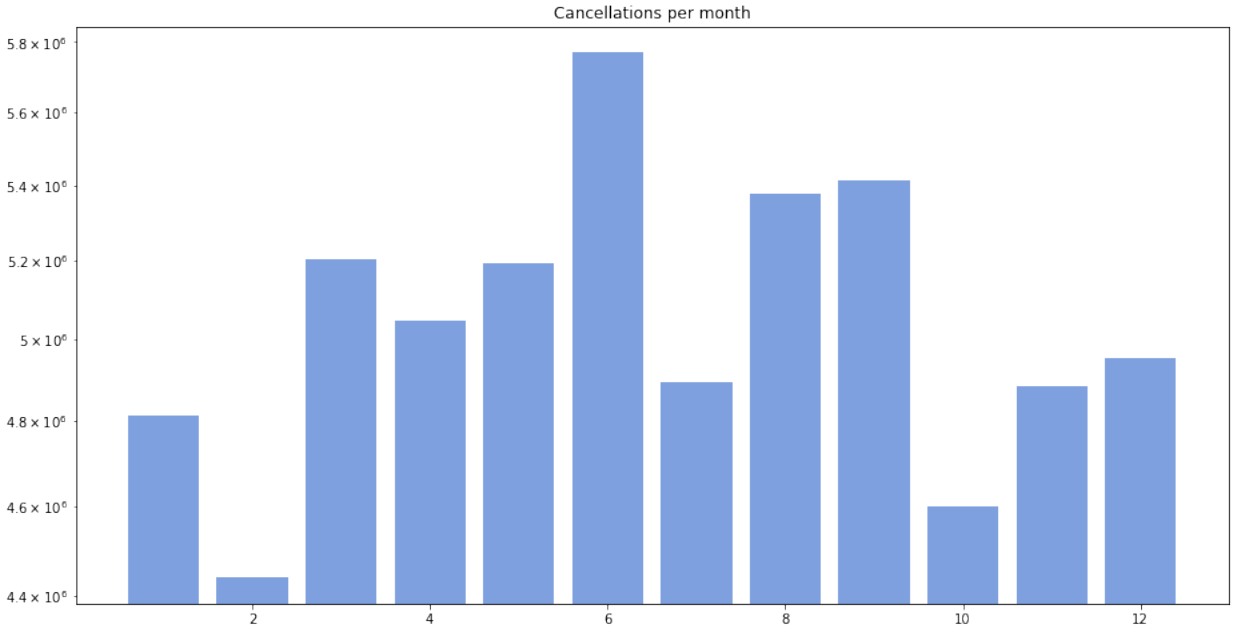

Fig. 4: No. of cancellations per month.

$(-750, 750)$ before underflowing to 0 or overflowing to positive infinity.

Thus, we need to implement the `sigmoid()` and `log_loss()` functions in a stable way. To do that, we resorted to `np.logaddexp()` function that implements a stable log-sum-exp routine.

4.6 Result Analysis

The metrics used to evaluate models' performances were chosen taking into account the highly imbalanced settings in which we are. Indeed, metrics such as Accuracy are not considered. Hence, we used:

- Logistic loss cost: it is the mean of log losses, it is the same metric used in minimization.
- F1-Score: harmonic mean of the precision and recall, for computing this metric we considered a threshold of 0.5;
- AUPRC: Area Under Precision Recall Curve, this metric is particularly useful for the detection of rare events;
- AUROC: Area Under Receiver Operating Characteristic Curve.

5 RESULTS

This section describes the experimental results, including a comparison with the `PySpark MLLib` implementation of Logistic Regression and the state of the art.

