UNIVERSITÀ DI PISA

CLOUD COMPUTING

K-MEANS

GIOVANNI BARBIERI
CLAUDIO DAKA
VITTORIA ACAMPORA

ANNO ACCADEMICO 2022 – 2023

Indice

Introduction

Clustering is a process of partitioning a given set into disjoint clusters, so that elements belonging to the same cluster can be considered similar while elements belonging to different clusters can be considered different. The purpose of clustering is to determine the intrinsic grouping of the belonging unlabeled data to a set. The division of data into groups of similar elements allows to represent the data through few clusters obtaining simplifications a at the expense of precision on small details. Clustering is subject research in many areas such as statistics, pattern recognition and machine learning.

In this project we want to investigate on how the K-means can be implemented in a parallelized way using the map-reduce approach with Hadoop, a Java framework.

Capitolo 1: K-means

The K-means is a non-hierarchical and geometric exclusive clustering algorithm. The goal of the algorithm is to find the best one division of n entities into k groups, such that the distance between the members of the group and the corresponding centroid, representative of the group, is minimized. Formally, the K-means algorithm partitions n entities into k partitions Pi, i = 1, 2, . . . , k such that:

$$MSE = \sum_{j=1}^{k} \sum_{i=1}^{n} \|x_i^j - c_j\|^2$$

is minimal, $\|x_i^j - c_j\|$ where provides the distance between the i −th entity and its relative centroid. Equation is the objective function of the algorithm. The K-means is a deterministic algorithm which converges to a local minimum. To carry out the clustering, the algorithm must be provided with a set of initial centroids, which are normally chosen in a pseudo-random manner. Since neither the global minimum point nor the set of initial centroids with which the optimal clustering can be obtained is known, it is necessary to re-run the algorithm by varying the initial centroids provided to obtain different clusterings and be able to choose the best.

1.1 Parameters

To implement the algorithm of k-means we need parameters:

- *n*: number of data points
- *d*: dimension of data points
- *k*: the desired number of clusters (a set of k points called *means*)

1.2 Structure algorithm

1.2.1 Mapper

The mapper calculates the distance between the data point and each centroid. Then emits the index of the closest centroid and the data point.

```
class MAPPER
    method MAP(key, values)
        closest_distance = MAX_VALUE
        closest_centroid_index = 0
        for all point in list_of_values
            distance = distance(centroid, point)
            if (distance < closest_distance)
                closest_centroid_index = index_of(centroid)
                closest_distance = distance
        EMIT(closest_centroid_index, point)
```

1.2.2 Combiner

At each stage we need to sum the data points belonging to a cluster to calculate the centroid (arithmetic mean of points). Since the sum is an associative and commutative function, our algorithm can benefit from the use of a combiner computing partial sums to reduce the amount of data to be transmitted to the reducers.

```
class COMBINER
    method COMBINER(key, values)
        partialSum.number_of_points = 0
        partialSum = 0
        for all point in values:
            partialSum += point
            partialSum.number_of_points += 1
        EMIT(key,sum)
```

1.2.3 Reducer

The reducer calculates the new approximation of the centroid and emits it. The result of the MapReduce stage will be the same even if the combiner is not called by the Hadoop framework.

```
class REDUCER
    method REDUCE(key, values)
        newCentroid=newPoint
        for all point in list_of_values:
            newCentroid += point
        output = newCentroid / number_of_Points
        EMIT(key, output)
```

Capitolo 2: Implementation on Hadoop

The project is implemented installing Hadoop and HDFS in a fully distributed mode to simulate as much as possible the behavior of the framework in a small real cluster of machines. We have used three virtual machines.

2.1 Configuration

IP	NAMENODE	DATANODE	HOSTNAME
10.1.1.42	Yes	Yes	Hadoop-namenode
10.1.1.101	No	Yes	Hadoop-datanode-2
10.1.1.37	No	Yes	Hadoop-datanode-3

- The first VM is the namenode and is identified by the local IP address 10.1.1.42.
- The first VM is the datanode-2 and is identified by the local IP address 10.1.1.101.
- The first VM is the datanode-3 and is identified by the local IP address 10.1.1.37.

