UNIVERSIDAD
POLITÉCNICA
DE MADRID

K-Means

Massively parallel machine learning

Group 12:

Hugo Garde
Corentin Ibos
Elise Maistre

19/12/2023

I. Introduction

In this practical assessment task, the objective is to implement and evaluate a parallelized version of the K-Means clustering algorithm using PySpark. The focus will be on enhancing the efficiency and scalability of the K-Means algorithm by leveraging the parallel processing capabilities of Spark. The MNIST dataset, consisting of 70,000 samples of handwritten digits, will be employed for experimentation and evaluation.

The assignment is divided into two main parts.
In the first part, a centralised version of the K-Means algorithm is to be implemented without parallelization. This involves creating functions for reading the dataset, assigning samples to clusters, and executing the K-Means algorithm in a serial manner.

The implementation will be made using three functions:

- ReadFile which receives the name of the csv file containing the MNIST dataset and returns it into an array.
- Assign2Cluster which receives a sample and the centroids, and returns the index of the closest one from the sample.
- KMeans which implements the kmeans algorithm. It initialises the centroids randomly, then it assigns each sample to a cluster using the previous functions and it computes the new centroids depending on the samples in the cluster.

The second part involves the adaptation of the K-Means algorithm for parallel execution on Spark. The functions will be the same but they will be implemented using parallelizing tools. The aim is to achieve a high level of parallelism, using Spark's map-reduce functions to enhance the performance of the algorithm.

II. Dataset description

The MNIST dataset is a widely used collection of handwritten digits, frequently employed in the field of machine learning and computer vision. It consists of 70,000 grayscale images of handwritten digits, covering the numerical range from 0 to 9. Each image is a 28 × 28 matrix of black-and-white pixels, forming a grid where each pixel represents the intensity of light at that specific position.

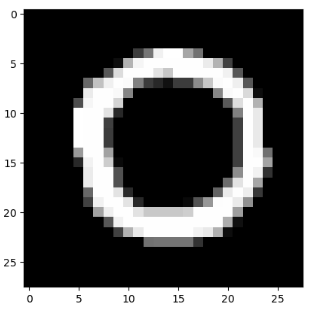


Fig 1: Example of a sample from MNIST data set

To facilitate machine learning tasks, the original 28 × 28 matrix for each image has been flattened into a vector of length 784 (28 × 28 = 784). In this assignment, the MNIST dataset is presented as a two-dimensional array with 70,000 rows and 784 columns, where each row corresponds to an individual image, and each column represents the intensity value of a specific pixel in the flattened vector. The pixel values are integers ranging from 0 to 255, where 0 represents a white pixel and 255 represents a black pixel. This grayscale representation allows for a simple and standardised input format for training and testing machine learning models on handwritten digit recognition tasks.

III. Algorithm implementation

A. Centralised

1. serialReadFile

The function serialReadFile reads a CSV file containing the MNIST dataset. It uses NumPy to load the data into a two-dimensional array, excluding the first row (which contains column names) and the first column (representing the predicted output label). The function then returns a NumPy array containing the data.

2. serialAssign2Cluster

The function determines the closest centroid to a given data point (x) from a list of centroids. It computes the Euclidean distance between x and each centroid, then returns the index of the closest centroid in the list. The implementation involves iterating over centroids, calculating distances, and finding the index of the minimum distance.

3. serialKMeans

The function performs the K-Means clustering algorithm on a given dataset X. It groups the instances into K different clusters over a specified number of iterations. The initial centroids are randomly initialised by sampling from a standard normal distribution.

During each iteration, samples are assigned to the nearest cluster based on the current centroids, and new centroids are computed by computing the centre of mass for each cluster.

The function returns a list of K centroids, each represented as a tuple of 784-dimensional values. The final centroids represent the centre of each cluster after the algorithm has converged through the specified number of iterations.

B. Parallelized

1. parallelReadFile

The aim was to read the rows in a parallelized fashion, eliminating the first row (header) as well as the first column. To achieve this, we used Spark's textFile function. The header is then retained using first or take(1).

