UNIVERSIDAD POLITÉCNICA DE MADRID

Parallel Implementation and Evaluation of Logistic Regression Algorithm

Academic year 2023-2024

Authors:

Nicola Cecere

Luca Petracca

Alessandro Rossi

Professor:

Mozo Velasco Bonifacio Alberto

Contents

1 Introduction

In this report, we analyze the use of the logistic regression algorithm for supervised classification tasks, and its application in a massively parallel processing environment using Spark and Python. Our investigation centers on the dataset 'botnet_tot_syn_l.csv', which comprises network traffic data, categorized as either normal or botnet-related. The primary objective is to assess the effectiveness of parallel computing in machine learning, especially in processing large datasets with speed and precision.

We explore four fundamental aspects of this process: data preparation (readFile), data normalization (normalize), model training (train), and model evaluation (accuracy). The readFile function involves loading and organizing the dataset, which contains 11 features and a label for each entry. Normalization is a critical step to format the data suitably, ensuring our logistic regression model can be effectively trained. The train procedure is implemented using Gradient Descent, as shown in Fig 1. The training phase is where our model learns to differentiate between data patterns, a process we optimize through parallel processing techniques. Lastly, the accuracy function is employed to analyze the model's performance, an essential aspect of any machine-learning task.

An additional critical aspect we address in this study is the implementation of cross-validation. Cross-validation is a robust method for assessing the generalizability of the model. By dividing the dataset into several subsets, we train our model on different combinations of these subsets and validate its performance on the remaining parts. This method not only provides a more comprehensive evaluation of the model's accuracy but also helps in mitigating the risks of overfitting. Employing cross-validation in a parallel computing environment poses its unique challenges, which we will explore, particularly focusing on its impact on computational efficiency and model reliability.

Throughout this report, we follow the process of implementing these machine learning functions in a parallelized setup using Spark. This includes discussing the challenges and specifics of parallel processing in machine learning. We aim to demonstrate how parallel computing can enhance the efficiency and speed of machine learning tasks, a vital consideration in today's data-intensive landscape. We will also analyze the impact of varying computational resources, like the number of cores, on the performance and acceleration of the process. The evaluation of our logistic regression model in a parallel context will include key performance indicators such as accuracy, precision, recall, F1-score, ROC and Precision-Recall curves.

This study contributes to the broader understanding of machine learning in the context of parallel computing. It aims to provide insights into developing more robust and efficient machine learning models, capable of managing large datasets with greater efficacy.

Gradient Descent Algorithm

```
initialize $w_1$; $w_2$; ... $w_n$; $b$
           $dw_1$; $dw_2$; ... $dw_n$; $db$
for $it$ in range ($iterations$):
    compute $dw_1$; $dw_2$; ... $dw_n$; $db$
    $w_1 = w_1 - \alpha * dw_1$
    $w_2 = w_2 - \alpha * dw_2$
    ...
    $w_k = w_k - \alpha * dw_k$
    $b = b - \alpha * db$
```

Cost Function

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(\hat{y}^{(i)}) + (1 - y^{(i)})log(1 - \hat{y}^{(i)})) + \frac{\lambda}{2m}\sum_{i=1}^{k} w_i^2$$

Derivatives

$$dw_1 = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) * x_1^{(j)} + \frac{\lambda}{m}w_1$$
$$dw_2 = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) * x_2^{(j)} + \frac{\lambda}{m}w_2$$
$$\dots$$
$$dw_k = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) * x_k^{(j)} + \frac{\lambda}{m}w_k$$

$$db = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)})$$

Figure 1: Algorithm Implementation

2 Dataset Description

The *botnet_tot_syn_l.csv* dataset is a structured collection of data points designed for the implementation and evaluation of a supervised machine learning algorithm, particularly a parallel version of logistic regression using Spark with Python.

- **Columns:** The dataset comprises 12 columns, including 11 feature columns (X) and 1 label column (Y).
- **Rows:** The dataset contains 1,000,000 rows. Each row in the dataset corresponds to a network traffic example, represented as a tuple (X, y) in the RDD format.
- **Features (X):** Each feature column contains float number values, representing various attributes of network traffic.
- **Labels (Y):** The label column contains integer values (0 or 1), where '0' denotes normal traffic and '1' indicates botnet traffic.

