UNIVERSIDAD
POLITÉCNICA
DE MADRID

Parallel Implementation and Evaluation of K-Means algorithm

Academic year 2023-2024

Authors:

Nicola Cecere

Luca Petracca

Alessandro Rossi

Professor:

Mozo Velasco Bonifacio Alberto

Contents

1 Introduction

In this report, we explore the K-Means clustering algorithm, focusing on both its traditional and parallelized forms. Our study starts with the K-Means algorithm as it's usually implemented in Python, using familiar tools to understand how it works and performs. We then shift our attention to a parallel version of the algorithm using Apache Spark, a powerful tool for handling large-scale data processing.

First, we look at the K-Means algorithm, Figure 1, running on a single machine. We use the MNIST dataset, which contains images of handwritten digits, perfect for testing our clustering method. In this part, we play with different settings of the algorithm and observe how they affect the results, giving us a good understanding of its strengths and weaknesses.

```
Input: Dataset $D$
$m^1, \dots, m^K \leftarrow random()$
while not stop:
    $C^1, \dots, C^K \leftarrow \emptyset$                      ┐
    for $x_i \in D$ :                                              │
        for $j = 1, .., K$:                                        │ Assign points to clusters
            $d^j \leftarrow \|x_i - m^j\|$                         │ (Voronoi partition)
        $j_0 \leftarrow argmin(d^1, \dots, d^K)$                   │
        $C^{j_0} \leftarrow C^{j_0} \cup \{x_i\}$                  ┘
    for $j = 1, .., K$:                                            ┐
        $m^j \leftarrow \frac{1}{|C^j|} \sum_{x_i \in C^j} x_i$    ┘ Recompute centroids
```

Figure 1: K Means Algorithm

Next, we move to the parallel version of K-Means using Spark. This part of the report explains how we adapted the algorithm to work in this new environment and compares the performance to the centralized version. We especially focus on how well it works with larger amounts of data.

Finally, we present a series of tests and analyses. These are designed to see how well both versions of the algorithm perform under different conditions. We look at things like how fast they run, how accurate they are, and the quality of the clusters they find. The goal is to give a clear picture of when and why you might use one version over the other.

2 Dataset Description

The MNIST dataset is extensively utilized for machine learning, characterized by the following attributes:

- **Size and Contents:** Contains 70,000 images, each representing a handwritten digit from 0 to 9.
- **Image Structure:** Every image is a grayscale 28x28 pixel grid, resulting in 784 pixels per image.
- **Feature Representation:** Each pixel is a feature, with its intensity varying from 0 (white) to 255 (black).

- **Data Format:** The dataset can be represented as a matrix with 70,000 rows (one per image) and 784 columns (one per pixel/feature).
- **Pattern Recognition Challenge:** The variety in handwriting styles presents a robust challenge for clustering algorithms, especially in high-dimensional spaces.
- **Application in Computing Environments:** Ideal for testing both centralized and distributed computing systems, given its size and complexity.
- **Benchmarking Utility:** Frequently used in the machine learning community as a benchmark for evaluating the performance of various algorithms.

In summary, the MNIST dataset is a fundamental resource for testing and refining machine learning algorithms, particularly in the field of image processing and clustering.

3 Algorithm Implementation

3.1 Reading

In this section, we discuss the reading steps for preparing the MNIST dataset for effective use in machine learning models ensuring that each image is represented consistently for accurate analysis. We also explore techniques for optimizing data for both centralized and parallelized computing environments, highlighting the significance of these steps in enhancing model performance and reliability.

Centralized Version

The `serialReadFile` function reads data from a CSV file into a Pandas DataFrame, using the 'pd.read_csv' method. It then removes a column named "label" from this DataFrame, modifying it in place. The function returns the modified DataFrame, now without the "label" column.

