UNIVERSITY OF PISA

Master's Degree Artificial Intelligence and Data Engineering

Cloud Computing Course

Parallel K-Means in MapReduce

Work Group:
Arduini Luca
De Marco Angelo
Pardini Marco

ACADEMIC YEAR 2022/2023

INTRODUCTION

Our project for the Cloud Computing course focuses on implementing the k-means algorithm using the distributed computing paradigm through the use of Hadoop. The main objective of the study was to develop and evaluate the implementation of the k-means algorithm in a distributed environment. To achieve this objective we utilized the Hadoop framework, which provides a distributed processing environment for handling large datasets.
After completing the implementation, we shifted our focus to performance evaluation. We conducted several tests, varying the input parameters such as the number of clusters, dataset size, and number of algorithm iterations. Through these evaluations, we were able to study the impact of these parameters on the overall performance of the k-means algorithm in a distributed environment.

The code is available on GitHub at: github.com/LucaArduini/k-means_in_MapReduce

CONTENTS

INTRODUCTION i

CONTENTS ii

1 K-MEANS ALGORITHM 1

1.1 THE ORIGINAL K-MEANS ALGORITHM 1

1.2 PARALLELIZING THE K-MEANS ALGORITHM. 2

1.2.1 Map Algorithm. 2

1.2.2 Reduce Algorithm 3

1.2.3 The necessity of a Combiner Algorithm 3

1.2.4 From Combiner to In-Mapper Combiner. 4

2 IMPLEMENTATION. 5

2.0.1 `KMeans.java`. 5

2.0.2 `Point.java`. 8

2.0.3 `ClusteringFeature.java` 8

3 RESULTS 9

3.0.1 Testing methodology 9

3.0.2 Results Discussion 10

A CODE 15

A.0.1 `Point.java`. 15

A.0.2 `ClusteringFeature.java` 16

1 K-MEANS ALGORITHM

1.1 THE ORIGINAL K-MEANS ALGORITHM

A clustering algorithm is a type of machine learning algorithm that is used to group similar data points together based on their inherent patterns or similarities. The goal of clustering is to identify natural groupings or clusters within a dataset, where data points within the same cluster are more similar to each other than to those in other clusters.

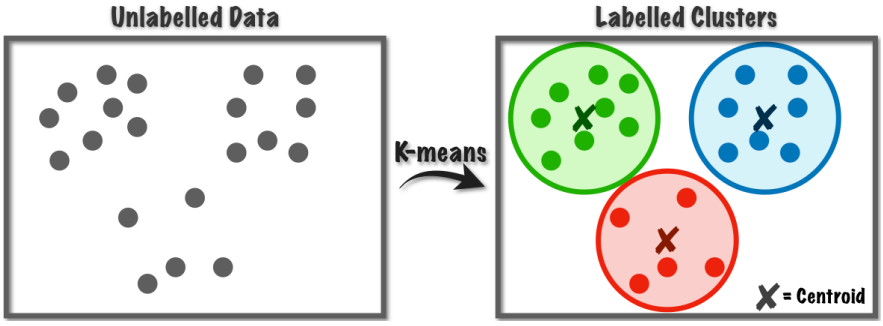

Figure 1.1. k-means algorithm

The K-means algorithm is one of the most powerful and popular data mining algorithms in the research community because of its simplicity and speed.
The fundamental principle behind K-means revolves around the concept of partitioning a set of objects into K groups or clusters. This partitioning is aimed at achieving a high level of similarity within each cluster while maximizing dissimilarity between elements belonging to different clusters. The process typically relies on a distance metric, such as the Euclidean distance, to assess the similarities and dissimilarities.
Clustering is carried out through an iterative algorithm. Initially, K centroids, which serve as representative points, are randomly selected. Each object is then assigned to the cluster whose centroid is closest in proximity. After the initial assignment, the centroids are updated by calculating the mean of the points assigned to each cluster. This iterative process continues until the centroids converge, indicating that they have stabilized and exhibit minimal changes.
Upon completion of the clustering process, each object is assigned to the cluster based on its

proximity to the corresponding centroid. Determining the appropriate number of clusters, K, typically depends on factors such as the dataset size and the desired level of granularity.
The popularity of the K-means algorithm stems from its undeniable strengths, such as its ease of implementation and relatively short execution time. However, it also has its drawbacks, including its dependence on the initial choice of centroids.