Given the size and scope of the dataset, it is a rich resource for examining the capabilities and limitations of logistic regression in a parallel computing environment. The dataset's complexity and volume provide a realistic and challenging test for our algorithm and computational strategies. The insights gained from this analysis are not only relevant to the specific task of network traffic classification but also contribute to the broader field of machine learning, particularly in the context of large-scale data processing.

3 Algorithm Implementation

3.1 Reading

In the context of processing the dataset for machine learning tasks, in this section we analyze the two implementations of the readFile function: the centralized version using NumPy, and the parallelized version using PySpark. This section compares these two implementations in terms of their design, functionality, and intended use cases.

Centralized Version

The *readFile* function reads data from a CSV file and converts it into a NumPy array. The function performs the following steps:

1. Reads each line of the file, performing the following operations:
   (a) Splits the line into columns.
   (b) Converts each column value to a float.
   (c) Adds the converted values as an array to a list.
2. Converts the list of float arrays into a NumPy array.

This function is utilized for loading and preprocessing CSV data, making it suitable for numerical computations and machine learning tasks.

Parallelized Version

The *readFile* function is designed for use with PySpark to process and load data from a file into a Spark RDD. The function performs the following steps:

1. Transforms each row into a tuple with two elements:
   - The first element is an array containing 11 feature values, each converted to a float.
   - The second element is a single float value representing the label.
2. Achieves this by splitting each line into columns and mapping them to the required tuple format.
3. The output is an RDD suitable for distributed processing in PySpark, ideal for large datasets and tasks requiring parallel processing.

This function leverages PySpark's capabilities to efficiently handle and process large-scale data in a distributed computing environment.

Comparison

The serial ‘readFile‘ function is suited for local execution with smaller datasets, reading CSV data directly into a NumPy array for in-memory processing. In contrast, the parallel version is designed for distributed environments, loading data into a Spark RDD and converting each row into a tuple format suitable for large-scale, parallel data processing. While the serial version is straightforward and efficient for smaller data, the parallel excels in handling large datasets across distributed computing resources.

3.2 Normalize

In this section, we examine two distinct implementations of the normalize function: one centralized using NumPy, and the other parallelized utilizing PySpark.

Centralized Version

The `normalize` function standardizes the feature columns of a NumPy array to have a mean of 0 and a standard deviation of 1, avoids division by zero for constant features, and then recombines these normalized features with their original labels. The function performs the following steps:

1. It extracts features (`X`) and labels (`y`) from the input NumPy array (`np_Xy`), where `X` includes all but the last column, and `y` is the last column reshaped for consistency.
2. The function then computes the mean and standard deviation for each feature in `X`.
3. To avoid division by zero, zeros in the standard deviation are replaced with ones.
4. The features in `X` are normalized to have a mean of 0 and a standard deviation of 1 using the NumPy vector operations to speed up the process.
5. Finally, the normalized features are recombined with the original labels to form a single array.

The function returns the normalized feature array concatenated with the label array, ensuring that each feature column has standardized values.

Parallelized Version

The parallel `normalize` function, using PySpark, takes an RDD as input and normalizes its features. It first calculates the total number of samples, through the `(count())`, and then proceeds to compute the sum and sum of squares for each feature across all samples. Utilizing the map and reduce functions, it aggregates these values to calculate the mean and variance for each feature. It then computes the standard deviation, handling cases to avoid division by zero. The mean and standard deviation are broadcast to all nodes. Finally, it applies normalization to each record (feature vector) in the RDD, adjusting each feature to have a mean of 0 and a standard deviation of 1, and returns the normalized RDD. The procedure is as follows:

1. Initiates a SparkContext, if not already existing.
2. Counts the number of samples in the input RDD.
3. Defines a function `compute_sum_and_squares` to calculate the sum and sum of squares for each feature within a record.

4. Aggregates these values across all records to compute the overall sum and sum of squares for each feature.
5. Calculates the mean and variance for each feature, then derives the standard deviation. This calculus is done following the mathematics rules explained in the section 3.3.
6. Sets any zero values in the standard deviation to one to prevent division by zero.
7. Broadcasts the mean and standard deviation to all worker nodes.
8. Defines a function `normalize_features` to apply feature scaling to each record using the broadcasted values.
9. Applies the `normalize_features` function to all records in the RDD, returning a new RDD with scaled features.

The function returns an RDD with features scaled to a mean of 0 and a standard deviation of 1, excluding the label from the scaling process.