2.2 Job

2.2.1 Mapper.java

```java
public static class KMeansMapper extends Mapper<LongWritable, Text, IntWritable,
Point> {
        private Configuration conf;
        private List<Point> centroids;

        //Aggiorna la lista di centroidi per la nuova iterazione (dalla seconda
iterazione in poi)
        @Override
        protected void setup(Context context) throws IOException,
InterruptedException {
            this.conf = context.getConfiguration();
            this.centroids = new ArrayList<>();

            for(int i = 0; i < this.conf.getInt("NUM_CENTROIDS", 0); i++){
                Point centroid = Point.parse(this.conf.get("oldCentroids" +
i));

                this.centroids.add(centroid);
            }
        }

        //Permette di calcolare quale sia il centroide più vicino ad ogni punto
appartenente al dataset
        @Override
        protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

            //Deserializza i punti del dataset
            String[] righe = value.toString().split("\\n");
            List<Point> points = new ArrayList<>();
            for(int i = 0; i < righe.length; i++){
                Point point = Point.parse(righe[i]);
                points.add(point);
            }

            int closestCentroidIndex = -1;
            double closestDistance = Double.MAX_VALUE;

            //Per ogni punto calcola l'indice del centroide più vicino al punto
            for(Point point : points){
                closestDistance = Double.MAX_VALUE;
                for (int i = 0; i < this.centroids.size(); i++) {
                    double distance =
point.euclideanDistance(this.centroids.get(i));
                    if (distance < closestDistance) {
                        closestCentroidIndex = i;
                        closestDistance = distance;
                    }
                }
                IntWritable closestCentroidInd = new
IntWritable(closestCentroidIndex);
                Point output = new Point(point);

                //Emette l'indice calcolato e il rispettivo punto
                context.write(closestCentroidInd, output);
            }
        }
    }
```

2.2.2 Combiner.java

```
public static class KMeansCombiner extends Reducer<IntWritable, Point, IntWritable,
Point> {

        //Viene effettuata una parziale sommatoria dei punti vicini al solito
centroide. Permette un minore traffico di dati tra Mapper e Reducer
        protected void reduce(IntWritable key, Iterable<Point> values, Context
context) throws IOException, InterruptedException {

            // Aggrega i punti parziali per il centroide
            Point sum = Point.copy(values.iterator().next());
            while (values.iterator().hasNext()) {
                sum.add(values.iterator().next());
            }
            // Emetti il centroide aggregato
            context.write(key, sum);
        }
    }
```

2.2.3 Reducer.java

```
public static class KMeansReducer extends Reducer<IntWritable, Point, IntWritable,
Point> {

        //Effettua la sommatoria totale delle sommatorie parziali e successivamente
divide per il numero di punti.
        @Override
        protected void reduce(IntWritable key, Iterable<Point> values, Context
context)
                throws IOException, InterruptedException {

            Point newCentroid = Point.copy(values.iterator().next());
            while (values.iterator().hasNext()) {
                newCentroid.add(values.iterator().next());
            }

            Point output = newCentroid.divide(newCentroid.getAggregatedPoints());
            //Emette l'indice del centroide ed il nuovo centroide
            context.write(key, output);
        }
    }
```

Capitolo 3: Performance analysis

We generate our synthetic dataset with a python script; we have tried some experiments with datasets of 1k, 10k, 50k, 100k points. We have also decided to analyze the cases of 5 and 10 centroids.

The images below (*fig. 3.1* and *fig. 3.2*) show examples with 500 points because they are more clearly visible than the cases tested by us.

