## Massively Parallel Machine Learning

A PROJECT REPORT

submitted by

**TOMMASO BARONI**
**LUCA BESTAGNO**
**FRANCESCO MATTIOLI**

on

# Parallel Implementation and Evaluation of Logistic Regression Alghorithm


POLITÉCNICA

**Master Universitario en Innovación Digital**
**Escuela Tecnica Superior de Ingenieros Informaticos**
December 2023

# Contents

# 1 Introduction

In the evolving landscape of machine learning and data analysis, the efficient handling of large datasets is becoming increasingly important. Supervised machine learning algorithms, particularly logistic regression, play an important role in classification tasks across diverse domains. This report presents a comprehensive study and implementation of a logistic regression classifier, adapted for a parallel computing environment using Apache Spark and Python.

The focus of our study lies in the implementation and evaluation of logistic regression in a parallelized setting. The primary dataset used for this purpose is "botnet_tot_syn_l.csv," a labeled dataset significant for its real-world implications in network security. This dataset, comprising 12 columns — 11 features and 1 label column — provides a rich ground for analyzing traffic data and distinguishing between normal traffic and botnet activities.

We begin by outlining the logistic regression model, a fundamental tool in the machine learning models for binary classification tasks. The logistic regression model, especially when combined with gradient descent for optimization, offers a robust framework for understanding and predicting categorical outcomes. Images illustrating the logistic regression model and gradient descent will be included to provide a visual comprehension of these concepts.

Subsequently, the report delves into the design and implementation of key functions — 'readFile', 'normalize', 'train', and 'accuracy' — in both centralized and parallelized formats. In the centralized approach, we use numpy arrays, while in the parallelized version, we leverage Apache Spark's Resilient Distributed Datasets (RDDs). This dual approach allows for a comparative analysis of performance in different computational settings.

Special emphasis is placed on the process of normalization, training procedure via gradient descent, and the accuracy measurement of the logistic regression model. The training process is detailed with gradient descent iterations, highlighting the convergence of the cost function, which is crucial for the effective training of the model.

The report also introduces a procedure for cross-validation, essential for estimating the performance of the model and selecting the best set of hyperparameters. The trade-off between bias and variance, a critical aspect in model performance, is analyzed through variations in learning rate and regularization rate.

An important aspect of this study is the analysis of parallelization effects, especially in the context of varying the number of cores. This involves a detailed examination of the performance and speedup curves, providing insights into the scalability and efficiency of the logistic regression model in a parallel computing environment.

In conclusion, the report will present a comprehensive analysis of machine learning error metrics such as cost error, accuracy, precision, recall, f1-score, confusion matrix, and ROC and Precision-Recall curves and also the performances of the model calculating the

speedup rate using different cores. These metrics will be critically discussed in the light of the results obtained, offering a thorough understanding of the model's performance in both centralized and parallelized contexts.

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

Figure 1: Gradient Descent Algorithm

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

Figure 2: Cross-Entropy cost fuction

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

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

Figure 3: Gradient Descent Derivatives

# 2 Dataset Description

The dataset fundamental to our study, labeled "botnet_tot_syn_l.csv," is an extensive collection of network traffic data, meticulously structured to facilitate the implementation and evaluation of a supervised machine learning algorithm, specifically logistic regression, in a parallelized computing environment using Spark and Python.

## Characteristics of the Dataset:

- **Format and Structure**: The dataset is presented in CSV (Comma-Separated Values) format, ensuring ease of use and compatibility with various data processing tools.
- **Dimensions**: It consists of 12 columns in total. Out of these, 11 columns represent the features or dimensions (X) of the data, and the remaining single column corresponds to the labels (Y).
- **Feature Description**: The 11 feature columns comprise a diverse range of network attributes, each encoded as a floating-point number. These features are critical in differentiating normal traffic from botnet activities.
- **Label Encoding**: The label column (Y) designates the type of network traffic, with '0' representing normal traffic and '1' indicating the presence of botnet traffic. This binary labeling is crucial for the classification task at hand.
- **Number of Examples**: The dataset encompasses '1.000.000' rows, where each row represents an individual example or record of network traffic.