Comparison

The centralized NumPy version operates on in-memory arrays, making it suitable for smaller datasets that can be accommodated by a single machine's resources. It is straightforward to implement and debug but is limited by the physical memory available. Conversely, the PySpark version is designed to handle normalization in a distributed manner across a cluster, ideal for large datasets that exceed the memory constraints of a single computer. While it introduces complexity due to the distributed nature of the data, it benefits from Spark's efficient handling of partitioned data and allows for scalable, parallel processing. Each implementation has its merits, with the choice between them guided by the scale of the data and the computational resources at hand.

3.3 Variance Explanation Parallelized

To decompose the summation $\sum(x_i-\mu)^2$ into its components, let's start by expanding the squared term inside the summation:

$$\sum(x_i-\mu)^2=\sum(x_i^2-2x_i\mu+\mu^2)$$

This can be broken down further into:

$$\sum(x_i-\mu)^2=\sum x_i^2-\sum 2x_i\mu+\sum\mu^2$$

Because μ is a constant (the mean of all x_i), the summation of μ^2 is simply N times μ^2, where N is the number of observations:

$$\sum\mu^2=N\mu^2$$

The term $\sum 2x_i\mu$ simplifies to 2μ times the sum of all x_i:

$$\sum 2x_i\mu=2\mu\sum x_i$$

And since the sum of all x_i divided by N is μ, we can rewrite $\sum x_i$ as $N\mu$:

$$\sum 2x_i\mu = 2\mu(N\mu)$$

$$\sum 2x_i\mu = 2N\mu^2$$

Putting it all together, we can rewrite the original summation as:

$$\sum(x_i - \mu)^2 = \sum x_i^2 - 2N\mu^2 + N\mu^2$$

Simplifying further, we combine the μ^2 terms:

$$\sum(x_i - \mu)^2 = \sum x_i^2 - N\mu^2$$

So, the variance can be calculated as:

$$\text{Variance} = \frac{\sum x_i^2}{N} - \mu^2$$

It allows us to calculate the variance in a two-step process that is more amenable to the map-reduce paradigm used in distributed systems like Apache Spark.

3.4 Training

In this section, we compare two distinct approaches for training machine learning models: the serial method, which utilizes in-memory computation ideal for smaller datasets, and the parallel method, which employs distributed computing frameworks like PySpark for handling large-scale data.

Centralized Version

The `train` function implements logistic regression using gradient descent and L2 regularization. It extracts features and labels from the NumPy array, initializes weights and bias randomly, then iteratively updates these parameters based on the computed gradients of a cost function that includes cross-entropy loss and regularization. The function outputs the optimized weights and bias after the specified number of iterations.

1. Extracts features (`X`) and labels (`y`) from the input NumPy array, determining the number of examples (`m`) and features (`k`).
2. Initializes weights (`w`) and bias (`b`) to random values, setting seed for reproducibility.
3. Iteratively computes the predictions (`y_hat`) concatenating the weight and the related column with NumPy array operations to speed up the computation.
4. Updates `w` and `b` using gradient descent while applying L2 regularization.
5. Maintains a history of the cost function to monitor the training process.
6. Returns the optimized weights, bias, and cost history after the specified number of iterations.

Parallelized Version

The `train` function in PySpark performs logistic regression training on an RDD. It iterates through a specified number of iterations, broadcasting weights and bias, computing gradients, and updating the model parameters using gradient descent. Auxiliary functions ‘compute_gradients‘ and ‘compute_cost‘ calculate gradients and cost for each data point, aiding in weight updates and cost monitoring for each iteration. The function is designed for distributed execution in a Spark environment. It conducts the following steps:

1. Establishes a Spark context and initializes model parameters: weights (`w`) and bias (`b`).
2. Iteratively performs the following for a specified number of `iterations`:
   (a) Broadcasts current `w` and `b` to the cluster nodes to improve the parallelization creating reading-only variables.
   (b) Maps the `compute_gradients` function across the RDD to compute gradients for each data point.
   (c) Aggregates the gradients and updates `w` and `b` using gradient descent.
   (d) Optionally computes and prints the cost for monitoring, using the `compute_cost` function.
3. Returns the final learned weights and bias after training.

Comparison

The serial train function performs logistic regression within a single machine’s memory using NumPy arrays, ideal for smaller datasets where the computational overhead of distribution is unnecessary. It processes data sequentially, which can lead to longer execution times for large datasets but is simpler and has fewer dependencies. In contrast, the parallel train function uses PySpark to distribute logistic regression training across multiple nodes in a cluster, handling large datasets efficiently by leveraging RDDs and Spark’s built-in parallelization.

