PRACTICAL ASSESSMENT TASK 2

Parallel Implementation and Evaluation
of an unsupervised Machine Learning algorithm

The aim of this assignment is to implement a parallelized version of the *K*-Means clustering algorithm on Spark with Python. The experiments will be carried out with the MNIST dataset, as available in the moodle site.

1. IMPLEMENTATION OF SERIALIZED K-MEANS (1 point)

In this first approach, we shall implement the standard version of the *K*-Means algorithm, without parallelization. The functions to be implemented are the following.

```
def serialReadFile(filename):
def serialAssign2cluster(x, centroids):
def serialKMeans(X, K, n_iter):
```

Specifically, the role of each function is:

- `serialReadFile`: It receives a String `filename` with name of the csv file of the dataset. The function returns a Pandas dataframe with the loaded data.
- `serialAssign2cluster`: It receives a list of *d*-dimensional tuples called `centroids`, representing the current state of the centroids, and a *d*-dimensional tuple `x` which represents the datum to be assigned to a cluster. It returns an integer with the index in `centroids` of the closest centroid to `x`.
- `serialKMeans`: Performs the serialized *K*-Means algorithm on the dataset `X`, grouping the instances into `K` different clusters. The number of iterations of the method to be executed is `n_iter`. The initialization of the centroids will be random, sampled from a standard normal distribution. It returns a list of length `K` with the *d*-dimensional centroids computed.

The implementation of the *K*-Means algorithm will be based on the provided pseudo-code in the slides of the course (available in Moodle).

2. IMPLEMENTATION OF PARALLELIZED *K*-MEANS (4 points)

In this second part, we will adapt the previous function to work in a highly parallelizable setting. The functions to be implemented are the following.

```
def parallelReadFile(filename):
def parallelAssign2cluster(x, centroids):
def parallelKMeans(data, K, n_iter):
```

- `parallelReadFile`: It receives a String `filename` with name of the csv file of the dataset. The function returns a Spark RDD with the loaded data.
- `parallelAssign2cluster`: It receives a list of *d*-dimensional tuples called `centroids`, representing the current state of the centroids, and a *d*-dimensional tuple `x` which represents the datum to be assigned to a cluster. It returns an integer with the index in `centroids` of the closest centroid to `x`.
- `parallelKMeans`: Performs the parallelized K-Means algorithm on the RDD dataset data, grouping the instances into K different clusters. The number of iterations of the method to be executed is `n_iter`. The initialization of the centroids will be random, sampled from a standard normal distribution. It returns a list of length `K` with the *d*-dimensional centroids computed.

The provided solution must show a high level of parallelism, with an intensive use of the Spark technology to implement a map-reduce based algorithm. In particular, the function `parallelKMeans` must take advantage of the parallelization procedure to speed up the task of assigning each datum to its nearest cluster.

3. ANALYSIS AND DISCUSSION (5 points)

Using the aforementioned version of the parallelized K-Means algorithm, analyze the execution time of the proposed solution on the MNIST dataset uploaded to Moodle. This dataset contains 70,000 samples of handwritten digits, from 0 to 9. Each image is represented by a 28×28 matrix of black-and-white pixels. In the dataset, this pixel matrix has been flattened into a $28 \times 28 = 784$ vector, so the dataset contains 28000 rows and 784 columns. Each pixel is represented by an integer ranging from 0 (white pixel) to 255 (black pixel).

3.1 **(2 points)** Study the execution time of the K-Means algorithm on the MNIST dataset versus the number of parallel workers and prepare plots showing:

(i) The performance curve (the execution time (y axis) versus the number of workers (x axis) using different values of K.

(ii) The speedup ratio (running time using 1 worker over running time using n workers) versus the number of workers using different values of K.

3.2 (**1 point)** Analyze the impact of parallelizing other parts of the algorithm as the computations of distances of an instance to each of the centroids. Plot the performance and speedup curves using different values of K and compare the results with respect the ones obtained in 3.1

3.3 (**0.5 points)** Analyze and interpret the obtained clusters with $K = 3, 5, 7, 8, 9,$ 10 and 11. Show plots of the obtained centroids and relate them with the original data.

3.4 (**1.5 points)** With these results, a written report must be prepared with the following sections:

1. Introduction.
2. Dataset description.
3. Algorithm implementation.
4. Experiments (including performance and speedup curves).
5. Conclusions.

4. SUBMISSION

With the obtained implementation and written report, prepare a submission to Moodle. A single zip file must be uploaded with the following three elements:

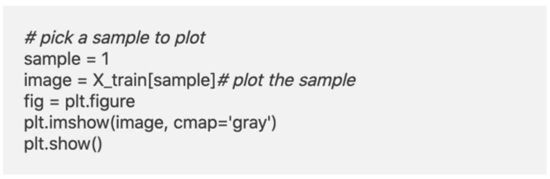
- Jupyter Notebook file (format `.ipynb`) with the Python code of the serialized implementation of *K*-Means.
- Jupyter Notebook file (format `.ipynb`) with the Python code of the parallelized implementation of *K*-Means.
- Pdf file with the written report.

The submitted code must work with the csv files of the MNIST dataset, as provided in Moodle. The submitted zip file must be named according to the format **MPML_nameStudent_surnameStudent_Assigment2.zip**.

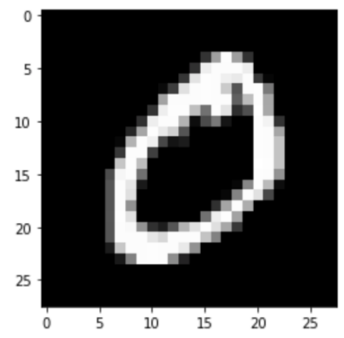
5. ORAL PRESENTATION

Additionally, an individual online oral presentation of 10 minutes maximum must be prepared. The presentation must outline the main ideas of the provided solution and parallel implementation, as well as some highlights of the obtained results. Accompanying slides are recommended but are not mandatory.

6. MNIST IMAGE PLOTTING HINTS



```
# pick a sample to plot
sample = 1
image = X_train[sample]# plot the sample
fig = plt.figure
plt.imshow(image, cmap='gray')
plt.show()
```



7. Pseudocode for a Simplified Version of K-Means

```
centroids = gen_random_points(K)

for it in range(iterations):
  centroids_with_points = [ [], [], ..., [] ]   // K empty lists

  // assign points to closest centroids
  for p in datapoints:
     i = assig2cluster(p, centroids)
     centroids_with_points[i].append(p)

  // handle empty clusters (reinitialize centroid)
  for i in range(K):
       if len(centroids_with_points[i]) == 0:
          centroids[i] = random_choice(datapoints)

  // recalculate new centroids
  for i in range(K):
       if len(centroids_with_points[i]) > 0:
          centroids[i] = sum(centroids_with_points[i]) / len(centroids_with_points[i])
```