Implementation of K-means Clustering in Hadoop MapReduce

Ullah, Md Azim
mullah@memphis.edu

Abstract. In this paper we describe our implementation of well known k-means algorithm in Hadoop Map Reduce. It introduces readers on the task level parallelism that is achieved in a MapReduce algorithm through the distribution of processing in a distributed system. A further comparative analysis of the implemented algorithm is done against Apache Spark in terms of time complexity.

1. Introduction

K-means clustering is a famous clustering algorithm that clusters n data points into k number of clusters typically known as centroids. The prerequisite is that k is known. An initial selection of k centroids starts the process. Each data point is then assessed based on its distance to the nearest centroid to form an initial grouping of the whole dataset into k groups. A reselection of the centroids precedes the same steps. Algorithm finishes when there is no more movement of centroids and a minimization of the objective function is achieved. Hadoop framework has been established for superior performance for computation and analysis of big data done over a distributed architecture. MapReduce generates the necessary flexibility and speed required for a distributed framework to facilitate unstructured data processing algorithms like K-means clustering. This project aims to analyze and report the advantages gained in using the distributed framework for a computationally expensive algorithm like k-means

1.1. Description of the Algorithm

Let X = $\{x_1, x_2\ \ . . . ,\ x_n\}$ be a set of n data points. The k-means problem seeks to find a set of k means M = $\{\mu_1, \mu_2\ \ . . . , \mu_k\}$ which minimizes the objective function

$$f(M)\ = \Sigma_{x \in X} min_{\mu \in M} \|x - \mu\|^2$$

So in plain terms we wish to find k points from the dataset known as means with the objective of minimizing the sum of squared distance from each point on the dataset to it’s nearest mean.

Although a NP-hard problem to begin with there have been a considerable number of heuristic solutions which enable a well enough approximate. The standard k-means algorithm from is described below with it’s iterative steps[1].

1.1.1. Standard k-means:

1. Choose k initial means $\mu_1, . . . , \mu_k$ uniformly at random from the set X.
2. For each point x ∈ X, find the closest mean μ_i and add x to a set S_i .
3. For i = 1, . . . , k, set μ_i to be the centroid of the points in S_i .
4. Repeat steps 2 and 3 until the means have converged.

The criterion for convergence is when the total error is static or in other words a local minimum of the objective function has been reached.

1.1.2. Distributed k-means in Mapreduce:

In the distributed algorithm each machine in the distributed system has a copy of the k mean points of each step. We distribute the number of points in the dataset to different machines.

Map step:

Then for each point in the dataset we calculate the distance of each point from the k means and return a key,value pair of **(i ,(x,1))** where i is the index of the mean that has the minimum distance from the point and x is the value of mean.

Reduce step:

In the reduce step we combine all the points with similar keys. For two key,value pairs of (i,(x,s)) and (i,(y,t)) with same keys we combine them like (i,(x+y,s+t)). After this reduce step we get k pairs of the form (i, $(\Sigma_{x\in S_i} x_i, |S_i|)$). Now the new mean to found in this step is

$$\mu_i \leftarrow \frac{1}{|S_i|} \Sigma_{x\in S_i} x$$

Next we define μ_i as the point within the cluster S_i which has the closest distance to μ_i.
The process stops when no new means are found.

1.2. Hadoop Distributed Computing Framework

Hadoop is an open-source software framework for distributed storage and distributed processing. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework. The framework is composed of the following four modules:

1. Hadoop Common – libraries and utilities for other Hadoop modules;

2. Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity

machines

3. Hadoop YARN – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications.

4. Hadoop MapReduce – an implementation of the MapReduce programming model for large scale data processing.

2. Related Works

K-means is an unsupervised method of clustering data. There have been some related works on parallelizing the algorithm on Hadoop based frameworks. From sensor area networks to general topologies to remote sensing parallelizing k-means over a cluster of nodes in a distributed computer network have been shown to improve the computational performance and efficiency. The most cited work on the implementation of k-means in Hadoop makes the use of the same principles as defined in the introduction section. The performance can be much improved with the initialization process modified to include pseudo-random data points as the k mean values. This does not increase the load on the distributed system much.