3.5 Predict

In this section, we delve into the predict function, a key component in logistic regression models for binary classification. The function remains consistent across both centralized and parallel implementations, applying a linear combination of weights and features, followed by a sigmoid function to determine the class label.

Centralized Version

The `predict` function calculates the linear combination of features and weights, adds bias, applies the sigmoid function to derive a probability, and then classifies the input as either class 0 or 1 based on this probability (default threshold at 0.5). This function is used for making predictions on individual data examples using trained logistic regression parameters.

1. Initializes a variable `z` to zero.
2. Iterates over each element in the weight vector `w` and the feature vector `X`, calculating the product of corresponding elements and accumulating the sum in `z`.
3. Adds the bias term `b` to `z`.

4. Applies the sigmoid function to `z` to obtain the probability `p`.
5. Returns a prediction of 1 if `p` is greater than or equal to the specified `threshold` (defaulting to 0.5), otherwise returns 0.

Parallelized Version

In both the centralized and parallelized versions, the `predict` function remains the same, performing binary classification based on weights, bias, and a given feature set.

Comparison

The distinction lies in their integration within the accuracy calculation: the centralized version iterates over a NumPy array, applying ‘predict‘ to each element and tallying correct classifications for an accuracy score. In contrast, the parallel version utilizes PySpark's RDD framework, mapping the ‘predict‘ function across distributed data and aggregating correct predictions using ‘reduce‘. While the centralized approach is straightforward for smaller datasets, the parallelized method efficiently scales to handle large datasets by distributing the computation across a Spark cluster.

3.6 Accuracy

Centralized Version

The `accuracy` function evaluates the accuracy of the logistic regression model by comparing its predictions with actual labels in the provided dataset. It iterates over each data point, uses the `predict` function with the model's weights and bias, and counts the number of correctly classified examples to calculate the overall accuracy.

1. Initializes a counter `correctly_classified` to zero.
2. Iterates over each element in the NumPy array `np_Xy`:
   - Applies the `predict` function using model weights `w`, bias `b`, and the first 11 elements of the element (features).
   - Compares the prediction with the 12th element (label) and increments `correctly_classified` if they match.
3. Calculates accuracy by dividing `correctly_classified` by the total number of elements in `np_Xy`.
4. Returns the computed accuracy.

Parallelized Version

The `accuracy` function evaluates the performance of a logistic regression model on an RDD dataset by calculating its accuracy. It maps each record in the RDD to 1 or 0, based on whether the model's prediction matches the actual label, using the `predict` function. The function then sums these values using `reduce` to count correct predictions and divides this sum by the total number of records in the RDD to compute the overall accuracy.

1. Applies the `predict` function to each record in the RDD (`RDD_xy`), mapping predictions to 1 or 0 based on their correctness.
2. Uses the `reduce` function to aggregate these values, summing the number of correct predictions into `correctly_classified`.
3. Calculates the accuracy by dividing `correctly_classified` by the total number of records in `RDD_xy`.
4. Returns the computed accuracy.

Comparison

The centralized accuracy function iterates sequentially over a NumPy array, applying the predict function to each data point and counting correct classifications, making it ideal for smaller datasets processed on a single machine. In contrast, the parallel accuracy function uses PySpark to distribute the prediction process across an RDD, aggregating correct predictions using reduce. This parallel approach efficiently handles larger datasets in a distributed computing environment, showcasing scalability and optimized resource utilization typical of Spark applications.

3.7 Testing

In this section we compare the procedures used to test the centralized and the parallelized systems.

Centralized Version

The procedure to test the logistic regression model in the centralized environment involves the following steps:

1. Reads and standardizes the dataset using the `readFile` and `normalize` functions, preparing it for model training.
2. Trains the logistic regression model using the `train` function with specified parameters (number of iterations, learning rate, and lambda for regularization).
3. Calculates the accuracy of the trained model on the same dataset using the `accuracy` function.
4. Prints the obtained accuracy to evaluate model performance.
5. Computes and analyses additional metrics like using the `compute_metrics` function.

This procedure effectively demonstrates the model's capability in a centralized system, focusing on accuracy and other performance metrics. The algorithm converges during training, the cost value decreases asymptotically at each iteration. The final reported accuracy of 93.02%, reached training with 10 iterations, 1.5 as learning rate, and 0.05 as lambda regularization, suggests that the model is performing well on the given dataset.