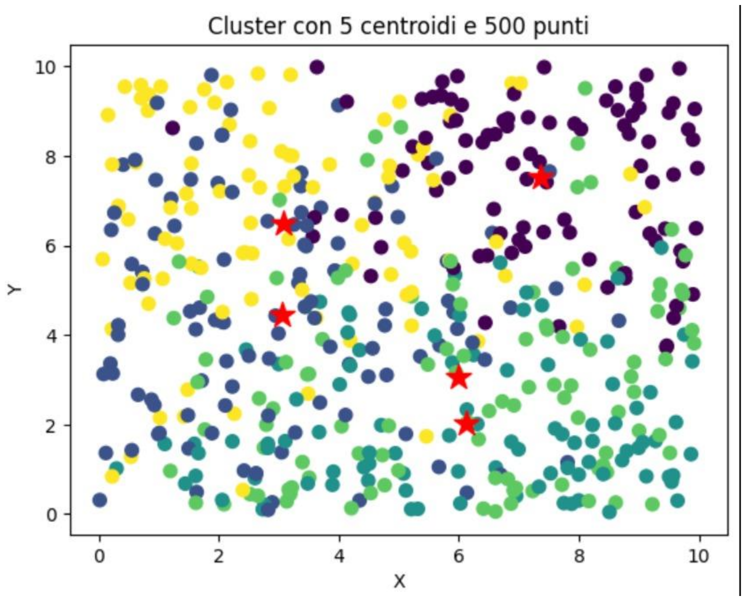

Fig 3.1

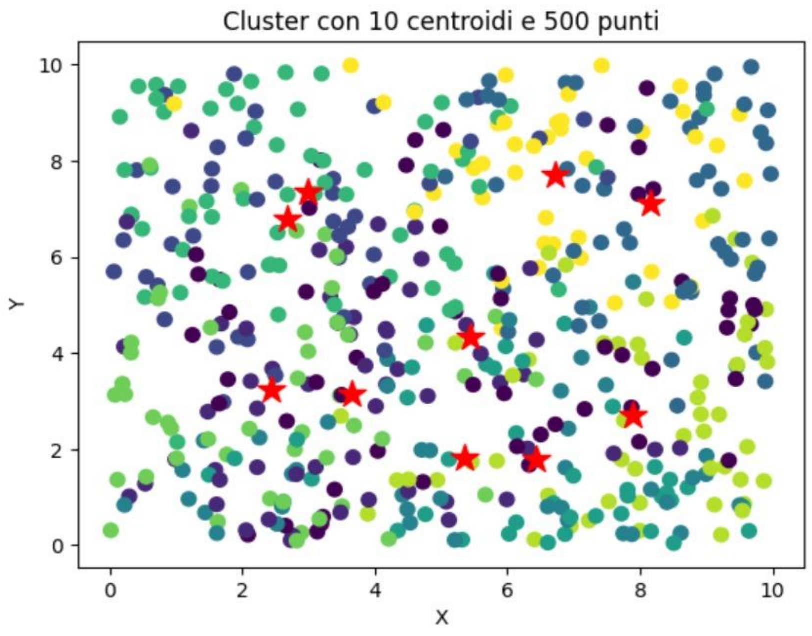

Fig. 3.2

3.1 Tests

For each dataset we have repeated the algorithm 5 times with a threshold of 0.1 and a max number of iterations of 15.

Considered that the k-means algorithm is sensitive to the initial centroids and that we used a random initialization, we will show the iteration average execution time (execution time/number of iterations).

