PC-2019/20 k-means Clustering with Apache Hadoop MapReduce

Lorenzo Pratesi Mariti
Matricola - 7037171
`lorenzo.pratesi@stud.unifi.it`

Abstract

The enlarging volumes of information emerging by the progress of technology, makes clustering of very large scale of data a challenging task. This paper presents a parallel k-means clustering algorithms based on Apache Hadoop MapReduce model which is a simple and powerful parallel programming technique. The tests were performed on intel i7 hexa core in a pseudo-distributed mode and then in a fully distributed mode with Amazon EMR. The results show that the proposed algorithm can scale well and efficiently process large datasets on commodity hardware.

Future Distribution Permission

1. Introduction

Clustering, which is one of the most fundamental problems in data analysis and management, has been applied in many areas of computer science and related fields, such as data mining, pattern recognition and machine learning. *k-means* is undoubtedly one of the most popular clustering algorithms owing to its simplicity and efficiency and has received significant research efforts.

Map-Reduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Apache Hadoop provide MapReduce runtimes with fault tolerance and dynamic flexibility support.

1.1. Hadoop MapReduce

Hadoop is a framework that allows for the distributed processing of large data sets across clusters of commodity computers using a simple programming models called MapReduce. A MapReduce program is designed by defining the Mapper function and the Reducer function. A Combiner (also called a Mini Reducer) function can be used to reduce the volume of data going from the Mapper to the Reducer and by the way reduce the communication cost on the network. Hadoop will partition the dataset and each partition is going to be processed by an instance of the mapper function. The output of each mapper is going to be processed by the combiner. The reducer will receive the output of all combiner to do the last computation. The following is the flow of a MapReduce program that counts the number of occurrences of each word in the input data.

1.2. k-means

k-means is an unsupervised algorithm for non-hierarchical clustering. It makes it possible to group a dataset in k distinct clusters such that data in the same cluster are most similar to each other while data in different clusters are very distinct. In the basic form of k-means k random points are selected as the centers of the cluster; then the algorithm iteratively assign each point to its closest centroid (a cluster's center) and then each centroid is recomputed as the mean of all data assign to it. The process continues until the centroids do not change anymore.

Algorithm 1: Basic k-means

```
  Result: $k$ centroids
1 Randomly choose $k$ centroids;
2 repeat
3     From $k$ clusters by assigning all points
        to the closest centroid;
4     Recompute the centroid of each cluster;
5 until The centroids don't change;
```

Algorithm 2: Basic k-means

```
  Result: $k$ centroids
1 Randomly choose $k$ centroids;
2 repeat
3     Map
4         Input is a data point and $k$ centers are
            broadcasted;
5         Finds the closest center among $k$ centers for
            the input point;
6     Reduce
7         Input is one of $k$ centers and all data points
            having this center as their closest center;
8         Calculates the new center using data points;
9 until The centroids don't change;
```

In k-means algorithm, the most intensive computation that occurs is the calculation of similarity between data and centroids. In each iteration, it would require a total of $(n \times k)$ similarity computations where n is the number of objects and k is the number of clusters being created. Since the computation of the similarity between a data point and a centroid do not required data from any other computation, parallelization can be used to reduce the computation time.

2. k-means based on MapReduce

To implement the k-means algorithm with MapReduce we have to define our main functions:

- Map
- Combine
- Reduce

The job of the map function is to assign each point to its closest centroid.

Algorithm 3: Map function

```
   Input: Global centroids, <key, datapoint>
   Output: <key', datapoint>
1  Function map(key, value):
2      Construct the sample instance from value
3      minDis = Double.MAX VALUE
4      index = -1
5      for i = 0 to centers.length do
6          dis=ComputeDist(instance, centers[i])
7          if $dis < minDis$ then
8              minDis = dis
9              index = i
10         else
11     end
```

Note that Step 3 and Step 4 initialize the auxiliary variable minDis and index; Step 5 computes the closest center point from the sample, in which the function *ComputeDist(instance, centers[i])* returns the distance between instance and the center point centers[i]; Finally the map function outputs the intermediate data which is used in the subsequent procedures. The intermediate values are then composed of two parts: the index of the closest center point and the sample information.

After each map task, we apply a combiner to combine the intermediate data of the same map task. The combiner takes in input the $< key, D >$ pairs where where key is the cluster index and D the list of data assigned to that cluster and will output the pairs $< key, < SUM, N >>$, where SUM is the sum of data in D and N the length of D. To sum up the combiner function outputs the sum and the number of data points assigned to same cluster by the mapper.

