UNIVERSIDAD POLITÉCNICA DE MADRID

Botnet

Massively parallel machine learning

Group 12:

Hugo Garde
Corentin Ibos
Elise Maistre

19/12/2023

I. Introduction

In this practical assessment task, the goal is to implement and evaluate a parallel version of the logistic regression classifier using PySpark. The provided dataset, "botnet_tot_syn_l.csv" contains labelled examples for training the machine learning model. The task involves parallelizing key functions such as reading the dataset, normalising data, training the logistic regression model, and evaluating its accuracy.

The project is divided into three main steps:

First, the centralised versions of the mandatory functions (readFile, normalise, train, accuracy and predict) were coded using NumPy arrays. We verified that the algorithm converges during training by seeing a decrease in the cost value over iterations.

Once the centralised implementation is working, the new focus is to parallelize these functions using Spark, to see the benefits of distributed computing.

Furthermore, the assessment includes the implementation of cross-validation procedures using Spark.

II. Data set description

As a data set, we are working with "botnet_tot_syn_l.csv" which is a CSV file containing 1 million lines and 12 columns. Each cell is a characteristic value as a real number except for the last one which is an integer label which indicates if it’s a normal traffic (0) or a botnet (1) used for the training.

III. Algorithm implementation

A. Centralised

1. ReadFile

The readFile function reads a CSV file containing a dataset with 12 columns (11 features and 1 label). It loads the data into a NumPy array, extracts features (X) and labels (y), and returns a NumPy array of tuples. Each tuple consists of an array with 11 features (floats) and an integer label (0 or 1).

2. Normalise

The normalise function takes a NumPy array npArray_Xy as input, where each element is a tuple (X, y). It extracts the array of features X, computes the mean and variance for each column in X, and then normalises each element in X by subtracting the mean and dividing by the standard deviation. The resulting array, npArray_Xy, contains the original labels y and the normalised features X. We are using a reduced centred normal law N(0,1) in this process. The normalisation process ensures consistent scales across features.

3. Train

The train function takes a labelled dataset represented as a NumPy array (npArray_Xy) and performs gradient descent to train a logistic regression model. The dataset is split into features (X) and labels (y). The function initialises the weights (ws) and bias randomly. It then iterates through the specified number of training iterations.

During each iteration:

- Predictions (Y_pred) are computed using the logistic function based on the current weights and bias. It involves initialising the model with random weights and a bias. For each example in the dataset, a linear combination of features and weights is calculated, passed through a sigmoid function to obtain a probability between 0 and 1. This probability is thresholded (at 0.5) to produce binary predictions (0 or 1).

- The cost function is calculated with the given formula.

- The derivatives of weights and bias are computed with the given formulas.

- Weights and bias are updated by subtracting their derivatives times the learning rate.

The cost values are stored. The function returns the final weights and bias after training.

Additionally, the function visualises the evolution of the cost function over iterations using matplotlib, providing insights into the training progress.

4. Accuracy

The accuracy function calculates the accuracy of a logistic regression model on a set of examples. It takes a NumPy array npArray_Xy with features X and labels y, as well as the weights and bias in ws. The function extracts features and labels, uses the predict function to obtain predictions, compares predictions with actual labels, counts correct predictions, and computes accuracy by dividing the count of correct predictions by the total number of examples. The resulting accuracy is then returned.

5. Predict

The predict function takes an example X and a set of weights and bias in ws as input. It computes the prediction Y_pred for the given example using logistic regression. Finally, the function returns a binary value (0 or 1) based on whether the computed prediction is greater than or equal to 0.5. This function is suitable for making predictions in a logistic regression model trained with binary classification tasks.

B. Parallelized

1. ReadFile

We first create a new RDD with the Spark function “textFile”.
Then we use two map transformations to process each line of the RDD in a parallelized manner:

- The first map operation splits each line into a list of fields using a comma as the delimiter. This step separates the features and labels in parallel across the dataset.

- The second map operation creates tuples for each line, where the first element is a NumPy array representing the feature vector (X), and the second element is the integer indicating the label (y).

2. Normalise

The first map transformation is applied to the input RDD (RDD_Xy) to extract the 11 features from each record into an RDD “X”.

To get the dimensions of the RDD X, we use X.count() which counts the number of rows with spark and to find out the number of columns, we measure the size of the first row (X.first(), we could also use X.take(1)).