## Data Representation in Spark:

- **RDD Format**: In the Spark environment, the dataset is represented as a Resilient Distributed Dataset (RDD). Each row of the file corresponds to one record in the RDD.
- **Record Structure**: Each record in the RDD is a tuple (X, y). Here, "X" represents an array containing the 11 feature values of an example, while "y" is the label, either 0 or 1.

In summary, the "botnet_tot_syn_l.csv" dataset serves as the foundation for this report, enabling a detailed exploration and evaluation of logistic regression in a parallelized computing context. Its structured and comprehensive nature makes it an invaluable resource for understanding and optimizing machine learning models in the realm of network security.

# 3 Algorithm Implementation

## Comparison of Serial and Parallel `readFile` Functions

The `readFile` function serves as the initial data processing step in logistic regression, designed to read and format the dataset. We compare the serial and parallel versions of this function, highlighting the parallelization techniques employed in the latter.

### 1. Serial Implementation: `readFile`

The serial version of `readFile` is intended for use in a non-distributed, single-machine environment.

- **Function Signature**: `def readFile(filename):`
  - `filename`: The path to the dataset file.
- **Workflow**:
  - Reads the file using standard Python I/O, processing each line to extract features (X) and the target (Y).
  - Splits each line by comma, converts feature values to floats and the target value to an integer.
  - Combines the features and target into a NumPy array, facilitating array operations and mathematical computations.
- **Return Value**: Returns the combined dataset as a NumPy array.

### 2. Parallel Implementation: `readFile`

The parallel version of `readFile` leverages Apache Spark's RDD for distributed data processing.

- **Function Signature**: `def readFile(filename):`
  - `filename`: The path to the dataset file.
- **Workflow**:
  - Utilizes Spark's `sc.textFile` to read the dataset file into an RDD, enabling parallel processing across the cluster.
  - Applies a chain of `map` transformations to split each line into 11 features and target, converting them into the required data types, floats for the features and integer for the target.

  - Each element in the RDD is transformed into a tuple, comprising a list of features and the target value.

- **Return Value**: Returns the processed dataset as an RDD of tuples.

**Summary**

The key distinction lies in the use of RDDs and parallel transformations, enabling distributed data reading and processing. This adaptation is crucial for logistic regression tasks that involve large datasets, necessitating efficient data processing and management in a distributed computing environment.

## Comparison of Serial and Parallel `normalize` Functions

The `normalize` function plays a critical role in the preprocessing phase of logistic regression, standardizing the dataset for consistency in model training. We examine the serial and parallel versions of this function, with an emphasis on the parallelization techniques in the latter.

### 1. Serial Implementation: `normalize`

The serial version of `normalize` operates in a centralized computing environment.

- **Function Signature**: `def normalize(data):`
  - `data`: The dataset (numpy array) to be normalized.
- **Workflow**: The function iterates over each feature (column) of the dataset, excluding the last column. For each feature, it calculates the mean by summing all the values and dividing by the number of values (len(column)). It then calculates the standard deviation by taking the square root of the average of the squared differences from the mean. The calculated mean and standard deviation for each feature are appended to the means and stds lists, respectively. The lists means and stds are converted into NumPy arrays to facilitate vectorized operations in the next step. The function again iterates over each feature of the dataset and it subtracts the mean of the feature from each element of the feature and divides by the standard deviation
- **Return Value**: Returns the normalized dataset.

### 2. Parallel Implementation: `normalize`

The parallel version of `normalize` leverages Apache Spark's RDD for distributed data processing.

