Massively Parallel Machine Learning

Practical Assessment Task (2025-26)

Parallel Implementation and Evaluation of a supervised ML algorithm

Description of tasks to be completed

Using the labelled dataset "botnet_tot_syn_l.csv", implement a parallel version of the logistic regression classifier on Spark with Python.

Implement parallel versions of readFile, normalize, train and accuracy functions:

```
def readFile (filename):
    Arguments:
    filename – name of the spam dataset file
      12 columns: 11 features/dimensions (X) + 1 column with labels (Y)
                  Y -- Train labels (0 if normal traffic, 1 if botnet)
      m rows: number of examples (m)

    Returns:
    An RDD containing the data of filename. Each example (row) of the file
    corresponds to one RDD record. Each record of the RDD is a tuple (X,y).
    "X" is an array containing the 11 features (float number) of an example
    "y" is the 12th column of an example (integer 0/1)

def normalize (RDD_Xy):
    Arguments:
    RDD_Xy is an RDD containing data examples. Each record of the RDD is a tuple
    (X,y).
    "X" is an array containing the 11 features (float number) of an example
    "y" is the label of the example (integer 0/1)

    Returns:
    An RDD rescaled to N(0,1) in each column (mean=0, standard deviation=1)


def train (RDD_Xy, iterations, learning_rate, lambda_reg):
    Arguments:
    RDD_Xy --- RDD containing data examples. Each record of the RDD is a tuple
    (X,y).
    "X" is an array containing the 11 features (float number) of an example
    "y" is the label of the example (integer 0/1)
    iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent
    lambda_reg – regularization rate

    Returns:
    A list or array containing the weights "w" and bias "b" at the end of the
    training process

def accuracy (w, b, RDD_Xy):
    Arguments:
    w -- weights
    b -- bias
    RDD_Xy – RDD containing examples to be predicted

    Returns:
    accuracy -- the number of predictions that are correct divided by the number
    of records (examples) in RDD_xy.
    Predict function can be used for predicting a single example
```

```
def predict (w, b, X):
    Arguments:
    w -- weights
    b -- bias
    X - Example to be predicted

    Returns:
    Y_pred - a value (0/1) corresponding to the prediction of X
```

The train procedure will be implemented using Gradient Descent.

```
initialize $w_1$; $w_2$; ... $w_n$; $b$

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:

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

Derivatives

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

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

$$\ldots$$

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

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

(1) Code the parallel versions of readFile, normalize, train and accuracy functions (4 points. MANDATORY).

Use the below main code for testing the whole system:

```
# read data
data= readFile(path)

# standardize
data=normalize(data)

ws = train(data, nIter, learningRate)
acc = accuracy(data, ws)
print ("acc:",acc)
```

(2) Implement a procedure to perform cross validation (2 points. MANDATORY).

You can read the file from disk only once. Use the following pseudo-code as a template for the cross-validation process.:

```
# read data
data = readFile(path)
# standardize
data = normalize(data)
num_blocks_cv = 10
# Shuffle rows and transform data, specifying the number of blocks
data_cv = transform(data, num_blocks_cv)

for i in range(num_blocks_cv):
    tr_data, test_data = get_block_data(data_cv, i)
    ws = train(tr_data, nIter, learningRate)
    acc = accuracy(test_data, ws)
    print("acc:", acc)

