UNIVERSITÀ DI PISA

MapReduce K-means Algorithm

Cloud Computing: Big Data Processing

Alessandro Noferi, Francesco Fornaini, Francesco Mione, Leonardo Lossi

2 luglio 2020

Indice

1 Introduction

The aim of this project is to design and develop two MapReduce implementations of the *K-Means Clustering Algorithm* using Hadoop and Spark Frameworks.
This section is a small introduction in which we show how we have produced the synthetic datasets of points we have used during the development and the testing of our two implementations.
In Section 2 and 3 we will describe respectively the Hadoop and the Spark Implementations with our design choices.
Finally in 4 we will compare the performances of our two implementations from the execution time point of view.

1.1 Dataset Generation

To create the data sets to test both the implementations, we have written a small script, called `generate_points.py` that uses the Python library `sklearn.datasets`. It allowed us to generate *ad-hoc* clustered points with a custom standard deviation (through the function `make_blobs`). In particular, we selected the number of points, the dimension and the number of clusters, and the script creates three text files, a first one with all the points and a second one with the initial means. The latter has been calculated in this step in order to make fair comparisons between the two implementations setting the same starting points.
In the third file, instead, are saved the optimal centroids calculated by the `sklearn.cluster` *Python* library, that provides an implementation of the K-means algorithm. This file is useful because it allowed us to compare these results with those found by our implementation.

Since the goal of this project was not to develop an optimum algorithm in terms of accuracy, we decided two select the initial mean points randomly from the total number of data points.

1.2 I/O organization

Both the implementations used the HDFS to load the input files with the points. The folders `inputs` and `initial_means` contain the input points and the initial means, respectively.

1.2.1 Stop Criteria

In our two implementation we have implemented two of the 3 most common stopping criteria of the *K-means Clustering algorithm*:

- **maximum number of iteration**: this first stopping criterion exploits a command line parameter to establish thee maximum number of iterations after which the execution is stopped;
- **minimum centroid movement**: this second stopping criterion exploits a command line parameter used as threshold to see if all the centroids no longer move beyond this parameter. In each iteration we verify that the Euclidean Distance between the previous iteration centroid and the currently computed one is below the threshold for all the centroids; in positive case the algorithm is terminated, otherwise it continues its execution.

2 Hadoop Implementation

2.1 Pseudocode

In order to develop an Hadoop implementation using the MaprReduce runtime support we had to think about the K-Means algorithm in MapReduce terms, that is in `map` and `reduce` (*pure*) `functions` terms. The following one is the pseudocode of our implementation, we need firstly to introduce some notations:

- M is the mean-vector, that is the vector containing the initial means μ_i;
- *cid* is the *cluster-id* produced by the python script `generate_points.py`;

2.1.1 Map function

The `map` function computes for each point the nearest mean: the prototype of this function should take as input a (key, value) pair where the value is given by a point (the key is ignored) and should produce as output a (*cid*, point) pair.

Algorithm 1 : **Map function**

1: $D = []$
2: **for** $\mu_i \in M$ **do**
3: D.append(euclideanDistance(*value*, μ_i))
4: *cid* = indexOfMin(D)
5: **Emit**(*cid*, *value*)

2.1.2 Combine function

The `combine` function is an optimization to reduce the quantity of data exchanged by all the mappers with all the reducers. In order to do it, this function takes as input a (*cid*, *list* of Point) pair produced locally by a Mapper and it produces as ouput a couple in which the key is given by the *cid* and the value is given by a tuple, composed itself by two fields *sum* and *count*:

- *sum* is a vector whose elements are given by the element-wise sum of all the points associated with that *cid*;
- *count* is the number of points associated with the cid that have contributed to produce the sum vector.

Algorithm 2 : **Combine function**

1: **Input:** *cid, list[value]*
2: $sum \leftarrow 0$
3: $count \leftarrow 0$
4: **for each** l_i **in** *list[value]* **do**
5: $sum \leftarrow sum + l_i$
6: $count \leftarrow count + 1$
7: **Emit** (*cid*, (*sum, count*))

2.1.3 Reduce function

The `reduce` function should produce the new set of centroids, starting from the pairs composed by the *cid* and its associated list of tuples (*sum, count*). The prototype of this function should take as input a *(cid, list(sum,count))* pair and should produce a (*cid*, newCentroid) pair.