Compute the mean :
We use a reduce transformation applied to the RDD X to compute the sum of feature values for each column. The reduce operation is performed in parallel across partitions of the RDD, where each partition computes a partial sum. The results from different partitions are then combined to obtain the overall sum. Then all of each sum for the 11 features are divided by the number of lines.

Compute the variance :
First, we use a map transformation on the RDD X to calculate the squared differences between each element and its mean for each row. This step prepares the data for the subsequent “reduce” operation. This operation is performed independently on each record in parallel, because if we do it in reduce we do not obtain the good result. Then all of each sum for the 11 features are divided by the number of lines.

Applying normalisation:
The final map transformation applies the normalisation function to each record in the initial RDD (RDD_Xy). This function rescales each feature in a row based on the computed means and variances. Like previous map operations, this transformation is performed independently on each record in parallel.

3. Train

As we did in the normalisation part, we use a map to extract the 11 features into an RDD X and we compute its dimensions.

Then, in the loop, we make the following computations using spark parallelization:

Computation of the predictions:
We use a map to create a new RDD called predictions with each line being a 3 dimensional tuple with an array of the feature, the prediction made from the features and the label. For each line, we sum each x multiplied by their weight and then apply the sigmoid to the sum. This gives us the label prediction. We map on each line to parallelize this calculation.

Computation of the derivatives:
Now that we have the predictions, we can create a new RDD called derivatives. To do that, we map the predictions RDD with a function computing the derivatives for each line. Therefore, on each line of the RDD we have the 11 derivatives and the bias.
We reduce the RDD derivatives to sum the corresponding derivatives of each line and to get a list of the final derivatives. To have the final value of the derivatives, we need to add a term to each of them but this is done in a central way.

Computation of the cost function:
We compute the cost by applying a map on the predictions RDD to have for each line:
$y \times log(\widehat{y}) + (1 - y) \times log(1 - \widehat{y})$
To compute the sum, we apply a simple reduce to add each line of the previous RDD.
We'll do the calculation in a first map, because if we did it directly with the reduce, the calculations would be different and would change with the lines, given the way spark works. The other computations needed to compute the cost are done in a central way.

One of the problems we encountered at the start was discovered thanks to spark's lazy evaluation. We were calculating the new weights in the centralised version before calculating the cost function. We realised the error because the results differed between the two versions. Finally, we calculated the new weights after calculating the cost function.

4. Accuracy

The accuracy function uses different map and reduce functions to find the number of well predicted values with trained final weights and the actual values to be determined.

The first map is created so that each of the lines (sent validation lines) is transformed into a tuple with the prediction between 0 and 1 in the first position and the y value (the real value) in the second position. This map uses the same prediction function as the centralised version.

The second map reuses the RDD created in tuple form, transforming each line into a 1 or 0 indicating whether or not the two values in the tuple (the predicted value and the actual value) are equal.

Finally, a reduce on the last RDD obtained is used to count the number of 1s (by summing) that correspond to the number of correctly predicted values.

The last step consists of dividing the result by the number of rows we have by performing a count() on the initial RDD.

IV. Cross validation procedure

A. Transform

First of all, the data must be transformed so that it can be split into 2 sets (training and testing) in order to perform the cross-validation.

This transformation is done by a map operation that transforms each line of the RDD into a tuple whose key is an integer between 0 and the desired number of division blocks K, and whose value is the line (containing the 11 features and the label). As there are a large number of lines, the law of large numbers tells us that assigning a random number according to a uniform distribution will give an equal distribution.

We are doing this because we want to shuffle the lines, in the sense that consecutive lines of the file will probably not end up in the same training set.

B. Get Block Data

At the end of this transformation, iterations are launched according to the desired number K. The data set is then divided into a train set and a test set. At each iteration, it will be a different partition.