1.2 PARALLELIZING THE K-MEANS ALGORITHM

In order to follow MapReduce paradigm, we need to divide in Map and Reduce task the two main procedures of the original algorithm:

1. At the start of the iteration, Points need to be assigned to the nearest centroid
2. After membership are assigned, a new centroid needs to be computed for that cluster, making the mean of all points belonging to it.

The idea is to make Mapper responsable for assigning membership, and Reducer responsable to compute the new centroid, receiving the ”list of membership” from the Mapper. Things will be slightly more complicated when it will be necessary to add a Combiner, that will lead us to make some modifications to original algorithms.

1.2.1 Map Algorithm

The role of the mapper in this first implementation of Map Reduce algorithm for parallelize K-Means is really simple: It gets a Key-Value pair, where the Key is the offset in byte of the Input Split the mapper is reading (default configuration) and so It won't be used in the algorithm.

Regarding the value, this represent a sample of the dataset, X. Moreover, It assumed that mapper knows centroids before starting the entire Task.

Algorithm 1: MR K-Means (Mapper)

Data: Centroid List C : $\{c_i\}$, Sample X
Result: Index of X' cluster $index$, Sample X
Procedure *MAP(k, X)*

1: index $\leftarrow indexNearestCentroid(C)$;
2: EMIT(index, X);

Basically, the function *indexNearestCentroids(C)* will return the below index:

$$index = argmin_i ||c_i - X||_2$$

1.2.2 Reduce Algorithm

The role of reducer is as simple as the mapper role. Since we are sure that the same key is forwarded to the same reducer, we know that if a reducer sees Its input (K, L(V)) pair, It will be ensured that L(V) will be the list of all samples assigned to K-th cluster. With this observation, we write this simple algorithm, to make reducer able to compute the new centroids.

Algorithm 2: MR K-Means (Reducer)

Data: Cluster Index : $index$, List of Samples X assigned to C_{index} : L
Result: Cluster Index : $index$, New Centroid for C_{index} : c_{new}
Procedure *REDUCE(index, L)*

```
1 sum ← 0;
2 for sample in L do
      sum ← sum + sample ;                          // Element-Wise Sum
  end
3 EMIT(index, sum/L.size())
```

1.2.3 The necessity of a Combiner Algorithm

The problem of Map function is that It outputs on the network one (Key, Value) pair *for each object* of the dataset. When dataset is really large, this could be an huge amount of data (O(n*dimension of points)). In order to reduce such a number, we implemented a Combiner.

The role of this combiner is to collect Mapper outputs and aggregate all points belonging to the same cluster, returning an object representing a couple (Partial Sum — Number of Points). The reason why the Combiner is not exactly equal to the reducer depends on the fact that the mean (computing the new centroid) is not an associative operation.

Algorithm 3: MR K-Means (Combine)

Data: Cluster Index : $index$, List of Samples X assigned to C_{index} : L
Result: Cluster Index : $index$, Descriptive Couple of the Centroid :
$(partialSum, numPoints)$
Procedure *COMBINE(index, L)*

```
1 sum ← 0;
2 for sample in L do
      sum ← sum + sample ;                          // Element-Wise Sum
  end
3 EMIT(index, (sum, L.size()))
```

By consequence, Reducer needs a little update, because It now receives a different kind of input (a list of descriptive couples):

Algorithm 4: MR K-Means (Reducer 2.0)

Data: Cluster Index : $index$, List of Descriptive Couple relative to C_{index} : L
Result: Cluster Index : $index$, New Centroid for C_{index} : c_{new}
Procedure *REDUCEv2(index, L)*

```
1 sum ← 0
2 howMany ← 0
3 for sample, num in L do
      sum ← sum + sample ;                          // Element-Wise Sum
      howMany ← howMany + num
  end
4 EMIT(index, (sum/howMany))
```

1.2.4 From Combiner to In-Mapper Combiner

The next phase of our development was to optimize the role of Combiner, in a way to reduce the number of data transferred on system buffers. In order to do that, Combiner has to be included in the Mapper class, using the *setup* and *cleanup* method.

The idea is to modify the mapper in a way such that he aggregates local data referred to the same cluster, creating descriptive couples (such as Combiner already does).

