UNIVERSITÀ DI PISA

SHREK GROUP

Cloud Computing Project

K-Means on MapReduce

Members:

Matteo Pasqualetti

Daniele Giaquinta

Francesco Londretti

Contents

1 Pseudo-Code of the MapReduce Algorithm

The implementation of a parallelized version of the K-Means algorithm doesn't leave much room to creativity; we decided to also adopt the optional phase of combining bringing us to a total of three main phases which will be now briefly described.

Let us suppose that our data are contained in a document (e.g. a CSV file) that gives us the ability to exactly identify each point of data with a unique key.

1.1 Map Phase

The map phase of the k-means algorithm is responsible for assigning each data point to the nearest cluster center; it computes the distance between the input data point and each centroid (all centroids are stored globally and available to every mapper) and it keeps track of the index of the cluster center with the minimum distance to the data point. The output of the map function is a key-value pair, where the key is the index of the nearest cluster center and the value is, again, the data point.

```
def map (key, value, cluster_centers):
  # key: document identifier of the point
  # value: features of the point
  # cluster_centers: global list of cluster centers

  nearest_center = null
  min_distance = infinity
  for i=0 to length(cluster_centers):
      distance = compute_distance(value, cluster_centers(i))
      if distance < min_distance:
          min_distance = distance
          nearest_center_index = i

  return (nearest_center_index, value)
```

1.2 Combine Phase and Reduce Phase

In this phase, the reduce function takes as input the key (cluster center index) and a list of values that are tuples containing a sum of points and the count of points assigned to that sum (in which all points belong to the same centroid). **It is important to note that the implementation of combiner and reducer are exactly the same** in our case, because we decided to implement the combiner traditionally, i.e. as a local copy of the reducer with the job to process the output data of a single mapper in order to reduce the computation effort performed by the reducer.

In the case of the combiner each value corresponds to the coordinates of a single point, therefore the sum is 1. The combiner sums the values of every point that belongs to a certain centroid and increments the sum of points at each addition. Finally, it divides the total sum of coordinates by the number of points that have been processed. Let's remember that every combiner only processes the points output by its mapper. In the pseudo-code for the combiner/reducer there is also a premultiplication which does not make any difference for the combiner but is essential for the reducer.

In the case of the reducer each value corresponds to a sum of coordinates of points belonging to a certain cluster, and the sum is the number of such points. There is one reducer per cluster. The sum function computes the following calculation:

$$(sum\,of\,coordinates\ \times\ n^\circ\,of\,points)_1 + ... + (sum\,of\,coordinates\ \times\ n^\circ\,of\,points)_n$$

The premultiplication in the reduce phase is needed to compute the multiplication between coordinates and number of points of the first input tuple that would otherwise not be computed by the sum auxiliary function (check implementation or pseudocode below).

Each sum of coordinates needs to be multiplied by its relative count of points because at the end of the combine phase such sum was divided by the same number; this operation is mandatory in order to keep the implementation of the combine phase and reduce phase the same.

After looping through all the values, the function computes the new centroid of the cluster by dividing the *sum_of_points* by the *count_of_points*. It returns a tuple containing the key (cluster center index) and the new centroid for that cluster.

```
def reduce (key, values):
  # key: cluster center index
  # values: list of (sum_of_points, count_of_points) from all
   the mappers

  # the count of points is initialized with the value of the
   sum of points of the first tuple
  count_of_points = value[1]
  # the sum of coordinates is initialized with the sum of
   coordinates of the first tuple
  # PREMULTIPLIED by the count of points in the first tuple
  # in the combine phase value[1] is always equal to 1
  sum_of_points = value[0] * value[1]

  while(value.hasNext)
      value = nextValue
      count_of_points += value[1]
      sum_of_points += value[0] * value[1]

  # the new centroid is obtained by computing the average
  new_cluster_center = sum_of_points / count_of_points
  return (key, new_cluster_center)
```

1.3 Generation of Synthetic Data

We decided to write a simple **Python program that generates syntethic data with clusterable distribution**; the code creates a dataset using the *make_ blobs* function from Scikit-learn **which allows us to specify number of samples, features, and clusters** (it retrieves these parameters as arguments from the command line using the sys.argv list), as well as the standard deviation of each cluster. It creates a pandas DataFrame to store the data, where each row corresponds to a data point with its features. The code then saves the data to a CSV file using certain rules for nomenclature specified by us.

2 Implementation

This chapter provides a detailed explanation of the implementation of the K-Means algorithm using MapReduce in Java.

2.1 Point Class

Before diving into the Mapper and Reducer, it's crucial to understand the '**Point**' class, which implements the '**Writable**' interface. This implementation enhances Hadoop's performance by enabling efficient serialization and deserialization.

