A7 - Clustering

Aditya, Vineet

November 3, 2017

System Information

The spark job with full data was run on:

1. AWS EC2 instance in local mode: >

- Ec2-Instance type - m4.2x.large
- vCPU: 4 RAM: 16GB Hard Disk: 50GB Storage: EBS-only
- Processors for all the above machine was 64 bit 2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors.
- Each vCPU is a hyperthread of an Intel Xeon core

2. Ubuntu VM on Windows 10 >

- VM-Ram GB ,Harddisk - 35GB
- Laptop -Ram - 8GB ,Hardisk 250 GB SSD

System Design:

K Means:

Agglomerative Clustering:

This involves combining the closest clusters into one cluster at each iteration. It is a bottom up approach where we start with each point as an individual cluster and move towards building 3 clusters. Then we stop. Let's take an example for song loudness: In the beginning, we have individual points (songId) and their centroids (loudness). What we want to do is sort the rdd and subtract consecutive rows, row1 => row2-row1, row2 => row3-row2 and so on. Then sort the differences and obtain the least among them. The idea is that the smallest difference will be among the differences of consecutive elements. However, as we cannot access the successive row we decided to take the following approach to produce a similar effect: We sort the input rdd, index it with zipWithIndex, make the index the key and create a copy which is shifted upwards. This we do by removing the first element of the copyRdd. We then do a join, get the difference of the 2 values per key and sort the resulting RDD by values. This gives us the closest clusters. Then it is just a matter of taking their average, removing the individual points from clusterRdd and adding the new point to it. Repeat this process till we have 3 clusters. This is still taking a lot of time even though it is much better than taking a cartesian product of all songIds, subtracting their loudness and sorting it to obtain the closest clusters.

Subproblem 2:

For this we decided to compute the KMedian instead of KMean. As we cannot compute the distance to an arbitrary mean, the next centroid needs to be one of our data points. Hence, we decided to use median to obtain the center point of the given cluster. Median is also comparatively less affected by outliers. To compute the KMedian we started with 3 rdds, table1 containing the list of all artists, table2 containg the list of top 30 popular artists and the graph containing a key pair (artist1, artist2) and the value being the distance between them (commonality).