The input of the reduce function is the data obtained from the combine function of each host. As described in the combine function, the data includes partial sum of the samples in the same cluster and the sample number. The role of the reducer is to compute the new centroid using the output from the combiner.

The reducer function takes in input a pairs of $< key, L >$, where key is the cluster index and L is the list of $< SUM, N >$ produced by the

Algorithm 4: Combine function

```
Input: <key, D>
Output: <key', <SUM, N>>
Function combine(key, D):
    sum := array to record the sum in D;
    num = 0;
    for val in D do
        sample := the sample instance for $val$
        for i=0 to D.dimension do
            sum[i] += sample[i];
        end
        num++;
    end
```

combiner and output a pairs $< key, centroid >$, where centroid is the new coordinate of the k-th centroid.

Algorithm 5: Reduce function

```
Input: $< key, L >$
Output: $< key, centroid >$
Function reduce(key, L):
    center := array to record the sum in L;
    NUM = 0;
    for val in L do
        sample := the sample instance for $val$
        for i=0 to L.dimension do
            center[i] += sample[i];
        end
        NUM += num;
    end
    for i=0 to L.dimension do
        center[i] /= NUM;
    end
```

2.1. Implementation with Apache Hadoop

The MapReduce solution for k-means clustering is an iterative solution, where each iteration is implemented as a MapReduce job. k-means needs an iterative version of MapReduce, which is not a standard formulation of the MapReduce paradigm. To implement an iterative MapReduce solution, we need a driver or control program on the client side to initialize the k centroid positions, call the iteration of MapReduce jobs, and determine whether the iteration should continue or end. The mapper needs to fetch the data point and all cluster centroids; the cluster centroids have to be shared among all mappers.

The developed program processes the k-means algorithm on a series of points that belong to an n-dimensional space. The input data are a set of text files which in each row have the id and the coordinates of a point. Eg:

$$\begin{gathered}\ldots\\ 14, -26.756167631, -7.4194447161\\ 15, -41.669074582, -19.845734930\\ \ldots\end{gathered}$$

The number of k clusters is specified by the user and is used to create a sequential file. The file contains the centers generated randomly and is initialized before the start of the MapReduce process.

The arguments to be indicated when starting the program are:

- The number of dimensions of each data point.
- The number of significant decimals to consider when comparing floats.
- The folder that contains the dataset files.
- The number of clusters.
- The folder where the output will be written.

2.1.1 Map

The `map()` function will do the following:

- Read the cluster centroids into memory from a text file. In Hadoop's implementation, this will be done in the setup() method of the mapper class.
- Iterate over each cluster centroid for each input key-value pair. In Hadoop's implementation, for the `map()` function, the key is generated by Hadoop and ignored (not used).
- Compute the Euclidean distances and save the nearest center with the lowest distance to the input point (as a *d*-dimensional vector).

- Write the key-value pair to be consumed by reducers, where the key is the nearest cluster center to the input point (and the value is a *d*-dimensional vector).

```java
public void map(Object ikey, Text ivalue, Context context) throws IOException, InterruptedException {
    int first_comma_index = ivalue.toString().indexOf(",");
    double[] sample = getSample(ivalue.toString().substring(first_comma_index + 1));
    double min = Double.MAX_VALUE, dist;
    long index = -1;
    for (int i = 0; i < centers.size(); i++) {
        dist = distance(sample, centers.get(i));
        if (dist < min) {
            min = dist;
            index = i;
        }
    }

    if (index != -1) {
        context.write(new LongWritable(index), new Text(ivalue.toString()));
    }
}
```

Listing 1. Map function.

2.1.2 Combine

After each map task, a combiner is applied to mix the map task's intermediate data. The combiner sums up the values, the sum is needed to compute the mean values for each dimension of vector objects. In the `combine()` function, we partially sum the values of the points (as vectors) assigned to the same cluster. The `combine()` function improves the efficiency of our algorithm because it avoids network traffic by transferring less data between slave (i.e., worker) nodes.

```java
public void reduce(LongWritable _key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    double[] sum;
    double[] sample;
    int num = 0;
    sum = IntStream.range(0, nb_dimensions)
        .mapToDouble(i -> 0.0).toArray();
    for (Text val : values) {
        int first_comma_index = val.toString().indexOf(",");
        sample = getSample(val.toString().substring(first_comma_index + 1));
        for (int i=0; i < nb_dimensions; i++) {
            sum[i] += sample[i];
        }
        num++;
    }
    String sb = IntStream.range(0, nb_dimensions).mapToObj(i -> sum[i] + ",").collect(Collectors.joining("", "", String.valueOf(num)));
    context.write(_key, new Text(sb));
}

public double[] getSample(String sampleText) {
    return Arrays.stream(sampleText.split(","))
            .mapToDouble(Double::parseDouble)
            .toArray();
}
```

Listing 2. Map function.

