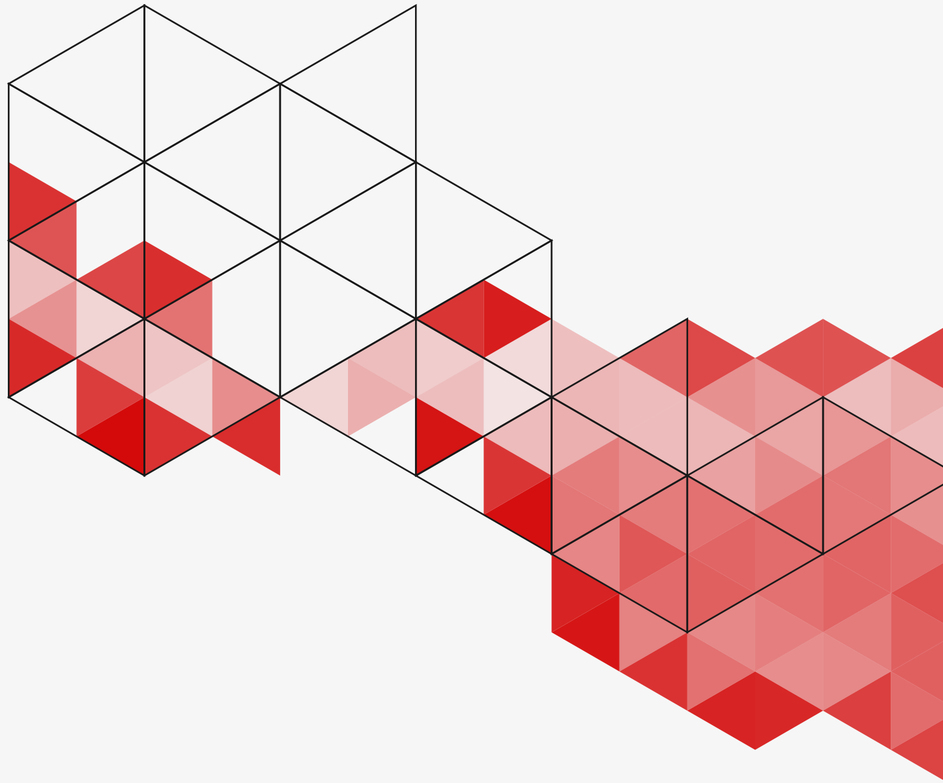
Massively Parallel Machine Learning

Parallel Implementation and Evaluation of Logistic Regression Algorithm



Nicola Cecere | Luca Petracca | Alessandro Rossi

The Goal

Project Goal: We analyze the use of the Logistic Regression algorithm for supervised classification tasks using Gradient Descent, and its application in a massively parallel processing environment using Spark and Python

Dataset: ’botnet tot syn 1.csv’, which comprises network traffic data, categorized as either normal or botnet-related

Process: data preparation (readFile), data normalization (normalize), model training (train), and model evaluation (accuracy)

Algorithm Implementation

readFile: centralized

1.Reads each line of the file, performing the following operations

- Splits the line into columns.
- Converts each column value to a float.
- Adds the converted values as an array to a list.

2. Converts the list of float arrays into a NumPy array.

readFile: parallelized

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.

readFile: parallelized

textFile(filename) →

[x_1_1 ... x_1_11]	y_1
⋮	
⋮	
[x_n_1 ... x_n_11]	y_n

→ normalize(np_Xy)

normalize: centralized

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 each feature's mean and standard deviation 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.