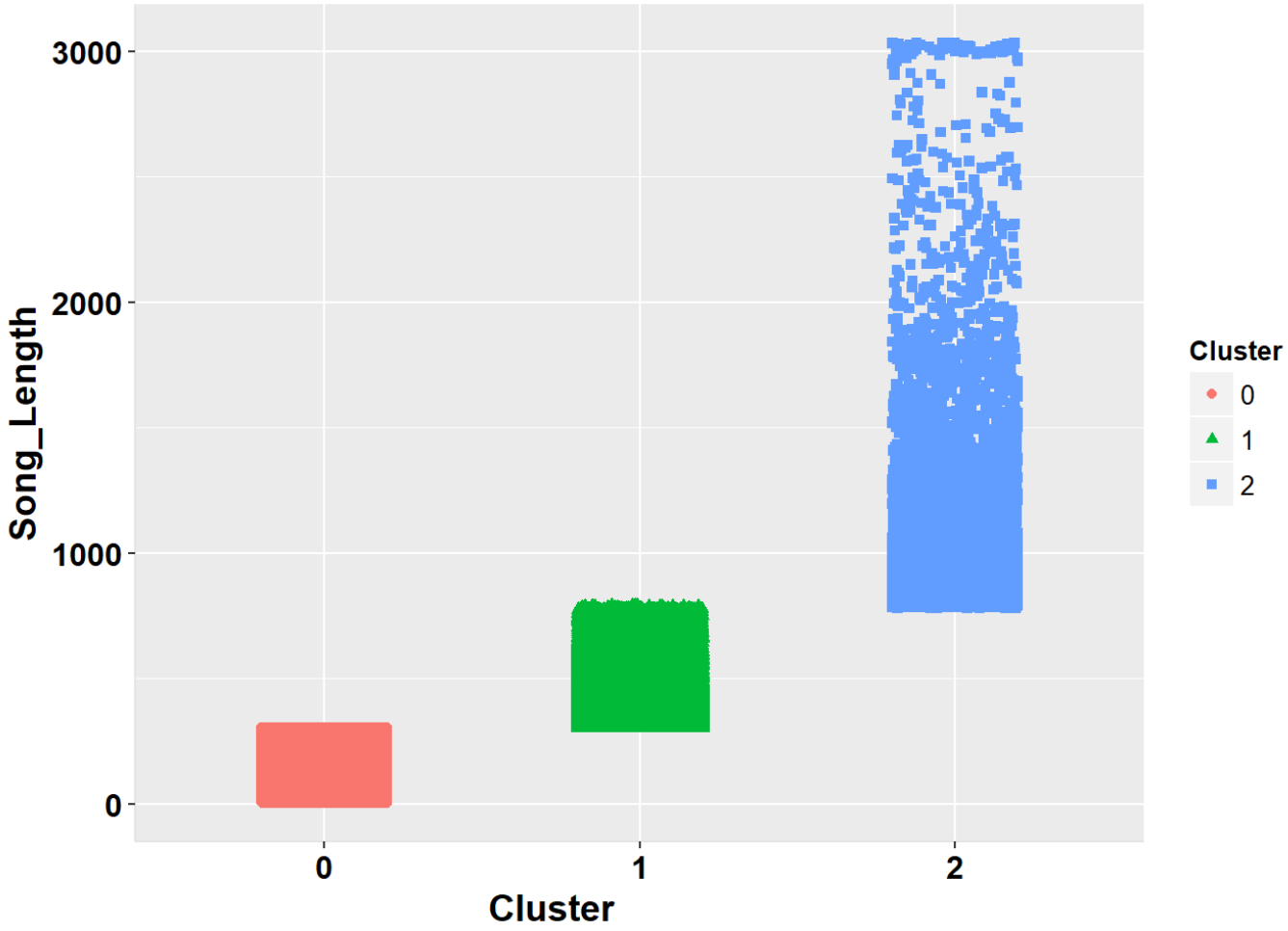
We performed a cartesian product of table1 and table2 to obtain a list of all possible (artist, centroid) pairs. We then joined it with graph to reduce the original list to contain only those artist centroid pairs that have any commonality. Rearranged the form (artist, centroid), weight => (artist, (centroid,weight)) A reduce by key gave the maximum of all centroids. A point would be considered closer to a centroid if it had greater commonality and hence a maximum needed to be found in this case. We then group all the values by centroids, and obtained a list of (artist,weight) against each centroid. Sorted the list and accessed middle element to obtain the new centroid. Repeated this process for 10 iterations.