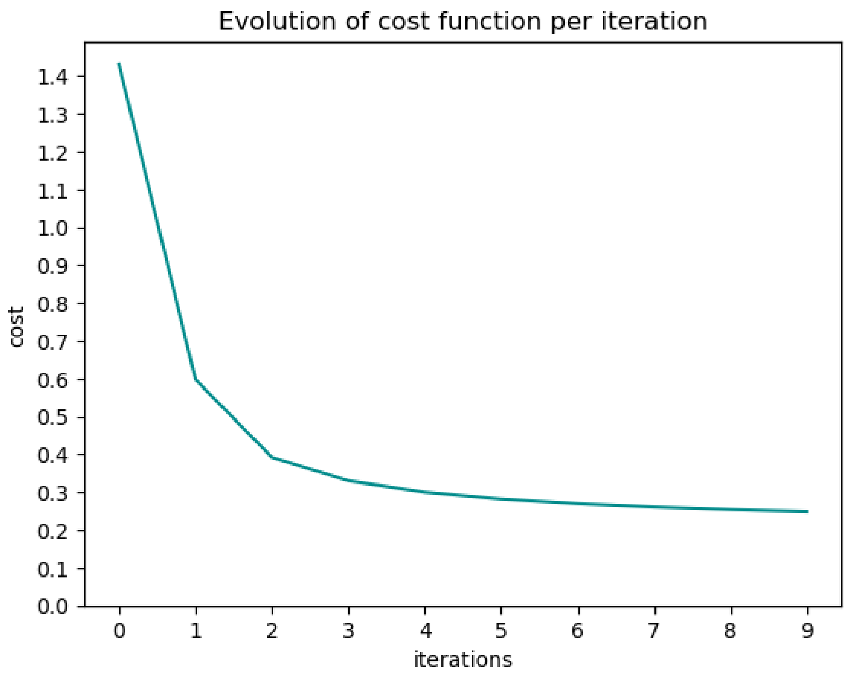
The function receives the transformed data and the iteration.
FlatMaps are used to obtain the train and test sets, because we want to obtain an RDD with a smaller dimension than the initial one (without using the forbidden function filter). We want to obtain two RDDs, one containing the training data and the other the test data. To achieve this, for the first RDD we check that the key of the tuple in the row does not correspond to the iteration. In this way, the RDD will have on each line an array X (containing the 11 features) that has not been selected for the iteration, or an empty list if it has (resulting in nothing in the RDD). The second RDD is built in the opposite way.

V. Experiments

<u>Results for the centralised version :</u>

Costs error of main with botnet_tot_syn, learning rate = 1.5, lambda reg = 0.1 and 10 iterations:

Iteration	Cost function
1	1.4305963790422633
2	0.5977780010718109
3	0.3920980778417579
4	0.33022681836789397
5	0.3000868615955463
6	0.28186073116478416
7	0.2695266091888207
8	0.26060717572163083
9	0.2538751822769308
10	0.24864170826989926

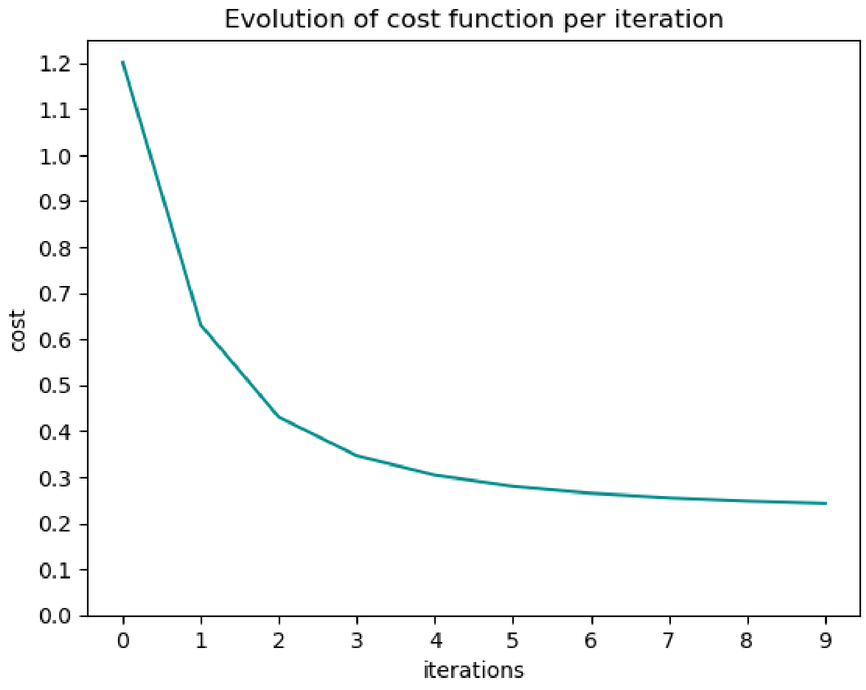


Accuracy: 0.9264
Execution time: 145.05 s

Results for the parallelized version :

Costs error of main_with_metrics with botnet_tot_syn, learning rate = 1.5, lambda reg = 0.1 and 10 iterations:

Iteration	Cost function
1	1.2012203933745902
2	0.6308910920380196
3	0.4303001879528821
4	0.34650140725485495
5	0.30427712882493907
6	0.28009853285605846
7	0.26496558566692274
8	0.2548786473297943
9	0.24783492607885854
10	0.2427356833068996



Accuracy: 0.9262
Execution time: 159.32 s

Observations:

Looking at the previous results for the parallelized version and the centralised version with different random start weights, we see a similarity. After 10 iterations, the results start to converge towards a value for J of around 0.24, whatever the starting weights. During the first iteration, the results of J are rather high, generally above one, then from the second iteration decreases very quickly towards a lower value. Finally, J tends to converge towards a value of around 0.2.

The cost function values for each iteration provide insights into the training process of the logistic regression model. Initially, the cost is relatively high, indicating a pretty high difference between predictions and actual labels. As the iterations progress, the cost rapidly decreases, signifying quick adjustments in model weights and bias to better fit the training data. As the iterations progress, we see a gradual reduction in cost, indicating smaller adjustments to optimise model parameters. The process aims to minimise the cost function, aligning model predictions more closely with the true labels and improving overall performance.

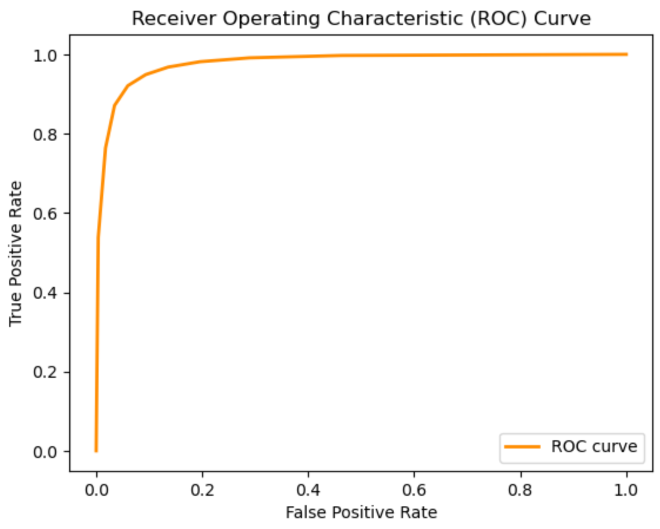
The accuracy for the two versions is very similar and quite good: at around 0.926.
The execution times are also very similar. This is because the spark engine takes time to set up, and the parallelization effect would be observed if we performed more iterations or input data (with a larger dataset).

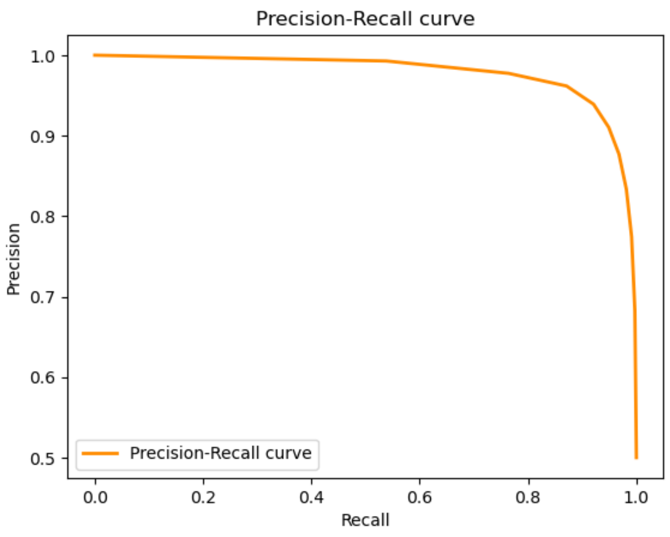
Metrics according to the threshold:

Threshold	**Accuracy**	**Precision**	**Recall**	**F1-Score**	**Specificity**
0.0	0.5	0.5	1.0	0.6666666666666666	0.0
0.1	0.76672	0.68255	0.99722	0.8104185290532304	0.53622
0.2	0.85115	0.77433	0.991156	0.8694305117692364	0.711144
0.3	0.892715	0.83350	0.981476	0.9014613859239998	0.803954
0.4	0.916037	0.87695	0.967882	0.9201755011432293	0.864192
0.5	0.927752	0.91043	0.948854	0.9292450705218478	0.90665
0.6	0.9306	0.93906	0.920956	0.929924188877535	0.940244
0.7	0.918245	0.96169	0.871186	0.9142077001619198	0.965304
0.8	0.87309	0.97746	0.76379	0.8575165600089816	0.98239
0.9	0.767091	0.99279	0.538086	0.6979111407985784	0.996096
1.0	0.5	1.0	0.0	0.0	1.0

The previous metrics for the different thresholds show that the best results are obtained when the threshold is around 0.5 or 0.6. They can be analysed more clearly by looking at the following curves.

