Project Final Report

Jiachi Liu & Peili Cao

December 5, 2014

1 Introduction

Music recommendation is a core feature for current online music applications. In this project, we implemented two data mining algorithms to recommend music to users. K-Means algorithm is used to recommend similar musics whereas frequent itemsets mining will find the combination of songs that most user will listen together.

1.1 K-Means

K-Means algorithm is a popular clustering algorithm aims to partition n data points into K clusters which each data point belongs to the cluster with the nearest mean. And it is also a unsupervised machine learning algorithm since the data points usually without labels. In this project, a Distributed version of KMeans algorithm is implemented with MapReduce to partition the millionSongs dataset into different clusters. Thus in future, given a new song data point, we can find the similar songs based on the clusters and recommend it to the user.

2 Data

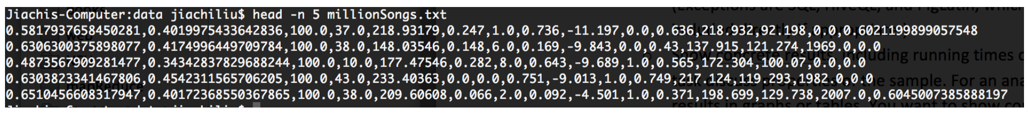
2.1 K-means

For K-Means major task, we are using MillionSongs dataset to divide songs into K different clusters. Since million songs dataset contains a lot of meta data of music, such as mode, key, segments, we are using a subset of those features to reduce the size of data, which contains 15 features. The features are: artist familiarity, artist hotness, number of similar artists, number of artist terms, duration, end of fade in, key, key confidence, loudness, mode, mode confidence, start of fade out, temp, year, song hotness. Figure 1 shows some lines of data as example.

Since the entire data set is very large, we are using a subset of million songs dataset which contains 10000 songs in H5 format files. This subset can be found at

http://labrosa.ee.columbia.edu/millionsong/pages/getting-dataset

In order to extract features from H5 file, we wrote a small java program to read the data from this format. The source code is under extractFeatures package and hdf5_getters.java is the official library to parse the H5 file.



```
Jiachis-Computer:data jiachiliu$ head -n 5 millionSongs.txt
0.5817937658450281,0.4019975433642836,100.0,37.0,218.93179,0.247,1.0,0.736,-11.197,0.0,0.636,218.932,92.198,0.0,0.6021199899057548
0.6306300375898077,0.4174996449709784,100.0,38.0,148.03546,0.148,6.0,0.169,-9.843,0.0,0.43,137.915,121.274,1969.0,0.0
0.487356790928147 7,0.3434283782968824 4,100.0,10.0,177.47546,0.282,8.0,0.643,-9.689,1.0,0.565,172.304,100.07,0.0,0.0
0.6303823341467806,0.4542311565706205,100.0,43.0,233.40363,0.0,0.0,0.751,-9.013,1.0,0.749,217.124,119.293,1982.0,0.0
0.6510456608317947,0.40172368550367865,100.0,38.0,209.60608,0.066,2.0,0.092,-4.501,1.0,0.371,198.699,129.738,2007.0,0.6045007385888197
```

Figure 1: Million Songs data example

3 Technical discussion

3.1 K-Means

For K-Means task, we have implemented both distributed and local version of KMeans algorithm. Our goal is to split the given songs into K different clusters based on selected features and get the centroid for each cluster. For local algorithm, parallel computing is applied to explore different K at same time. Both distributed and local algorithm will output the final centroids into file which can be used to assign data point to corresponding clusters.