Algorithm 3 : **Reduce function**

1: **Input:** *cid, list(sum, count)*
2: $totalSum \leftarrow 0$
3: $totalCount \leftarrow 0$
4: **for each** sum_i, count_i **in** *list[(sum, count)]* **do**
5: $\quad totalSum \leftarrow totalSum + sum_i$
6: $\quad totalCount \leftarrow totalCount + count_i$
7: $newCentroidValue = totalSum \ / \ totalCount$
8: **Emit** (*cid, newCentroidValue*)

2.2 Design Choices

The application master of this implementation is given by the class `Kmean`. The control-logic of the algorithm is implemented in the `main` function.
Firstly the `main` function performs the **input parsing**: the provided command line parameters are used to configure some `Configuration` parameters (see subsection 2.2.1).
Then there is the *core* of the algorithm; the function basically iterates in a `while` cycle based on the check of the convergence conditions: in each iteration a new job is created and configured, and after the `waitForCompletion` statement it performs the check to see if the algorithm has converged.
The most important thing to notice is the possibility to set more than one reducer; this is done by the `job.setNumReduceTasks(int reducerNumber)` method. Having a multiple number of reducer means having multiple output file with name-pattern "`part-r-*`". Since we assumed that the algorithm reads the centroids from file, in the `main` function we implemented the `computeNewCentroids` function: this function simply reads from all the files produced by all the reducers and writes all the data in the file used by the mapper to load the means.

2.2.1 Command Line Parameters

The parameter passed throught the command line are:

- `input`: path of the input file;
- `output`: path of the output file;
- `k`: this argument is pretty important because have a double function. In case it is given as a numeric parameter, it will be the number of clusters in which the data points will be grouped. The initial centroids in this case are randomly chosen among all the input points. In case the value is given as a string, this parameter is interpreted by the program as the centroid file path. This processing has been fundamental in our study in the comparative analysis between the Hadoop and the Spark implementations. In order to perform a fair comparison the two implementations should process the same dataset starting from the same set of centroids: the initial set of centroids is critical in reaching the optimal solution (it often conduces to suboptimal solutions).
- `maximum number of iterations`: this command line parameter is the one which allows us to set the maximum number of iterations that the algorithm should perform according the first stopping criterion (see subsection 1.2.1);
- `threshold`: this command line parameter is the one which allows us to set the threshold for the movement of the centroids in consecutive iterations for the second stopping criterion (see subsection 1.2.1).
  In order to implement this second stopping criterion we have used the Hadoop Counter `NUMBER_OF_UNCONVERGED`. This counter assumes as value, iteration after iteration, the number of centroids that have not moved more than the threshold.
- `reducers`: this command line parameter allows to select the number of reducers to use.

2.2.2 How to launch the program

Once the input file and the eventual centroid file have been uploaded on the hdfs, the command to launch the execution of the program is:

```
hadoop jar Kmeans-1.0-SNAPSHOT.jar it.unipi.hadoop.Kmeans.Kmean

<input_file> <output_dir> <k> <max_iter> <threshold> <reducers>
```

2.3 Components

2.3.1 Custom Classes

2.3.1.1 Point This class implements a data point. Since it will be sent through the network, the class must implement the `writable` interface. To store the point coordinates has been used the `ArrayPrimitiveWritable` class, a wrap *Writable* implementation around an array of primitives (`double` type in this case). The class has all the methods that permit to sum and calculate the Euclidian distance between points.

2.3.1.2 Mean This class implements a mean point. It basically extends the Point class with the addition of a text *label ID* that identifies the cluster. It also implements the `WritableComparable` interface because, as we will see, the mean will be the *key* of the (Key, Value) pairs exchanged between the components and so it must have a compare method used by the *shuffle and sort* for the sorting. This method is implemented by comparing the *label ID* between two centroids.

2.3.2 Mapper

The Mapper class role has been described in subsection 2.1.1. In order to perform the related actions the Mapper has to read the centroids file in the `setup` method. We decided to read line by line the textual file using a `BufferedReader` and its method `readLine()`.
The read centroids are stored in the `ArrayList<means>` data structure: we decided to proceed in this way assuming that in most of the cases clustering tasks have to group data in a quite small number of clusters. So this data structure will never use a large quantity of memory. One important thing to notice is that the Mapper has two member field marked with `final`, this is because in this way the map task will reuse the same portion of memory in order to store the `Writable` objects.