5.1 Fixed Model Results

In this subsection, we show and comment on the results obtained by the Model using the fixed hyperparameters' configuration on S_{test}. As showed in Tables VI-VII, it is easy to notice that the Logistic Regression model is under-fitting, indeed the cost function is both high on training and test sets. Moreover, while AUROC is acceptable (both in training and test sets) the AUPRC and F1 metrics are higher in the training set, this is an effect of the data set balancing procedure. The problematic metric is the Precision (that is defined as the rate of true positives over all the positives predicted by the model), while in training set we have less chances to be wrong (because of under-sampling), in the test set we have many more negatives that are predicted as positives because of under-fitting.

Predicted \ **Actual**	Cancelled	Not Cancelled
Cancelled	73228	2728318
Not Cancelled	43353	4368537

TABLE VI: Confusion Matrix of Fixed Model

Cost	F1-score	AUROC	AUPRC	Dataset
0.65	0.62	0.66	0.65	Train
0.65	0.05	0.66	0.03	Test

TABLE VII: Figures of merit of Fixed Model.

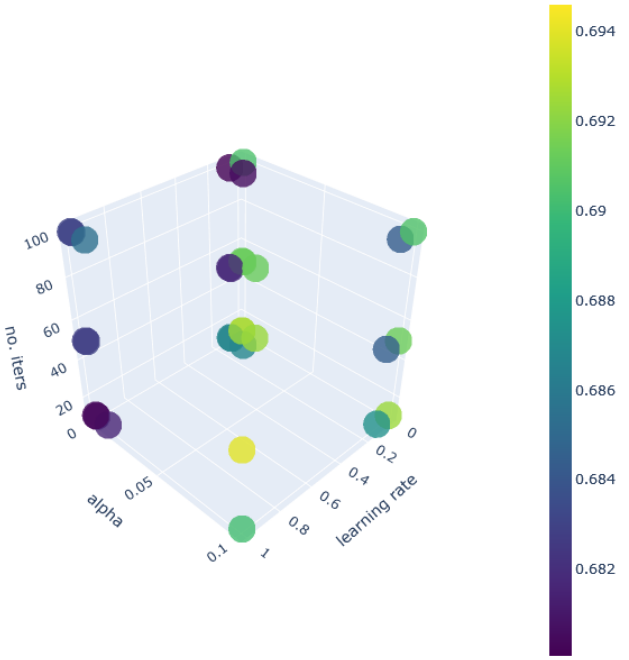

Fig. 5: Hyper-parameters' space

5.2 *Meta Model Results*

In this subsection, we show and comment on the results obtained on S_{test} using hyperparameters' tuning. As showed in Tables VIII-IX, there is not much changing from the fixed configuration, confirming that the task is too difficult for a linear classifier. Indeed, even though 30 configurations were tried, none of them produced performances that left behind fixed configuration's ones.

Predicted \ **Actual**	Cancelled	Not Cancelled
Cancelled	73887	2877164
Not Cancelled	42694	4219691

TABLE VIII: Confusion Matrix of Meta Model

Cost	F1-score	AUROC	AUPRC	Dataset
0.66	0.04	0.03	0.65	Test

TABLE IX: Figures of merit of Meta Model.

Since we considered only three hyper-parameters, we plotted in Figure 5 their space and the value of cost reached in each trial.

5.3 *Comparison With Mini Batch Gradient Descent*

Mini Batch gradient descent is halfway between Deterministic gradient descent and Online gradient descent: it computes the gradient of the cost by considering only a fraction of the training set. When the dataset is enormous, this is the only viable solution, but we should consider that the convergence will be slowed down and more iterations will be required to reach the minimum.

Expecting to reduce the execution time of training, we implemented the mini batch variant. This was achieved by using the method `sample()` from `PySpark`, which, as the name suggests, given a `fractionSize` samples the `RDD` and returns the new `RDD`. This is done at each training iteration.

We considered a fraction size of 0.01, obtaining from various experiments very similar learning curves. While regarding execution times the mini batch version is not as faster as expected. This is probably due to the fact that even though the train `RDD` is cached before training, Spark could recompute some jobs on the original `RDD` before sampling it, causing waste of time. Figure 6 presents an example of learning curves and execution times.

