Clustering Algorithm Implementation Based on Mapreduce

Shuxin Li, Yilun Yang and Chang Liu
Instructor: Professor Cho-Jui Hsieh

1. Introduction

Clustering has been important approach in data mining fields, widely used in document retrieval, image recognition and pattern classification. The enlarging volume of data makes clustering being complicated and time-consuming. We would propose and implement a K-Means Clustering algorithm based on the Mapreduce computing framework.

This report represents our work mainly in deployment, configuration, algorithm design/implementation, experimental results and conclusion

1.1 Hadoop Installation and Deployment:

Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.
Since we are going to write Mapreduce code and then run it on HDFS, so a Hadoop platform should be deployed first. Our groups use at most four computers as our computing machines, details about how to deploy a Hadoop is in the below chapter.

1.2 Mapreduce computing framework:

Mapreduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
The term Mapreduce actually refers to two separate and distinct tasks that Hadoop programs perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples. As the sequence of the name Mapreduce implies, the reduce job is always performed after the map job.

1.3 Mrjob

Although Hadoop is primarly designed to work with Java code, it supports other languages via Hadoop Streaming, his jar opens a subprocess to your

code, sends it input via stdin, and gathers results via stdout. mrjob is a framework that assists you in submitting your job to the Hadoop job tracker and in running each individual step under Hadoop Streaming.
It lets you write Mapreduce jobs in Python 2.6+ and run them on Hadoop.

1.4 Sample Data Generator and result visualization

Since clustering algorithm will cluster and group data, so test data or sample data must be input as well. We used MASS package on R language to generate sample data, most observations we created are multi-normal distributed with different mean vectors and the same covariance.
For clustering visualization and result demo, we used ggplot. ggplot is a plotting system for R, based on the grammar of graphics, which tries to take the good parts of base and lattice graphics and none of the bad parts. It takes care of many of the fiddly details that make plotting a hassle (like drawing legends) as well as providing a powerful model of graphics that makes it easy to produce complex multi-layered graphics.

2. Hadoop Platform Deployment

2.1 Cloudera Manager

2.1.1 Cloudera introduction

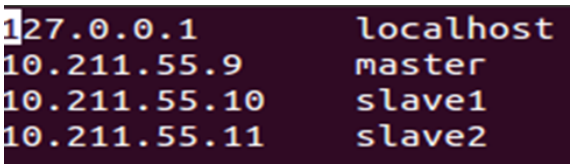
Cloudera Provides open-source Apache Hadoop distribution called CDH and also the Monitor and manage software cloudera manager. CDH contains the main, core elements of Hadoop that provide reliable, scalable distributed data processing of large data sets (chiefly Mapreduce and HDFS), as well as other enterprise-oriented components that provide security, high availability, and integration with hardware and other software.

2.1.2 Local configuration

We used cloudera manager to help achieve the deployment. To begin with, we firstly set some configuration on each machine.

(a) install openssh-server
sudo apt-get install openssh-server

(b) Modify hosts and hostname
Each /etc/hosts file would looks like:

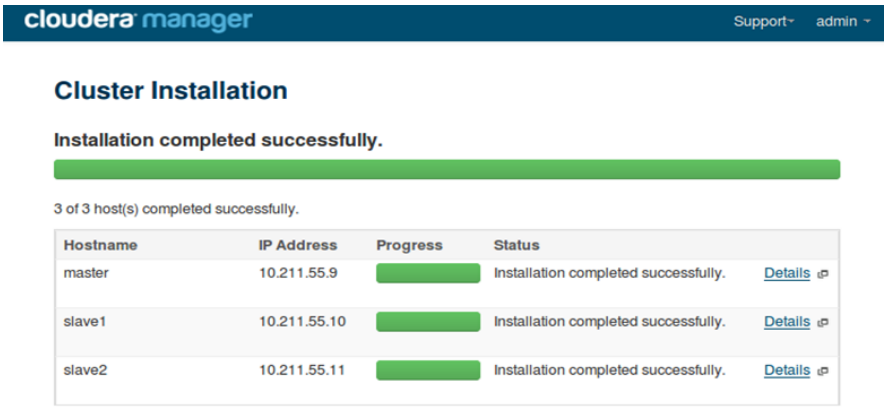


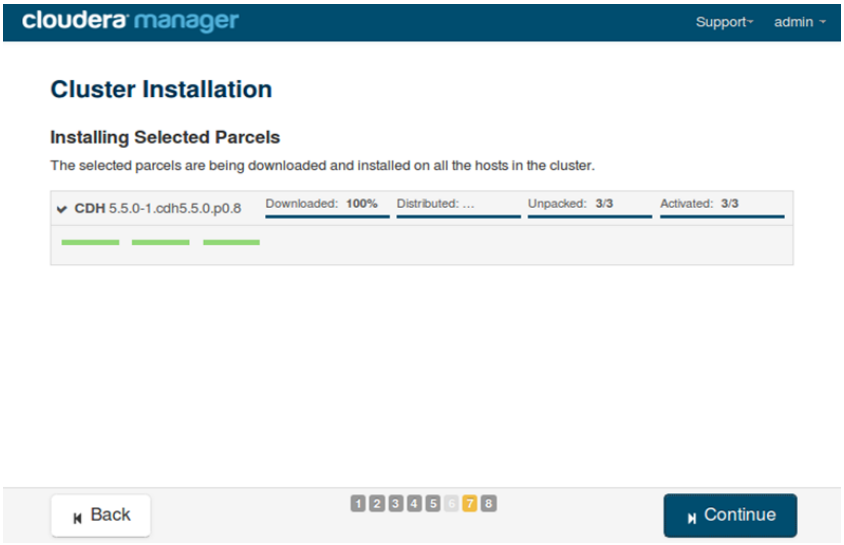
(c) Modify root privilege
To access with no password, we should add:
waterman(our username) ALL=(ALL) NOPASSWD:ALL
At the very end of /etc/sudoers file.

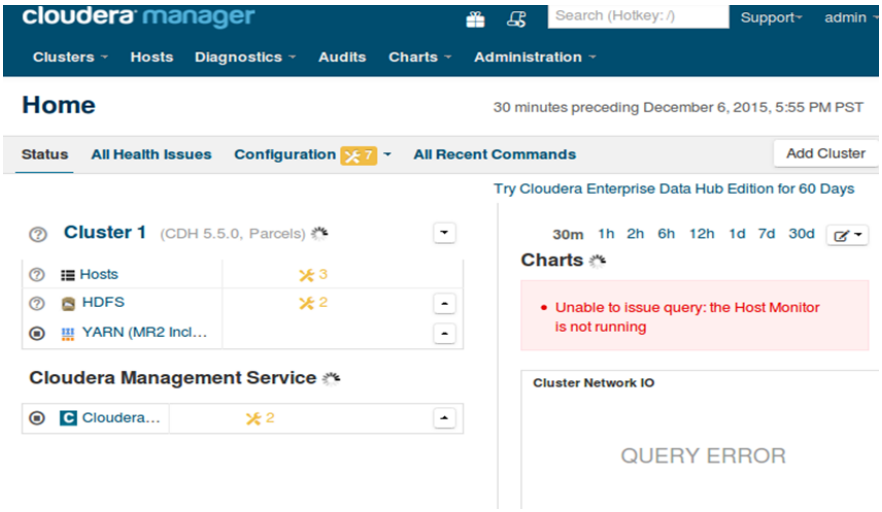
2.1.3 Cloudera deployment

(a) Cloudera manager installation
With Cloudera manager installer, we could install cloudera manager and also postgresql database. We are able to access to cloudera manager web UI over 7180 port after then.

(b) Starting services
Finally, we finished the deployment process on the web by downloading, distributing, unpacking CDH5.5, setting configuration and starting HDFS and Mapreduce services. In this process, all nodes in the cluster use the same username(waterman)and password. Snapshots are like the following:

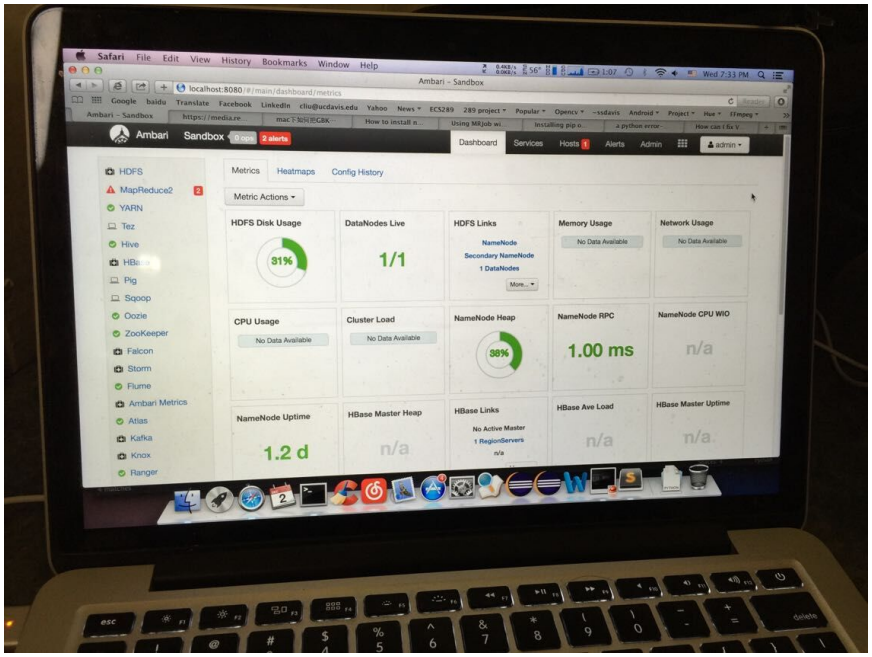




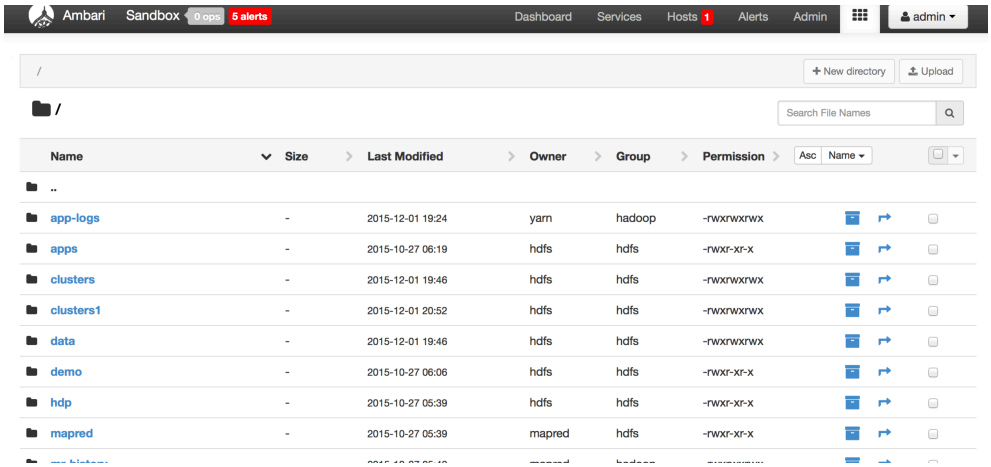


2.2 Hortonworks Sandbox

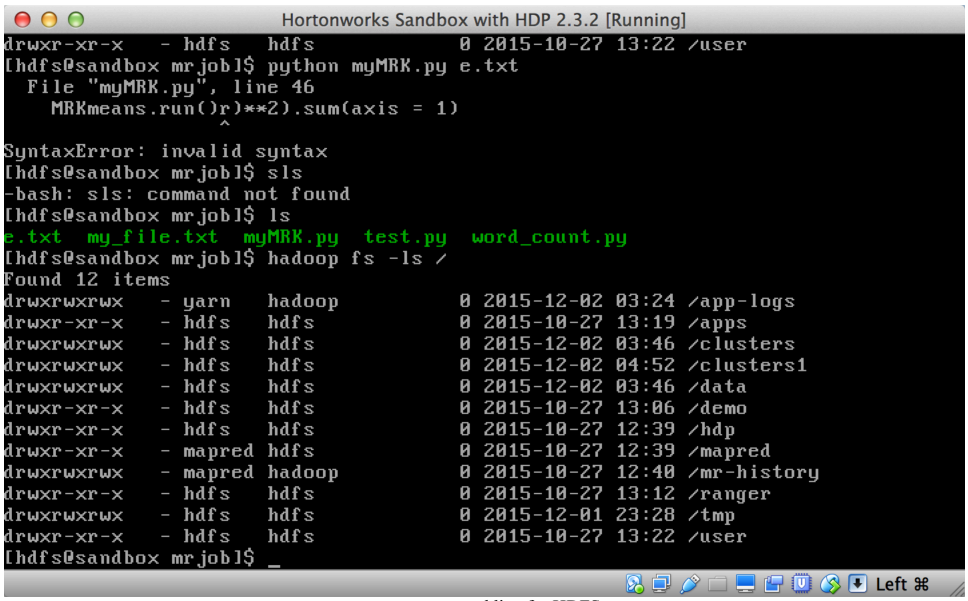
The Hortonworks Sandbox is delivered as a virtual appliance. The virtual appliance (indicated by an .ovf or .ova extension in the filename) runs in the context of a virtual machine (VM), a piece of software that appears to be an application to the underlying (host) operating system (OS), but that looks like a bare machine, including CPU, storage, network adapters, and so forth, to the operating system and applications that run on it. We imported Sandbox into VirtualBox and started this single-node cluster successfully.