- **Function Signature**: `def normalize(RDD_Xy):`
  - `RDD_Xy`: The dataset (Spark RDD) to be normalized.
- **Workflow**:
  - The function begins by computing the means and standard deviations for each feature across the distributed dataset. This is achieved through a combination of Spark's `map` and `reduce` operations, applied in `compute_mean_stds`. Each node in the cluster processes a subset of the data, calculating partial sums and sums of squares, which are then aggregated to obtain the global mean and standard deviation for each feature. To enhance performance, the feature vector x is converted to a Numpy array within the readFile function before any computations. Numpy arrays are designed for high-performance matrix and vector operations, enabling efficient numerical computation that is significantly faster than native Python
  - Following this, the function normalizes each data point in the RDD. The normalization (subtracting the mean and dividing by the standard deviation) is performed in parallel across the cluster using a `map` transformation. This step ensures that each feature is standardized, accounting for variations in scale and distribution, which is crucial for effective logistic regression analysis.
- **Return Value**: Returns the normalized RDD.

**Summary**

By utilizing Spark's map and reduce functions, computations of means and standard deviations are performed in parallel across nodes. This reduces the computational burden on a single worker and leverages the processing power of multiple workers.

## Comparison of Serial and Parallel `train` Functions

The `train` function is crucial in logistic regression for learning model parameters. Here, we compare the in-depth workflows of both the serial and parallel implementations, with particular attention to the parallelization techniques in the parallel version. Before going in the detail with the two version of the train functions here follows the explaination of the sigmoid and the linear combination of the weights and features. The sigmoid function, which is applied to the linear combination of features and weights, is defined as:

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

where $z$ is the linear combination of the weights and features plus the bias term:

$$z = \mathbf{w}^\top \mathbf{x} + b \tag{2}$$

Here, $\mathbf{w}$ represents the weight vector, $\mathbf{x}$ represents the feature vector, and $b$ represents the bias term.

### 1. Serial Implementation: `train`

The serial `train` function is designed for execution in a single-machine environment.

- **Function Signature**: `def train(data, iterations, learning_rate, lambda_reg):`
  - `data`: The dataset.
  - `iterations`: Number of iterations for gradient descent.
  - `learning_rate`: Learning rate.
  - `lambda_reg`: Regularization parameter.
- **Workflow**:
  - Initializes weights and bias randomly, sets up a loop for the specified number of iterations.
  - In each iteration, calculates the gradient of loss with respect to each feature and the bias. This involves iterating over each data point, computing the sigmoid of the linear combination of features and weights, and determining the error based on the actual label.
  - Aggregates these gradients across the entire dataset to update the model parameters, adjusting weights and bias in the direction that minimizes the cost function.
  - Regularization is applied to prevent overfitting, modifying the weight update rule to include a term proportional to the weights themselves.
  - The cost (error) is calculated in each iteration to monitor the convergence of the algorithm.
- **Return Value**: Outputs the optimized weights and bias.

### 2. Parallel Implementation: `train`

The parallel `train` function leverages Apache Spark's RDDs for distributed computing.

- **Function Signature**: `def train(RDD_Xy, iterations, learning_rate, lambda_reg):`
  - `RDD_Xy`: The dataset (Spark RDD).
  - `iterations`: Number of iterations for gradient descent.
  - `learning_rate`: Learning rate.
  - `lambda_reg`: Regularization parameter.
- **Workflow**: In the parallel train function, the setup begins similarly to the serial version with the initialization of weights, bias, and the iteration loop. However, the parallel approach leverages Spark's distributed computing capabilities to enhance the logistic regression training process:

  - **Parallel Gradient Computation**: During each iteration, a `map` transformation is used to calculate the gradient and cost contribution for each data point across the cluster. Here, the computeGradientsAndCost function applies the sigmoid and linearCombination functions to compute predicted values (y_hat), errors, and individual gradients (dw) for each feature, as well as the bias gradient (db). This step utilizes the cluster's nodes to perform calculations concurrently, resulting in a significant speed-up compared to the serial computation.
  - **Aggregation of Results**: Following the gradient computation, Spark's reduce action aggregates the results from all nodes. It sums up the individual gradients and cost contributions to yield the total gradients and cost. This distributed aggregation harnesses the full power of the cluster, ensuring a swift combination of intermediate results for global parameter updates.
  - **Parameter Updates**: The model's parameters, which include weights and bias, are updated in a collective manner based on the aggregated gradients. This is done using the standard gradient descent algorithm, where the regularization term (lambda_reg) is also factored into the weight updates to prevent overfitting.
  - **Cost Calculation**: Alongside gradients, the cost calculation is parallelized. Each node computes a portion of the total cost, which, when combined, provides a comprehensive measure of the model's performance after each iteration.

- **Return Value**: Returns the trained weights and bias, optimized over the distributed dataset.

**Summary**

The parallel version employs Spark's `map` and `reduce` transformations to compute and aggregate gradients across the cluster, leading to substantial efficiency gains. This parallel gradient computation allows simultaneous processing of data points across different nodes, effectively reducing the time complexity compared to the serial approach, which processes data points sequentially. Finally, the cost calculation, which is crucial for evaluating model performance, is also parallelized, ensuring that the algorithm's convergence can be monitored effectively after each iteration.

## Comparison of `predict` Functions

The `predict` function in logistic regression is used to make predictions based on learned model parameters. We compare the workflows of the serial and parallel versions of the `predict` function.

### 1. Serial Implementation: `predict`

The serial version of `predict` is straightforward and is typically used in a non-distributed environment.

- **Function Signature**: `def predict(w, b, x):`
  - `w`: The weight vector.
  - `b`: The bias term.
  - `x`: A single data point (feature vector).
- **Workflow**:
  - Calculates the linear combination of features and weights, adding the bias term this is done by iterating on each element.
  - Applies the sigmoid function to this linear combination to obtain the probability (`y_hat`).
  - Makes a binary prediction (1 or 0) based on the threshold of 0.5.
- **Return Value**: Returns the binary prediction for the input data point.

### 2. Parallel Implementation: `predict`

The parallel version of `predict`, while structurally similar to the serial version, is typically part of a larger parallelized pipeline in a distributed computing environment.

- **Function Signature**: Same as the serial version.
- **Workflow**:
  - Follows the same steps as the serial version: computes the linear combination of weights and features, adds the bias, applies the sigmoid function, and makes a binary prediction.
- **Return Value**: Identical to the serial version, returning a binary prediction.

### Summary

Both versions of the `predict` function essentially perform the same operation of making predictions based on the logistic model.

## Comparison of Serial and Parallel `accuracy` Functions

The `accuracy` function is pivotal in logistic regression for evaluating the performance of the trained model. Below, we compare the serial and parallel versions of this function, with special emphasis on the parallelization techniques in the parallel version.

### 1. Serial Implementation: `accuracy`

The serial version of `accuracy` is used in a single-machine environment.

- **Function Signature**: `def accuracy(w, b, data):`
  - `w`: The weight vector.
  - `b`: The bias term.
  - `data`: The dataset (numpy array).
- **Workflow**:
  - Separates features and true labels from the dataset.
  - Iterates over each data point, using the `predict` function to obtain predictions.
  - Compares predictions with true labels, counting the number of correct predictions.
  - Calculates the accuracy as the proportion of correct predictions to total data points.
- **Return Value**: Returns the accuracy of the model on the given dataset.

### 2. Parallel Implementation: `accuracy`

The parallel version of `accuracy` leverages Apache Spark's RDDs for distributed computation.