3.1.1 Datasets with dimension = 5 and centroids = 5

Number of samples	1k	10k	50k	100k
Avg value (sec)	24,518	24,417	25,496	26,702

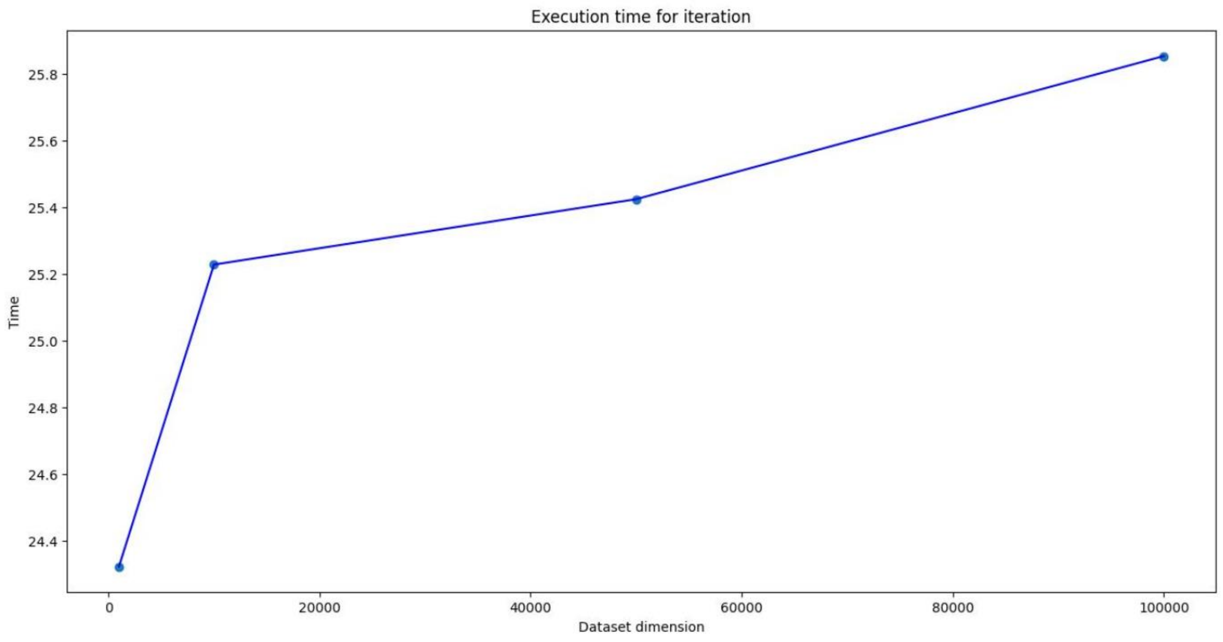

Fig. 3.3

In this test we have studied the variation of the execution time of the single iteration as the size of the dataset varies, testing it for the following sizes: 1k, 10k, 50k, 100k. The size of the space and the number of centroids were both fixed to the value of 5.

We can observe that the growth of the execution time is much faster going from 1k to 10k, while going to 50k and 100k the curve is less sloping.
The increase in execution time is coherent and it is a result that we expected, given that as the number of points in the dataset increases, we will have greater complexity.
Having a lower slope for large dataset values is a positive note that demonstrates how the software scales well, thus adapting perfectly to the context of use in big data.