A flatMap is then applied to filter out all rows and not retain the header. The result is not the same size (one line less, the first).

Finally, a map is used to process each line of data. This map deletes the first value of each line. The rows are split into different values, and the first value is not returned. Each column is thus returned in 784 floats.

2. parallelAssign2Cluster

This function remains the same as the serial one as it does not include a big data computation.

3. parallelKMeans

In this part, the aim is to parallelize the allocation of data to each cluster and to obtain a new cluster corresponding to the centre of mass of all the data in the previous cluster. This stage is carried out iteratively.

Initially, the first centroids are defined at random. A list of K tuples with 784 (28 x 28) values is obtained.

Since the aim is to calculate the centre of mass of each of the new clusters, it is necessary to sum all the images belonging to the cluster and divide the sum by the number of images in the cluster. To do this, we assign the nearest cluster to each image using the parallelAssign2cluster function above. In this way, for each image we obtain a tuple with the closest cluster to the image and the image itself as the key. To avoid two different calculations, a 1 is added at the end of the image data, so that when the data is summed, the number of images for each cluster is calculated at the end of the data.

This tuple is obtained by performing a map on the RDD containing all the images. This key-value tuple can then be used for a reduceByKey.

If we want to obtain a centroid for each of the K clusters, we can use the reduceByKey to obtain the desired result. Each of the tuples with the same keys is summed, and we obtain the first version of our future centroid.

Next, each of the values must be divided by the number of values in the cluster. Fortunately, this value was summed when the reduceByKey was used, having been placed in the last position. So a final map is created which divides each of the image values by the number of images in the cluster. As a tuple cannot be modified, it is transformed into a list. Each value is divided and then a tuple without the last value for the number of images in the cluster is returned.

Finally, a collect is performed to obtain these K clusters and use them in the next iteration. The next iteration repeats the previous steps, using the K centroids calculated.

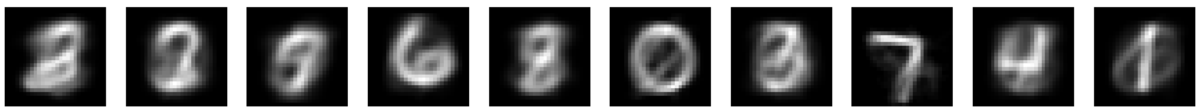
At each iteration, the K images are displayed in order to check the quality of the centroids created.

IV. Experiments

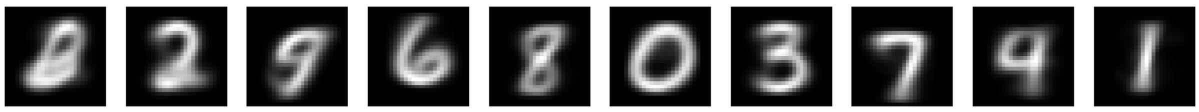
<u>Results for centralised version :</u>