Parallelized Version

The `parallelReadFile` function uses PySpark to read a CSV file into an RDD, removing the header and the first column (label). It achieves this by filtering out the header row and converting the remaining rows into lists of floats, excluding the label. This function enables efficient, distributed processing of large datasets in Spark environments, returning an RDD of data points suitable for further analysis or machine learning tasks. The function performs the following steps:

1. Establishes or retrieves an existing SparkContext.
2. Reads the specified CSV file into an RDD using the `textFile` method.
3. Applies two main operations to the RDD:
   - **Filter out Header:** Utilizes the `flatMap` operation with the `is_header` function to remove the header row, identified by a known feature such as 'label'.
   - **Convert to Floats:** Transforms each line into a list of floats, excluding the label column, through the `map` operation with the `convert_to_float` function.

4. Returns the processed RDD, where each data point is now a list of floats representing the features, excluding the label.

Comparison

The centralized ‘readFile‘ implementation processes CSV files using standard Python libraries, reading data directly into a NumPy array, ideal for smaller datasets manageable on a single machine. In contrast, the parallel ‘readFile‘ implementation uses PySpark to read data into an RDD, suitable for large-scale datasets that require distributed processing. While the centralized approach is simpler and straightforward, the parallel approach efficiently handles vast amounts of data, leveraging Spark’s robust data distribution and parallel computation capabilities.

3.2 Assign to Cluster

In this section, we examine the distinct implementations of the assign2cluster function for the centralized and parallelized systems.

Centralized Version

The `serialAssign2cluster` function assigns a data point ‘x‘ to the nearest centroid from a list of centroids ‘centroids‘, based on the Euclidean distance. For each centroid, it calculates the distance between ‘x‘ and the centroid, updating the minimum distance and the index of the closest centroid when a smaller distance is found. The function returns the index of the closest centroid to the data point ‘x‘. This process is essential in clustering algorithms like K-means, where data points are iteratively assigned to the nearest centroid.

1. Initializes the minimum distance as infinity and the index of the closest centroid as -1.
2. Iterates over each centroid in the given list of centroids:
   - Converts the current centroid to a NumPy array for computation.
   - Calculates the Euclidean distance between the data point `x` and the centroid.
   - Updates the minimum distance and the index of the closest centroid if the current distance is smaller than the recorded minimum.
3. Returns the index of the centroid that is closest to the data point `x`.

This function is a key part of the K-Means clustering algorithm in a centralized environment, determining the nearest centroid for each data point based on Euclidean distance.

Parallelized Version

The `parallelAssign2cluster` function in Python computes the Euclidean distance between a given data point ‘x‘ and each centroid in a list of centroids, identifying the index of the closest centroid. It returns a tuple containing the index of the closest centroid and the data point ‘x‘ (converted to a NumPy array) along with the count ‘1‘, facilitating further aggregation in clustering algorithms like K-means. This function is designed for distributed computation in a PySpark environment.

1. Starts by initializing the minimum distance as infinity and the index of the closest centroid as -1.

2. Iteratively computes the Euclidean distance from the data point `x` to each centroid in the list:
    - Transforms each centroid into a NumPy array for accurate distance calculation.
    - Updates the closest centroid index and minimum distance if a closer centroid is found.
3. Returns a tuple consisting of the closest centroid's index and a pair: the data point `x` (as a NumPy array) and the count 1. This format is particularly suited for subsequent aggregation in a distributed computing environment.

This function is essential for the K-Means algorithm in a parallel computing context, such as with PySpark, enabling efficient assignment of data points to centroids across multiple nodes.

Comparison

The serialAssign2cluster function operates in a centralized computing environment, iteratively calculating Euclidean distances between a data point and each centroid, suitable for smaller datasets processed on a single machine. In contrast, the parallelAssign2cluster function, while maintaining a similar distance calculation logic, returns data structured for distributed computing, making it ideal for use in parallelized environments like PySpark. The parallel version's output format, a tuple with the data point and a count, facilitates aggregation in subsequent steps of the K-Means algorithm across multiple nodes, offering scalability and efficiency in handling larger datasets.