print("average acc:", avg_acc)
```

If any of the functions apply a transformation involving randomness to the elements, the RDD must be persisted. Otherwise, performing multiple actions on the same RDD could result in different RDD instances, since each action will trigger a recomputation of the RDD from the very beginning of the DAG. This would include regenerating the random elements in both cases.

Note that ONLY the following Spark functions can be used for (1) and (2):
textFile, count, *map, flatMap, reduce and reduce*ByKey.
Do not use any other Spark function.

Hints:

- Convert python lists to numpy arrays and use numpy vectorized operations (e.g. dot, multiply) for operating vectors and matrices and speed up the computations.
- MANDATORY: Print the value of the cost function at the end of each gradient descent iteration to observe how the training process is converging (i.e. the cost value decreases asymptotically).
- Try to parallelize with Spark computations that involve processing all the elements of the dataset.
- Do not apply the "collect" method to any potentially big data RDD.

(3) Written report (4 points, OPTIONAL)

This report will study and analyse the parallelization effect of different parameters: number of partitions, workers, and rdd caching in the code developed in (1).

The following five experiments need to be included in the report.

Note that in the five experiments detailed below, the large dataset (1 million rows) is used.

A fixed number of iterations for **Gradient Descent** is set to 10 in all experiments.

Indicate in the report:

- the number of cores on the machine where the experiments are executed.
- Whether a virtual machine or a Docker container was used for the experiments.

EXPERIMENTS

1) Hyperparameter exploration: learning rate and regularization (0.5 points)

1a) Generate a single graph with the **iteration** (X-axis) **vs. cost** (Y-axis) curves for different values of *learning rate*.
Explain the results shown in the graph.

1b) Generate a single graph with the **iteration** (X-axis) **vs. cost** (Y-axis) curves for different values of *regularization*.
Explain the results shown in the graph.

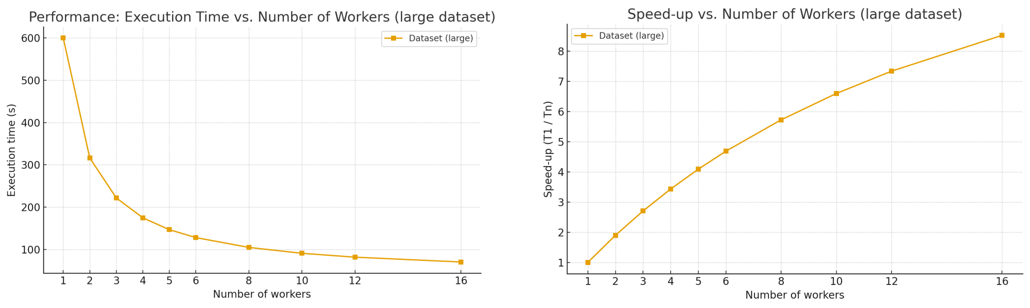
For the remaining experiments (2, 3, 4 and 5), use a learning rate of 1.5 and set the regularization term to 0.0.

2) Performance and speed-up analysis based on the number of workers and partitions (1 point)

- Generate a single graph with the **performance and speed-up** curves for different numbers of workers and partitions, taking values in the range [1, 2, 3, ..., MaxCores + 3]. Alternatively, additional values may be tested to identify the *elbow of the curve* and determine the optimal value.

- The performance curve shows execution time (y axis) versus number of workers (x axis). The speedup curve shows the ratio (running time using 1 worker) /(running time using n workers) versus the number of workers. Comment critically the results with respect to parallelization trying to discover and explain changes in the behaviour of the curves.

- The same number of workers and partitions should be set for each point of the curve.
- **RDDs should not be persisted** (i.e., do not use cache() or persist()).
- It is recommended to use tools such as top or equivalents to verify that the worker processes are executing correctly.
- Using getPartition() is recommended to ensure that the number of partitions in the RDD is as expected.
- Explain the results observed for each dataset separately and then compare the results of both datasets.



Performance and Speedup curves example.

3) Use of the cache() function and its impact on execution time (1.5 points)

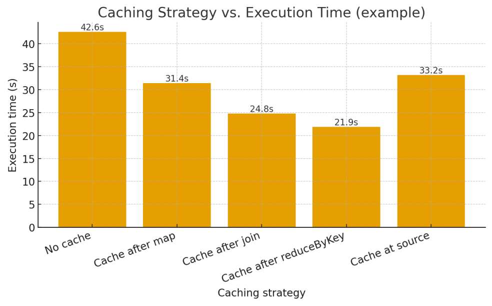
For this experiment, the **optimal number of workers and partitions** obtained in experiment 2 should be used.

- Apply different caching strategies to the RDDs and determine which one offers the best performance (shortest execution time) with the fewest calls to cache().
- Remember that the cache() function is not an action in Spark.

3a) Bar chart of caching strategies (1 point)
Create a **bar chart** summarizing your results:

- **X-axis:** Caching strategy (e.g., "No cache", "Cache after X", "Cache after Y", …).
- **Y-axis:** Execution time (in seconds).
- Label bars clearly; optionally add error bars if you repeat runs.

Explain the observed results (e.g., why the best strategy helps).



Bart chart example.

3b) Comparison between cache and no cache (0.5 points)

- Using the best caching strategy found in exercise **3a**, recalculate the **performance and speed-up curves (**defined in experiment 2**)**.
- In the same graph, include the curve **without caching** (from experiment 2) and the **optimal caching curve**.
- Explain and reason the differences in the results for each worker count when comparing **cache vs. no cache results**.

4) Study of the effect of the number of partitions (0.5 points)

- Set the number of **workers to 4** and vary the number of partitions in [1, 2, 3, 4, 5, 6, 7, 8].
- Generate a graph where:
    - **X-axis**: Number of partitions.
    - **Y-axis**: Execution time.
- Analyze whether increasing or decreasing the number of partitions improves performance. Explain and justify the observed results in the graph.

5) Study of the effect of the number of workers (0.5 points)

- Set the number of **partitions to 2** and vary the number of workers in [1, 2, 3, ..., MaxCores + 3].
- Generate a graph where:
    - **X-axis**: Number of workers.
    - **Y-axis**: Execution time.
- Analyze whether increasing or decreasing the number of workers improves performance. Explain and justify the observed results in the graph.

6) Code submission

The code used to perform the different experiments (**1, 2, 3, 4 and 5**) must be submitted.

In addition to the written report, separate jupyter notebooks will be delivered for the parallelized version and the code used for deploying the experiments (**1, 2, 3, 4 and 5**) of the written report.