3.1.1 Distributed KMeans

The Mapper of KMeans Algorithm will read the current K centroids from HDFS file and save them into the memory in setup function. In map function call, for each line of data, we assign it to the centroid that has minimum distance. The distance is calculated as Euclidean distance between the data point and centroids. The mapper then emits the centroid id as a key and the data point as value.

```
1: function MAP(key, data)
2:     $minDistance \leftarrow Infinity$
3:     $centroidId = -1$
4:     for each centroid $c$ in centroids do
5:         $dist \leftarrow distance(c, data)$
6:         if $dist < minDistance$ then
7:             $minDistance \leftarrow dist$
8:             $centroidId = c.id$
9:         end if
```

```
10:     end for
11:     $emit(centroidId, data)$
12: end function
```

The reducer will update the centroid vector by calculating the average vector among the input data point list. And emit the new centroid to output file.

```
1: function REDUCE(cid, $[d_1, d_2 ...., d_n]$)
2:     $sumVector \leftarrow 0$
3:     $count \leftarrow 0$
4:     for each data $d$ in input list do
5:         $sumVector+ = d$
6:         $count + +$
7:     end for
8:     $newCentroid = sumVector/count$
9:     $emit(newCentroid, Null)$
10: end function
```

The Driver class will repeatedly create map reduce job for each iteration of KMeans. And it will set the number of reducers as same as K. It will first initialize centroids based on input K before start KMeans algorithm, and then start create jobs for each iteration to get new computed centroids. After that, it will copy the output file to currentCentroid folder so the mapper can read current centroids from it. Also, it will delete the output file in order to avoid exceptions on map reduce job. And to stop the iterations, it will read and compute the sum of distance of all centroids between two iterations and determines whether to stop.

The following pseudo code shows the process of driver program:

```
1: function TRAIN(Input arguments)
2:     read the dataset file path, current centroid file path, and new centroid file path
       from program input arguments.
3:     read K from input arguments
4:     init K centroids from dataset and write them to new centroid file.
5:     while Not converge do
6:         copy the new centroid file to current centroid file
7:         delete the new centroid file
8:         run map reduce job on current centroid file and input dataset to get new
           centroid file
9:     end while
10: end function
```

The following pseudo code shows the logic on determining converge. This program will read new centroids and current centroids from S3 that output by map reduce jobs.

```
1: function ISCONVERGE(oldCentroids, newCentroids)
2:     $dist \leftarrow 0$
```

```
3:     for each old centroids o, and new centroids n do
4:         $dist+ = distance(o, n)$
5:     end for
6: return $dist <= threshold$
7: end function
```

The source code related to Distributed KMeans algorithm is shows as following:

- KMeansMapReduce.java the map reduce job
- KMeansController.java the driver program
- DenseVector.java the vector class to compute the distance
- FileReadWriteUtil.java helper class for file read, write, delete and copy

3.1.2 Distributed KMeans - Running Result

Table 1 shows the running time of Distributed KMeans on AWS for different K. We are using two configurations, one is 5 small machine and one is 10 small machines.

Table 1: K-Means Runtime Table

K	# of Iterations	5 workers	10 wokers
2	2	4m01s	3m32s
3	11	21m43s	20m41s
5	18	38m32s	34m38s
10	32	1h09m12s	1h00m33s
20	58	2h30m41s	2h00m07s
30	76	3h52m58s	2h56m49s

As the table shows, under same configuration, with the increasing number of K, the total number iterations is also increased and thus more time is needed to finish the algorithm. However, when we double the worker machines, the running time is not decreased to half as we expected. Based on the algorithm introduced above, the number of reducers is same as the K values. When K is small, not all workers is computing the new centroid during reduce phase. Thus the running time is almost same when K is 2,3,5,10. But when K becomes larger, for instance 30, it is possible for all reducers participated in computing thus less time is needed.

