Massively Parallel Machine Learning

A PROJECT REPORT

submitted by

TOMMASO BARONI
LUCA BESTAGNO
FRANCESCO MATTIOLI

on

Parallel Implementation and Evaluation of K-Means Algorithm

POLITÉCNICA

Master Universitario en Innovación Digital
Escuela Tecnica Superior de Ingenieros Informaticos
December 2023

Contents

1 Introduction

In the growing field of unsupervised machine learning, our report embarks on an explorative journey with the K-Means clustering algorithm, a cornerstone technique for data analysis and pattern recognition. This exploration is uniquely positioned at the intersection of machine learning and high-performance computing, employing Apache Spark and Python to realize a parallelized version of the K-Means algorithm. The MNIST dataset, a collection of 70,000 handwritten digits transformed into 784-dimensional vectors, serves as the challenging yet fitting subject of our study due to its size and complexity.

Our approach begins with implementing a serialized version of K-Means, laying the groundwork for understanding its core mechanics. This initial phase, involving functions like 'serialReadFile', 'serialAssign2cluster', and 'serialKMeans', establishes a foundational understanding and a performance baseline. The journey then progresses towards the realm of parallel computing, adapting the algorithm within Spark's robust ecosystem. This transition to a parallelized implementation is pivotal, aiming to leverage the distributed processing power to handle the MNIST dataset more efficiently.

The essence of this report lies in the comparative analysis between the serialized and parallelized implementations. We meticulously evaluate the algorithm's performance, focusing on execution times and the impact of increasing parallel workers, reflected in the speedup ratios. This analysis is crucial for quantifying the benefits of parallelization and optimizing the processing of large-scale data sets like MNIST.

Furthermore, the study extends into the realm of cluster interpretation and visualization, particularly for varying cluster counts (K values). This examination not only illustrates the algorithm's effectiveness in data grouping but also offers practical insights into its real-world applicability.

The report concludes with a comprehensive synthesis of our findings, documenting the dataset characteristics, algorithmic implementations, experimental results, and conclusions drawn from the study. Complementing the report, an oral presentation will encapsulate the key insights and outcomes of this exploration into parallelizing the K-Means algorithm.

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 $j = 1, .., K$:
      $d^j \leftarrow \|x_i - m^j\|$
    $j_0 \leftarrow argmin(d^1, \dots, d^K)$
    $C^{j_0} \leftarrow C^{j_0} \cup \{x_i\}$
  for $j = 1, .., K$:
    $m^j \leftarrow \frac{1}{|C^j|} \sum_{x_i \in C^j} x_i$

Figure 1: K-Means Algorithm

2 Dataset Description

The MNIST dataset is a cornerstone in the field of machine learning, particularly for tasks involving image recognition and unsupervised learning algorithms like K-Means clustering. In this assignment, we delve into the implementation of a parallelized version of the K-Means algorithm using Apache Spark and Python, with a focus on efficiently handling the MNIST dataset.

Characteristics of the MNIST Dataset:

- **Nature of Data**: The MNIST dataset is a collection of 70,000 images of handwritten digits, ranging from 0 to 9. It is widely used for training and testing in the field of machine learning.
- **Image Representation**: Each image in the dataset is a 28×28 matrix of black-and-white pixels, providing a straightforward yet challenging task for clustering algorithms.
- **Data Format**: In this dataset, each 28×28 pixel matrix is flattened into a 784-dimensional vector. Therefore, the dataset is structured into 70,000 rows, with each row representing one image, and 784 columns corresponding to the pixel values.
- **Pixel Values**: The pixel intensity in each image is represented by an integer ranging from 0 (white pixel) to 255 (black pixel), providing a grayscale representation of the handwritten digits.

Data Representation in Spark:

- **RDD Format**: For the parallelized implementation, the dataset will be loaded into Spark as a Resilient Distributed Dataset (RDD). This format is suitable for distributed computing and will enable efficient data handling and processing.
- **Record Structure**: In the RDD, each record represents an image, where each image is a 784-dimensional vector corresponding to the flattened pixel matrix.

In summary, the MNIST dataset offers a comprehensive and challenging environment for testing and evaluating the K-Means clustering algorithm in a parallelized setting. Its structure and complexity are ideal for demonstrating the capabilities of Spark in handling high-dimensional data and for showcasing the efficiency improvements gained through parallel computing techniques.