Execution time for 10 iterations : 1607.58 s

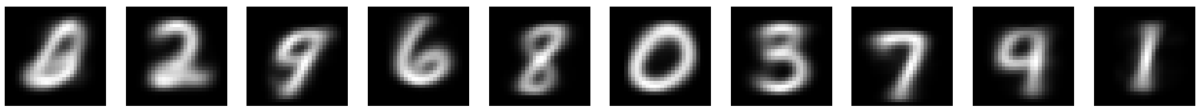
It 1



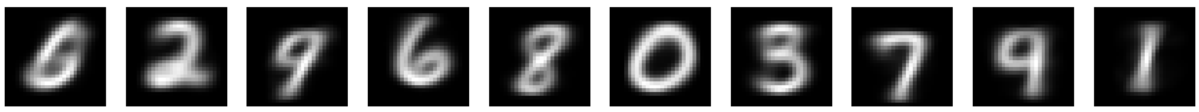
It 3



It 5



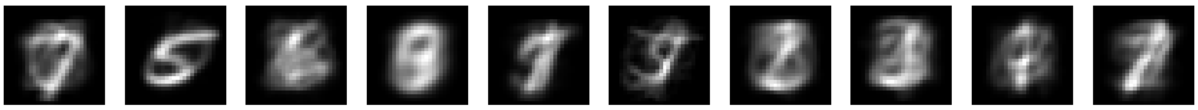
It 10



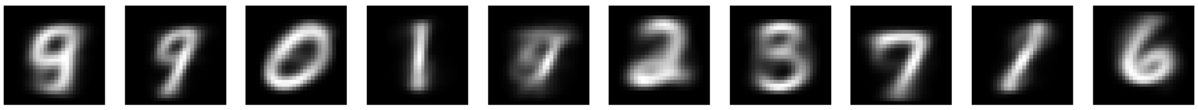
<u>Results for parallelised version:</u>

Execution time for 10 iterations : 1124.60 s

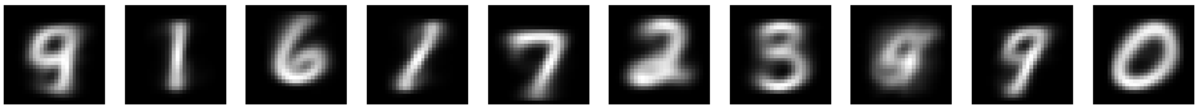
It 1



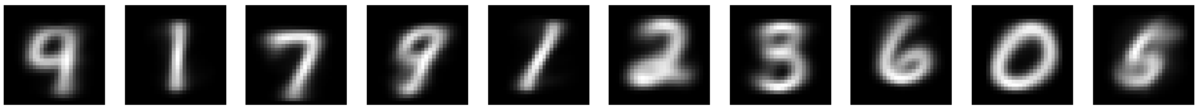
It 3



It 5



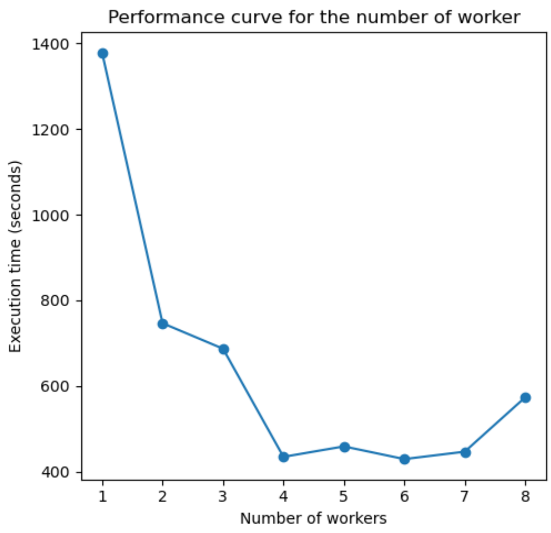
It 10

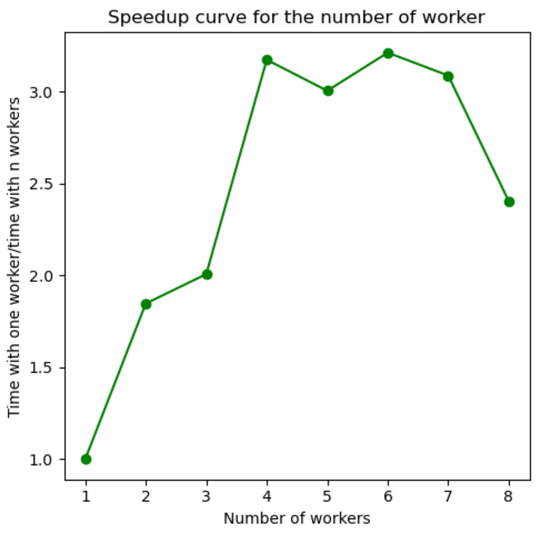


From the previous results we observe a difference in execution time between the centralised and parallelized versions. In fact, the parallelized version is ⅔ faster than the centralised version. This seems to show that our parallelization works since some calculations are performed on different cores at the same time.