Precision-Recall and ROC Curves:





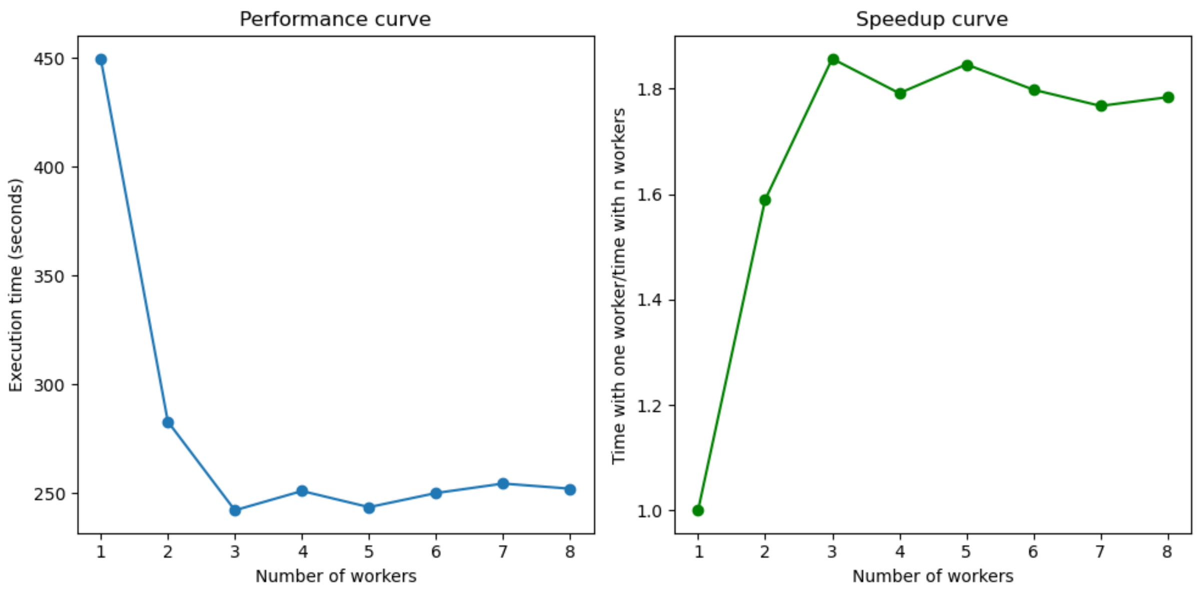
- ROC curve:

The ROC curve starts at (0,0) and moves towards the top-right corner, which is a good indication. It suggests that the model is able to achieve a high True Positive Rate while maintaining a relatively low False Positive Rate.
The AUC, which represents the area under the ROC curve, allows us to appreciate the overall performance of the model. This suggests that the model is effective in distinguishing between positive and negative instances.
The false positive rates and true positive rates for various thresholds give insights into the model's behaviour at different decision points. A balanced default threshold of 0.5 shows reasonably low false positive rate and high true positive rate. Thresholds closer to 0.0 yield high sensitivity, and thresholds closer to 1.0 yield high specificity.

- Precision-Recall curve:

The Precision-Recall curve starts at (1,0) and moves towards the top-left corner, which is a positive sign. It suggests that the model is able to achieve a high precision while maintaining a high recall.
The area under the Precision-Recall curve suggests that the model is effective in achieving high precision across different recall levels.
Similar to the ROC curve, the Precision-Recall curve provides insights into the model's behaviour at different decision points. A balanced default threshold might show reasonably high precision and recall. Lowering the threshold could improve recall but might decrease precision.

Performance and speedup curves



- Performance curve:

As the number of workers increases, the execution time decreases. This decreasing trend suggests that the workload is being distributed more efficiently across multiple workers, leading to faster execution times. This is a positive sign of scalability, indicating that the parallelization of tasks is effective.
While performance improves with an increasing number of workers, there are diminishing returns beyond a certain point. Having more than 3 workers provides diminishing benefits or even starts to have a negative impact.