3.1.3 Local KMeans with MapReduce

We also implemented local version of KMeans algorithm and running multiple K-s with mapreduce. The basic idea is that each mapper running the local version KMeans to get the final centroids. And NLineInputFormat is used in order to let different mapper

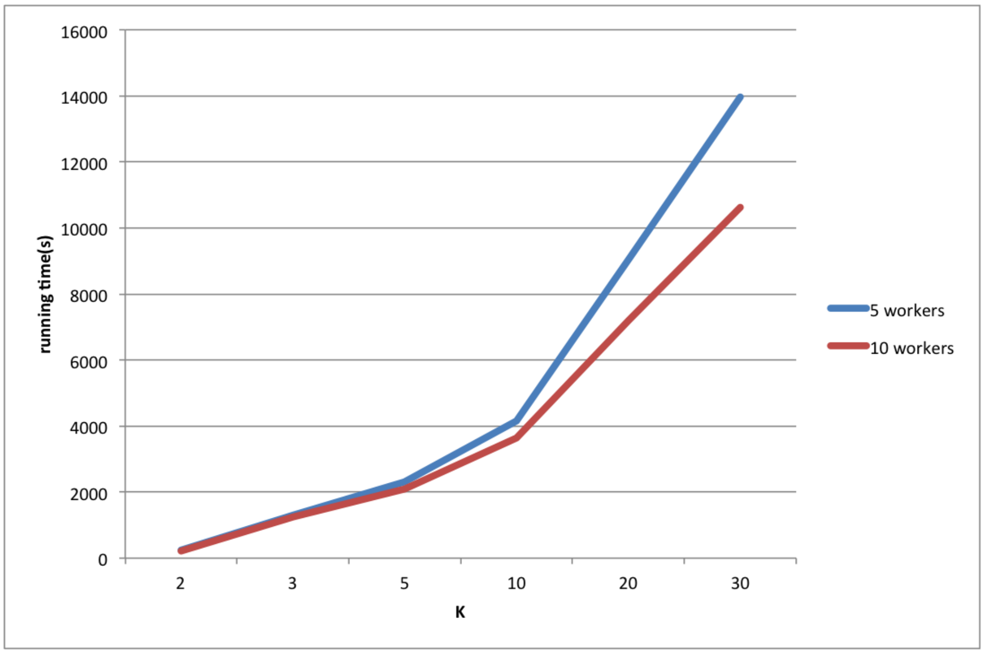


Figure 2: Distributed KMeans - Running Time

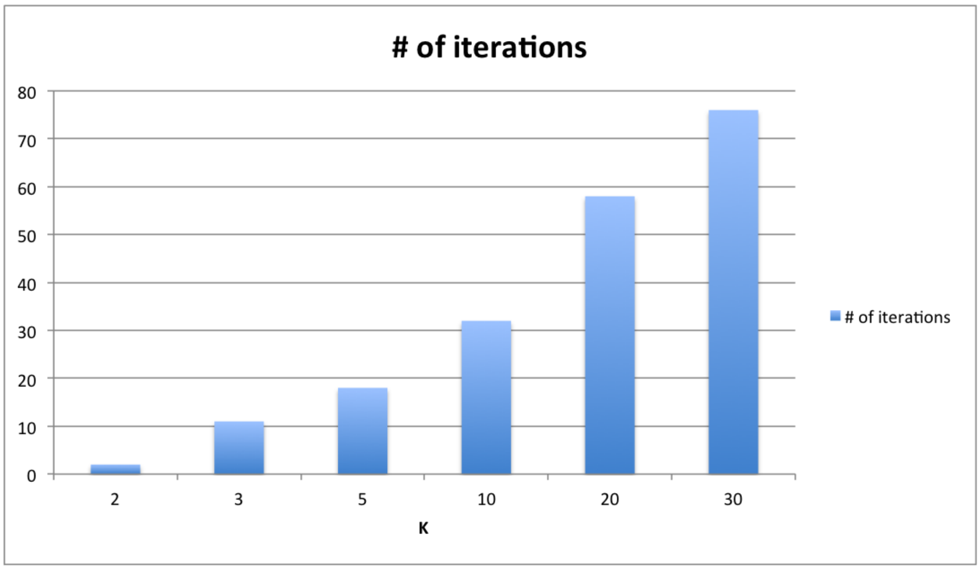


Figure 3: KMeans - Number of Iterations with Different K

handle different K in input file. For this problem, the input data is a file that each line is a single number that represents a value of K. The dataset will be put in Distributed Cache and load into memory during the map phase, since the size of dataset can be fit into memory.