Algorithm 5: MR K-Means (Mapper 2.0)

Data: Centroid List C : $\{c_i\}$, Input Split I
Result: List of indexes $index$, Descriptive Couple of the $Cluster_{index}$: $(partialSum, numPoints)$
Procedure *INITIALIZE()*

```
1: H ← InitializeListDescriptiveCouples();
```

Procedure *MAP(key, X)*

```
1: index ← indexNearestCentroid(C);
2: H[index].numPoints++
3: H[index].partialSum ← H[index].partialSum + X ;        // Element-Wise Sum
```

Procedure *CLOSE()*

```
1: for index, entry in H do
2:     EMIT(index, entry)
       end
```

2 IMPLEMENTATION

The code of our project is divided into three files, which will be presented in the following subsections.

2.0.1 `KMeans.java`

This is indeed the most important file. It contains the main() function, which encompasses the entire logic of the application. Here, the configuration and submission of jobs take place, along with the preparation phase of the k-means algorithm. Within this file, you will also find the KMeansMapper and KMeansReducer classes.

The **main()** function takes 7 input parameters:

- `input`, the path of the input file
- `k`, the value of **k** to be used in the algorithm
- `max_iter`, the maximum number of iterations allowed before a job is stopped
- `output`, the path of the output file
- `dim`, indicates the dimension of the input points
- `epsilon`, indicates the maximum error tolerated by the cost function
- `num_reducer`, indicates the number of reducers to be used

These and other parameters will be used to configure the algorithm. Specifically, they will set the input and output files, the number of reducers to be used, the output classes of the algorithm, and many other settings. Once these parameters are configured, before submitting the job, it is necessary to choose the initial values for the centroids. After initializing the centroids, the first iteration of the k-means algorithm can begin, which corresponds to launching the first job.

At the end of each job, an immediate check is performed to see if a cluster is empty. This occurs when no points are assigned to a centroid during the mapping phase. In such cases, the algorithm

must resolve this issue, and it can do so using various policies, which we will discuss later.
After this check, the cost function is computed. In our algorithm, this function corresponds to the sum of movements of the k centroids compared to the previous iteration:

$$\sum_{i=1}^{k} \|c_i^k - c_i^{k-1}\|_2$$

If the total movement of centroids is less than the user-defined epsilon parameter given as input, no further iterations are required, as the desired precision has been achieved.
If the centroids have moved significantly, it means that additional iterations are needed, and the algorithm is restarted with the updated centroid positions as input.

Initialization of the centroids

There are different techniques to choose from in the literature. We have considered the following ones:

- One method is to randomly select the positions of the k centroids, in which case the random generation must be limited to the values taken by the other points in the dataset. Although this policy aligns perfectly with the basic idea of the algorithm, the algorithm's performance with these initial points is not optimal.
- The method we have chosen to use involves randomly selecting the centroids from within the pool of points available in the input file, which is stored in HDFS. This approach yields statistically better results. Additionally, this method ensures that in the first iteration there are no empty clusters, as they will contain at least the point with the same coordinates as the centroid.

Managing an empty cluster

As mentioned earlier, an empty cluster occurs when no points are assigned to a centroid during the mapping phase. If this event occurs, it is crucial to take action because in the algorithm, the centroids move based on the points that form their cluster. If a cluster is empty, the respective centroid will not move for that iteration and likely not in the subsequent ones, which poses a significant problem.
Also in this case, there are multiple strategies to address this issue:

- One possibility is to reduce the number of centroids from k to k-1. This is because, according to the logic of the algorithm, if a cluster remains empty, it is likely because the user-defined parameter k is not suitable for that specific dataset. However, we have chosen not to use this strategy as it goes against the algorithm's goal of finding exactly k clusters. It is up to the user, based on the results obtained from various tests, to determine the most appropriate value of k for their dataset.

- Another option is to manually ”reposition” the problematic centroid. In this case, only the problematic centroid is re-initialized, while leaving the other centroids unchanged. However, we believe that this option may not be the best approach, as the new position of the centroid may disrupt the cluster of another centroid that was already well-positioned. This could lead to numerous iterations to reposition both centroids.
- The solution we consider to be the best is to re-initialize all the centroids if an empty cluster occurs. This is because the k-means algorithm is known to converge in fewer iterations when the initial centroids are well-distributed. By starting over and re-initializing all the centroids, we can expect to converge in fewer iterations compared to the previous case where there were the possibility to have two centroids very close to each other and required more iterations to move apart.