NumPy

normalize: parallelized

1. Count the number of samples in the input RDD.
2. Calculate the sum and sum of squares for each feature within a record.
3. Aggregates these values across all records to compute the overall sum and sum of squares for each feature.
4. Calculate the mean and variance for each feature, and derive the standard deviation.
5. Sets any zero values in the standard deviation to one to prevent division by zero.
6. Broadcasts the mean and standard deviation of all worker nodes.
7. Apply feature scaling to each record using the broadcasted values.
8. Applies the normalize features function to all records in the RDD, returning a new RDD with scaled features.

normalize: parallelized

map →

[x_1_1 ... x_1_11]	[x_1_1 ... x_1_11]^2
⋮	⋮
⋮	⋮
[x_11_1 ... x_11_11]	[x_11_1 ... x_11_11]^2

reduce →

sum[x_1 ... x_11]	sum[x_1 ... x_11]^2

$$\mu = \frac{\sum X_N}{N}$$

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

$$\sigma = \sqrt{\frac{\sum x_i^2}{N} - \mu^2}$$

normalize: parallelized

[x_1_1 ... x_1_11]	y_1
⋮	
⋮	
[x_n_1 ... x_n_11]	y_n

Normalized RDD

$$\sigma = \sqrt{\frac{\sum x_i^2}{N} - \mu^2}$$

Cached to optimize performance!

train: centralized

1. Extract 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 the seed for reproducibility.

3. Iteratively computes the predictions (y_hat).

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.

NumPy

train: parallelized

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:

- Broadcasts current w and b to the cluster nodes to improve the parallelization creating reading-only variables.
- Maps the compute gradients function across the RDD to compute gradients for each data point.
- Aggregates the gradients and updates w and b using gradient descent.
- Optionally computes and prints the cost for monitoring, using the compute cost function.

3. Returns the final learned weights and bias after training

train: parallelized

map →

[dw_1_1 ... dw_1_11]	db_1
⋮	⋮
⋮	⋮
[dw_n_1 ... dw_n_11]	db_n

reduce →

[dw_1 ... dw_11]	db

$$dw_1 = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) \times x_1^{(j)} + \frac{\lambda}{k}w_1$$

$$dw_2 = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) \times x_2^{(j)} + \frac{\lambda}{k}w_2$$

$$\vdots$$

$$dw_k = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) \times x_k^{(j)} + \frac{\lambda}{k}w_k$$

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

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

predict: serial/parallel

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

NumPy

accuracy: centralized

1. Initializes a counter correctly classified to zero.

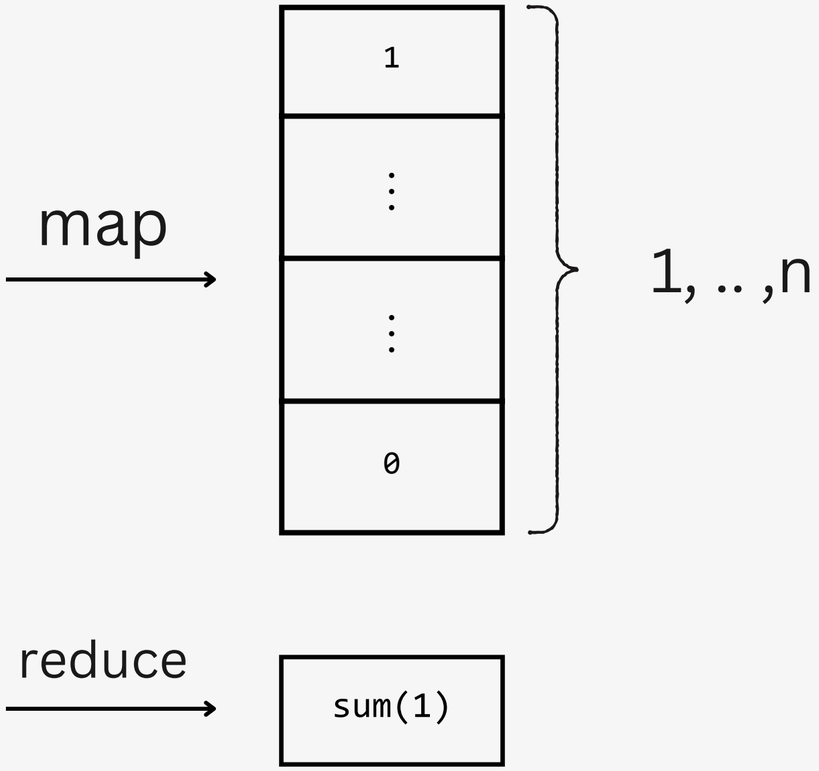
2. Iterates over each element in the NumPy array np_Xy:
   - Applies the predict function using model weights w, bias b, and the first 11 elements of the element (features).
   - Compares the prediction with the 12th element (label) and increments correctly classified if they match.