3 Algorithm Implementation

Comparison of Serial and Parallel `ReadFile` Functions

In the pursuit of implementing the K-Means algorithm for both centralized and parallelized environments, the `ReadFile` function plays a pivotal role in data ingestion. This function is developed in two forms: `serialReadFile` for the centralized approach and `parallelReadFile` for the parallelized approach using Spark. Below is a detailed comparison of these two implementations, highlighting the adaptations made for parallelization.

1. Serial Implementation: `serialReadFile`

The `serialReadFile` function is designed for a centralized environment, typically run on a single machine. It uses the Pandas library, a popular choice for data manipulation in Python.

- **Function Signature**: `def serialReadFile(filename):`
  - `filename`: Path to the CSV file.
- **Workflow**:
  - The function reads a CSV file into a Pandas DataFrame using `pd.read_csv`. This approach is straightforward and effective for datasets that fit into the memory of a single machine.
  - It then drops the 'label' column from the DataFrame using `df.drop`. This is a simple operation in Pandas, signifying the removal of a non-feature column. Then we have also removed the header since it us not useful for our purposes.
- **Return Value**: The function returns the processed DataFrame.

2. Parallel Implementation: `parallelReadFile`

The `parallelReadFile` function, on the other hand, is crafted for a distributed environment, using Apache Spark's RDD (Resilient Distributed Dataset) abstraction. This approach is essential for handling large datasets that exceed the memory capacity of a single machine.

- **Function Signature**: `def parallelReadFile(filename):`
  - `filename`: Path to the CSV file.
- **Workflow**:

  - The CSV file is read into an RDD with `sc.textFile`. Unlike Pandas, Spark's RDD is designed for distributed computing, allowing the data to be processed across multiple nodes.
  - The first line of the CSV, usually the header, is identified using `rdd.first()`.
  - A `flatMap` transformation is used to filter out the header, ensuring that subsequent transformations are applied only to the data.
  - The `map` transformation is then used to process each line of the RDD. It removes the first column (assuming it's the 'label') and converts the remaining elements to floats. This step showcases the power of Spark's lazy evaluation, where transformations are applied in a distributed manner across the dataset.

- **Return Value**: The function returns the processed RDD.

Summary

In the parallelReadFile function, flatMap is strategically employed to exclude the header from the data processing steps, ensuring that operations like converting string values to floats are only applied to actual data. The map function then transforms each line of data, indicating Spark's capability to execute complex transformations in a distributed manner, efficiently preparing the data for analysis across the cluster. This is in contrast to the serialReadFile function, which uses Pandas for more direct and less distributed data manipulations.

Comparison of Serial and Parallel `Assign2cluster` Functions

The `Assign2cluster` function is a critical component of the K-Means algorithm, responsible for assigning data points to the nearest centroid. We examine the differences between the serial version, `serialAssign2cluster`, and its parallel counterpart, `parallelAssign2cluster`.

1. Serial Implementation: `serialAssign2cluster`

The `serialAssign2cluster` function is intended for a serial or centralized computing environment.

- **Function Signature**: `def serialAssign2cluster(centroids, x):`
  - `centroids`: A list of centroids.
  - `x`: The data point for which the nearest centroid is to be identified.
- **Workflow**:
  - The function calculates the Euclidean distance from the data point `x` to each centroid in the list `centroids`. It is calculated as it follows: $\sqrt{\sum_{i=1}^{n}(x_i - c_i)^2}$

– It then identifies the closest centroid by finding the index of the smallest distance in the list.

- **Return Value**: Returns the index of the nearest centroid.

2. Parallel Implementation: `parallelAssign2cluster`

The `parallelAssign2cluster` function adapts the same core logic for a parallelized environment, typical in distributed computing frameworks like Spark.

- **Function Signature**: `def parallelAssign2cluster(centroids, x):`
  - `centroids`: A list of centroids.
  - `x`: The data point for which the nearest centroid is to be identified.
- **Workflow**:
  - Similarly, it calculates the Euclidean distance from the data point `x` to each centroid.
  - Finds the index of the nearest centroid.
- **Return Value**: In addition to returning the index of the closest centroid, it returns the data point `x` itself, converted to a numpy array. This modification is particularly useful in a parallelized setup for further distributed computations.