Subproblem 1: Clustering

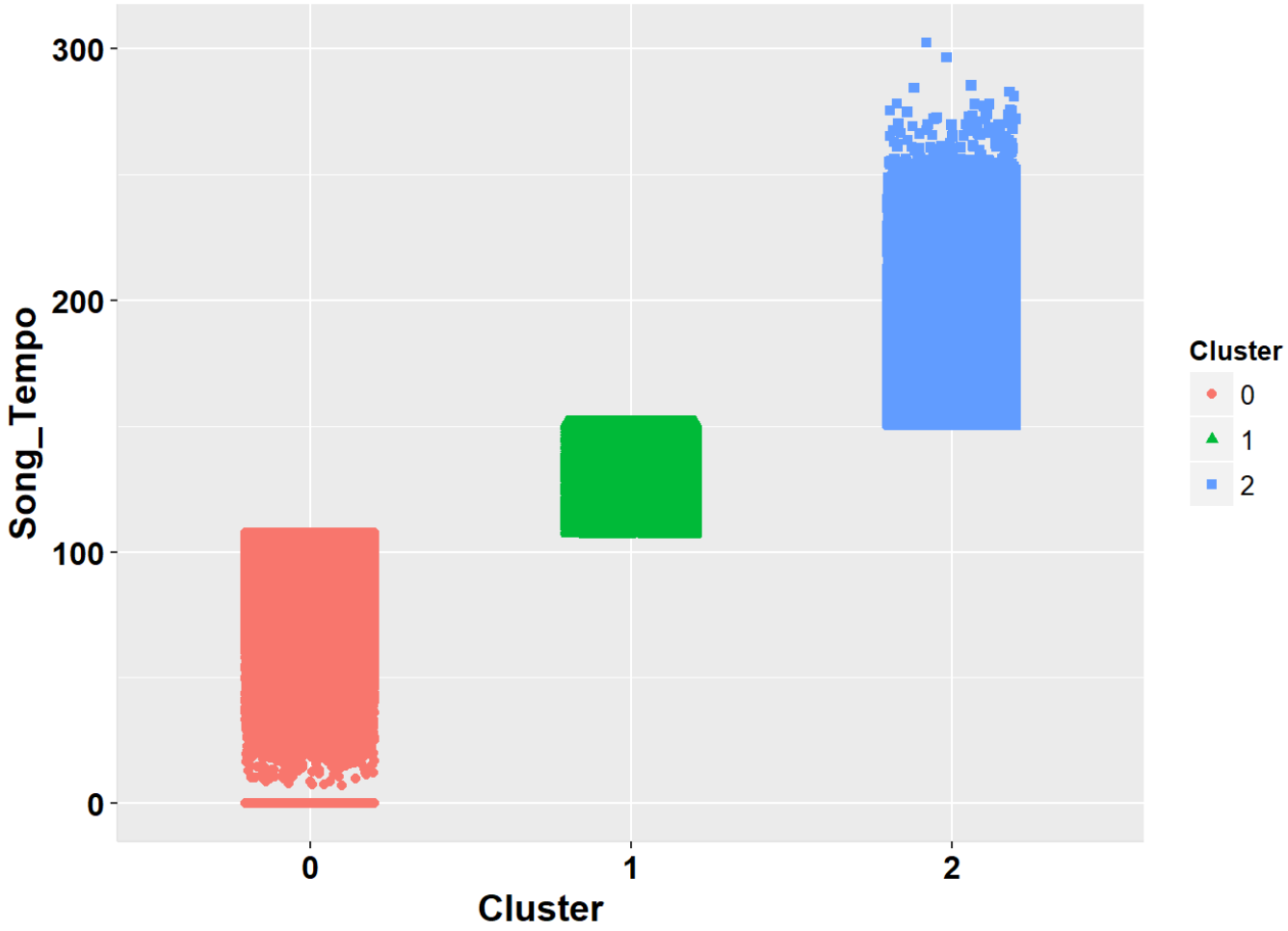
Using Spark, perform hierarchical agglomerative clustering and k-means clustering as follows:

> - Fuzzy loudness: cluster songs into quiet, medium, and loud
> - Fuzzy length: cluster songs into short, medium, and long
> - Fuzzy tempo: cluster songs into slow, medium, and fast
> - Fuzzy hotness: cluster songs into cool, mild, and hot based on song hotness
> - Combined hotness: cluster songs into cool, mild, and hot based on two dimensions: artist hotness, and song hotness