3.3 K-Means Implementation

In this section, we provide an in-depth comparison between the centralized and parallelized implementations of the K-Means clustering algorithm. Centralized K-Means, traditionally executed on a single machine, is well-suited for small to medium-sized datasets and offers simplicity and ease of implementation. Conversely, the parallelized version leverages the distributed computing power of frameworks like Apache Spark, efficiently handling larger datasets by dividing the workload across multiple nodes. Our analysis aims to highlight the distinct advantages, challenges, and performance characteristics of each approach.

Centralized Version

The `serialKMeans` function implements the K-Means clustering algorithm in a serial manner. It converts a Pandas DataFrame 'X' into a NumPy array, initializes 'K' centroids using the `initialize_centroid` function, and then iteratively updates these centroids over 'n_iter' iterations. In each iteration, each data point in 'X' is assigned to the closest centroid using the `serialAssign2cluster` function. The centroids are then updated to be the mean of the points in their cluster. If a cluster is empty, a new centroid is randomly generated. The function returns the final positions of the centroids after completing the specified number of iterations. This serial implementation is suitable for smaller datasets that can be processed without the need for distributed computing frameworks.

1. Converts the input data `X` to a NumPy array.
2. Initializes `K` centroids using the `initialize_centroids` function, which generates centroids from a standard normal distribution based on the number of features in `X`.

3. Iterates over a specified number of iterations (`n_iter`):
    - Initializes a list of clusters, with each cluster corresponding to one centroid.
    - For each sample in `X`, identifies the closest centroid using the `serialAssign2cluster` function and assigns the sample to the respective cluster.
    - Updates each centroid to be the mean of all samples assigned to it. If a cluster is empty, reinitializes its centroid randomly.
4. Returns the final centroids after completing all iterations.

This function embodies the core logic of the K-Means algorithm, clustering data based on feature similarity, and is well-suited for datasets that can be processed in a centralized way.

Parallelized Version

The `parallelKMeans` function implements the K-Means clustering algorithm in a distributed manner using PySpark. It initializes centroids randomly, iterates over a specified number of iterations, and in each iteration, assigns data points in RDD 'X' to the nearest centroid, then updates the centroids based on the mean of the assigned points. The function handles cases where some centroids have no assigned points by reinitializing them, ensuring that the number of centroids remains constant throughout the process. This implementation is suited for large-scale datasets, leveraging Spark's distributed computing capabilities.

1. Obtains or creates a SparkContext instance.
2. Initializes `K` centroids using the `initialize_centroids` function. This function generates centroids from a standard normal distribution, considering the number of features in the RDD `X`.
3. Iterates over the specified number of iterations (`n_iter`), performing the following steps in each iteration:
    - Broadcasts the current centroids to all nodes in the cluster.
    - Assigns each data point in `X` to the nearest centroid using the `parallelAssign2cluster` function and a map operation.
    - Aggregates data points assigned to each centroid and updates the centroids' positions. This is done using the `reduceByKey` and `map` operations, where the latter applies the `averageCentroid` function to calculate the new centroid position.
    - Collects the updated centroids back to the driver node.
    - If the number of collected centroids is less than `K`, additional centroids are initialized to maintain the total number of `K` centroids.
4. Returns the final centroids after completing all iterations.

This function leverages Spark's distributed computing capabilities, making it suitable for clustering large-scale datasets. It demonstrates the scalability and efficiency of parallelized K-Means clustering.

Comparison

The serialKMeans function, designed for centralized computing, processes datasets on a single machine, iterating over data points and centroids to perform clustering. This approach, while straightforward and easy to implement, is limited by the memory and computational constraints of a single system. It excels with smaller datasets where simplicity and direct control over the process are prioritized.