Parallelized Version

The procedure to test the logistic regression model in the parallelized environment using PySpark involves the following steps:

1. Configures the PySpark environment for varying numbers of cores, iterating from 1 to 12 to assess performance across different parallelization levels.
2. Initializes a SparkContext with the specified number of cores for each iteration.
3. Reads and standardizes the dataset, applying the `readFile` and `normalize` functions.
4. Employs caching (`cache()`) on the standardized data to optimize RDD operations, enhancing performance by reducing data retrieval times for subsequent actions. The cache is applied before the train to avoid the same RDD to be built for each iteration. This operation helps to reduce time despite the occupancy of more memory and also avoids to change every time the order of the data.
5. Trains the logistic regression model using the cached data, then calculates its accuracy.
6. Records execution time for each iteration to analyze the performance speed-up with increased parallelization.
7. If the maximum number of cores is used (12 in this case), computes additional performance metrics.
8. Stops the SparkContext at the end of each iteration.

This testing procedure highlights the efficiency gains from caching and parallel processing in PySpark, particularly in handling large datasets. The algorithm converges during training, the cost value decreases asymptotically at each iteration. The final reported accuracy of 93.09% suggests that the model is performing well on the given dataset, independently from the level of parallelization used that modifies only the time of execution and not the final performances of the algorithm.

4 Cross Validation Implementation

In this section, we explore the implementation of cross-validation, a crucial technique for assessing model performance and determining the optimal hyperparameters, such as learning rate ("alpha") and regularization level ("lambda"). This approach allows for a detailed analysis of the bias-variance trade-off, guiding the selection of hyperparameters that best balance model accuracy and generalization. The `readFile` and the `normalize` functions are implemented with the same structure as the parallelized version, as well as the `train`, `predict` and `accuracy`.

4.1 Transform

The `transform` function in Python, using PySpark, adds a random index between 0 and ‘num_blocks - 1‘ to each record in an RDD, effectively assigning each record to a random fold for cross-validation. It achieves this through the ‘map‘ function, which applies the ‘add_index‘ nested function to each record, using the random uniform distribution `random.randint(0, num_blocks - 1)` . This operation returns a new RDD where each element is a tuple of the original record and its randomly assigned index.

1. Defines an inner function `add_index` that assigns a random index to each record. The index is a random integer between 0 and `num_blocks` - 1.
2. Applies the `add_index` function to each record in the RDD, using the `map` operation. This process adds a fold index to each record, essential for cross-validation.
3. Returns the transformed RDD, now with each record associated with a random cross-validation fold index.

4.2 Get Block Data

The `get_block_data` function in Python, using PySpark, splits an RDD into two subsets based on a specified block number. It creates training data (‘tr_data‘) by excluding records with the given block number, and testing data (‘test_data‘) by including only records with that block number, using the ‘flatMap‘ function for filtering.

1. The function accepts an RDD (`data_cv`) and a block number (`block_numb`) for splitting.
2. Defines two inner functions, `filter_block_train` and `filter_block_test`, to filter the RDD based on the block number:
   - `filter_block_train` includes records not belonging to `block_numb`, creating the training dataset.
   - `filter_block_test` includes records belonging to `block_numb`, creating the testing dataset.
3. Applies these filters using `flatMap` and returns a tuple of two RDDs: training data (`tr_data`) and testing data (`test_data`).

4.3 Testing

We measure the performance of a machine learning workflow with varying numbers of cores. To test the cross-validation implementation, We set up the Spark environment, read and normalize data, then perform 10-fold cross-validation using a logistic regression model, calculating accuracy for each fold. The script iterates over three different numbers of cores (3,9 and 12) to measure execution times, which are recorded in ‘execution_times‘. After each run, SparkContext is stopped and reinitialized with a different configuration. The script aims to evaluate the impact of parallelization on the overall execution time of the machine learning workflow.

1. Sets up the PySpark environment, configuring the number of cores for parallel processing.
2. Reads and standardizes data, implementing cross-validation with 10 folds.
3. For each fold, splits the data into training and testing sets:
   - Utilizes caching (`cache()`) to optimize the performance of RDD operations, reducing data retrieval times and improving overall efficiency.
4. Trains a logistic regression model and calculates accuracy for each fold.
5. Computes the average accuracy across all folds and measures execution time for different core configurations.