As we can see in this photo, we have launched the Hadoop cluster. This webpage is the hue web interface in which we can operate and maintain HDFS or job tracker fairly easily.



photoscreen for HDFS directory



command line for HDFS

3. Sample Data Generator

Since we want to explore the performance of K-means based on our own MapReduce and Mahout in Hadoop platform, we need a large dataset. There are several solutions. The first solution is that we can find some practical datasets on Kaggle, however, we are not sure that if these datasets are suitable to using clustering method and we never know the real centroids. The second solution is that we can simulate a big dataset by high-level language(R language), in which we already know the centroids

We choose to apply the second solution and we plan to simulate three datasets of multivariate normal distribution of a 2-dimensional random vector in which $x = [X_1, X_2]$, $x : N(\mu, \sum)$

Where $\mu = [E[X_1], E[X_2]]$ and $\sum = [Cov[X_i, X_j]], i = 1, 2 j = 1, 2.$

We generate three datasets $N(\mu_1, \sum), N(\mu_2, \sum), N(\mu_3, \sum)$ where $\mu_1 = [1, 1]$ $\mu_2 = [5, 5], \mu_3 = [10, 10], \sum = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Therefore we can combine the three datasets together and we know that our real centroids are μ_1, μ_2, μ_3.

To generate a big dataset, we use statistics department server to run the R script and we generate different size of datasets which have 10000 and 1000000 observations.

4. K-Means Mapreduce Code Implementation

We proposed two kinds of python code implementions for Mapreduce K-Means algorithm. We will firstly present and explain the core code for both versions code.

4.1 Mapreduce Overview

We write map, combine, and reduce functions that are submitted to the job tracker for execution.

A mapper takes a single key and value as input, and returns zero or more (key, value) pairs. The pairs from all map outputs of a single step are grouped by key.

A combiner takes a key and a subset of the values for that key as input and returns zero or more (key, value) pairs. Combiners are optimizations that run immediately after each mapper and can be used to decrease total data transfer. Combiners should be idempotent (produce the same output if run multiple times in the job pipeline).

A reducer takes a key and the complete set of values for that key in the current step, and returns zero or more arbitrary (key, value) pairs as output.

After the reducer has run, if there are more steps, the individual results are arbitrarily assigned to mappers for further processing. If there are no more steps, the results are sorted and made available for reading.

4.2 VERSION ONE

Mapper Function:

```
def mapper(self, _, lines):
        centroids = self.get_centroids()
        for l in lines.split('\n'):
            x,y = l.split(',')
            point = [float(x),float(y)]
        min_distance=INTEGER_MAX
        class = 0
        for i in range(cluster_num):
            distance = self.dist_vec(point,centroids[i])
```

```
            if distance < min_distance:
                min_distance = distance
                class= i
        yield class, point
```

Combiner Function:

```
def combiner(self,k,v): # Calculate for centroid for each
class at the end of the                    mapper
    count = 0
    moy_x=moy_y=0.0
    for t in v:
      count += 1
      moy_x+=t[0]
      moy_y+=t[1]
    yield k, (moy_x/count,moy_y/count)
```

Reducer Function:

```
def reducer(self, k, v):  # Calculate for centroid for each
combiner
   moy_x=moy_y=0.0
   for t in v:
       count += 1
       moy_x+=t[0]
       moy_y+=t[1]
   print str(moy_x/count)+","+str(moy_y/count)
```

Main Function:

```
if __name__ == '__main__':
    MRKMeans.run()
```

Helper Functions:

```
def get_centroids(self):
   f = open(self.options.c,'r')
    centroids=[]
    for line in f.read().split('\n'):
        #print line
        line = line.strip()
        if line:
            x,y = line.split(',')
            centroids.append([float(x),float(y)])
    f.close()
    #print centroids
    return centroids
```

note: This function will read the initial cluster centroid file. Besides input files must follow some specific format

4.3 Version Two (Latest version, more efficient and I/O unblocked)

Our group also tried another way to implement Kmeans in Mapreduce, we tried to let the program iterate automatically instead of tapping keyboard for each iteration. Here is the code for this version:

4.3.1 Data:

Data format in this version is a little bit different. The first column is data ID, the second column is initial cluster ID, the last column is each data's coordinates. A sample dataset is like the following:

```
1|3|1.78742833671961,2.76729907770636,3.05047780993627
2|2|0.102735230852052,2.16284054379424,3.82873946593105
3|2|2.44628126803956,2.34106573809756,1.71589626250546
4|2|0.238090195979469,2.74096567048553,3.2386548938678
5|3|1.20879622349637,1.09388906144333,3.59922957666696
```