2.1.1 Serialization - write()

The '**write()**' method in the '**Point**' class specifies how the object's fields are converted into a binary representation that can be stored or transmitted.

2.1.2 Deserialization - readFields()

The '**readFields()**' method in the '**Point**' class reconstructs the object from its serialized form. It reads the binary data and assigns the values to the appropriate fields of the object.

2.1.3 Comparison - comparator()

The '**comparator()**' method in the '**Point**' class is used to compare the previous centroids with the newer ones outputted from the reducers.

This method calculates the relative shift between two points in each dimension using the formula:

$$shift = \frac{|a-b|}{|a| + 10^{-9}}$$

Here, a represents the new centroid, and b represents the old centroid. To account for the case where $a = 0$, the absolute value of a is added to 10^{-9} in the denominator. After calculating the relative shift for each dimension, an average is computed and multiplied by 100 to obtain the percentage value. This value can be compared to the user-defined variable '**Epsilon**' to determine convergence.

2.2 Mapper

The Mapper's objective is to take each tuple in the form of <**Key, Value**> and output a new intermediate <**Key', Value'**> pair. In this implementation, the Mapper uses the Euclidean distance to assign each input point to the key of the closest cluster.

2.3 Reducer

In this implementation, there is a Reducer for every cluster, resulting in K reducers. Each Reducer receives all the points with the same key (clusterID) as input and calculates the new centroid for that cluster.

2.4 Main

Before discussing the main function, it's necessary to introduce the auxiliary functions that ensure its smooth operation.

2.4.1 verifyStop()

The '**verifyStop()**' method checks if it's time to stop the algorithm and return the final centroids. For each cluster, the method compares the average shift calculated using the '**comparator()**' method with a user-specified variable called '**EPS**'.

2.4.2 recoverResults()

The 'recoverResults()' method is used to fetch the new centroids outputted by the reducers. Since each reducer writes to a file, additional steps are necessary to collect and process the intermediate results.

2.4.3 main()

The '**main()**' method sets up the Hadoop system based on the configuration file and starts the jobs during various iterations until the stop conditions are met. The configuration file includes attributes such as dimensionality, the number of clusters (**K**), the threshold for the stop condition (**eps**), and the maximum number of iterations (**max_iter**).

During each iteration, a new output path is created to recover the intermediate centroids returned by the reducers. The current job is configured with parameters such as the appropriate JAR file, the classes for Map, Reduce, and Combine functions, the number of reducers (equal to the number of clusters), the input and output paths, and the input/output operation formats.

At the end of the process, the final centroids obtained from the last iteration of the algorithm will be written to the output file.

3 Results

Our impementation works correctly and in accordance with the results obtainable by performing the k-means algorithm by scikit-learn on Python. Of course in order to obtain the same results in a given dataset it is necessary to force both the contexts to start from the same centroids. For our testing we decided to generate random centroids in our implementation and to store them in a file so that we could use them as initialization in Python as well.

It is important to note that **the k-means function provided by scikit-learn adopts the so called k-means++ initialization method**, a technique used to select starting centroids that allow to (almost always) find the optimal final centroids leading to an optimal clustering; without this technique initial centroids are generated randomly in the space and this often leads to a sub-optimal solution.

Such technique is however unfeasable in a distribuited approach that exploits Hadoop mapReduce; in a real-world scenario the parallelized implementation of k-means is used when the dataset is too large to be stored or/and processed by just one machine and the k-means++ implementation requires to cycle through the dataset multiple times to select the initial centroids based on their mutual distances; a certain probability to be chosen as initial centroid is assigned to each point and recomputed at each newly chosen centroid. Such computation must be computed on a single machine and such machine must store the whole dataset, which is exactly what we cannot afford when we decide to use a parallelized approach.

That is why the typical solution adopted to avoid selecting a sub-optimal clustering while exploiting parallelized k-means does not rely on k-means++ but on multiple runs of the algorithm with a consequent selection of the best solution using some kind of metric (e.g. the silhouette technique).