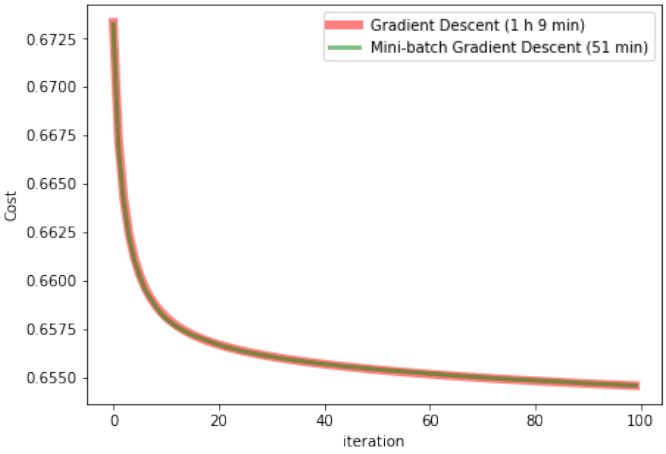

Fig. 6: GD vs Mini Batch GD.

5.4 Comparison With PySpark MLLib implementation

`PySpark` provides its own implementation of the logistic regression algorithm exploiting mini batch gradient descent[10][11]. The parameters are quite easy to configure, just by setting `fractionSize=1.0` we transformed mini batch gradient descent into classic (deterministic) gradient descent and by setting `convergenceTol=0.0` we avoided early stopping.

Aiming at a reliable comparison we used the same hyper-parameters' values for both implementations: 1) learning rate = 1.0; 2) regularization factor = 0.01; 3) regularization type = L2. The difference between the two implementations are not notable (Table X), confirming that our implementation works properly and can reach the minimum.

Algorithm	Cost	Execution Time
LogisticRegressionWithSGD	0.7066	57min 48s
Our Implementation	0.6501	1h 11min 28s

TABLE X: Comparison with LogisticRegressionWithSGD.

Additionally, we used `np.allclose()` function to assert that the vectors of weights are component wise close with a tolerance of 0.1. This means that, as expected, the learned weights are very similar.

5.5 Comparison With the State of the art

As already mentioned, there are many works in the literature tackling our task.

The actual state of the art was reached by Lambelho et al. [2] who used LightGBM, MLP and RF to predict Flight Delays and Cancellations based on the major London Airport Data, confirming that non-linear classifiers may improve performances. However, it should be noted that considering data coming from just one airport makes the solution only valid for that airport.

Moreover, there are other works that use Logistic Regression Model and consider exactly our dataset, but the results of some of these are not reliable.

The second work we analyzed was carried out by Yanying et al. [9] who did a very interesting analysis, considering 5 million rows of flight data and exploiting an actual Spark cluster to run the computations. As they show, the execution time of various learning algorithms (comprising Logistic Regression) rapidly goes down with the increase of cluster's nodes.

Concerning evaluation, they used AUPRC and AUROC metrics, allowing us to perform a one-to-one comparison with our results and understand 1) if our results are correct, and 2) if their model is struggling on the same metric.

Indeed, they obtained the following results ($AUROC = 0.5$ and $AUPRC = 0.108$) that are perfectly in line with ours. These results confirm that the Logistic Regression fails in Precision assessment and this is due to a potential under-fitting both in our work and in Yanying et al. one.

Lastly, we analyzed the work by Navoneel Chakrabarty [10]. In this work only a limited fraction of the dataset was considered (27019 data points). Firstly, he kept the dataset unbalanced, obtaining a confusion matrix suffering from False Positive errors. Secondly, he used SMOTE for oversampling the Cancelled flights data, decreasing False Positive errors. However, we can not say that this approach is correct, because 1) SMOTE could lead to overoptimistic results, and 2) in the dataset there are almost 1 million cancelled flights' examples, making it unreasonable to generate synthetic data when we have real ones.