4.3.2 Mapper:

```
def relabel_data(self, _, line):
    data_ID, Cluster_ID, Coord= line.split('|')
    Coord = Coord.strip('\r\n')
    Coord_arr = np.array(Coord.split(','), dtype = float)
    global Centroid
    Centroid = self.get_centroids()
    global Centroid_old
    Centroid_old = Centroid.copy()
    Centroid_arr = np.reshape(Centroid, (-1, len(Coord_arr)))
    global nclass
    nclass = Centroid_arr.shape[0]
    global ndim
    ndim = Centroid_arr.shape[1]
    Distance = ((Centroid_arr - Coord_arr)**2).sum(axis = 1)
    Cluster_ID = str(Distance.argmin() + 1)
    Coord_arr = Coord_arr.tolist()
    yield Cluster_ID, (data_ID, Coord_arr)
```

We firstly defined a get_centroids function to read the Centroids from a txt file, the centroids of different clusters are aligned in a single line in the file. In the whole Mapreduce code we will primarily use numpy to process the data except when data is passed from mapper to combiner and from combiner to reducer. We need to use list instead because the default JSON protocol in Mrjob doesn't support numpy data stream.

In mapper, we read each line and cut the line into data Id, cluster ID and coordinate values. We also read in centroids and reshape it so that distance can be easily calculated at once. After that, we relabeled all the data according to distance. The output key of mapper is cluster ID, value is a combination of data_ID and the list of coordinates. Note we also declared global variables so that combiner and reducer can use them as well and we can also verify the convergence of clustering.

4.3.3 Combiner:

```
def node_combine(self, Cluster_ID, values):

    member = []
    Coord_set = []
    Coord_sum = np.zeros(ndim)
    for data_ID, Coord_arr in values:
        Coord_set.append(','.join(str(e) for e in Coord_arr))
        Coord_arr = np.array(Coord_arr, dtype = float)
        member.append(data_ID)
        Coord_sum += Coord_arr
        Coord_sum = Coord_sum.tolist()
    yield Cluster_ID, (member, Coord_sum, Coord_set)
```

In the Combiner, we collected all members and coordinates for each cluster IN EACH MACHINE. The reason to do this will be introduced in reducer function. We also calculated the coordinate summation of all data within each cluster IN EACH MACHINE. We do this because we want to avoid the speed delay due to data transformation to reducer.

4.3.4 Reducer:

```
def update_centroid(self, Cluster_ID, values):

    final_member = []
    final_Coord_set = []
    final_Coord_sum = np.zeros(ndim)
    for member, Coord_sum, Coord_set in values:
        final_Coord_set += Coord_set
        Coord_sum = np.array(Coord_sum, dtype = float)
        final_member += member
        final_Coord_sum += Coord_sum

    n = len(final_member)
    new_Centroid = final_Coord_sum / n
    Centroid[ndim * (int(Cluster_ID) - 1) : ndim * int(Cluster_ID)] = new_Centroid
    if int(Cluster_ID) == nclass:
        self.write_centroids(Centroid)

    for ID in final_member:
        ind = final_member.index(ID)
        yield None, (ID + '|' + Cluster_ID + '|' + final_Coord_set[ind])
```

Reducer generally does the same thing as combiner except that it works on ALL MACHINES. We defined a function called write_centroids so that we can update the centroids and write them into a file. At the end, I reformatted

the reducer output following rawprotocol in mrjob with no key and with data ID, cluster_ID, coordinates separated by '|' as value. As we can see, the output is exactly the same as original input file except the cluster ID has been updated. Note that the coordinates and members recorded by combiner and reducer are necessary to get our reformatted data. That is why they were stored and passed for many times.

4.3.5 Configuration and steps procedure:

Mrjob is special because it can run multi-step Mapreduce code as in JAVA by passing the output from last reducer to the next mapper. Remember our reducer output has exactly the same format as mapper input thus the steps procedure is workable. To achieve this, one way is to add an option to the linux command line which specify how many iterations to go then pass this variable to the steps function. The code is like the following:

```
def configure_options(self):
    super(MRKmeans, self).configure_options()
    self.add_file_option('--infile')
    self.add_file_option('--outfile')
    self.add_passthrough_option('--iterations', dest='iterations', default=10, type='int')
def steps(self):
    return [MRStep(mapper=self.relabel_data,
                   combiner=self.node_combine,
                   reducer=self.update_centroid)] * self.options.iterations
```

Following our Mapreduce function, we can perform any times of Mapreduce job we want until convergence.

5. Mahout K-Means

5.1 Mahout Introduction

Mahout is a library of scalable machine-learning algorithms, implemented on top of Apache Hadoop and using the Mapreduce paradigm. Once big data is stored on the Hadoop Distributed File System (HDFS), Mahout provides the data science tools to automatically find meaningful patterns in those big data sets. The Apache Mahout project aims to make it faster and easier to turn big data into big information.

5.2 Mahout Usage

We use the Hortonworks Sandbox which can implement a single node Hadoop environment. Since Mahout is not included in Sandbox, we use the command line “yum install mahout” to install it.