Regarding the displays according to the iterations we observe similar cases whether for the parallelized version or for the centralised version. In the first iteration, the centroids seem to be rather mixed up, with some numbers standing out slightly depending on the randomness of the first centroid. During the 3rd iteration, the numbers are already more formed, with a blur linked to certain poorly clustered numbers. By the 5th iteration, the final centroids have already been formed and will change very little as subsequent iterations produce very similar results.

<u>Performance and speedup curves for the number of worker</u>





Performance curve:

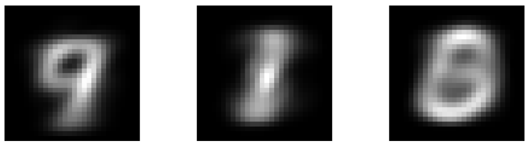
The initial transition from a single worker to 2 workers results in a reduction in execution time, indicating that parallelization brings about immediate improvements. This aligns with the expectation that parallel execution can enhance performance by distributing computational load.
Beyond the transition from 1 to 2 workers, the performance gains become less significant as the number of workers increases. The curve shows a diminishing trend, with execution time not improving further after 4 workers.
The point where the curve starts to plateau suggests an optimal number of workers for this specific task. Beyond this point, adding more workers does not yield proportional reductions in execution time and even increases the execution time because of Spark engine distribution management of work.

Speedup curve:

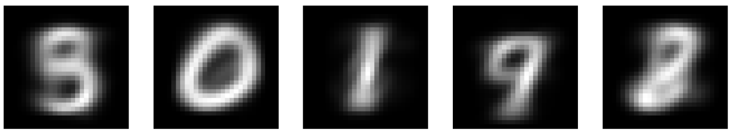
We see that the execution time with 2 workers is almost 2 times shorter than with only 1. It is the case because some parts of the algorithm are not parallelized. As the number of workers increases, the speedup experiences smaller returns.
The speedup curve provides insights into the optimal number of workers for this specific workload. Beyond a certain point, the benefits of adding more workers become less pronounced, and the system experiences diminishing returns. In this curve, we can see that we reach this point with 4 workers.