3.1.2 Datasets with dimension = 5 and centroids = 10

Number of samples	1k	10k	50k	100k
Avg value (sec)	28,762	29,622	29,471	30,075

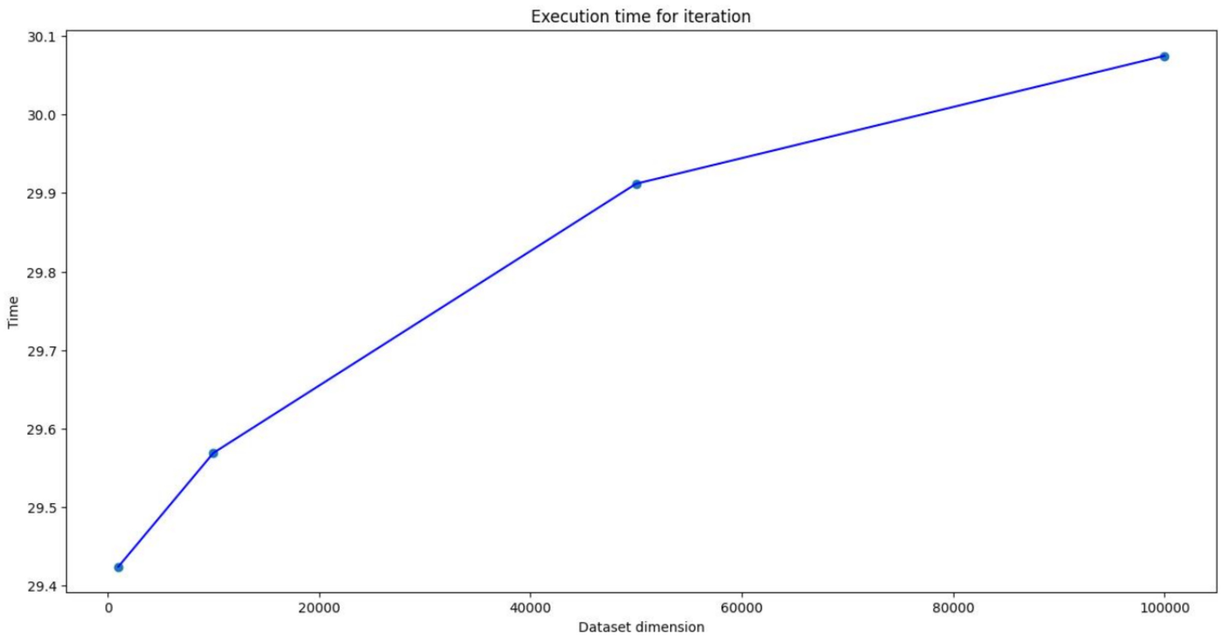

Fig. 3.4

We performed the same test as before but this time setting the number of centroids to 10. We have observed the same trends as in the previous test, thus confirming the results already obtained with a further try.
We noticed also that the execution times are greater than in the previous test.
In the following histogram this difference is most clearly illustrated (*fig. 3.5*).
Having set the number of reducers equal to the number of centroids, theoretically the execution time should not be affected by the increase of the number of centroids (except for the few points where we have for loops with a number of iterations equal to the number of centroids, which in this case it would be just 5 more iterations).
The increase we see is due to the fact that the parallelization of the work done by the reducers can never have the maximum theoretical speedup, i.e. equal to the number of centroids in this case, due to the overheads due to creation and any synchronizations.