- Speedup curve:

The positive speedup values suggest that, in general, parallelization has led to improved performance compared to using a single worker. This is a positive outcome as it indicates that distributing the workload among multiple workers has reduced the overall computation time.
The initial speedup values show an increasing trend as the number of workers grows, which aligns with the expected behaviour of parallelization.
The speedup values start to stabilise and even slightly decrease as the number of workers increases. As in the previous curve, having more than 3 workers provides diminishing benefits or even starts to have a negative impact.

As we assigned four workers to the virtual machine, we expected the benefits to diminish after four workers but it is not the case.

Cross validation:

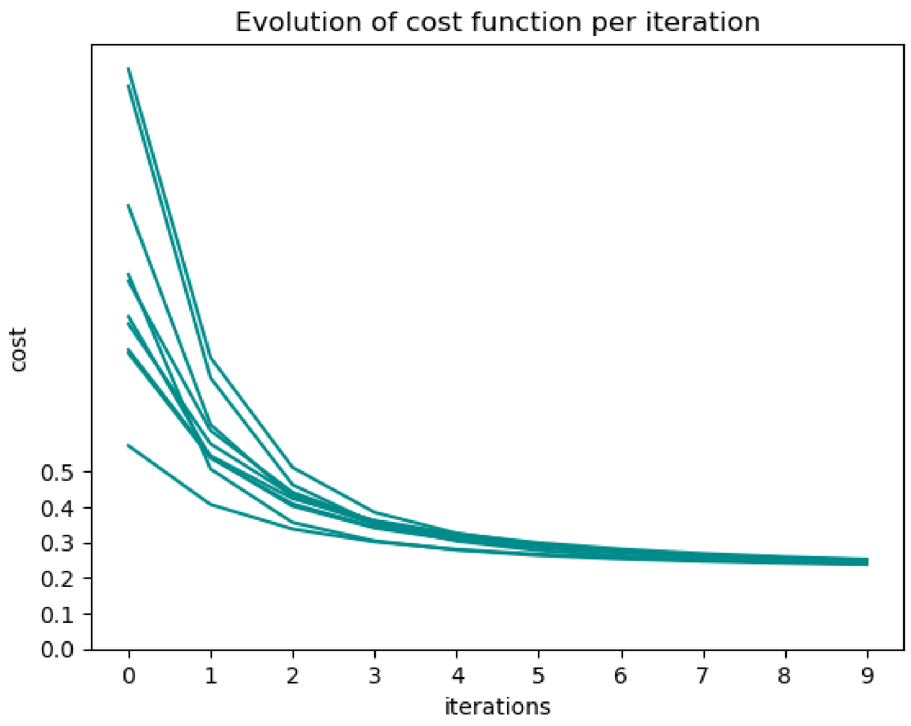
Test Block Number	Accuracy
1	0.9218
2	0.9256
3	0.9356
4	0.9241
5	0.9278
6	0.9297
7	0.9181
8	0.9241
9	0.9327
10	0.9313

Average of accuracy: 0.9271
Execution time: 1218.19 s

The cross-validation results allow us to observe the distribution of accuracies when we divide our data into blocks in 10 different ways.

Each train and test is different for each iteration. The accuracys results obtained therefore also differ. However, the results are still approximately the same, which means that our data is very homogeneous.

When we look at the average accuracy, it remains close to the previously obtained results of 0.927.



This curve shows that the evolution of cost function per iteration differs each time we launch the algorithm as it is initialised at random but converges to the same cost value.

VI. Conclusion

In conclusion, we saw the ability of the logistic regression model to converge and learn from the provided dataset. We used the gradient descent optimization to update the model's weights and bias during the training process, contributing to the model's overall predictive accuracy.

The incorporation of parallelization strategies was a significant aspect of this project, aimed at enhancing computational efficiency. The performance analysis indicated large improvements with parallelization. However, we observed a smaller efficiency with an increasing number of workers which shows the need for a balanced approach. Indeed, using too many workers is counterproductive.

Performance evaluation metrics, including ROC and precision-recall curves, showed the model's predictive capabilities at different decision thresholds. The computation of accuracy, false positive rates, and true positive rates allowed us to better understand the model's performance.

In this project, we used as hyperparameters the values given by the teacher. However, hyperparameter tuning could further enhance the model's predictive accuracy.