KMeansMapper

The `KMeansMapper` class, as the name suggests, implements the Mapper of our algorithm. It consists of three functions:

- The `setup()` function is responsible for populating the two data structures used to store cluster information. To do this, it reads the positions of the centroids computed in the previous iteration of the k-means algorithm. In the case of the first iteration, it will use randomly chosen values as the initial centroids.
- The `map()` function, which is called on every single line of the split input, is responsible for parsing the line and initializing a new Point object. Once created, it calculates the nearest centroid for that point. After identifying the nearest centroid, the point is added to the collection of points assigned to that centroid. This pre-aggregation of points assigned to each centroid helps anticipate the total sum of points that will be processed in the Reducer.
- The `cleanup()` function simply emits the obtained results so that they can be read by the Reducer.

It is important to note that the Mapper implementation follows the ”in-mapper combiner” programming paradigm. This approach allows the Mapper functions described above to perform some of the tasks that would traditionally be handled by the Reducer. This design choice aims to minimize network and system memory overhead by aggregating data locally within the Mapper before sending it to the Reducer. By reducing the volume of data transmission, the ”in-mapper combiner” approach improves the overall efficiency and performance of the MapReduce job.

KMeansReducer

The `KMeansReducer` class represents the Reducer of our algorithm and it has a single function called reduce(). This function is straightforward. For each centroid, it collects all the partial sums calculated by the Mappers, calculates their total sum, and computes the average, which indicates

the new location of that centroid. Subsequently, a key-value pair is emitted, consisting of the centroid index and its new coordinates.
It is worth noting that during the calculation of the average if a cluster is found to be empty, the function marks the corresponding centroid with symbolic values. These values are later detected by the `main()` function, which then re-initializes the centroids before starting the next iteration.

2.0.2 `Point.java`

This class provides a logical representation of a point within our dataset. It consists of two fields: an integer variable indicating the vector dimension of the point, and a vector containing its coordinates.
The class contains numerous functions that perform various calculations on a Point object within the algorithm. For example, the `distance()` function takes a centroid as a parameter and returns the distance between the point and the centroid. Another function, `nearestCentroid()`, utilizes the `distance()` function to find the closest centroid to the point on which it is called.
Two important functions, `write()` and `readFields()`, are introduced by the Writable interface. They enable the serialization and deserialization of data, facilitating the exchange of information within the job.
There are also other utility functions that are not mentioned here for brevity, but you can find an excerpt of the code for this class in Appendix A of the document.

2.0.3 `ClusteringFeature.java`

This class does not specifically represent a logical object but was created to hold information about a specific cluster. In fact, an instance of this class will be created for each cluster in order to contain its information.
This class has two fields:

- The first field is an object of type ”Point”. However, it does not represent an actual data point from the dataset. Instead, it is used to hold the partial sum of the points in a cluster that is calculated by the map() function. The reason for using the Point class is that it provides convenient functions that are already implemented for points, saving the need for redundant code.
- The second field is a numeric variable that stores the count of points contributing to the partial sum. This count is essential for calculating the average of the points in the cluster.

Apart from these two fields, there isn’t much else. There are just a few utility functions that facilitate the final calculation of the average.
An excerpt of this file is also available in Appendix A.

3 RESULTS

In this chapter we're going to deeply analyze the results obtained by the moltitude of tests ran on Hadoop.

3.0.1 Testing methodology

In this subsection, we will discuss the testing approach used to evaluate the K-means algorithm. To ensure reliable and accurate results, a systematic testing methodology was followed.

- `Test Design:` The testing process began with the design of appropriate test cases. The test cases were designed to cover various scenarios and edge cases to thoroughly evaluate the performance and correctness of the implemented K-means algorithm.
- `Data Generation:` To simulate real-world scenarios, a diverse set of data points were generated by using the Python Sklearn method makeblobs(). These data points were used as input for the K-means algorithm during the tests. Care was taken to ensure that the generated data sets were representative of the expected input the algorithm would encounter in real-world applications. In particular:
  - `Number of objects:` 10k, 100k, 400k;
  - `Number of dimensions:` $10, 30$;
  - `Number of clusters:` $3, 4, 10$;
  - `Number of reducers:` $1, 2$;
  - `Epsilon:` $0.1, 0.01$.
- `Test Execution:` Each test was executed using the MapReduce framework on Hadoop. The tests were performed multiple times to ensure consistency and reliability of the results. Specifically, five test runs were carried out for each test case to account for any variations or outliers in the results.
- `Result Aggregation:` The results obtained from the multiple test runs were aggregated to provide a more reliable and accurate representation of the algorithm's performance. For each test case, the average of the five test results was calculated to determine the final outcome.