2.1.3 Reduce

The reducer does recentering: it recreates the centroids for all clusters by recalculating the means for all clusters. In the `reduce()` phase, the outputs of the `map()` function are grouped by `Cluster-ID`, and for each `Cluster-ID` the centroid of the points associated with that `Cluster-ID` is calculated. Each `reduce()` function generates `(Cluster-ID, Centroid)` pairs, which represent the newly calculated cluster centers. The `reduce()` function can be summarized as follows:

- The main task of the `reduce()` function is to recenter.
- Each reducer iterates over each value vector and calculates the mean vector. Once you have found the mean, this is the new center; your final step is to save it into a persistent store.

```java
public void reduce(LongWritable _key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    double[] center;
    double[] sample;
    int num = 0, last_comma_index;
    center = IntStream.range(0, nb_dimensions)
        .mapToDouble(i -> 0.0).toArray();

    for (Text val : values) {
        last_comma_index = val.toString().lastIndexOf(",");
        if (last_comma_index < 1) {
            continue;
        }
        sample = getSample(val.toString().substring(0, last_comma_index));
        for (int i = 0; i < nb_dimensions; i++) {
            center[i] += sample[i];
        }
        num += Integer.parseInt(val.toString().substring(last_comma_index + 1));
    }
    for (int i = 0; i < nb_dimensions; i++) {
            center[i] /= num;
    }
    StringBuilder center_sb = new StringBuilder();
    for (int i = 0; i++) {
        center_sb.append(dFormater.format(center[i]));
        if (i < nb_dimensions - 1) {
            center_sb.append(" ");
        }
    }
    centers.set((int) _key.get(), center);
    context.write(_key, new Text(center_sb.toString()));
}
```

Listing 3. Map function.

2.1.4 Driver

The driver is important to control the program on the client side to initialize the K centroid positions, call the iteration of MapReduce jobs, and determine whether the iteration should continue or end.

The main class that contains the configuration and manages the jobs that execute the process MapReduce is PointsKMeans. The main functions for the job are `initJob` and `run`. The first function initializes all the variables necessary for the execution of the jobs using the input arguments. The second function is the actual MapReduce process. It initializes all the classes used and the input and output elements. After each `waitForCompletion` we check the convergence criteria using the `isDifferent` function that checks if the distance between the old center values and the new one does not exceed the threshold limit.

```
public int run() throws Exception {
    do {
        initJob();
        deleteOutputDirIfExists();
        centers.clear();
        centers.addAll(KMeansReducer.centers);
        KMeansMapper.centers.clear();
        KMeansMapper.centers.addAll(centers);

        if (!job.waitForCompletion(true)) {
            logger.info("job failed");
        }

        nb_iter++;
    } while (isDifferent(centers, KMeansReducer.centers)
    );

    logger.info("Finished with " + nb_iter + "
    iterations");
    return nb_iter;
}
```

Listing 4. Main class - run().

Running the program will create in the output folder a file containing the coordinated of each centroid.

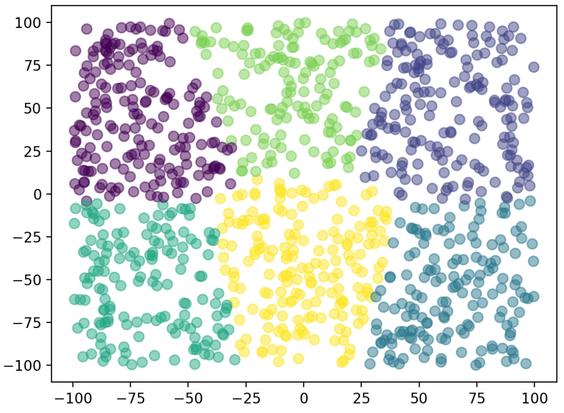

Figure 1. Clustering 1000 2D point with K = 6

3. Experimental Results

In this section, we evaluate the performance of the algorithm with respect to speedup. Performance experiments were run in a pseudo-distributed mode and in a fully distributed mode thanks to the service Amazon Elastic Map Reduce (ERM) with Hadoop verison 2.8 and java 1.8.

The dataset is composed of a set of csv files in which each line is a data point. Each data point is represented by its identifier followed by its coordinates. To measure the speedup, we kept the dataset constant and increase the number of node in the ERM cluster. The perfect parallel algorithm demonstrates linear speedup: a system with m times the number of cluster yields a speedup of m.