Summary

Both the serial and parallel versions of the `Assign2cluster` function fundamentally perform the same task of assigning data points to the nearest centroid based on Euclidean distance. The key distinction lies in the return value of the parallel version, which is tailored for subsequent distributed operations, a necessity in parallel computing environments like Spark. This adaptation showcases the changes required when transitioning algorithms from a serial to a parallelized context.

Comparison of Serial and Parallel `KMeans` Functions

The implementation of the K-Means clustering algorithm in both serial and parallel environments is a crucial aspect of this study. Below, we compare the ‘serialKMeans‘ and ‘parallelKMeans‘ functions, highlighting the key differences and adaptations for parallelization.

1. Serial Implementation: `serialKMeans`

The 'serialKMeans' function is tailored for execution in a centralized, single-machine environment.

- **Function Signature**: `def serialKMeans(X, K, n_iter):`
  - `X`: The dataset.
  - `K`: The number of clusters.
  - `n_iter`: The number of iterations.
- **Workflow**:
  - Randomly initialize centroids using numpy's random function.
  - Iterate for a given number of iterations:
    - For the assignment step, use `serialAssign2cluster` to assign each data point to the closest centroid.
    - Append each data point to the corresponding cluster based on the assignment.
    - In the update step, recalculate each centroid by computing the mean of all points assigned to it.
    - If a cluster is empty, reinitialize its centroid randomly.
  - After completing all iterations, return the final positions of the centroids.
- **Return Value**: Returns the final centroid positions after all iterations.

2. Parallel Implementation: `parallelKMeans`

The 'parallelKMeans' function adapts the K-Means algorithm for a distributed computing environment using Spark.

- **Function Signature**: `def parallelKMeans(data, K, n_iter):`
  - `data`: The dataset (Spark RDD).
  - `K`: The number of clusters.
  - `n_iter`: The number of iterations.
- **Workflow**:
  - Similar to the serial version, centroids are randomly initialized.
  - For each iteration:
    - The Spark `map` transformation assigns each data point in the RDD to the nearest centroid. This step is parallelized across the cluster, allowing each node to process a subset of the data independently.

    * After assignment, the `reduceByKey` operation is used to aggregate data points belonging to each cluster. This operation efficiently combines values with the same key (in this case, the centroid index), performing sum and count in a distributed manner. It significantly optimizes the process of updating centroids by reducing network communication and distributing the load across multiple nodes.
    * The new centroids are calculated by averaging the points in each cluster. This step is conducted using the `map` transformation, which processes the sum and count results from `reduceByKey` to compute the mean. The use of `map` here allows for parallel processing, as each centroid's mean calculation is independent of others.
    * Finally, the `collect` action is used to gather the updated centroids back to the driver program. While `collect` is not parallelized itself, it is essential to consolidate the results of the distributed computations.

- **Return Value**: Returns the final centroids, calculated using parallel operations.

Summary

In the serial serialKMeans, data points are assigned to the nearest centroids and centroids are updated, using NumPy for operations. The usage of `map` and `reduceByKey` speeds up both the assignment of points to centroids and the recalculation of centroids.

4 Experiments

In this section we will delve into our model performances with different workers, in particular we will analyze the execution speed compared with the number of workers and the speedup ratio, caluclated as $\frac{t_{1\text{worker}}}{t_{n\text{workers}}}$. We have also calculated the speedup over the number of clusters, we fixed the number of workers at 10 and we runned the k-means algorithm with a number of clusters k = [3, 5, 7, 8, 9, 10, 11] to see the different results. In the computational experiments, logical cores beyond the physical ones were utilized. This meant engaging the system's hyper-threading capabilities to parallelize operations further. While hyper-threading can increase the throughput of processes by leveraging idle CPU cycles, it's not always a guarantee of performance improvement, particularly for compute-intensive tasks that can already fully engage physical cores. The choice to use logical cores was a deliberate one to explore the potential for performance gains in the context of the experiments conducted.

It follows an example of centroids obtain after the 10 iteration of the kmeans with 10 clusters, we will show only four of them, Fig. 2 .

