Project Progress Report

Jiachi Liu & Peili Cao

December 2, 2014

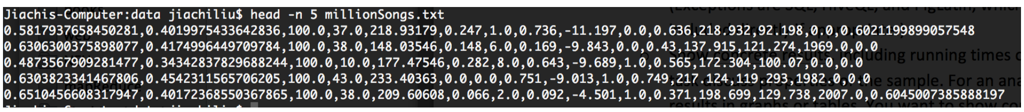
1 Data Description

1.1 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.

1.2 Frequent itemset mining

For Frequent itemset mining, we are using Taste Profile subset from MillionSongs dataset. This dataset contains two columns: the first column represents user and the second column represents song. Table 1 shows the statistics of this dataset.



```
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
0.4873567909281477,0.34342837829688244,100.0,10.0,177.47546,0.282,8.0,0.643,-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

Table 1: Statistics of Taste Profile Subset

Unique Users	Unique MSD songs	User-Song-Playcounts Triplets
1,019,318	384,546	48,373,586

2 Task Progress

2.1 K-Means

For K-Means task, we have implemented the distributed version of KMeans and run it on amazon AWS among 10,000 songs.

2.1.1 Pseudo Code

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.

```
function MAP(key, data)
    minDistance ← Infinity
    centroidId = −1
    for each centroid c in centroids do
        dist ← distance(c, data)
        if dist ¡ minDistance then
            minDistance ← dist
            centroidId = c.id
        end if
    end for
    emit(centroidId, data)
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.

```
function REDUCE(cid, [d_1, d_2...., d_n])
    sumVector ← 0
    count ← 0
    for each data d in input list do
        sumVector+ = d
        count + +
    end for
    newCentroid = sumVector/count
    emit(newCentroid, Null)
end function
```

The Driver class will repeatedly create map reduce job for each iteration of KMeans. 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.

function isConverge(oldCentroids, newCentroids)
    $dist \leftarrow 0$
    for each old centroids o, and new centroids n **do**
        $dist+ = distance(o, n)$
    end for
return $dist <= threshold$
end function

2.1.2 Running Result

Table 2 shows the running time on AWS for different K.

Table 2: K-Means Runtime Table

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

2.2 Frequent itemset mining

2.2.1 Preprocessing data

Since the user-id and song-id provided in original data file are long string, we first replace the data with new sequential id. For user, id starts from 0 to 1,019,317. For song, id starts from 0 to 384,546. After reformatting ids, we need to generate transactions to perform Apriori Algorithm. Figure 2 shows an example of the transactions. Row i represents the list of songs that listened by $user_i$. The lists are ordered.

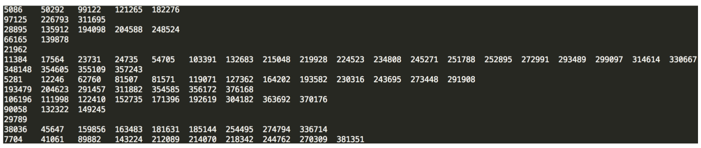


Figure 2: Transaction Example

2.2.2 Pseudo Code

```
itemset_1 ← (Size1_Frequent_Items)
for k = 1, Itemset_k notempty; k++ do
    candidates_{k+1} = itemset_k sort_join itemset_1
    for each transaction t in transactions do
        each map will have a copy of candidates_{k+1}
        emit(candidates_{k+1}, count)
    end for
    //In Reducer
    for each candidate in candidates_{k+1} do
        if count >= min_support then emit(candidates)
        end if
    end for
    itemset_{k+1} = reduce_outputs
end for
```

2.2.3 Generate Size K Frequent Item Candidate

To generate size K($K >= 2$) frequent item candidates, we join size $K - 1$ frequent items with size 1 frequent items. In order to improve efficiency of algorithm, we make sure that ids of size $K - 1$ frequent items are in order. There will be two mappers. One is for reading size 1 frequent items, the other is for reading size $K - 1$ frequent items. The key of two mappers are ("*dummy*", *tag*). And Partitioner and GroupComparator will only consider "*dummy*". Pseudo Code

```
function MAP(offset,value)
    emit("dummy", tag, value)
end function
```

In Reducer, it will store records into two separate lists based on tag and then join the two lists. Since the two lists are all ordered, there will be no duplicates candidates.

```
function REDUCE("dummy", tag, listofitemsets)
    for each item in list do
        TagAList.add(itemfromA)
        TagBList.add(itemfromB)
    end for
    for each item in A do
        for each item in B do
            candidate = join(A.item, B.item)
            emit(candidate)
        end for
    end for
```

end function

2.2.4 Analysis

We analysis the number of listening users for each song ids. Table 3 shows mean, max, min for all songs. Figure 3 shows the distribution of the number of listening users for each song. We split songid into 20 bins. X coordinates means the number of bin. Y coordinates means the number of songs located in that bin. The scope of first bin is (1,5525)

Table 3: Statistics of Taste Profile Subset

Mean	Max	Min
125.79	110479	1

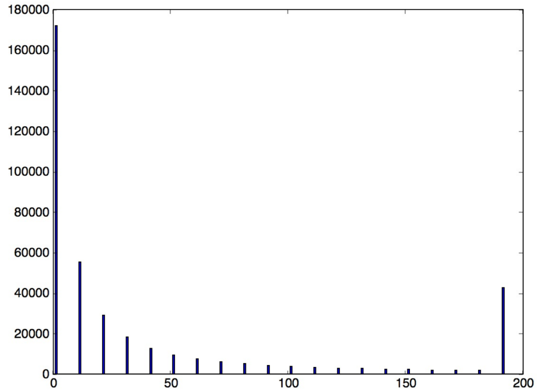


Figure 3: Distribution of # of listening users for songs

3 Remain Work

- For K-Means problem, we will need a helper task to calculate the Mean Root Square Error between the centroid and songs in same cluster to measuring the quality of the algorithm. And we will also implement a local version K-Means algorithm to compare the performance with distributed version.
- For Frequent Itemsets, we will continue to implement Apriori Algorithm. Till now, we've implemented generating size1-3 frequent itemset. After implementing, we will run it on AWS. We also notice that there are transactions that can be eliminate when there is no frequent itemset in it, but doing this may make code more complicate.