Caching is crucial in this process, as it significantly speeds up repeated access to RDDs, enhancing the performance of iterative machine learning tasks in a distributed environment. More specifically the caching operation is used 2 times:

1. after the `transform` function to speed up the `get_block_data` function.
2. after the `get_block_data` function to speed up the computation of the iterative process in the `train`.

Due to computational constraints we manually evaluated different values for the hyperparameters iterations, learning_rate and lamda_reg. The best results were achieved with 10 iterations, learning_rate = 1.5 and lamda_reg = 0.05

5 Experiments

In this section we are going to visualize and explain the effects of the amount of Spark workers assigned to our program, both on a execution time and speed-up perspective. Additionally, we will examine various classification metrics to gain a more in-depth evaluation of the classifier's performance.

5.1 Classification Metrics

The table 1 presents a detailed overview of the performance metrics. The evaluation is conducted at various threshold levels, each impacting the model's sensitivity to detecting botnet connections.

Threshold	TP	FP	TN	FN	Accuracy	Precision	Recall	F1
0.1	498734	220063	279937	1266	0.778671	0.693845	0.997468	0.818404
0.2	495443	133998	366002	4557	0.861445	0.787116	0.990886	0.877324
0.3	489957	89783	410217	10043	0.900174	0.845132	0.979914	0.907546
0.4	482392	61153	438847	17608	0.921239	0.887492	0.964784	0.924526
0.5	471848	40850	459150	28152	0.930998	0.920323	0.943696	0.931863
0.6	456648	25813	474187	43352	0.930835	0.946497	0.913296	0.929600
0.7	427903	14772	485228	72097	0.913131	0.966630	0.855806	0.907848
0.8	371003	6698	493302	128997	0.864305	0.982266	0.742006	0.845397
0.9	256147	1713	498287	243853	0.754434	0.993357	0.512294	0.675974

Table 1: Performance Metrics at Different Thresholds

True Positives and False Positives:

At lower thresholds, such as 0.1 and 0.2, we observe a high number of True Positives (TP), indicating the model's strong ability to identify botnet connections. However, this comes with a significant number of False Positives (FP), suggesting a tendency to misclassify normal connections as botnet at these levels.

True Negatives and False Negatives:

As the threshold increases, there is a notable decrease in FP, enhancing the model's precision in identifying normal traffic accurately. However, this improvement is accompanied by an increase in False Negatives (FN), indicating missed botnet detections.

Accuracy:

The accuracy of the model shows a general increase with the threshold. It starts from 0.778671 at a threshold of 0.1 and reaches its peak at 0.930998 for a threshold of 0.5, suggesting that the model achieves its best overall performance at this point.

Precision and Recall:

Precision shows a steady improvement with increasing thresholds, indicating fewer false alarms in botnet detection. Recall, however, decreases, highlighting a trade-off between precision and the ability to detect all actual botnet connections.

F1 Score:

The F1 Score, which balances precision and recall, suggests that the model performs optimally at a middle-range threshold. At threshold levels either too low or too high, the balance between recall and precision is disrupted, leading to less effective overall performance.

ROC Curve:

The ROC curve (Figure 2) demonstrates the logistic regression model's performance in classifying botnet connections, with a steep initial rise indicating a high True Positive Rate against a low False Positive Rate for lower thresholds. This initial performance suggests strong sensitivity and specificity. The curve stabilizes at higher thresholds marking the optimal balance between detecting true botnet connections and minimizing false alarms. The area under the curve is substantial, indicating excellent model discrimination ability. Overall, these characteristics suggest that the results are very good, and the model is well-suited for identifying botnet activity within network traffic.