Conversely, the parallelKMeans function utilizes PySpark for distributed computing, dividing the workload across multiple nodes in a cluster. This method effectively handles larger datasets, leveraging Spark's optimized data processing and in-memory computation. It involves broadcasting centroids to nodes, parallel assignment of data points to clusters, and aggregating updates to centroids across the cluster. While more complex in setup and requiring a Spark environment, it offers scalability and improved performance for big data scenarios.

Both implementations achieve the same objective of clustering data via the K-Means algorithm, but they differ significantly in scalability, computational efficiency, and suitability for varying dataset sizes and computing environments.

3.4 Testing

Centralized Version

The following procedure employs the serial K-Means clustering algorithm on the MNIST dataset. It reads the dataset using `serialReadFile` into a DataFrame, performs clustering to find 10 centroids using `serialKMeans`, and then visualizes these centroids using the `plot_centroids` function. Each centroid is reshaped and displayed as a grayscale image, representing the average of the cluster's images. The script showcases a complete workflow of reading data, applying a machine learning algorithm, and visualizing the results.

1. Accepts an array of centroids and an image size tuple as input. Each centroid in the array represents a cluster's center in the feature space.
2. Iterates through the centroids array:
   - Reshapes each centroid to the specified image dimensions (e.g., 28x28 for MNIST dataset images), converting the flat array back into a 2D image format.
   - Utilizes Matplotlib to plot each reshaped centroid as a grayscale image.
   - Assigns a title to each plot indicating the centroid number for easy identification.
3. Displays the images, providing a visual representation of the average or most typical data point in each cluster.

Parallelized Version

This script sets up a PySpark environment and runs a parallel K-Means clustering algorithm on MNIST dataset images. It reads the data using `parallelReadFile`, caches (`cache()`) it for performance optimization, and performs clustering with `parallelKMeans` to identify K centroids. Finally, it visualizes these centroids as images using `plot_centroids`, displaying each centroid reshaped to the original image dimensions. The script integrates various components, including PySpark setup, data processing, clustering, and visualization within a Python executable script.

1. Configures the PySpark environment for varying numbers of cores to assess the performance across different levels of parallelization.
2. Initializes a SparkContext for each iteration, specifying the number of cores for the current run.
3. Reads the MNIST dataset using the `parallelReadFile` function, which returns an RDD of data points.
4. Applies caching to the RDD (`data_cache`) to enhance performance. Caching is crucial as it reduces the time to access RDDs in subsequent operations, particularly important in iterative algorithms like K-Means.
5. Performs K-Means clustering using the `parallelKMeans` function, which adapts the algorithm for distributed processing.
6. Visualizes the centroids obtained from clustering using the `plot_centroids` function. This function reshapes each centroid to the image size (28x28) and plots it, providing insights into the clustering results.
7. Records the execution time for each parallelization level to analyze performance gains.
8. Stops the SparkContext at the end of each iteration.

This procedure demonstrates the scalability of K-Means clustering in a parallelized setting and the effectiveness of caching in improving the efficiency of large-scale data processing with PySpark.

Comparison

The serial K-Means implementation processes datasets in a centralized manner, operating efficiently on small to medium-sized datasets within the constraints of a single machine's resources. It is straightforward in execution, with direct access to data and simplified debugging, but can be limited by memory and computational power when handling larger datasets. Visualization of centroids in this context is direct and uncomplicated, reflecting the clustering of more manageable datasets.

In contrast, the parallel K-Means implementation utilizes PySpark for distributed processing, effectively managing large-scale datasets by distributing computations across multiple nodes. This approach leverages the power of parallel processing to handle complex computations and vast amounts of data, significantly reducing execution time for large datasets. Caching plays a crucial role in enhancing performance, especially in iterative processes like K-Means, by minimizing data retrieval times. While setup and management are more complex due to the distributed nature of the computation, the scalability and efficiency gains are substantial, making it ideal for big data applications. Visualization in a parallel context also reflects the distributed nature of the computation, offering insights into the clustering of more complex datasets.