The `map` function takes as inputs a generic `Object` as key (we implemented it this way because we have completely ignored the key in the map phase) and a `Text` as value, since we set as `TextInputFormatClass` as `InputFormatClass`. The `Text` value represents a line read from the input file, so it has to be parsed in an array of double in order to build a Point.
Once the point has been built we have computed the index of the nearest mean using the `ArrayList<means>` and we produced as output a (Key, Value) pair consisting in nearest mean and the Point.

2.3.3 Combiner

The aim of the Combiner class has been described in subsection 2.1.2. The only thing to notice is that in order to perform its task we needed to perform a deep copy of the first point of the `Iterable<Point>` taken as input.

2.3.4 Reducer

The aim of the Reducer class has been describer in subsection 2.1.3. In order to perform its task in the `setup` method it reads the Configuration parameter `threshold`, this will be used in the reduce function in order to verify the convergence according the second stopping criterion (see subsection 1.2.1). The considerations about the member data structures are similar to the ones of the Mapper class.

3 Spark Implementation

3.1 Steps

1. Load input points text file into a Spark RDD;
2. Tansform the lines RDD into a `Point` RDD;
3. Extract initial centroids from the `Point` RDD
4. While not converged:
   4.1. Broadcast the centroids to all the nodes;
   4.2. Assign each `Point` to the closest centroid;
   4.3. Calculate the new centroids;
   4.4. Verify convergence;
5. Save the centroids as text file.

3.2 Design Choices

The first task that the `main` function has to perform is to initialize the `SparkContext` so that the application can run on the cluster using the value `"yarn"` for the master parameter.

Once the context has been initialized we have built the Point RDD. In order to do it we have created a lines RDD starting from the input file and we applied the `map` trasnformation on it taking as input the function `create_point`. Once created the Point RDD we decided to cache it because it will be reused many times until the algorithm will not converged according one of the stopping criteria.
The next task was to build the `numpy` array for the centroids according to the value of the `k` parameter. The core of the algorithm is a while cycle which iterates based on the convergence condition. At each iteration the algorithm performs basically two steps:

1. first of all it associates to each point the nearest mean. This is done applying firstly a `map` tranformation to all the points giving as input the `assign_to_closest_mean` function; the produced RDD is then given as input to a second transformation `groupByKey`, which is a wide transformation because it demands data-shuffle to generate an RDD with cluster id and the list of points associated to it. So far there is only lazy evaluations;
2. the second step applies firstly a `map` transformation to the RDD produced by the previous step in order to generate the new centroids; they are then sorted by key with the `sortByKey` transformation and finally they are collected in the Spark driver using the `collect()` action, the latter triggers the entire computation.

Once the algorithm has converged the centroids are written in a file.

3.2.1 Command Line Parameters

The description of the parameters passed from command line interface is exactly the same of the one provided for the Hadoop implementation in subsection 2.2.1, except for the number of reducers.

3.2.2 How to launch the program

Once the input file has been uploaded on the hdfs with HDFS commands, the command to launch the execution of the program is:

```
spark-submit kmeans.py -k <clusters or file> -i <input_file> -o <output_file>
-m <max_iter> -t <threshold>
```

4 Experimental Results

In this part we will discuss the test and validation phase of our two implementations and we will show a fair comparison between the Hadoop and the Spark implementations.
"Fair comparison" means that the executions of the algorithm have been fed with the same input file, the same initial vector of centroids, the same threshold for the second stopping criterion and the same maximum number of iterations.

4.1 Testing on datasets

We performed a validation phase comparing the centroids configurations computed by our two implementations. The produced centroids were always the same and they have been produced in the same number of iterations.
So we proceeded in comparing them with the well-known results produced by the `generate_point` script. The Figure 1 shows the result of the K-means algorithm we implemented in a simplified case where the clusters are quite separated:

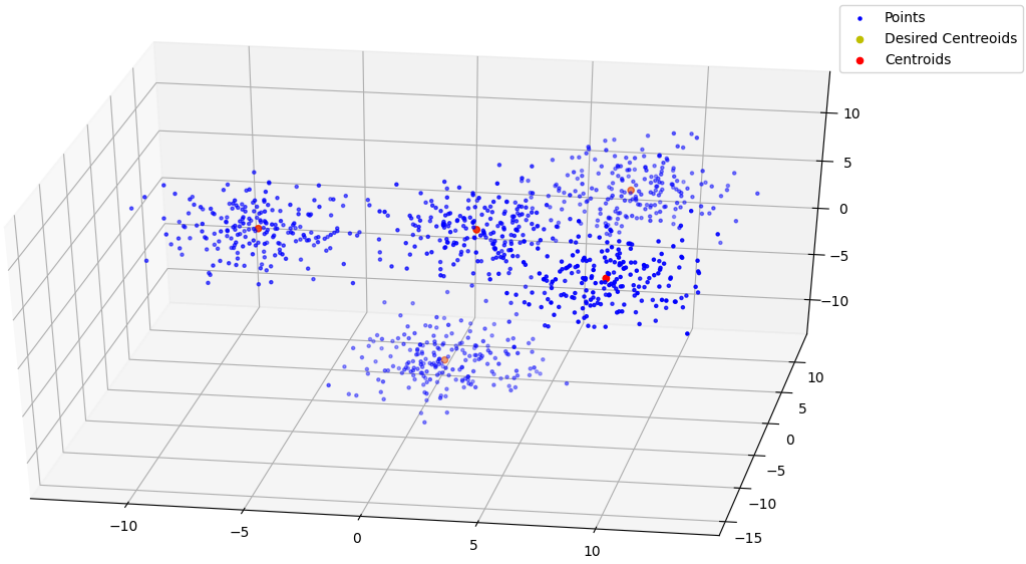

Figura 1: K-means algorithm with 1000 points and K = 5

As we can see the calculated centroids (the red dots) overlap the desired ones (the yellow ones) calculated by the script.
Anyway, the results were not exactly the same, the reason of the difference is simply the randomicity in the choice of the initial centroids set which lead us to a suboptimal solution of the clustering problem. In any case we can consider our results acceptable.

4.2 Execution time comparison between the two implementations

Finally, we made a comparison, in terms on time elapsed to run the algorithm, between both the developed implementations. We have tested them combining the following parameters:

Parameter	**Set of values**
Number of points	{1000, 10000}
Number of clusters	{5, 10, 15, 20, 25}
Number of Point dimensions	3
Threshold	0.01
Maximum number of iterations	100

Tabella 1: Configuration Parameters for testing executions.

Figures 2 and 3 show the execution time spent by Hadoop and Spark implementations in order to successfully complete the algorithm

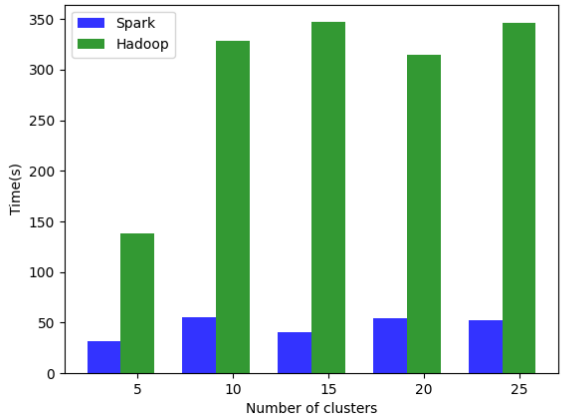

Figura 2: 1000 points

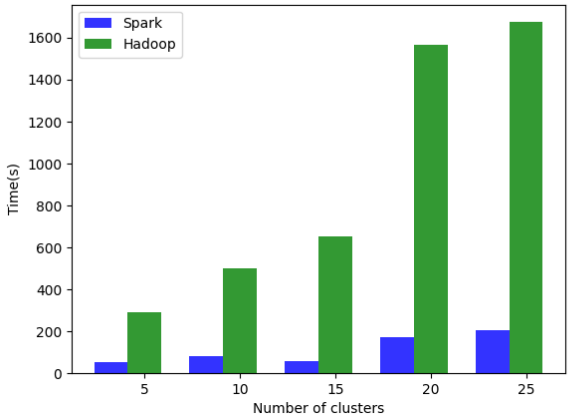

Figura 3: 10000 points

As we can see, the Spark implementation results always faster with respect to the Hadoop one. This is due to the fact that Spark exploits the RAM memory in order to store the intermediate RDDs, while the Hadoop implementation always stores the intermediate output data into the disk.
As an additional confirmation Table 2 the execution times ratio between the two implementations related to the number of iterations that the algorithm has exploited in order to converge.

# Clusters	# Points	Iterations to converge	Execution Times Ratio
5	1000	7	4.3
10	1000	36	9.1
15	1000	17	6.7
20	1000	16	5.8
25	1000	18	6.6
5	10000	14	5.5
10	10000	25	6.2
15	10000	33	11.3
20	10000	78	9
25	10000	84	8.2

Tabella 2: Relationship between number of iterations to converge and Execution Time ratio.