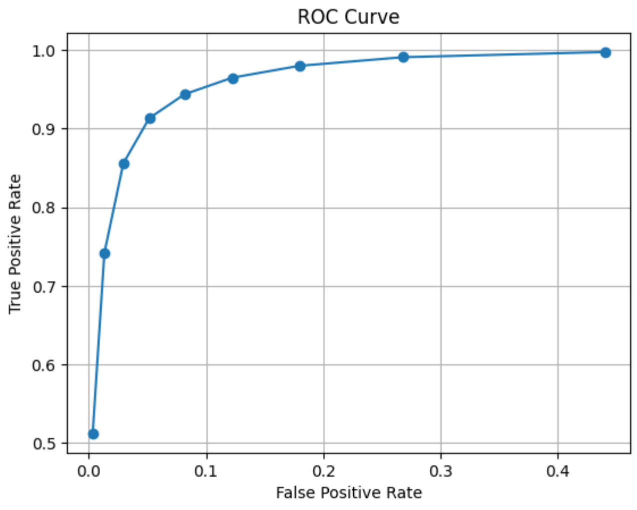

Figure 2: ROC Curve for the logistic regression model

Precision-Recall Curve

The Precision-Recall curve shown in Figure 3 offers a detailed look at the trade-off between precision and recall for our logistic regression model when classifying botnet connections. The curve starts with high precision at lower recall levels, indicating that when the model predicts a connection to be a botnet, it is highly likely to be correct. However, as recall increases, precision gradually declines, suggesting a decrease in confidence as the model attempts to identify all positive instances. The model maintains a relatively high precision until a steep drop as recall approaches 1, which is typical due to the increased number of false positives in the effort to capture all true positives.

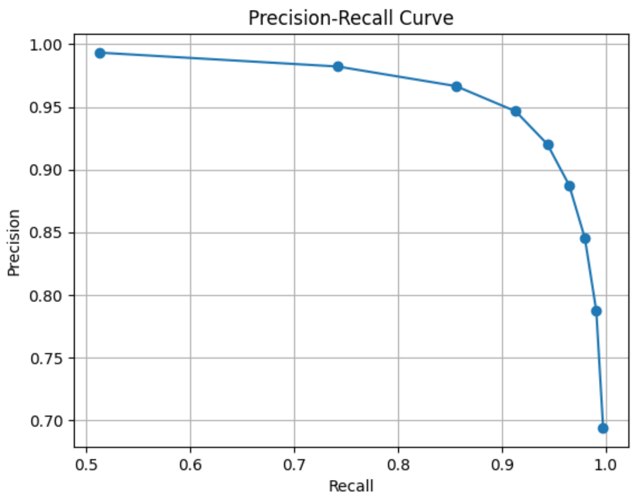

Figure 3: Precision-Recall Curve for the logistic regression model

5.2 Parallelization Analysis

For this part, we have run 12 times our model, each time with an increasing number of workers assigned to it; so starting with a single workers up to a 12 workers assignment. It's important to say that the physical machine on which these experiments have been executed on a Apple M2 Pro, 10 cores processor, because we have tested also the behaviors of the system using more workers than cores available.

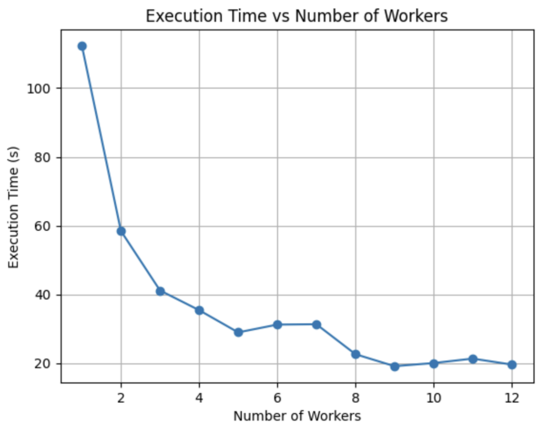

Figure 4: Performance Curve

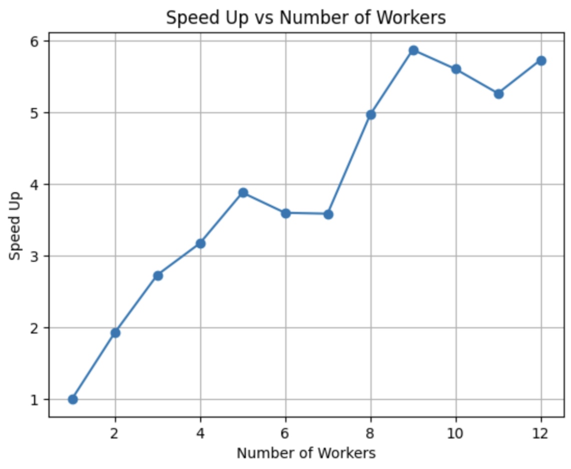

Figure 5: Speed Up Curve

From the Figure 4 we can identify 4 macro behaviours:

- **Deep Decrease (1 to around 5 workers):** Initially, when the number of workers increases from 1 to around 5, there is a steep decline in execution time. This suggests that the program benefits significantly from parallelization. The workload is being divided amongst more processors, which means each worker has less to do, and tasks can be carried out concurrently.
- **Gradual Leveling Off (Around 5 to 7 workers):** As the number of workers increases beyond 4, the decline in execution time starts to level off. The benefit of adding more workers decreases; a possible reason could be caused by the overhead of managing more workers and the inter-worker communication may start to outweigh the performance benefits.
- **Plateau (Beyond 7 workers):** the execution time seems to plateau. This indicates that adding more workers does not result in further significant reductions in execution time. It's possible that all the parallelizable parts of the program are already being executed in parallel to the maximum extent.