4 Experiments

In this section, we are going to visualize and explain the effects of the amount of workers, both from an execution time and speed-up perspective.

4.1 Parallel K-means

For this part, we have run our model 12 times, each time with an increasing number of workers assigned to it; so starting with a single worker up to 12 workers. It's important to say that the physical machine on which these experiments have been executed on an Apple M2 Pro, 10 cores processor, because we have tested also the behaviors of the system using more workers than cores available.

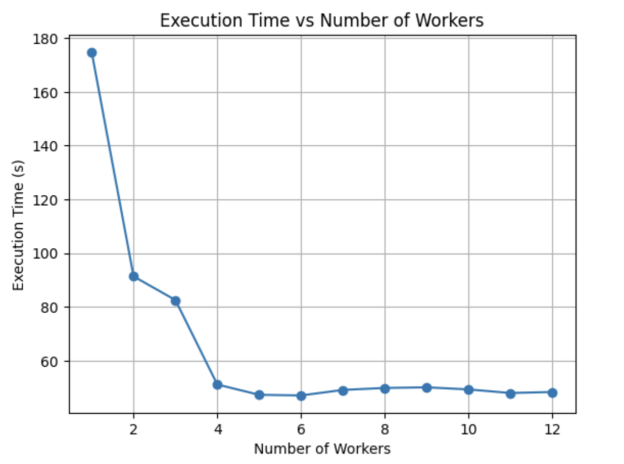

Figure 2: Performance Curve

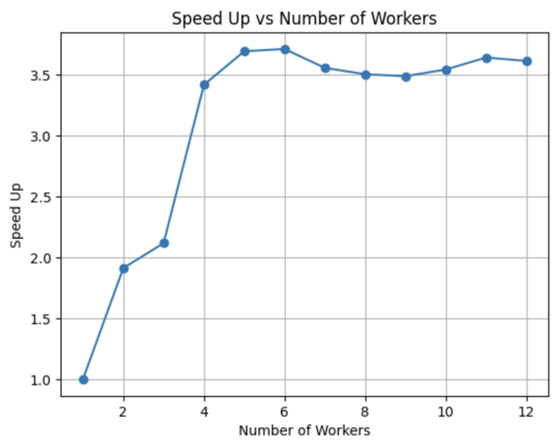

Figure 3: Speed Up Curve

From the plot in figure 2, we can identify 4 macro behaviors:

- **Sharp Decrease with Few Workers**: The execution time decreases sharply as the number of workers increases from 1 to 4. This suggests that the kNN algorithm benefits greatly from parallel processing initially, as the task is divided among more processors.
- **Plateau Beyond 4 Workers**: As more workers are added beyond 4, the execution time curve flattens, indicating that the addition of workers has little to no impact on decreasing execution time. This plateau suggests that there might be a limit to the parallel efficiency of the kNN algorithm.

The curve shows the speed-up changing the number of workers as the number of workers increases. This is because the number of workers increases the amount of parallelism that can be achieved, which can lead to significant speedups for tasks that can be parallelized. However, as the number of workers continues to increase, the curve starts to flatten out and eventually even decreases. This is due to the overhead of communication and coordination between the workers.

4.2 Clusters experiments

Another experiment we have done was to run the algorithm using different amounts of clusters to understand better:

1. If the speed-up was highly affected
2. How the algorithm behaves with a number of clusters different from the digits in the dataset, both with fewer than 10 clusters and with more clusters than digits.

We decided to run the algorithm using [4, 6, 8, 10, 12] for the number of clusters and we obtained the results showed in Figure 4.