Training of 10 iterations for each K: 3,5,7,8,9,10,11

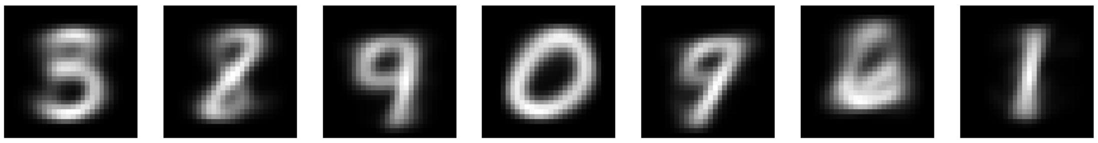
K = 3



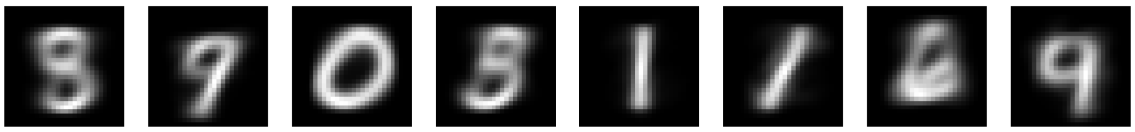
K = 5



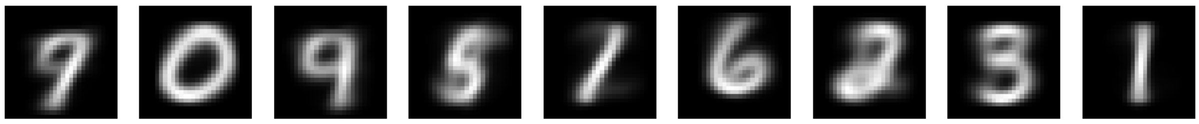
K = 7



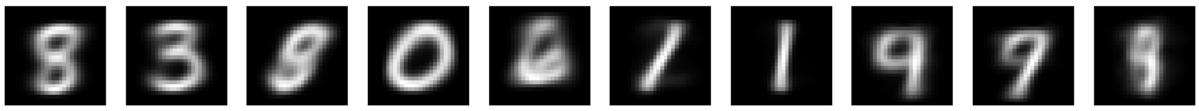
K = 8



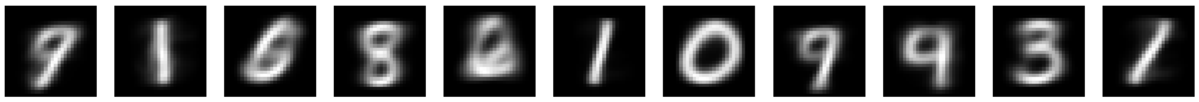
K = 9



K = 10

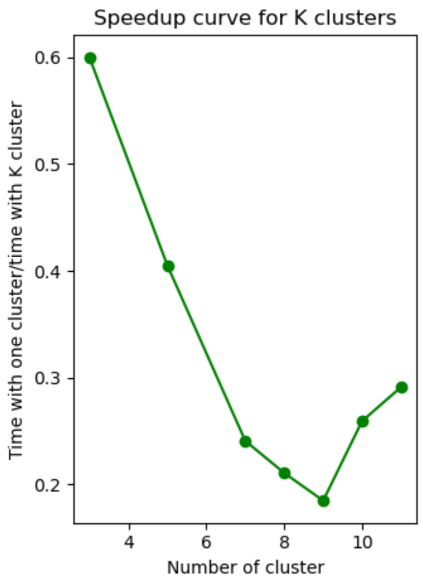


K = 11



After displaying the centroids for K = 3,5,7,8,9,10,11, the results can be analysed. It can be seen that the centroids formed do not correspond completely to the desired digits, even for K = 10. The 10 digits from 0 to 9 are not obtained separately. But some digits stand out from the others, even at K = 3 or 5. This is particularly the case for 9, 1 and 0. We could even guess these digits from K = 3, with a mixture of the other advantage digits formed for the centroids which would correspond to 0. But from K = 5, these 3 digits really stand out from the others. And when differences occur, these digits, particularly 1 and 9, are duplicated and form several clusters. For K = 11 we even have three clusters forming the 1, in several shapes. When the desired number of centroids is larger, the algorithm is also able to separate other digits such as 3 and 8. The other digits are difficult to separate correctly every time. Sometimes the results are better than others. These results depend on the first random clusters formed. We should test the case where the clusters are assigned non-randomly, with one cluster for each of the ten digits from 0 to 9.

<u>Speedup curves for the number of cluster:</u>



As the number of clusters increases from 3 to 11, the speedup diminishes. This diminishing trend implies that the parallelization efficiency decreases with a higher number of clusters.
The speedup curve suggests that the optimal number of clusters lies between 3 and 5. Beyond this range, the efficiency gains diminish more rapidly.
However, this is when only taking into account the execution time. As we know that there are 10 different digits, optimising the execution time won't give us the best result.

V. Conclusion

In conclusion, the implementation of a parallelized version of the K-Means clustering algorithm using PySpark for the MNIST dataset has demonstrated improvements in efficiency and scalability. From the centralised to the parallelized version, the algorithm showed a reduction in execution time, highlighting Spark's parallel processing capabilities.

Performance curves, and speedup analyses provided valuable insights into the optimal configuration for the number of workers and "clusters". We saw that having too many workers was not a good idea as the execution time remained the same or could even increase.

However, despite the evident speedup and efficiency gains, challenges remain in achieving perfect digit separation, especially when faced with the randomness in initial centroid assignments. Further investigations into non-random initialization methods may help refine the accuracy of the K-Means algorithm for handwritten digit recognition tasks.