The usage details of Mahout K-means are as below. There are mainly three steps.

The input dataset synthetic.control.data includes two columns numbers which are separated by a space.

- place the dataset into HDFS.

```
Hadoop fs -put /<local_paht>/synthetic_control.data  /<Hadoop path>/
```

- run k-means clustering.

```
mahout org.apache.mahout.clustering.syntheticcontrol.kmeans.Job
-i <Hadoop input path>: Path to job input directory.
-o <Hadoop output path>: The directory pathname for output.
-dm <distanceMeasure>:Default is SquaredEuclidean
-k <number of clusters>: The number of clusters to create
-t1 <threshold value>: T1 threshold value related to Canopy
method
-t2 <threshold value>: T2 threshold value related to Canopy
method
-x <number of iterations>: The maximum number of iterations.
-ow: If present, overwrite the output directory before running
job
-h: Print out help
```

- convert the clustering output of SequenceFile format which is not human readable into human readable format.

```
mahout clusterdump
-i <Hadoop input path>: Path to job input directory.
-o <local ouput path>: The directory pathname for output.
-p <points Dir>: The directory containing points sequence
files mapping input vectors to their cluster. If specified,
then the program will output the points associated with a
cluster
```

6. Experimental Results

6.1 SMALL Data Set:

Observation number: 9999
Data points file size: 351KB

Sample data is generated by MASS, following multi-normal distribution. Observation has three clusters, which are (1,2);(4,5);(9,10).

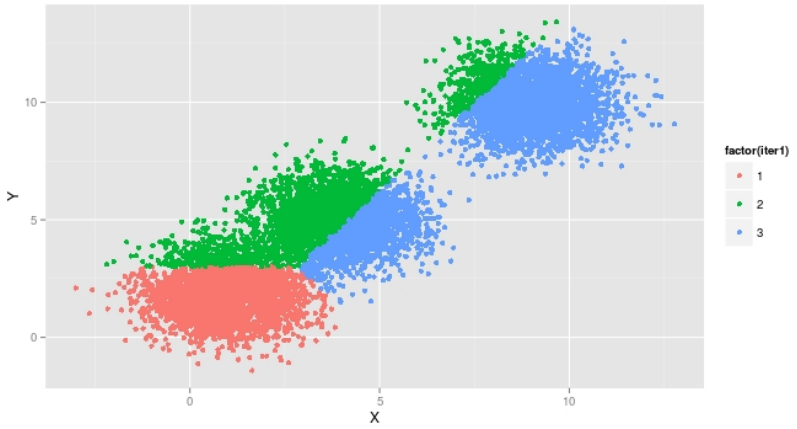
If running our Mapreduce code and finally get converged cluster centroids close to (1,2);(4,5);(9,10), we can conclude that our code can work and get correct clustering centroids

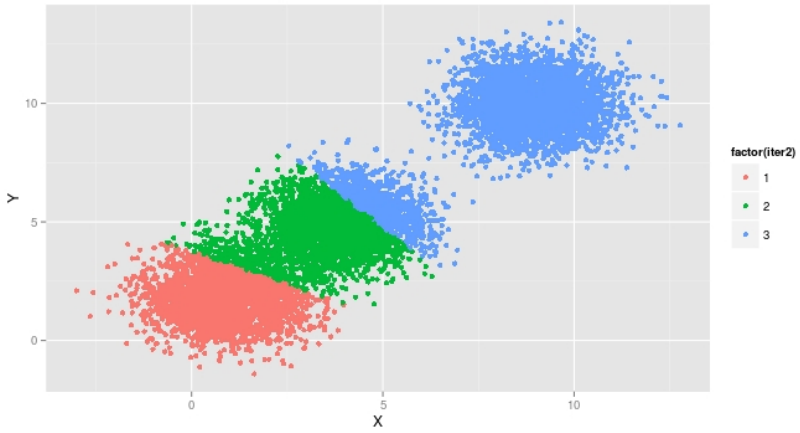
- VERSION ONE

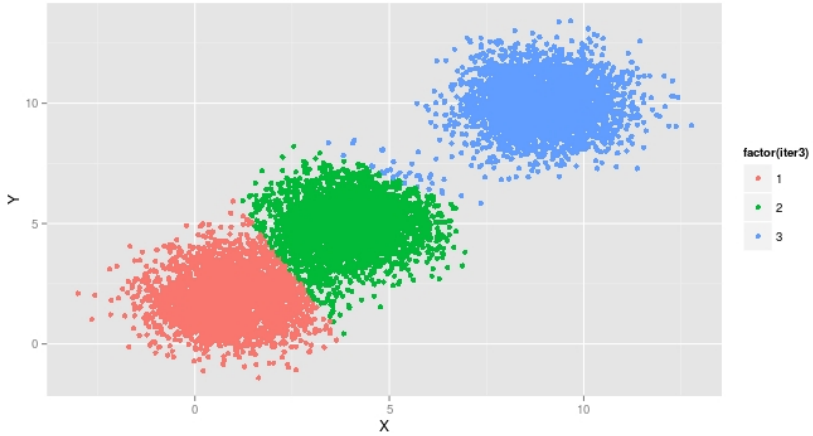
Small data set and visualization result:
Data points dimension : 2
Cluster Number K : 3
Sample Data fragment:

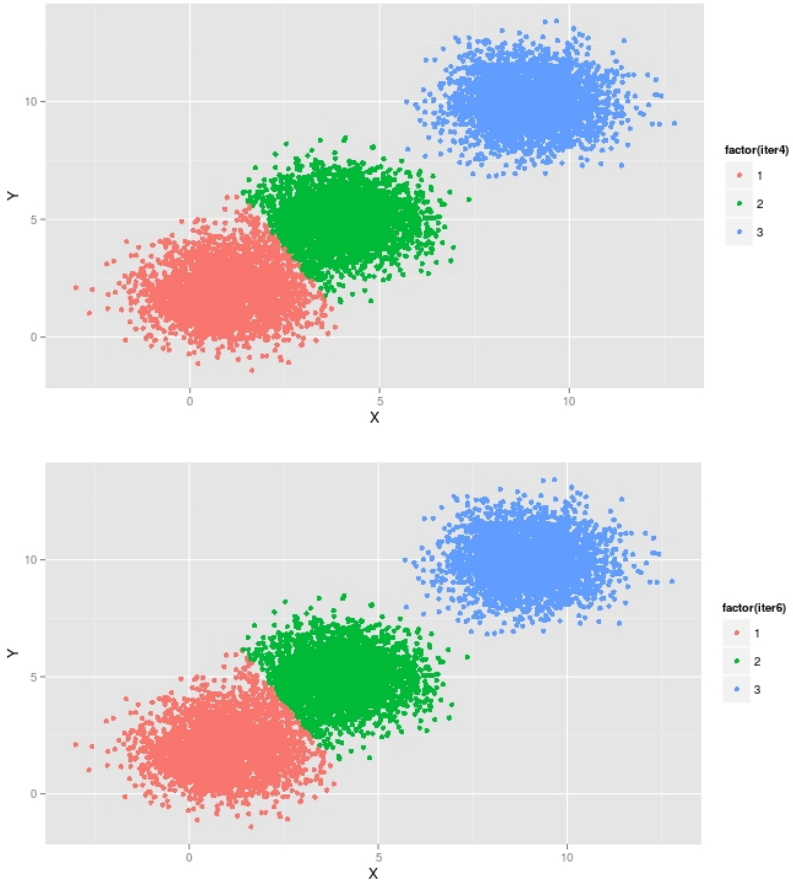
2.41222176553237, 2.03083353049084
0.789274342728431, 2.64271807174558
-0.753716953903275, 1.60623703895319
1.58822353291241, 2.03322477981346
-0.131034903476554, 1.85734157467443
1.83955563192581, 2.26715157587592
1.59804608097208, 2.78315822963477
0.151521997032367, 1.64954045304973
0.432868061935464, 2.17902844527602
1.67795432247325, 1.85091113216999
0.501254692923928, 2.54389593934225
1.32534997875302, 2.79097773942509
0.994448517713719, 1.0026383776995
-0.107167977276981, 2.91853755525468
0.981428012883737, 1.18222823630342
-0.708432387010828, 1.52471139855828
1.30209741130723,1 .42721639740726
1.74121540713026, 2.35013240905996
1.08798739259279, 0.850512246124881
0.276715903319533, 4.44444184106394
…

Cluster Assignment Sequence Plots(From iteration 1 to 10):


Y
10
5
0
0
5
10
X
factor(iter1)
1
2
3


Y
10
5
0
0
5
10
X
factor(iter2)
1
2
3


Y
10
5
0
0
5
10
X
factor(iter3)
1
2
3



Conclusion: We can find that the Mapreduce K-Means code works correctly. Three centroids are (0.972587182033,1.97315197858); (3.99484710407,5.01486630249); (9.00785403589,9.99782697969)

Our final clustering centroids are very close to (1,2);(4,5);(9,10), which matches the observation distribution.

Centroids are being updated during the computation. After six iterations, cluster centroid will be converged, and input data points will be assigned to cluster correctly(Respectively, Red, Green and Blue means three clusters)

6.2 MEDIUM Data Set:

Observation number: About 1000000 (One Million)
Data points file size: 30MB

Sample data is generated by MASS, following multi-normal distribution. Observation has three clusters, which are (1,1);(5,5);(10,10).