For the Speed curve graph, Figure 5,we can observe this behavior the graph begins with a consistent increase as the number of workers rises from 1, indicating significant performance gains with each additional worker and a highly parallelizable workload. However, as the worker count approaches 5/6 workers, the rate of speed-up gain slows down, marking the onset of diminishing returns likely due to increased communication overhead, synchronization delays, and resource contention. Notably, around 8 workers, there's an unexpected spike in speed-up, hinting at a super-linear performance increase possibly because the workload benefits from a more efficient use of the system's cache memory or because the dataset fits entirely in the cumulative memory of the workers, thus reducing I/O overhead. Lastly, the system decreased the performance when the number of workers exceeded the number of cores.

5.3 Parallel cross-validation

In the cross-validation process, due to the time-intensive nature of the operations, we limited our experiments to three distinct combinations, specifically utilizing 3, 9, and 12 workers. These numbers were deliberately chosen to reflect three different scenarios:

1. A significantly smaller number of workers compared to the available cores.
2. A number of workers approximately equal to the number of available cores.
3. A greater number of workers than the available cores.

This approach led to the generation of two plots, Figure 6 and Figure 7, demonstrating the system's performance under each of these conditions.

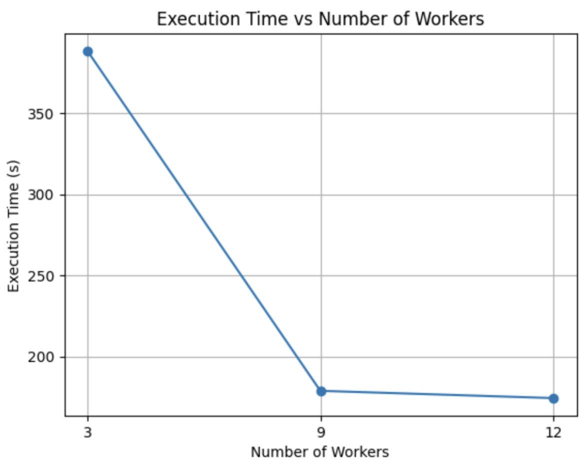

Figure 6: Performance Curve Cross Validation

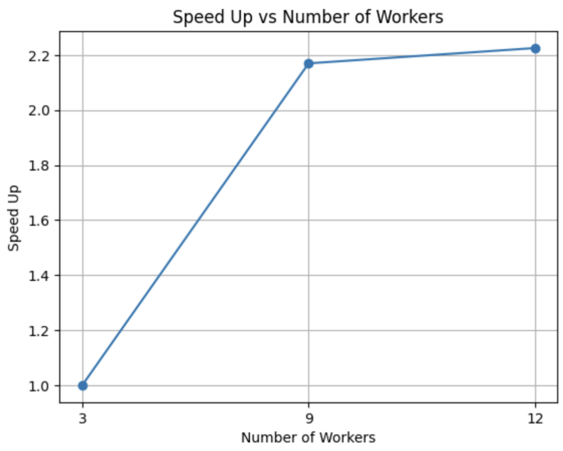

Figure 7: Speed Up Curve Cross Validation

Also in this case, we have a very sharp decreasing curve in the execution time (and so a sharp increasing in the speed-up curve) as in the previous case. Exactly like in the previous case, once we reach 9 workers adding more workers does not result in a big improvement.

5.4 Conclusions

In conclusion, the logistic regression model developed in this project, implemented using PySpark, demonstrates a strong ability to identify botnet connections within network traffic. The model's performance varies with different threshold levels, with the best overall performance observed at a threshold of 0.5. In terms of parallelization, the model benefits significantly from an increase in the number of workers up to around 5, after which the decline in execution time starts to level off and eventually plateaus. This suggests that the program is effectively utilizing parallelization to speed up computation, but there is a limit to the performance benefits that can be gained from adding more workers. The use of PySpark for parallelization has been instrumental in handling large datasets and achieving efficient computation. The final reported accuracy of 93.09% suggests that the model is performing well on the given dataset, independently from the level of parallelization used that modifies only the time of execution and not the final performances of the algorithm.