- **Function Signature**: `def accuracy(w, b, RDD_Xy):`
  - `w`: The weight vector.
  - `b`: The bias term.
  - `RDD_Xy`: The dataset (Spark RDD).
- **Workflow**:
  - Utilizes Spark's `map` transformation to apply the `predict` function to each data point in parallel. This step computes a binary value (1 or 0) for each prediction based on its correctness.
  - Aggregates these binary values using `reduce`, summing up the total number of correct predictions.
  - Divides the total number of correct predictions by the count of the RDD to calculate the accuracy.
  - This parallel processing of predictions and aggregation significantly enhances computational efficiency, especially for large datasets.
- **Return Value**: Outputs the model's accuracy computed over the distributed dataset.

**Summary**

In the serial version, the accuracy computation iterates over each data point, predicts outcomes, and compares them with actual labels to calculate accuracy as the ratio of correct predictions. In contrast, the parallel version leverages Spark's distributed computing. It simultaneously predicts and assesses correctness across data points using the map function across cluster nodes. The reduce operation aggregates these results, and the total correct predictions divided by the RDD's size gives the model's accuracy.

## Metrics

In this section of our study, we have evaluated the performance of our logistic regression model using several metrics: accuracy, precision, recall, F1 score, and confusion matrix at different thresholds.

The **accuracy** metric, ranging from 0.774 to 0.932, indicates the proportion of true results among the total number of cases examined. Precision, which assesses the model's ability to identify only relevant instances, spanned from 0.693 to 0.993, signifying a high level of exactness in the model's predictions.

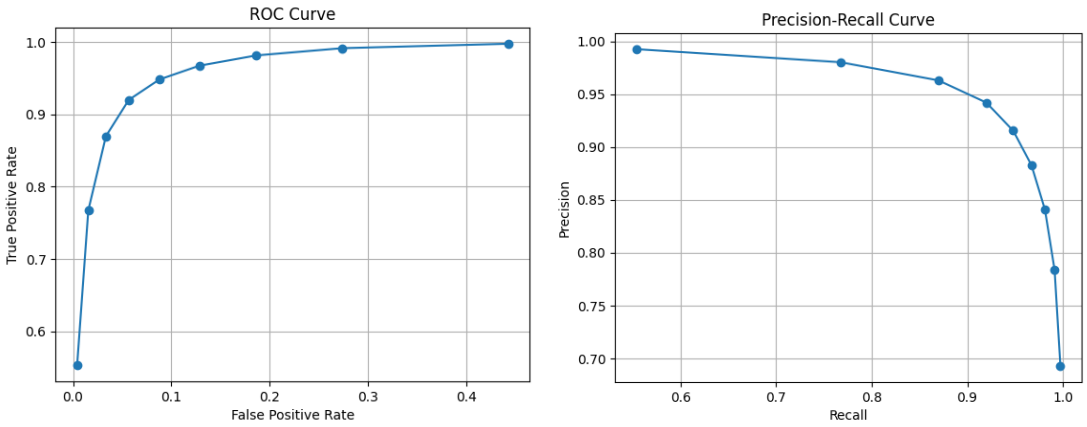
**Recall** measures the model's ability to find all relevant instances within a dataset. Our model's recall varied between 0.553 and 0.997, indicating that at a lower threshold, the model was almost perfect in identifying all relevant instances. However, as the threshold increased, it started to miss more positive cases.

The **F1 score**, which balances precision and recall, was highest (0.931) at a threshold of 0.5 and significantly lower (0.711) at 0.9. This suggests that the model's best balance between precision and recall was achieved at the median threshold.

The **confusion matrix** provides a detailed breakdown of the model's performance, with true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). At the lower threshold of 0.1, the model had a high number of true positives and true negatives but also a considerable amount of false positives, which decreased with a higher threshold.

The **Receiver Operating Characteristic (ROC)** and **Precision-Recall** curves visualize the trade-offs between true positive rate and false positive rate, and precision and recall, respectively. Both plots exhibit the model's performance across different thresholds, with the ROC curve showing a strong true positive rate at low false positive rates and the Precision-Recall curve demonstrating a high precision even with decreasing recall, Fig. 4.