If running our Mapreduce code and finally get converged cluster centroids close to (1,1);(5,5);(10,10), we can conclude that our code can work and get correct clustering centroids

We run TWO version of Mapreduce codes that we wrote and Mahout.
Below is the results of our codes:

- VERSION ONE

  Iteration number until convergency: 10
  Average running time for one iteration: 58 seconds
  Total running time: 10 minutes
  Converge to correct cluster centroids: YES

Iteration NUM	Centroid 1 X-Cord	Centroid 1 Y-Cord	Centroid 2 X-Cord	Centroid 2 Y-Cord	Centroid 3 X-Cord	Centroid 3 Y-Cord
1	-0.18539	0.84588	1.12097	0.10263	6.35894	6.34251
2	0.19741	1.47361	1.62675	0.71945	7.53059	7.53305
3	0.28651	1.51410	2.21171	1.38404	7.83485	7.84839
4	0.33794	1.04740	3.11564	2.55487	8.26571	8.28968
5	0.75372	0.86638	4.35175	4.23113	9.26474	9.28231
6	0.97147	0.97593	4.95615	4.95681	9.98363	9.98259
7	0.99566	0.99870	4.99643	4.99886	10.00028	9.99881
8	0.99691	1.00003	4.99792	5.00026	10.00041	9.99895
9	0.99697	1.00010	4.99798	5.00033	10.00041	9.99895
10	0.99697	1.00010	4.99798	5.00033	10.00041	9.99895

- VERSION TWO

  Iteration number until convergency: 7
  Average running time for one iteration: 20 minutes
  Total running time: About 2 hours
  Converge to correct cluster centroids: YES

Iteration NUM	Centroid1 X-Cord	Centroid1 Y-Cord	Centroid2 X-Cord	Centroid2 Y-Cord	Centroid3 X-Cord	Centroid3 Y-Cord
1	5.70894	4.93751	6.49546	7.91586	0.20005	0.27447
2	4.94566	5.00242	7.44070	8.34529	0.39477	0.37477
3	5.06521	4.96794	8.00214	8.88744	1.25588	0.57845
4	4.99221	4.99201	9.79433	9.58544	0.98934	0.99611
5	5.00474	4.99806	9.99405	9.99914	0.99921	0.99945
6	4.99738	4.99961	9.99835	9.99918	0.99968	1.00022
7	4.99740	4.99962	9.99835	9.99918	0.99969	1.00024
8	4.99740	4.99962	9.99835	9.99918	0.99969	1.00024
9	4.99740	4.99962	9.99835	9.99918	0.99969	1.00024
10	4.99740	4.99962	9.99835	9.99918	0.99969	1.00024

- Apache Mahout

Command Line:

mahout org.apache.mahout.clustering.syntheticcontrol.kmeans.job –i /kmeans/inputfile/synthetic.control.data –k 3 –t1 3 –t2 6 –x 10 –output /kmeans/outputfile

mahout clusterdump –i /kmeans/outputfile/clusters-3-final –pointDir /kmeans/outputfile/clusteredPoints –output /home/hdfs/result.txt

Iteration number until convergency: 10
Average running time for one iteration:40s
Total running time:400s
Converge to correct cluster centroids:Yes

7. Conclusion and Summary

In the step of deployment, we must pay much attention to some details. In most circumstances, errors are keeping occurring. For Cloudera Manager or Sandbox, network configuration and permission control are very important. A clear understanding of Local files/HDFS files/Remote hosts is needed.

In the second version of Mapreduce code, although we realized our goal to let the program iterate automatically by passing coordinates and members to reducer, the computation speed is too slow (nearly 20 minutes for one iteration) to make the method for practical use. As a comparison, the first method runs and converges fast even if human work should be involved to control the iteration. In conclusion,

although the two Mapreduce implementations both converge, the first method is preferable.

Meanwhile, although we successfully deployed our multi-node Hadoop system, it is very unstable that we often lost the connection to cloudera-scm-server (the error log shows connection failure and a pid file already exists). We tried our best to search the solution to this issue. However, it still remained very unstable that we sometimes couldn't even run a complete Mapreduce clustering job before we lost the connection finally. Even though we couldn't get a convinced result, we are confident that we could run the job on multi-node once this problem is solved since multi-node routine is nearly the same as single node.

For Mahout, it reached to convergency faster compared with others. However, it's like a black-box that we couldn't access or modify the core algorithm implementation inside. What we can do is following the Mahout references or Docs to write commands. We put a lot of efforts on the real coding implementation for K-Means as described above. Compared to the performance of Mahout build-in function, our codes allow us to manipulate any data operation or processing in the intermediate steps.