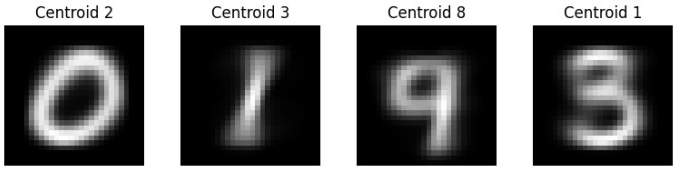

Figure 2: Some centroids

Performances

Different Workers Performances

To analyze the performances of the model with different number of workers we have used a number of worker ranging from 1 to 12. Analyzing the KMeans performance based on charts in Fig. 3, the execution time decreases sharply as the number of workers increases from 1 to around 4. Beyond this point, the improvement plateaus, indicating that adding more workers doesn't significantly reduce execution time, possibly due to overheads or limits in parallelization efficiency. The speedup curve corroborates this, showing a rapid increase up to 4 workers and then flattening, suggesting diminishing returns on speedup with more workers. This performance pattern is common in parallel computing where, beyond a certain point, additional resources do not translate into proportional gains.

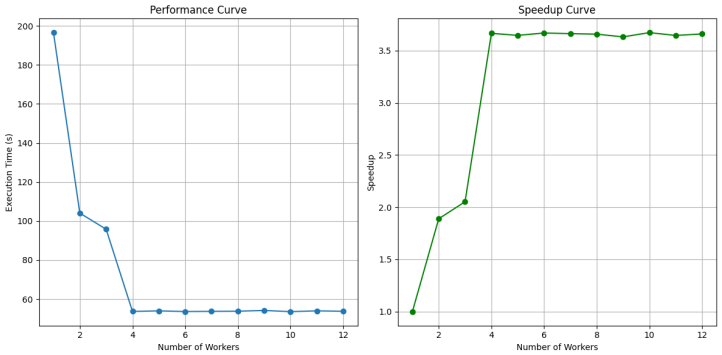

Figure 3: Performances of K-means with different workers

Different Clusters Performances

K-Means clustering performance with varying cluster numbers shows the speedup ratio as the number of clusters K increases, Fig. 4. A clear downward trend in the speedup ratio is visible as K grows, indicating a decrease in performance gains. This suggests that larger cluster counts intensify computational complexity, leading to diminished relative speedup. Additionally, while smaller cluster numbers yield poor cluster definition, increasing K improves cluster clarity, yet the perfect distinction of all 10 categories remains unachieved, hinting at limits to performance scaling with cluster number augmentation.

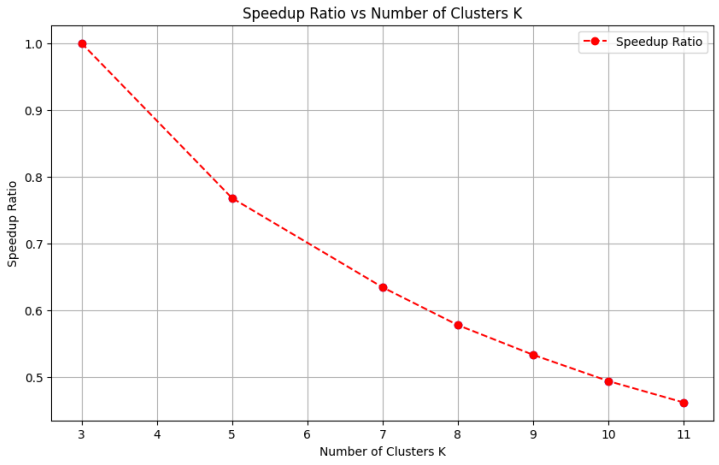

Figure 4: Performances of K-means with different number of clusters

5 Conclusions

In conclusion, this report has demonstrated the significant advantages of parallel processing over traditional serial methods in machine learning tasks. By distributing computations across workers using Apache Spark, we achieved notable improvements in execution times and efficiency. Our k-means clustering experiment underscored the impact of cluster size on performance, revealing the trade-offs between computational load and the clarity of results. The adoption of logical cores for hyper-threading sometimes led to performance gains, but it also highlighted the importance of matching workload characteristics to system capabilities.