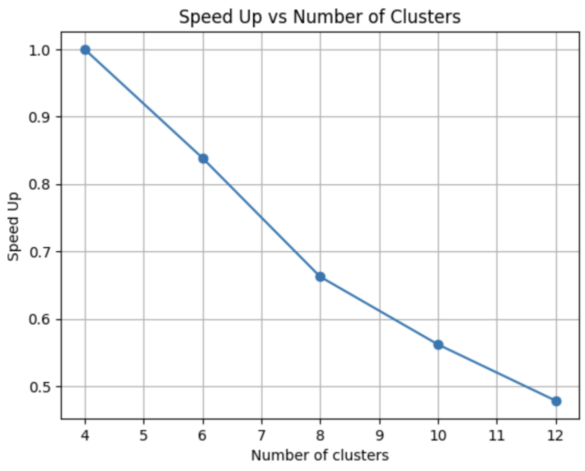

Figure 4: Speed Up vs Number of Clusters

With fewer clusters, each contains a larger subset of the dataset, meaning that during the clustering phase, a new sample only needs to be compared with a few cluster centroids to find its nearest cluster. This is a relatively low overhead process. Once the appropriate cluster is identified, the algorithm only needs to perform distance calculations within this cluster. As the number of clusters increases, the initial benefit is that each cluster becomes smaller, however, the increase introduces significant overhead. For each new sample, the algorithm must now compute and compare distances to more cluster centroids to determine the appropriate cluster. This overhead can quickly outweigh the benefits of having fewer comparisons to make within each cluster, especially as the number of clusters becomes very large relative to the size of the dataset.
As you can see in this example, the algorithm executed with 4 clusters obtained much more approximate centroids than the one with 12 clusters

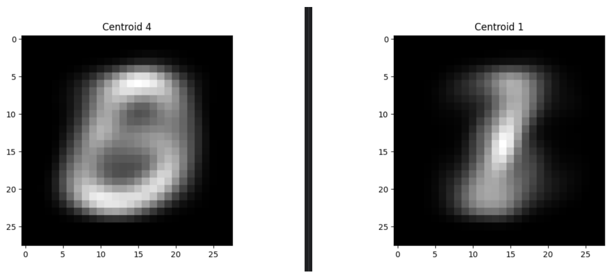

Figure 5: Examples of centroids of KNN ran with 4 clusters

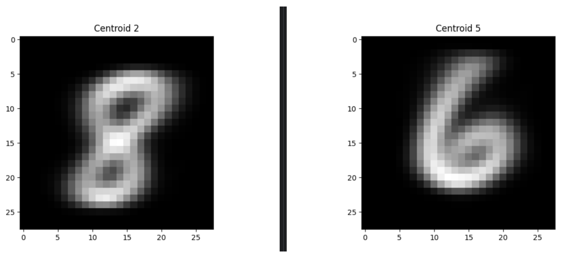

Figure 6: Examples of centroids of KNN ran with 12 clusters

5 Conclusions

The report comprehensively analyzes the K-Means clustering algorithm in both its traditional and parallelized forms. The traditional version, suitable for smaller datasets, is implemented using standard Python libraries and processes data into a NumPy array. Its simplicity and ease of use are highlighted, but it is limited by the computational constraints of a single system.

In contrast, the parallelized version, employing PySpark, is tailored for large-scale datasets. This approach leverages distributed processing, making it highly efficient and scalable. It utilizes Resilient Distributed Datasets (RDDs) for processing, highlighting its capability to handle vast amounts of data.

The comparison between the two versions underlines that the choice depends on the dataset size and the computing environment. The traditional version is ideal for smaller datasets due to its straightforward implementation, while the parallelized version excels in big data scenarios, offering enhanced scalability and performance. Despite the detailed exploration and comparison of both versions of the K-Means algorithm, more complex models may be necessary for certain tasks. This is particularly evident in the final results where, after extensive trials, the K-Means algorithm produced some unclear centroids. This suggests that for certain complex datasets or specific analytical needs, the simplicity of the K-Means algorithm might not suffice, and the exploration of more sophisticated machine learning models could be required to achieve clearer and more definitive clustering results.