Despite experimenting with various iterations and learning rates, there were no significant changes in the results. Therefore, the study opted for 10 iterations and a learning rate of 1.5 and a regularization term of 0, indicating that further adjustments to these parameters did not yield improvements in model performance. This suggests that the chosen iteration count and learning rate were sufficient for the convergence of the model on this particular




(a) ROC curve (b) Precision-Recall Curve

Figure 4: Metrics representation

dataset.

# 4 Cross Validation

Cross-validation is a statistical method used to estimate the performance of a predictive model. It helps ensure that the model is generalizable and performs well on unseen data. The method involves dividing the dataset into a fixed number of "folds" or "blocks," then systematically using one block at a time for testing the model while the remaining blocks are used for training.

In our project, instead of randomly shuffling all rows of our dataset and then splitting it into blocks, we took a different approach. We decided to assign an index between 1 and 10 to each row uniformly. This assignment allowed us to efficiently create 10 distinct blocks for cross-validation purposes. The uniform assignment of indices ensures that each block is representative of the whole, reducing the chance of bias that might occur with random shuffling, especially when certain patterns or sequences within the data can influence the learning process.

The function `getUniformNumber` is responsible for generating a uniform random number within the specified range of indices. It guarantees that every record in our dataset receives an index that's uniformly chosen between 1 and 10. This method maintains the diversity and distribution of the dataset across all blocks.

The `assignUniformBlocks` function takes our dataset as an RDD (Resilient Distributed Dataset) and assigns a block index to each record. It uses the map transformation provided by Spark to apply the `assignBlock` function to every record. The result is an RDD with records augmented with a block index, effectively partitioning the dataset into 10 blocks.

With the `crossValidate` function, we perform the actual cross-validation process. Each iteration of the function focuses on a different block of the dataset, treating it as the test set, while the remaining blocks are merged to form the training set. The dataset is processed using the flatMap method, replicating the function of the index-based filter. The model is trained on the training set, and its performance is evaluated on the test set. The accuracy of the model is recorded for each block.

After all blocks have been used as a test set, we calculate the average accuracy across all iterations. This average gives us a robust estimate of our model's performance. It minimizes the impact of any anomalies or peculiarities in the data, providing a more reliable measure of the model's predictive power.

In conclusion, our uniform cross-validation method ensures that each subset of data receives equal representation in both the training and testing phases, enhancing the model's ability to generalize well to new data. This methodological choice aligns with the overarching goal of our project: to develop a model that not only performs well on our current data but also maintains its accuracy and reliability in the face of new, unseen data.

# 5 Experiments

In this section we will delve into our model performances with different workers, in particular we will analyze the execution speed compared with the number of workers and the speedup ratio, caluclated as $\frac{t_{1\text{worker}}}{t_{n\text{workers}}}$ . In the computational experiments, logical cores beyond the physical ones were utilized. This meant engaging the system's hyper-threading capabilities to parallelize operations further. While hyper-threading can increase the throughput of processes by leveraging idle CPU cycles, it's not always a guarantee of performance improvement, particularly for compute-intensive tasks that can already fully engage physical cores. The choice to use logical cores was a deliberate one to explore the potential for performance gains in the context of the experiments conducted.

## Performances

### Performances of logistic regression

In the performance analysis of your parallel computing experiment, the relationship between execution time and the number of workers as well as the speedup achieved provides a clear indication of how well the system scales with additional resources.

The Performance Curve plot displays a steep decline in execution time as the number of workers increases from 1 to 2, indicating that parallel processing can significantly reduce the time taken for tasks. This decline continues, though less sharply, until it stabilizes around 6 workers, suggesting that adding more workers up to a certain point can enhance performance, but beyond that, the benefits begin to plateau.