- `Performance Evaluation:` In addition to the average results, performance metrics such as execution time and resource utilization were measured and analyzed. These metrics helped assess the efficiency and scalability of the K-means algorithm implementation on the Hadoop framework.

3.0.2 Results Discussion

In this subsection, we will analyze the collected results from the testing phase of the K-means algorithm implemented using MapReduce on Hadoop. Through multiple test runs, we have obtained a comprehensive dataset to evaluate the algorithm's performance.

Number of objects

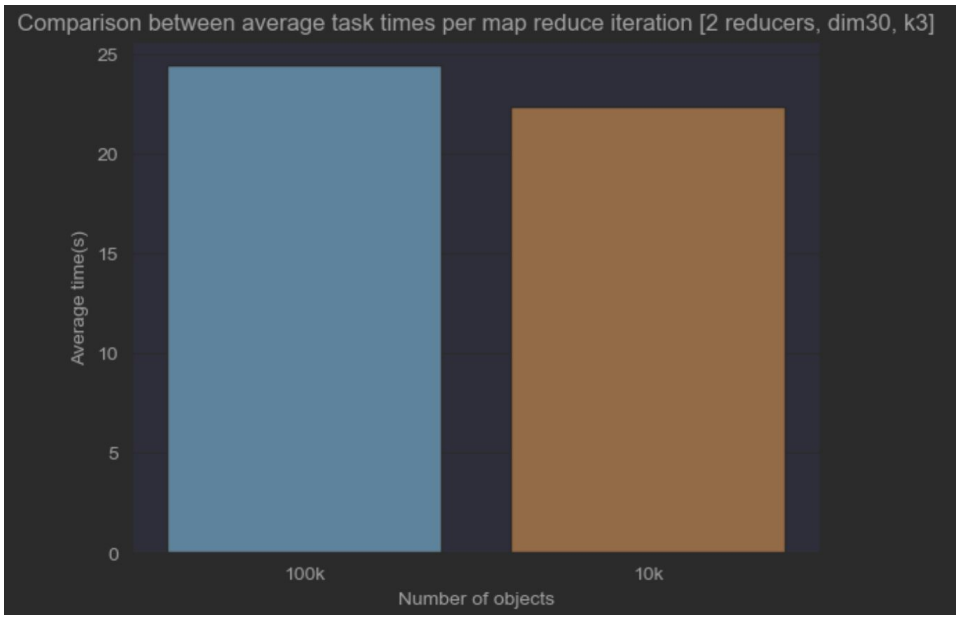

Figure 3.1. Average execution time per iteration varying the number of objects.

Based on the test results obtained, it was observed that the average time per iteration increased as the number of objects in the dataset increased. Specifically, when working with a dataset of 100,000 objects, the average time per iteration was approximately 24 seconds. In contrast, with a dataset of 10,000 objects, the average time per iteration was around 22.5 seconds.

These results indicate a clear correlation between the dataset size and the average time per iteration. As the number of objects in the dataset increases, the computational complexity of the K-means algorithm also grows. This leads to longer execution times per iteration, as the algorithm needs to process and update a larger number of data points in each iteration.

The observed trend highlights the importance of considering the dataset size when assessing the performance of the K-means algorithm. As the dataset becomes larger, the algorithm requires more computational resources and time to converge. Therefore, it is essential to take into account

the scalability and efficiency of the algorithm when working with larger datasets to ensure timely and accurate results.