3. Calculates accuracy by dividing correctly classified by the total number of elements in np_Xy.

4. Returns the computed accuracy

NumPy

accuracy: parallelized



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

Cross Validation

Transform function

1. Defines an inner function that adds an index that assigns a random index [0, num_blocks-1] to each record.

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.

transform function

transform(RDD_Xy, num_blocks)

[x_1,1 ... x_1_11]	y_1	0
⋮		⋮
⋮		⋮
[x_n_1 ... x_n_11]	y_n	9

Random

get_block_data(data_cv, block_n)

get_block_data function

1. The function accepts an RDD (data_cv) and a block number (block_n) 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_n, creating the training dataset.
- filter_block_test includes records belonging to block_n, 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)

get_block_data function

filter_block_train

[x_2,1 ... x_2_11]	y_2	1
⋮		⋮
[x_n_1 ... x_n_11]	y_n	9

Blocks 1-9

filter_block_test

[x_1_1 ... x_1_11]	y_1	0

Block 0

Experiments

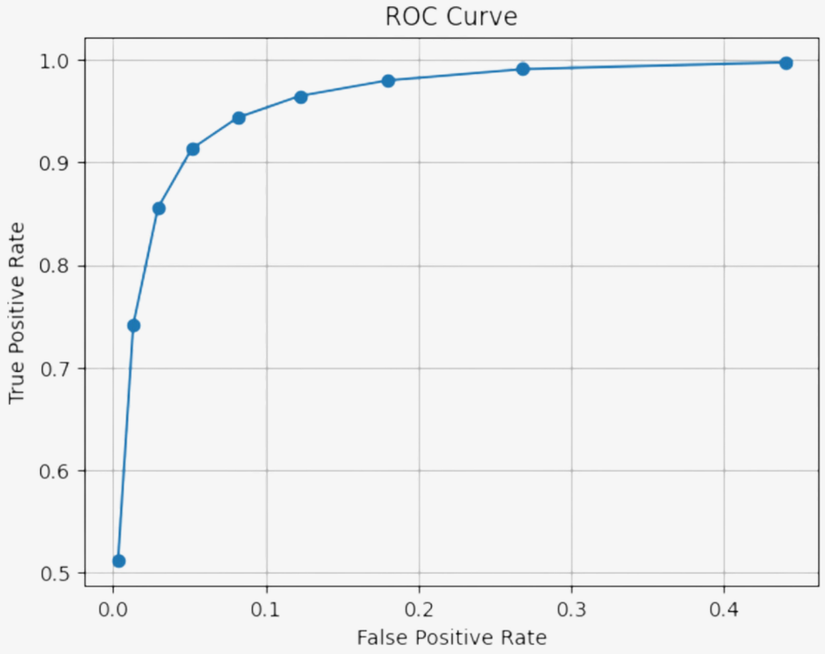
Different tresholds

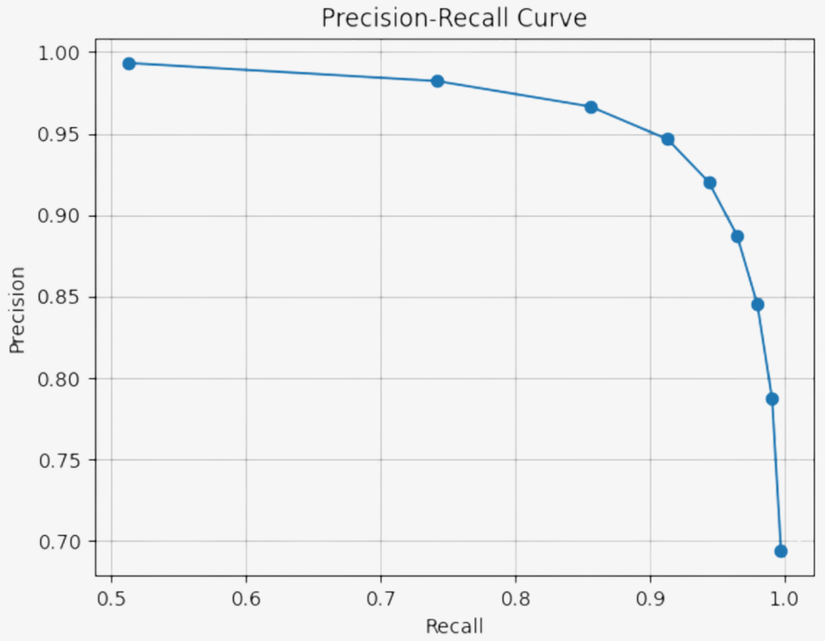
1. Executed different time the algorithm using different tresholds for the classification

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

Metric Curves

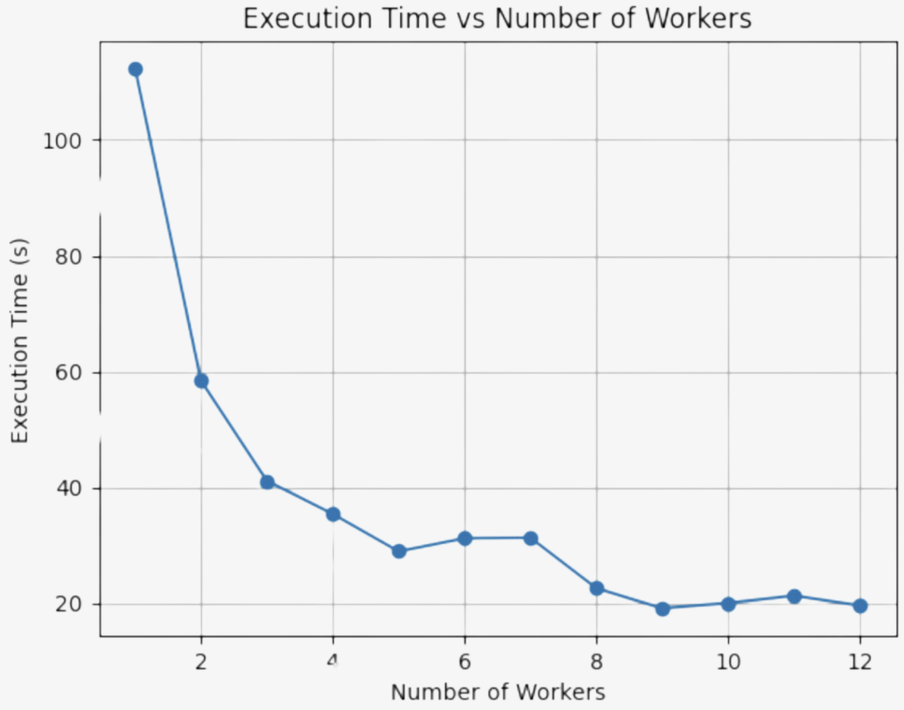
To better understand the performance of our model we decided to plot both ROC and Precision-Recall curves