Figure 1 shows the result of the process on 1000 points and 6 clusters. In Figure 2 the number of points is increased to 10000 and the clusters to 30. In Figure 3 the example of a result is shown on 1000 three-dimensional points with 10 clusters. In Figure 4 the example of a result is shown on 10000 three-dimensional points with 30 clusters.

As you can see in Figure 5, the speed increases according to the number of nodes, but also according to the data load.

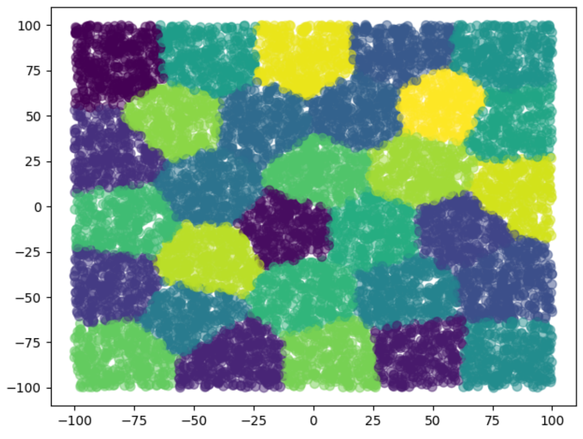
Figure 2. Clustering 10000 2D point with K = 30

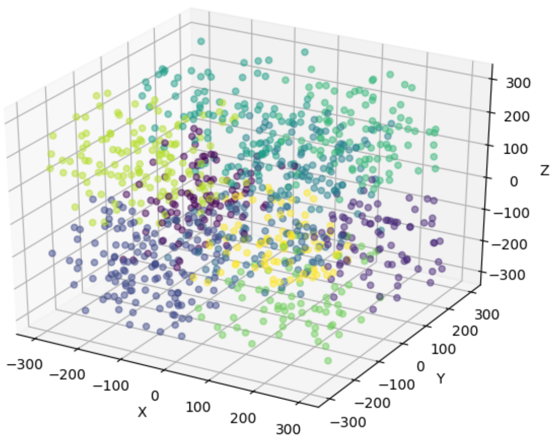
Figure 3. Clustering 1000 3D point with K = 10

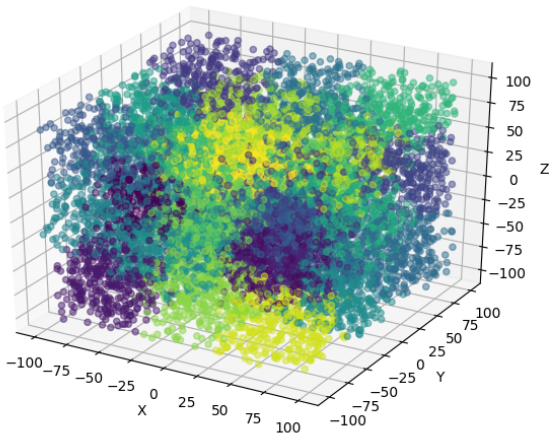
Figure 4. Clustering 10000 3D point with K = 30

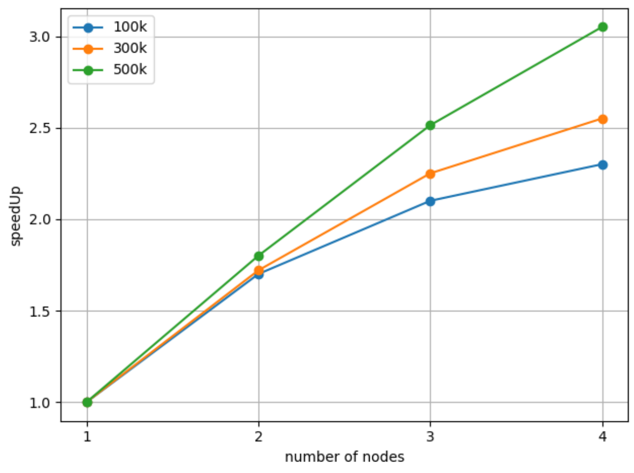

Figure 5. Speedup with respect to number of nodes

References

[1] Mahmoud Parsian - *Data Algorithms - Recipes for scaling up with hadoop*.

[2] Weizhong Zhao1, Huifang Ma and Qing He *Parallel k-means Clustering Based onMapReduce*.

[3] Rajashree Shettar, Bhimasen, V. Purohit *A review on clustering algorithms applicable for map reduce*. Proceedings of the international conference computational systems for health sustainability (April, 2015).

[4] S. Bawane Vinod, M. Kale Sandesha *Clustering algorithms in MapReduce:a review* . Int J Comput Appl (2015)