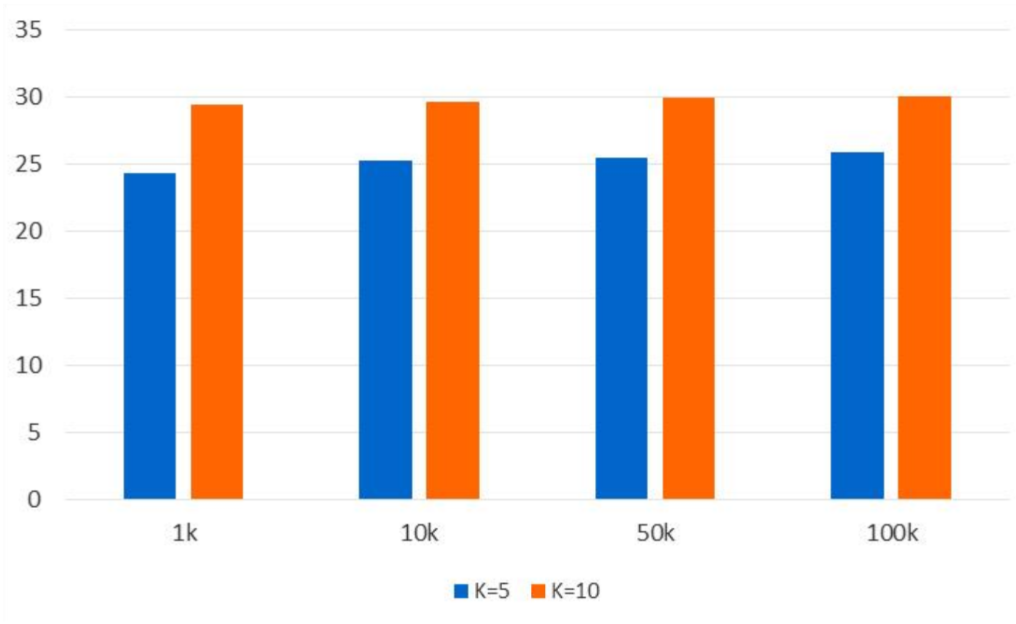

Fig. 3.5

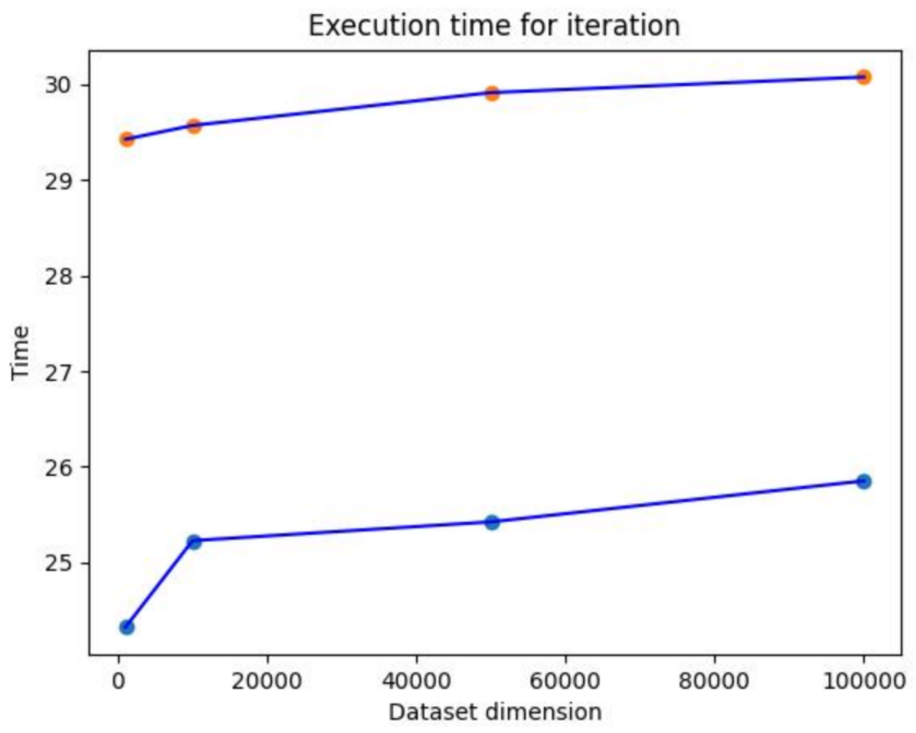

Fig. 3.6

The graph in *fig. 3.6* shows both tests we have ran so far.