The following pseudo code shows the local KMeans algorithm:

```
function LOCAL KMEANS(input dataset, K)
    init K centroids from input dataset
    while True do
        for each data t in dataset do
            find the closest centroid c from centroids
            assign t to cluster respect to c
        end for
        for each cluster cl do
            calculate the new centroid from data point that assigned to cl
            add new centroid to new centroid list
        end for
        if Converge(newCentroids, currentCentroids) then
            break
        end if
    end while
end function
```

Mapper pseudo code is shows as following. The input value is the parameter K which is also used as output key to distinguish different centroids from mapper output.

```
function MAP(key, value)
    read dataset from distributed cache to memory
    run local KMeans algorithm on dataset and get the centroids
    for for each centroid c in centroids do
        emit(value, c)
    end for
end function
```

Reducer pseudo code is shows as following. The reducer will collect all centroids from the current K and emit it to output file.

```
function REDUCE(key, values)
    read dataset from distributed cache to memory
    run local KMeans algorithm on dataset
    for each value in values do
        emit(key, value)
    end for
end function
```

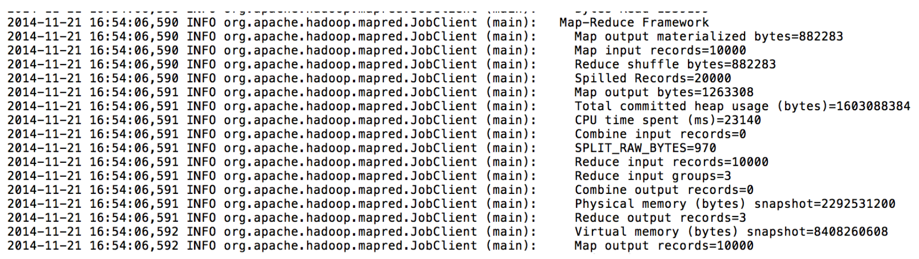
3.1.4 Local KMeans - Running Result

We also apply two configurations(5 workers and 10 workers) for this problem and set the converge condition as same as distributed version. The input file contains 6 different K, which is 2,3,5,10,20,30. For 5 workers, it took 2min31s to finished the job. For 10 workers, it took 2min37s. The running time is not change accordingly with number of workers. Due to the NLineInputFormat, the number of mappers is fixed during the execution thus increasing the number of machines will not have significant different on running time when input parameters is small.

3.1.5 Local KMeans v.s. Distributed KMeans

Now comparing with distributed version, local version took much smaller time to finished the job. The reason is that:

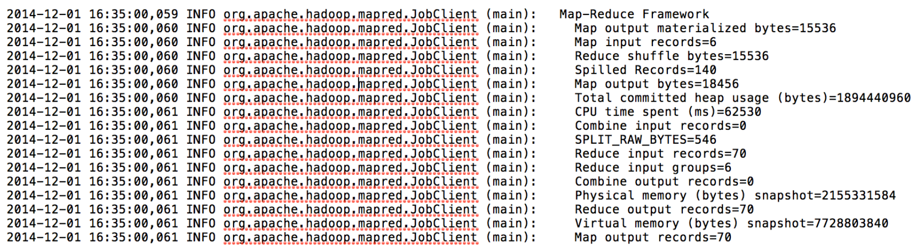
- Distributed version KMeans repeatedly setup new job for each iteration. On AWS, job setup takes a lot of time which heavily extend the total execution time. Where the local version only have one job to setup.
- The local version only send final centroids from mapper to reducer one time, whereas distributed version needs to send the entire dataset in each iteration. The data transition between the mapper and reducer also takes time. This can be verified from the log file. Figure 4 shows one iteration of Distributed KMeans when $K = 3$, we can see that the total number of mapper output is 10000 which is the entire dataset. And Figure 5 shows the local version of KMeans mapper output is 70, which is the sum of all different numbers of K in NLineInput file.
- Distributed version also needs to copy and read centroids from S3 file, where local version finished everything in memory.



```
2014-11-21 16:54:06,590 INFO org.apache.hadoop.mapred.JobClient (main):   Map-Reduce Framework
2014-11-21 16:54:06,590 INFO org.apache.hadoop.mapred.JobClient (main):     Map output materialized bytes=882283
2014-11-21 16:54:06,590 INFO org.apache.hadoop.mapred.JobClient (main):     Map input records=10000
2014-11-21 16:54:06,590 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce shuffle bytes=882283
2014-11-21 16:54:06,590 INFO org.apache.hadoop.mapred.JobClient (main):     Spilled Records=20000
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Map output bytes=1263308
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Total committed heap usage (bytes)=1603088384
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     CPU time spent (ms)=23140
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Combine input records=0
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     SPLIT_RAW_BYTES=970
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce input records=10000
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce input groups=3
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Combine output records=0
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Physical memory (bytes) snapshot=2292531200
2014-11-21 16:54:06,591 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce output records=3
2014-11-21 16:54:06,592 INFO org.apache.hadoop.mapred.JobClient (main):     Virtual memory (bytes) snapshot=8408260608
2014-11-21 16:54:06,592 INFO org.apache.hadoop.mapred.JobClient (main):     Map output records=10000
```

Figure 4: Syslog for Distributed Version KMeans - K=3

The source code for local version is also in DistributedKMeans folder, which contains LocalKMeans.java, LocalKMeansController.java and KMeansLocalMapReduce.java.



```
2014-12-01 16:35:00,059 INFO org.apache.hadoop.mapred.JobClient (main):   Map-Reduce Framework
2014-12-01 16:35:00,060 INFO org.apache.hadoop.mapred.JobClient (main):     Map output materialized bytes=15536
2014-12-01 16:35:00,060 INFO org.apache.hadoop.mapred.JobClient (main):     Map input records=6
2014-12-01 16:35:00,060 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce shuffle bytes=15536
2014-12-01 16:35:00,060 INFO org.apache.hadoop.mapred.JobClient (main):     Spilled Records=140
2014-12-01 16:35:00,060 INFO org.apache.hadoop.mapred.JobClient (main):     Map output bytes=18456
2014-12-01 16:35:00,060 INFO org.apache.hadoop.mapred.JobClient (main):     Total committed heap usage (bytes)=1894440960
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     CPU time spent (ms)=62530
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Combine input records=0
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     SPLIT_RAW_BYTES=546
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce input records=70
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce input groups=6
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Combine output records=0
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Physical memory (bytes) snapshot=2155331584
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Reduce output records=70
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Virtual memory (bytes) snapshot=7728803840
2014-12-01 16:35:00,061 INFO org.apache.hadoop.mapred.JobClient (main):     Map output records=70
```

Figure 5: Syslog for Local Version KMeans

4 Conclusion

4.1 K-Means

Local version of K-Means is much faster than distributed version in our experiments. However, our implementation of local version is loading the dataset into memory which may not feasible in practical problems. For distributed version, the job setup takes a lot of times compare with the actual computing time, a better way should be found to avoid repeatedly create jobs in each iteration.

Another issue for distributed K-Means is that we need to save the generated centroids to some where and also keep tracking the old centroid generated in last iteration. In our implementation, we copy the new centroids file to other place and read it back when checking the converge condition, which also requires additional time.

In order to compare the performance of two implementation, we fixed the initial centroids and converge condition to ensure the number iteration is same in both distributed version and local version. However, practically, the initial centroids should be carefully determined since KMeans algorithm is very sensitive with the start points. And in order to reduce the number of iterations, the initial centers should be as far as possible to each other. In future work, a carefully designed algorithm, such as Furthest First Traversal, should be applied to find the optimal initial centers.