Number of dimensions

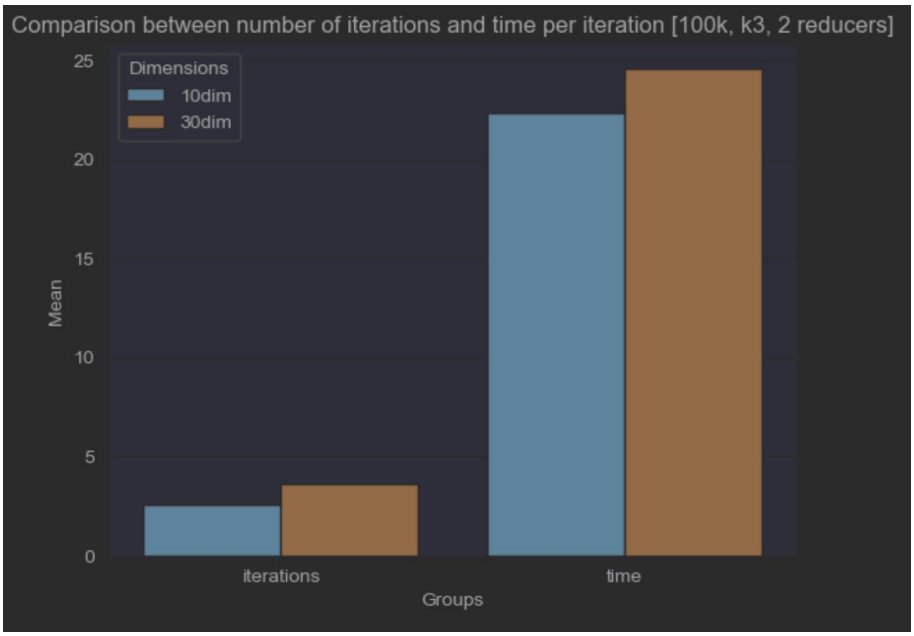

Figure 3.2. Average iterations and execution time per iteration varying the number of dimensions.

The influence of the number of dimensions on the performance of the K-means algorithm is another important aspect to consider. Upon conducting tests with different dimensionality settings, we observed interesting results.

For a dataset consisting of 100,000 objects and 3 clusters, with 10 dimensions, the algorithm took an average of 2.5 iterations to converge. In contrast, with 30 dimensions, the average number of iterations increased to 3.5. This indicates that as the dimensionality of the dataset grows, the algorithm requires more iterations to reach convergence.

Furthermore, we examined the average execution time per iteration for each dimensionality scenario. With 10 dimensions, the average execution time per iteration was approximately 22 seconds, while for 30 dimensions, it was slightly less than 25 seconds. Although the difference in execution time per iteration is relatively small, it is worth noting that higher dimensionality tends to lead to slightly longer execution times.

These results demonstrate the impact of dimensionality on the performance of the K-means algorithm in a MapReduce and Hadoop environment. The increase in the number of dimensions introduces additional complexity, resulting in a higher number of iterations needed for convergence and slightly longer execution times per iteration.

Epsilon

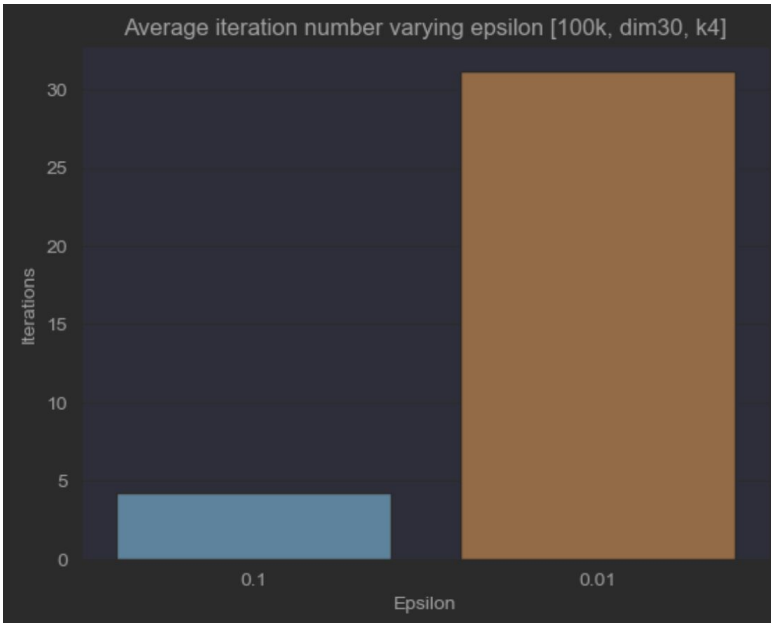

Figure 3.3. Average number of iterations varying Epsilon.

In our K-means algorithm implementation, we have incorporated an epsilon parameter, which serves as a threshold for determining when to terminate the iterations. Epsilon represents the sum of the distances between centroids in the previous iteration and the current iteration. By examining the behavior of the algorithm with different epsilon values, we gain insights into the impact of this parameter on the convergence process.

During our tests, we set the epsilon value to 0.1 for a dataset consisting of 100,000 objects, 30 dimensions, and 4 clusters. This configuration resulted in an average of 4 iterations to reach convergence. However, when we lowered the epsilon value to 0.01, the algorithm required an average of 32 iterations to converge. Notably, in three out of the five executions, the algorithm reached the maximum allowed number of iterations, which was set to 50 in our implementation.