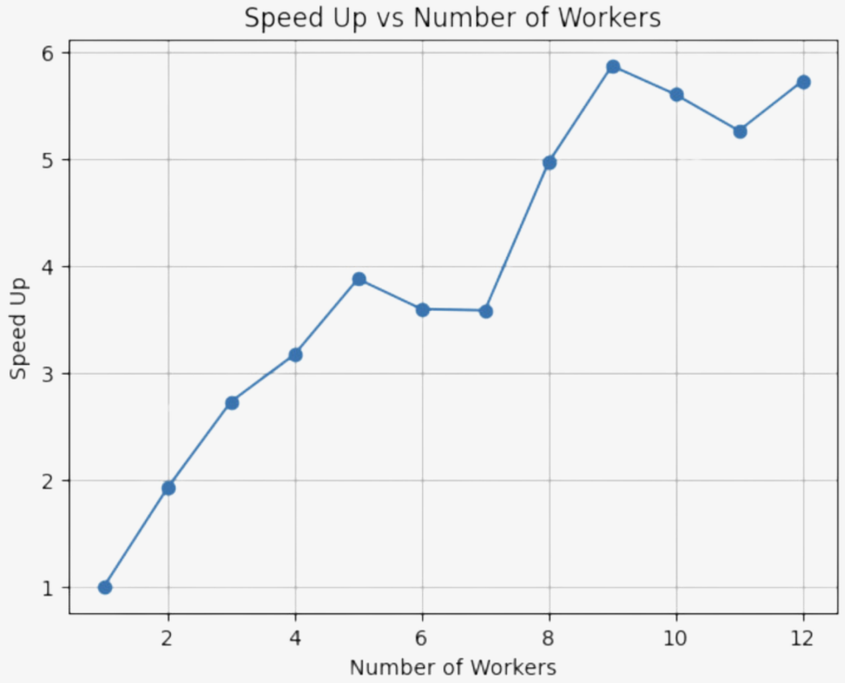




Different workers execution

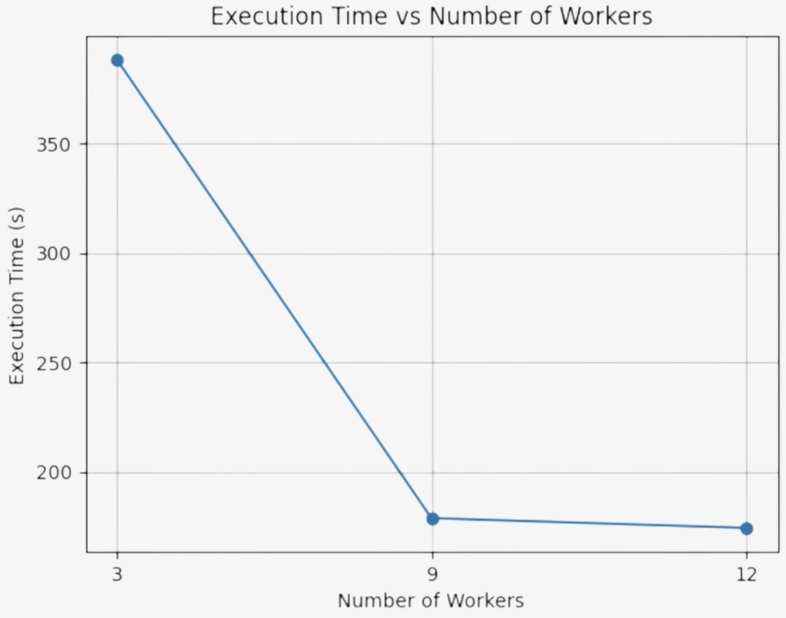
1. Try the execution of the logistic regression with different workers
2. From 1 to 12 workers (executed on a 10-core processor)