3.1.3 Dataset with centroids=5 and number of samples=10k

Dimensions	2	5	8
Avg value (sec)	24,513	25,229	25,315

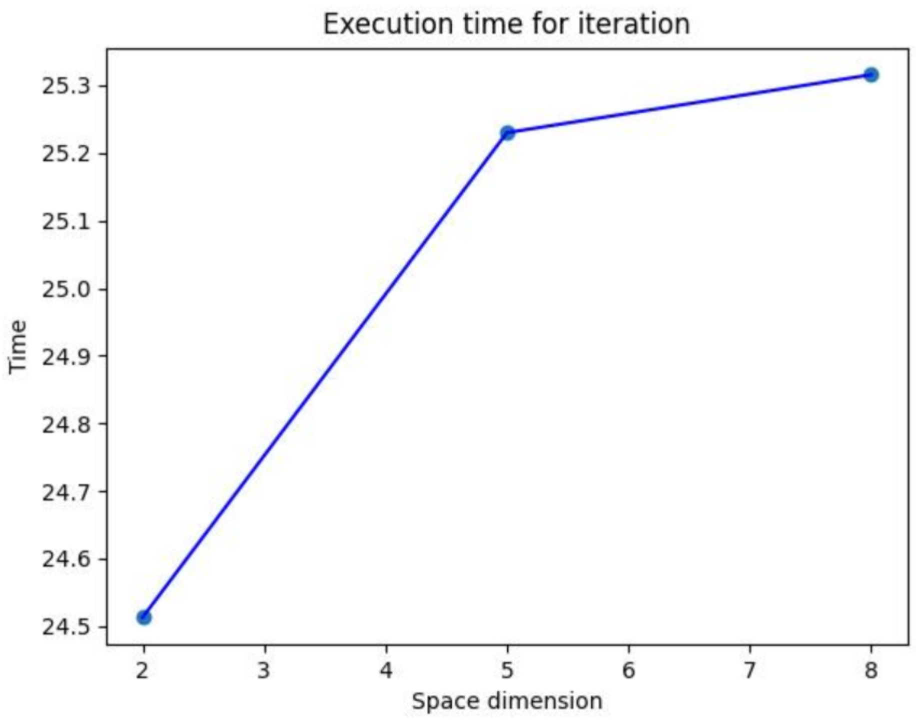

Fig. 3.7

We then proceeded to carry out tests by varying the size of the space, we studied the variation of single iteration execution time for the following space dimension values: 2, 5, 8.

The dataset size and the number of centroids were set respectively to 10k and 5.

We can see that the execution time increases, also in this case it is a consistent result: since the size of the space increases, all the parts of the code that access to the components of the point become more onerous.

3.1.4 Dataset with centroids=5 and number of samples=10k

Dimensions	2	5	8
Avg_iterations	7	12,6	13,8

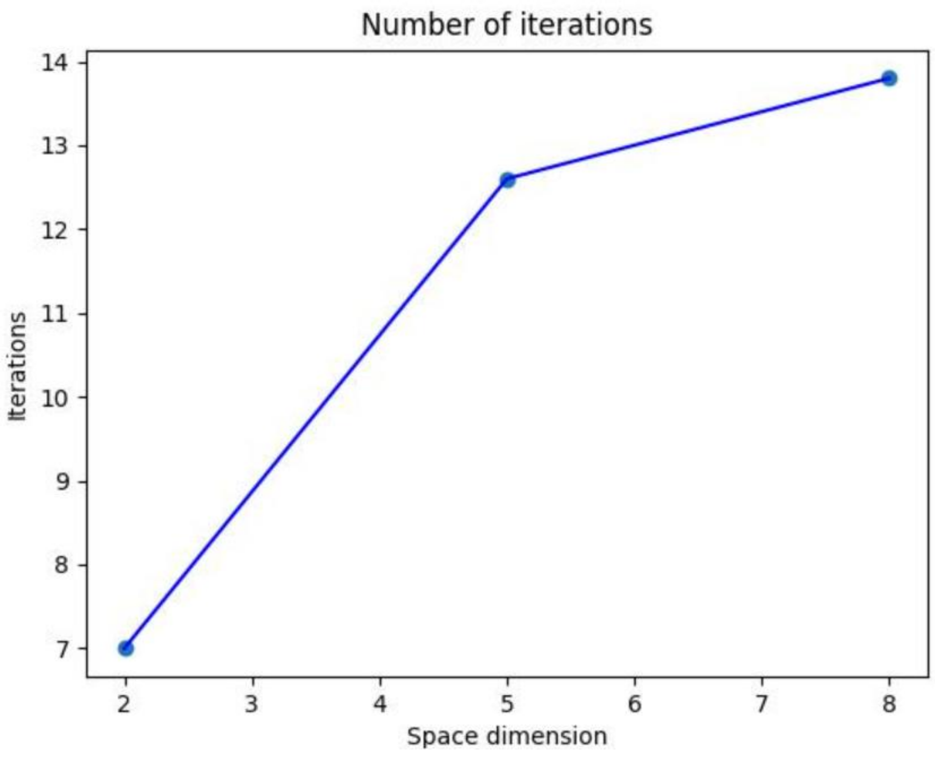

Fig. 3.8

We noticed a clear trend in this last test: the number of iterations needed to converge increases as the space size increases.

This trend can be explained by the fact that our stop condition is based on the Euclidean distance between the centroids of the current iteration and those of the previous one, this must be less than a certain threshold. Having a greater number of components, each component will contribute to the Euclidean distance increasing the probability that this exceeds the threshold.