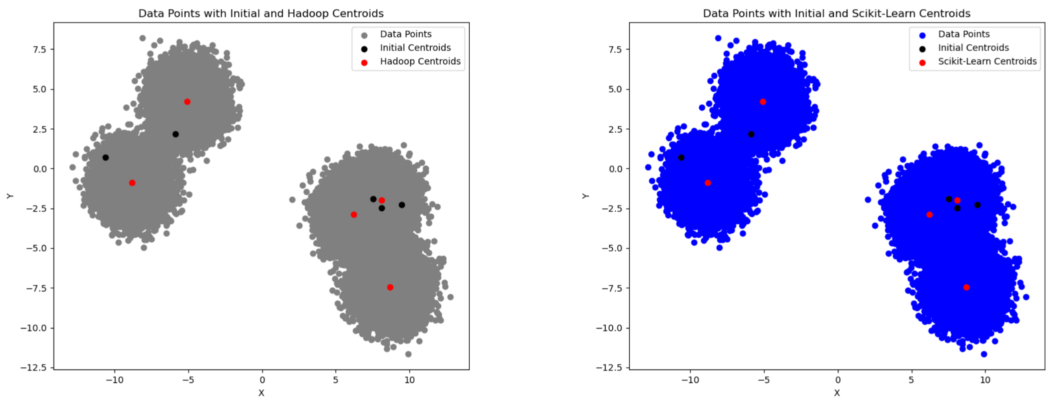

Figure 1: Hadoop and Python results ($n = 100,000$, $k = 5$, $d = 2$, $minEps = 1$)

3.1 Benchmarks

In order to compare and benchmark our implementation of the k-means algorithm using Hadoop MapReduce with the scikit-learn implementation, **we decided to systematically vary one factor at a time. The factors being considered are:**

- The Number of Points in the Dataset
- The Number of Clusters in the Dataset
- The Number of Features in the Dataset
- The Number of Iterations of the Algorithm

The main focus of our testing is to check the influence these variations have on execution time. **It is important to note that execution times are always longer on Hadoop;** with these benchmarks we are only trying to expose how well the two implementations scale and that there will surely be a breaking point at which execution times in Hadoop become lower. It must also be considered that we are just using three datanodes and with very limited resources, in a real-world scenario with more nodes and bigger datasets the Hadoop implementation advantages would become even more clear.

3.1.1 Impact of the Dataset Size

The impact of dataset size on algorithm execution time was tested with a fixed maximum number of iterations (6), features (2), and clusters (5). **Execution times were measured for two algorithms using three datasets: 1000 points, 100000 points, and 10000000 points.** The results are as follows:

	Hadoop	**Python**
1000 points	135.786 s	0.026 s
100,000 points	140.307 s	0.072 s
10,000,000 points	196.399 s	3.608 s

Table 1: Execution Times (6 iterations, 2 features and 5 clusters)

- Hadoop execution time increase: 44%
- Python execution time increase: 13777%

3.1.2 Impact of the Number of Clusters

To assess the influence of the number of clusters on execution times, a similar setup was employed. The experiments utilized 100,000 points, 2 features, and 6 iterations. **We tested different numbers of cluster/centroids (5, 9, 14, and 19):**

Table 2: Execution Times (6 iterations, 2 features and 100,000 points)

	Hadoop	**Python**
5 clusters	143.734 s	0.0579836368560791 s
9 clusters	160.972 s	0.11588025093078613 s
14 clusters	183.02 s	0.13141202926635742 s
19 clusters	209.135 s	0.16048359870910645 s

- Hadoop execution time increase: 45.5%
- Python execution time increase: 176.8%

3.1.3 Impact of the Number of Features

For this test we kept the number of points constant and equal to 10,000,000, the number of iterations to 6 as well as the number of clusters. **The test has been conducted with 2, 7 and 14 features.**

Table 3: Execution Times (6 iterations, 5 clusters and 10,000,000 points)

	Hadoop	**Python**
2 features	196.399 s	3.608 s
7 features	231.315 s	5.640 s
15 features	397.926 s	10.283 s

- Hadoop execution time increase: 102.6%
- Python execution time increase: 186.0%

3.1.4 Impact of the Number of Iterations

Lastly, **we tested the impact of the number of iterations on our Hadoop implementation** with 100,000 points, 2 features and 5 clusters.

Table 4: Execution Times (6 iterations, 5 clusters and 10,000,000 points)

	Hadoop Time (s)
3 iterations	68.690
4 iterations	93.718
6 iterations	140.307
8 iterations	190.680
11 iterations	264.627
12 iterations	287.078
14 iterations	336.110
19 iterations	456.585

It comes as no surprise that the number of iterations emerged as the key time-consuming factor in the process. The obtained results vividly demonstrate a direct and proportional relationship between the execution time and the number of iterations.

The ability to dispose of more nodes and more powerful ones can reduce the time needed to complete each iteration though this is not a solution. The only way to control the number of iterations itself is a tuning of the stopping conditions which is highly dependent on the needs of the user. **Our implementation allows for the setting of the maximum number of itereations (which is self-explainatory) and for the choice of a minimum epsilon.**

Epsilon determines the percentage of centroid movement between consecutive iterations. Setting a minimum epsilon value of zero enables the algorithm to achieve convergence, where each centroid remains unchanged from one iteration to another because it has already reached an optimal or sub-optimal position. **Higher values of minimum epsilon reduce the number of iterations at the expense of an approximate solution.**