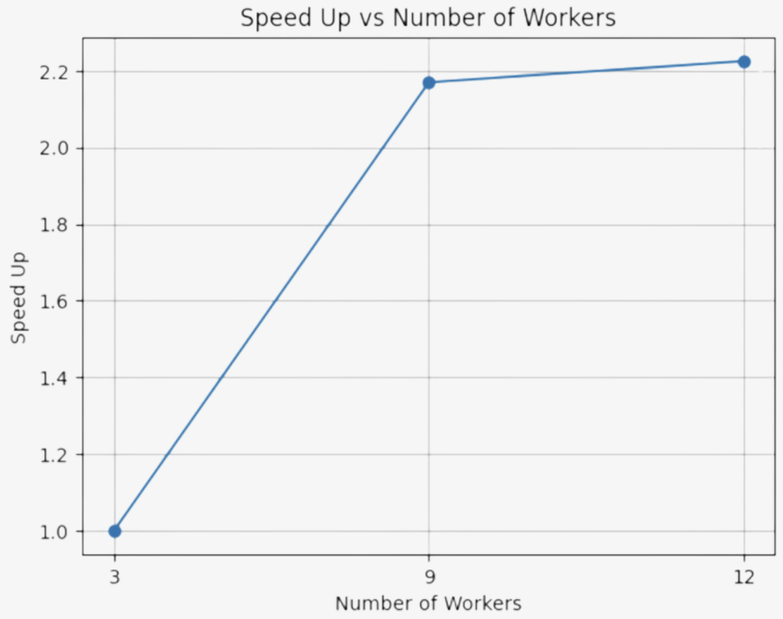




Different workers execution: Cross Validation

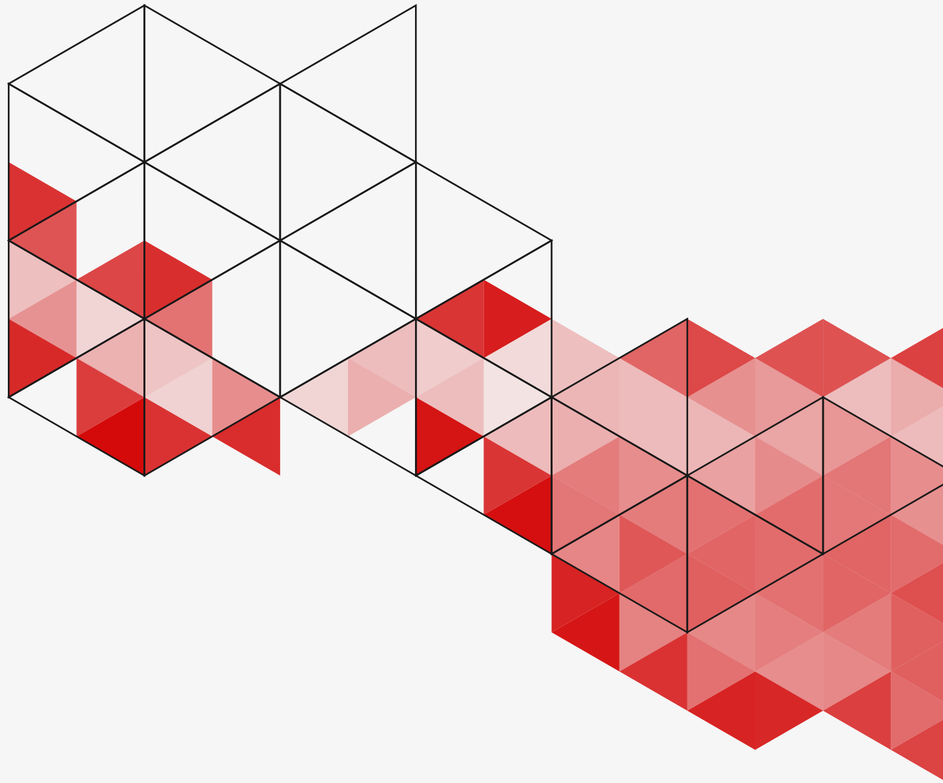
1. Try the execution of the cross validation with different workers
2. Using 3, 9, and 12 workers. (executed on a 10-core processor)





Massively Parallel Machine Learning

Thank you!
Q&A?



Nicola Cecere | Luca Petracca | Alessandro Rossi