These results highlight the significant influence of the epsilon threshold on the convergence behavior of the K-means algorithm. A higher epsilon value allows for faster convergence with a smaller number of iterations. On the other hand, setting a smaller epsilon value necessitates more iterations to achieve convergence, potentially leading to a longer execution time.

Choosing an appropriate epsilon value requires a trade-off between convergence speed and accuracy. A smaller epsilon can ensure a more precise clustering result but may lead to increased computational cost due to the higher number of iterations required.

Number of reducers

In order to investigate the impact of the number of reducers on the average time per iteration, we conducted tests using three different datasets: one with 3 clusters, another with 4 clusters, and a

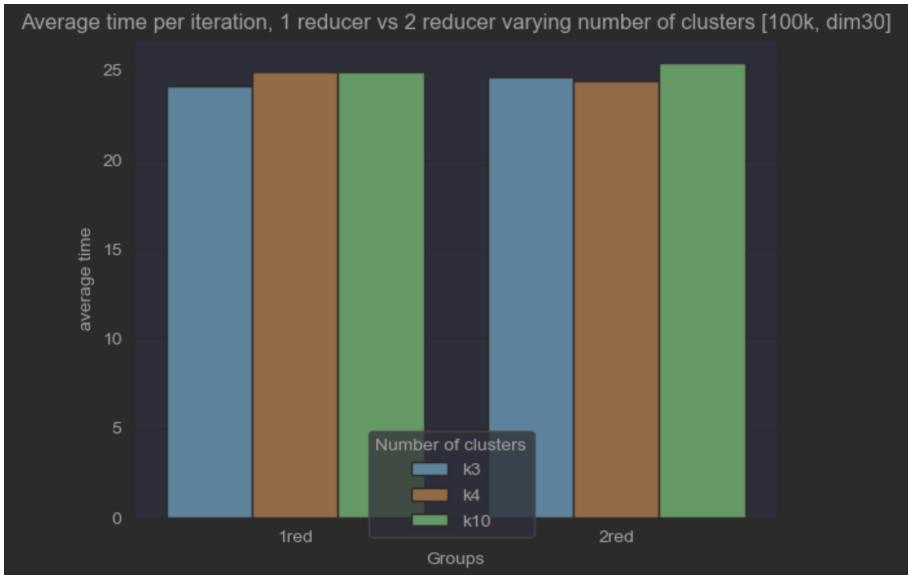

Figure 3.4. Average time per iteration varying the number of reducers.

third with 10 clusters. All datasets consisted of 100,000 objects and 30 dimensions.

During the tests, we found that when using a single reducer, the average time per iteration was consistently around 24.5 seconds, regardless of the number of clusters in the dataset. Similarly, when utilizing two reducers, we obtained similar results with an average time per iteration also around 24.5 seconds.

These findings suggest that, in our particular implementation, the number of reducers does not significantly impact the average time per iteration for the tested scenarios. The observed consistency across different cluster counts indicates that the algorithm's performance remains stable in terms of time per iteration, regardless of the reducer count.

Based on these results, it appears that with the given dataset sizes and dimensionality, employing a single reducer is sufficient for achieving optimal performance. There is no discernible advantage in introducing a second reducer in terms of reducing the average time per iteration.

It is worth noting that these conclusions are specific to our implementation and the particular dataset configurations tested. Different datasets, larger dataset sizes, or more complex algorithms may exhibit varying behavior in response to the number of reducers.

Parallelization vs Sequential approach

To assess the benefits of utilizing MapReduce in terms of execution time, we conducted a final test comparing the performance of our algorithm implemented in Hadoop against the Python algorithm. The test involved two datasets: one consisting of 100,000 objects and another with 400,000 objects, both with 30 dimensions and 4 clusters.