6 DEPLOYMENT OF THE SOLUTION

Clearly, the proposed solution can not be deployed and used as it is. This is due to two main reasons:

- First, the parallelism is just simulated. And such simulation is even more expensive than a sequential solution. Google Colab does not provide a cluster, indeed, if we open the Spark UI [12] we can see that

[10] despite the name (`LogisticRegressionWithSGD`) they provide an implementation of mini batch gradient descent, since we need to specify the fraction of samples to consider and do not consider one sample at a time as in Stochastic gradient descent

[11] https://spark.apache.org/docs/latest/api/scala/org/apache/spark/mllib/classification/LogisticRegressionWithSGD.html

[12] https://spark.apache.org/docs/latest/web-ui.html

we just have one machine where all the computations take place, both driver's and workers' ones. To make matters worse, Colab standard environment offers only two single-core CPUs, thus, all the computations will be extremely slow, indeed a local machine with a more performing CPU was employed for heavier computations.

- Secondly, `PySpark` is just a nice addendum to the original Scala APIs, that has gained a lot of popularity because of its gentle learning curve. However, the code ran using `PySpark` is more or less 10 times slower than Scala code, because of the marshalling operations. In a setting like ours, using `PySpark` is not suggested and we should prefer Scala APIs.

7 CONCLUSIONS

This report described an Airline Cancellation prediction system based on a from scratch implementation of Logistic Regression with gradient descent.

After extensive experiments and a comparison with the state of the art, it was shown that Logistic Regression and in general linear classifiers are not suitable for the task because of under-fitting problems.

We believe that - as was shown by Lambelho et al. [2] - the employment of more complex classifiers will improve performances. Moreover, among the three algorithms used in their work, only the Random Forest algorithm is available in `MLLib`[13]. Thus, it is important to continue researching and extending classical Machine Learning algorithms to work in a distributed fashion.

It should be pointed out that the present experimental set-up is based on a publicly available dataset, while the results are fully reproducible and available on github[14].

8 DECLARATION

I declare that this material, which I now submit for assessment, is entirely my 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 my work. I 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 me or any other person for assessment on this or any other course of study.

[13]https://spark.apache.org/mllib/

[14]https://github.com/vincenzoconv99/flight_cancellations_prediction

REFERENCES

[1] R. Henriques and I. Feiteira, "Predictive modelling: flight delays and associated factors, hartsfield–jackson atlanta international airport," *Procedia computer science*, vol. 138, pp. 638–645, 2018.

[2] M. Lambelho, M. Mitici, S. Pickup, and A. Marsden, "Assessing strategic flight schedules at an airport using machine learning-based flight delay and cancellation predictions," *Journal of Air Transport Management*, vol. 82, p. 101737, 2020.

[3] J. J. Rebollo and H. Balakrishnan, "Characterization and prediction of air traffic delays," *Transportation research part C: Emerging technologies*, vol. 44, pp. 231–241, 2014.

[4] Y. Ding, "Predicting flight delay based on multiple linear regression," in *IOP Conference Series: Earth and Environmental Science*, vol. 81, no. 1. IOP Publishing, 2017, p. 012198.

[5] J. Berkson, "Application of the logistic function to bio-assay," *Journal of the American statistical association*, vol. 39, no. 227, pp. 357–365, 1944.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[7] J. Moody, "Fast learning in multi-resolution hierarchies," *Advances in neural information processing systems*, vol. 1, 1988.

[8] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1113–1120.

[9] Y. Yanying, H. Mo, and L. Haifeng, "A classification prediction analysis of flight cancellation based on spark," *Procedia Computer Science*, vol. 162, pp. 480–486, 2019.

[10] N. Chakrabarty, "A data mining approach to flight arrival delay prediction for american airlines," in *2019 9th Annual Information Technology, Electromechanical Engineering and Microelectronics Conference (IEMECON)*. IEEE, 2019, pp. 102–107.