Apache Spark has an distributed implementation of K-means in Spark Mllib library. This project aims to deal with the MapReduce implementation of the algorithm and understanding the distributed framework that implements the MapReduce code.

3. Hardware Setup

For setting up distributed hadoop environment I setup a hadoop cluster in a pseudo distributed way within my machine. The configuration of the machine is as follows

Processor: AMD Quad-Core A10
Memory: 6GB

For setting up Spark I used the precompiled binaries that are available from Apache website. The necessary interactions with the Spark environment was done using python through Pyspark library

4. Design and Implementation

Pseudo code of our implementation is described first with the inclusion of Map Reduce code written in Java.

4.1. Pseudo Code

The pseudo code for implementation of k means in Mapreduce is given below in steps:

1. Given k number of clusters we select k random points as initial cluster centroids. We initialize one arrays of centroids with these k initial points. Let us assume their names are collectively stored in a list “Centroids”. We initialize a global counter equal to 0.

2. In the map step for each data point in the input(2D cartesian point) we find out the centroid which has the minimum distance from the point and emit the key value pair of

   key = the particular centroid and its index, *value* = the coordinate of the point in question.

3. In the reduce step we are given the values generated from the map step for each key. As explained in 1.1.2 we find the new value of the centroid by averaging over all the values for a given key out of k possible keys. We find the new centroid as the point which is the closest to the centroid that we found after averaging.

4. Insert the value of the point in the index of "Centroids" that was already emitted in the map step.

5. While replacing the centroid value in a given index we take note of a global counter which increments if and only if the centroid value at that index is required to change.

6. At the end we check if the global counter is 0 or not. If it is zero that means in the previous iteration no centroid update occurred. We stop the computation and the output of the previous iteration as saved in the output directory is our answer.

7. If we do not find the global counter to be then we reinitialize the counter to be 0 and start from step 2.

4.2. Code

We have completed and tested the MapReduce code for only the first iteration. All the logics in Map step and Reduce Step has been implemented. In terms of Pseudo Code only step 4 remains to be included. For viewer's judgement here we include only the Mapper and Reducer steps.

Mapper:

```
public void map(Object key, Text value, Context context) throws
                                        IOException, InterruptedException {

 String line = value.toString();     // converting the input to string
 StringTokenizer tokenizer = new StringTokenizer(line);
 String[] x_y_map_input = new String[2];
 while (tokenizer.hasMoreTokens()) {
         x_y_map_input = tokenizer.nextToken().split(",");   // splitting the input point into x,y values
 }
 double[] distance = new double[Centroids.size()]; // initialize an array which will store the distance
                                                           from centroids
 for(int i=0;i<Centroids.size();i++) {
         distance[i] = computedistance(Centroids.get(i),x_y_map_input);  // compute distance of each
                                                           centroid to the input
 }

 int index = findmindistance(distance);     // finds the index of the point which has minimum distance
```

```
    word =  new Text(index+","+Centroids.get(index)[0]+","+Centroids.get(index)[1]);   // key = (index,
                                                                                          centroid value)
    word1 = new Text(x_y_map_input[0]+","+x_y_map_input[1]);    // value = input point

    context.write(word,word1);  // emit key value pair

   }
```

Reducer:

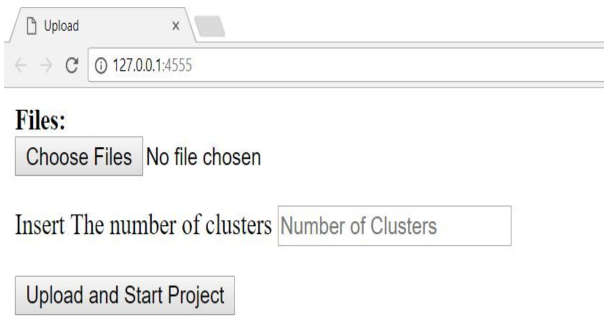
```
  public void reduce(Text key, Iterable<Text> values, Context context) throws
                                                   IOException,      InterruptedException {
  String temp = key.toString();
  String[] rt = temp.split(",");               // our key = index,x value of centroid, y value of centroid
  int sumx = 0; int sumy = 0;  int n = 0;
  ArrayList<String[]> store = new ArrayList<>();           // to store all the values for a given key
  for(Text val:values ) {
            String temp1 = val.toString();
            String[] temp2 = temp1.split(",");
            store.add(temp2);
            sumx = sumx+Integer.parseInt(temp2[0]);
            sumy = sumy+Integer.parseInt(temp2[1]);
            n++;
  }

  int centroid_x = (int)sumx/n;                           //averaging over the x values in coordinates
  int centroid_y = (int)sumy/n;                           //averaging over the y values in coordinates
  String[] ncn = new String[2];                           // storing New centroid found by averaging
  ncn[0] = Integer.toString(centroid_x);
  ncn[1] = Integer.toString(centroid_y);

  double[] distance = new double[store.size()];
 for(int i=0;i<store.size();i++) {
            distance[i] = computedistance(ncn,store.get(i));   // compute distance of centroid to all the
                                                                  input points for this key
   }
 int index = findmindistance(distance);  // finds the index of the point which has minimum distance =
                                                        this is the new centroid to be inserted

  if(!checkequal(store.get(index),Integer.parseInt(rt[0]) )) {     // check if stored centroid and the new one are
            Centroids.set(Integer.parseInt(rt[0]),store.get(index));      are equal else store the new one
  }
  }
```

4.3. Iterative kmeans

As described in pseudo code iterative implementation is done by keeping track of the counter. Since its inception iterative algorithms have been a bottleneck for Hadoop. As will be described by the following sections Apache Spark vastly outperforms Hadoop when it comes to iterative algorithms.

4.4. Web Interface

As the goal of this project is not only to implement the mapreduce code in hadoop distributed systems rather perform detailed diagnostics on the performance of the code with increased friendliness to the technical nuances we have implemented an interface in python with the help of a well known python web framework "Flask". This allows the users to input the original input text file and select the number of clusters that they want. The screenshots of this interface are given in figures 1 and 2.



Upload

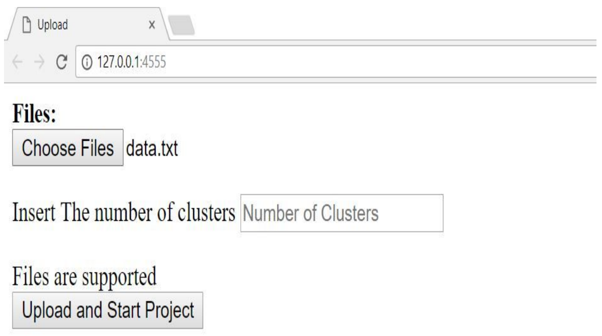
127.0.0.1:4555

Files:

Choose Files No file chosen

Insert The number of clusters Number of Clusters

Upload and Start Project



Upload

127.0.0.1:4555

Files:

Choose Files data.txt

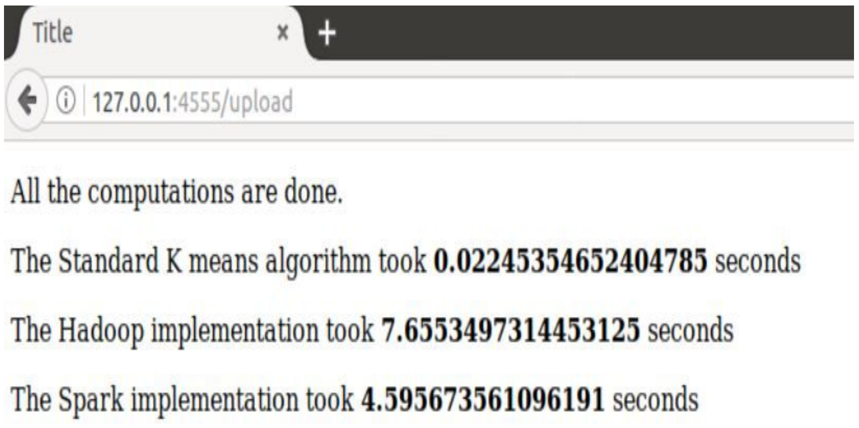
Insert The number of clusters Number of Clusters

Files are supported

Upload and Start Project

Figure 1: Initial Look of the interface which asks the user to input the data file and number of clusters

Figure 2: The interface checking if the input data file is supported



Title

127.0.0.1:4555/upload

All the computations are done.

The Standard K means algorithm took **0.02245354652404785** seconds

The Hadoop implementation took **7.6553497314453125** seconds

The Spark implementation took **4.595673561096191** seconds

Figure 2: Final phase of interface showing comparative results of different methods

5. Results and Comparison with Apache Spark

The efficiency of our implementation is tested in comparison to the Apache Mllib implementation of K-means algorithm for different size inputs. Table 1 documents the time complexity comparisons with the same number of clusters specified for each method.

No. of Data Points	Time taken by Hadoop Map Reduce Implementation(our method) in seconds	Time Taken by Apache Spark Mllib Implementation in seconds
20	8. 64	4.70
2324	165.79	15.019
10000	735.69	18.1174
20000	1372.29	21.65

Table 1. Comparison of Hadoop Map Reduce with Apache Spark

The striking difference between our mapreduce implementation and the implementation as done by Apache Mllib library is the definition of the cluster centroid itself. Here I define the cluster centroid to be one of the given points which is closest to the original centroid as found by averaging over all the values. But in other frameworks the original centroid does not need to be a given point in the data. This introduces additional complexity in our implementation but the motivation is to assign given points as cluster centroids not any floating point number that can be intuitively thought to be within our data but is not really a member.

All factors considered Apache Spark has a 40 fold improvement in time efficiency. Kmeans is an iterative machine learning algorithm. Map Reduce introduces task level parallelism towards a problem. But in case of multiple iterations this introduces sufficient I/O bottlenecks to performance. Spark uses Mesos which is a distributed system kernel. By caching the intermediate dataset after each iteration and running multiple iterations on this cached dataset reduces I/O bottlenecks and helps the algorithm to converge in a fault tolerant manner.

6. Limitations

The project has significant limitations in some areas. The key limitation is the lack of research done on the initialization of cluster centroids. A big overhead of K means is the number of iterations to converge. An intelligent initialization of the cluster centroids in step 1 would drastically reduce the time complexity of any K means algorithm by reducing the number of iterations itself. Spark and Hadoop being environments superimposed on our local operating system, the web interface should have taken account of the system variables so as to make it possible to run on any machine.

7. Conclusion

The project taught us valuable lessons in the introduction of parallelism in an expensive algorithm. The fundamentals of Map Reduce made us realize the importance of a distributed file system in handling these algorithms. Further implementation and smooth execution of Map Reduce code in a distributed system was achieved. The most important finding of this project is the strength of Apache Spark to handle big data and analytics. Our future work will be on understanding the operating principles of Spark and use it towards better implementation of distributed algorithms.

8. References

[1] Stuart P Lloyd. Least squares quantization in pcm. Information Theory, IEEE Transactions on, 28(2):129–137, 1982.

[2] Zhao, Weizhong, Huifang Ma, and Qing He. "Parallel k-means clustering based on mapreduce." *IEEE International Conference on Cloud Computing*. Springer, Berlin, Heidelberg, 2009.