In the Python implementation, without leveraging the parallelization capabilities of Hadoop, we obtained an average time per iteration of 0.07 seconds for the smaller dataset and 0.24 seconds

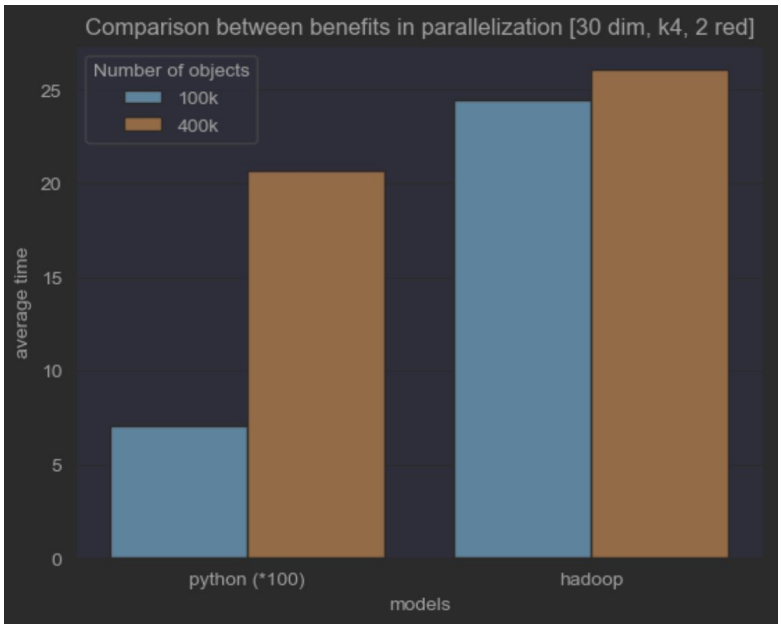

Figure 3.5. Benefits obtained using MapReduce parallelization.

for the larger dataset. It is important to note that in order to make these results comparable on the graph, we multiplied the Python execution times by 100.

In contrast, when running the algorithm on Hadoop using MapReduce, we achieved an average time per iteration of 23 seconds for the smaller dataset and 26 seconds for the larger dataset. Although the Hadoop execution times were significantly higher than the Python results, it is crucial to consider the significant difference in scalability between the two approaches.

The key objective of the graph generated from these results is to highlight the advantage of MapReduce in handling larger datasets. The sequential approach in Python exhibited a substantial increase in execution time as the number of objects rose. On the other hand, the MapReduce framework provided by Hadoop demonstrated more consistent execution times, irrespective of the dataset size.

By parallelizing the computation across multiple nodes in a Hadoop cluster, MapReduce enables efficient processing of large-scale data. This parallelization effectively mitigates the performance degradation experienced in a sequential execution, as demonstrated by the relatively stable execution times observed in the Hadoop-based implementation.

In summary, the test results indicate that MapReduce, with its parallelization capabilities, offers significant benefits in terms of execution time when dealing with larger datasets. While the average execution times obtained from the Hadoop implementation were slightly higher than those from the Python implementation, the scalability and stability provided by Hadoop make it a favorable choice for processing big data in a distributed environment.

A CODE

Here are excerpts of the code that make up this project. Please note that the complete code can be found in the GitHub repository, the link to which is provided in the introductory page of this documentation.

A.0.1 Point.java

As mentioned in Chapter 2, this class serves as a logical representation of a point within our dataset. It includes several functions that perform various calculations on a Point object and also enables the serialization and deserialization of data.

```
public class Point implements Writable{
    private ArrayList<Double> features = new ArrayList<>();
    private int dim;

    public Point();
    public Point(int size);
    public Point(ArrayList<Double> features);
    public ArrayList<Double> getFeatures();
    public int getDim();
    public void setFeatures(ArrayList<Double> features);
    public void sumPoint(Point p);
    private Double distance(Point p);
    public int nearestCentroid(ArrayList<Point> centroids);
    public void scale(int n);
    public static ArrayList<Point> getPoints(int k, int dim);
    public boolean equals(Point p);

    @Override
    public String toString();
    public void write(DataOutput dataOutput);
    public void readFields(DataInput dataInput);
}
```

A.0.2 ClusteringFeature.java

This class represents the logical pair of information related to a cluster: the partial sum of points in the cluster and the count of points. It also includes a few utility functions that facilitate the final calculation of the average.

```
public class ClusteringFeature implements Writable{
    private Point partialSum;
    private int numPoints;

    public ClusteringFeature();
    public ClusteringFeature(Point partialSum, int numPoints);
    public ClusteringFeature(int dim);
    public ClusteringFeature(ArrayList<ClusteringFeature> list);
    public Point getPartialSum();
    public void setPartialSum(Point partialSum);
    public int getNumPoints();
    public void setNumPoints(int numPoints);
    public Point computeMean();

    @Override
    public String toString();
    public void write(DataOutput out);
    public void readFields(DataInput in);
}
```