The Speedup Curve complements this analysis by showing the efficiency gains from parallel processing. Initially, the speedup increases linearly, which is the ideal scenario when doubling the workers cuts the execution time in half. However, after 4 workers, the speedup curve shows some fluctuations. This could be due to several factors, including the overhead of managing additional workers or the fact that the tasks being executed may not be perfectly parallelizable. Interestingly, there is a notable increase again at 8 workers, which could indicate that for specific tasks or data segments, having more workers can still provide a performance boost.

Overall, these results demonstrate the efficiency of parallel processing in reducing computation time. It also underscores the point of diminishing returns, where adding more workers does not necessarily lead to proportional decreases in execution time. This is a critical consideration for cost and resource management in distributed computing environments, where the goal is to achieve the best performance with the optimal number of workers.

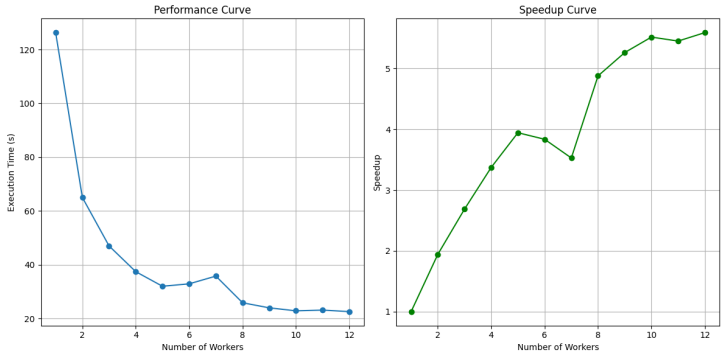



Figure 5: Performances of logistic regression with different workers

**Performances of logistic regression**

The performance curve indicates a sharp decrease in execution time as the number of workers increases from 3 to 6, suggesting that distributing tasks across more workers significantly speeds up computation. However, beyond 6 workers, the reduction in execution time plateaus, showing marginal gains. This could be due to the overhead of managing more workers or the limitations of parallel task distribution.

The speedup curve shows a steady increase in speedup as more workers are added, with a peak speedup occurring with 12 workers. This suggests that the system benefits from additional workers up to a point, after which resource management or data communication may become a bottleneck, preventing further speedup.

The choice of workers [3, 9, 12] for cross-validation reflects a need to optimize resource usage and reduce time spent on validation, while still gaining insight into the system's scalability. The absence of workers between 3 and 9 in the speedup curve is due to the desire to observe the system's performance under low, medium, and high loads without the exhaustive time investment required for every incremental increase in the number of workers.

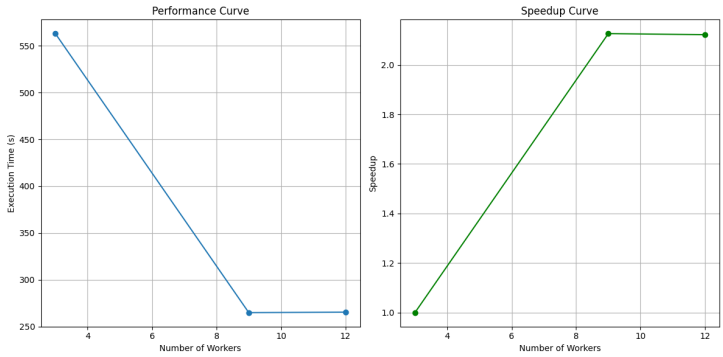



Figure 6: Performances of the cross validation in logistic regression

# 6 Conclusion

In conclusion, this report presents a comprehensive study of logistic regression adapted for parallel computing using Apache Spark and Python. It emphasizes the model's effectiveness in classifying network traffic data, focusing on distinguishing normal traffic from botnet activities. The report details the implementation of logistic regression functions in both centralized and parallelized formats, highlighting the benefits of parallelization. It also explores the impact of varying the number of cores on performance and speedup. The parallel approach showcases significant efficiency gains in computation time, especially for large datasets, underlining the importance of optimal resource management in distributed computing environments.