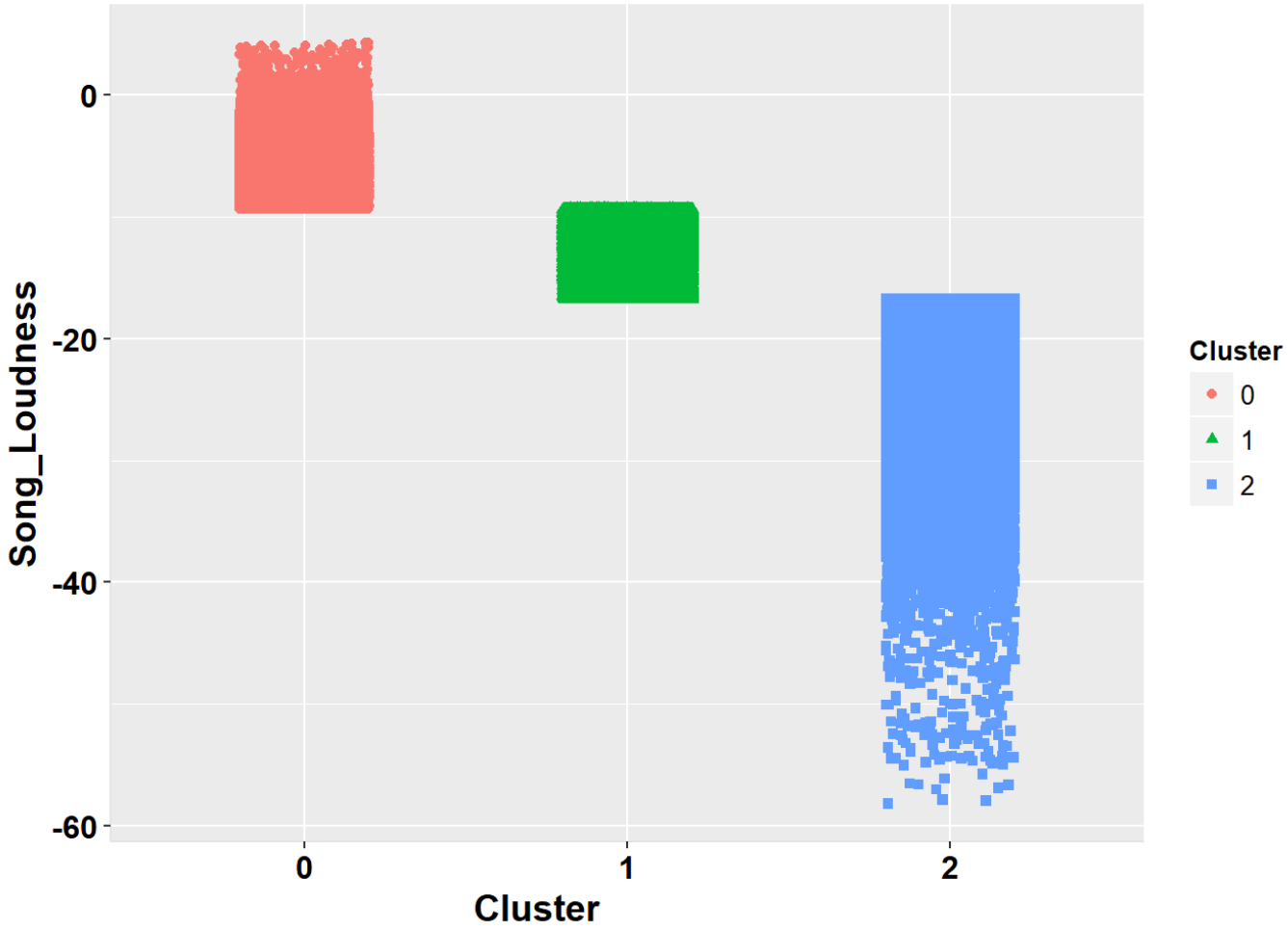
K-Means Clustering - Local Runs



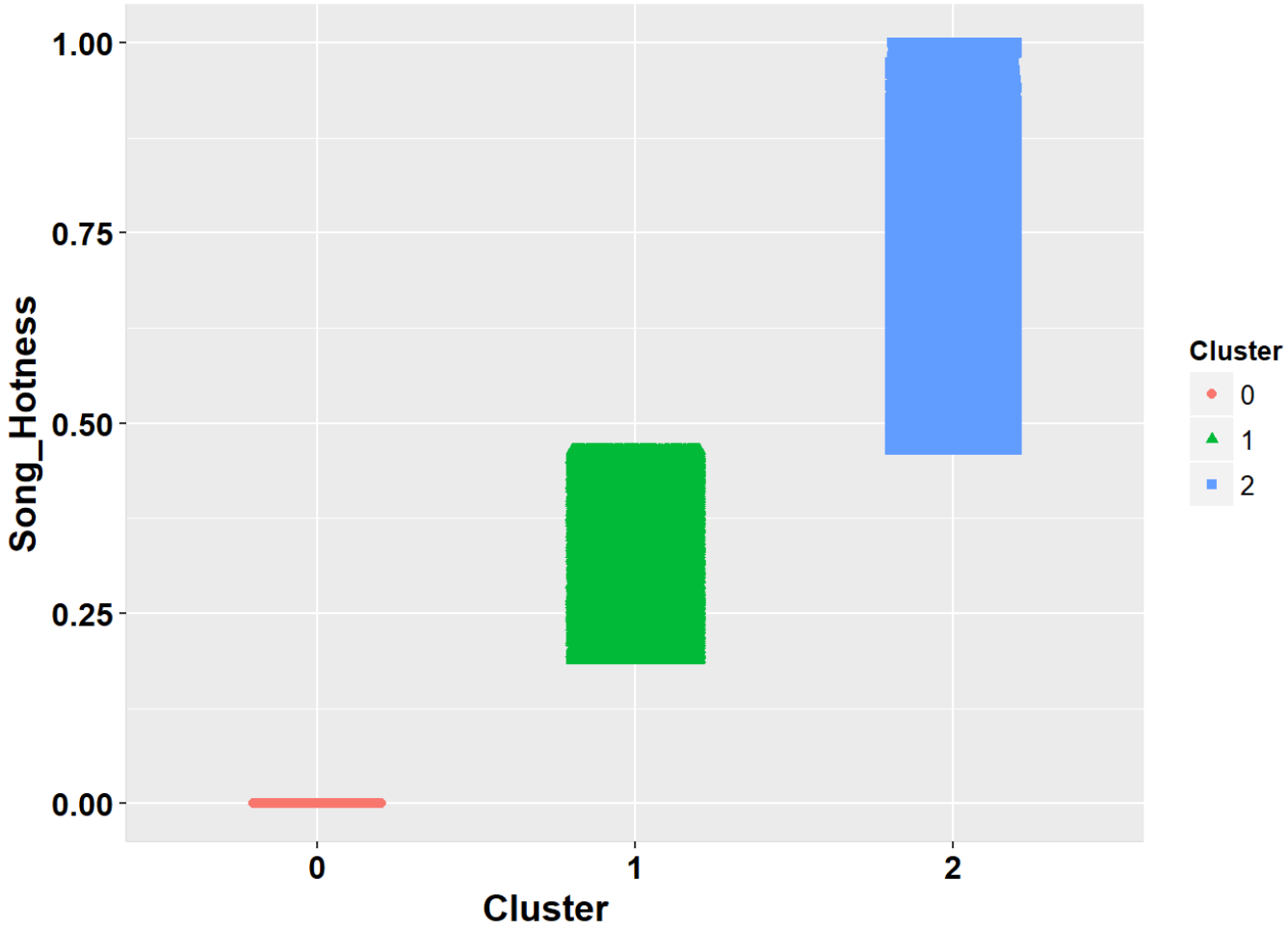
The graph above has 3 clusters on the x-axis and song duration on the y axis. The duration ranges from 0 to 3000. Cluster 0 is for short songs, Cluster 1 is for medium songs and Cluster 2 is for long songs. There are few songs with song duration greater than 1500.



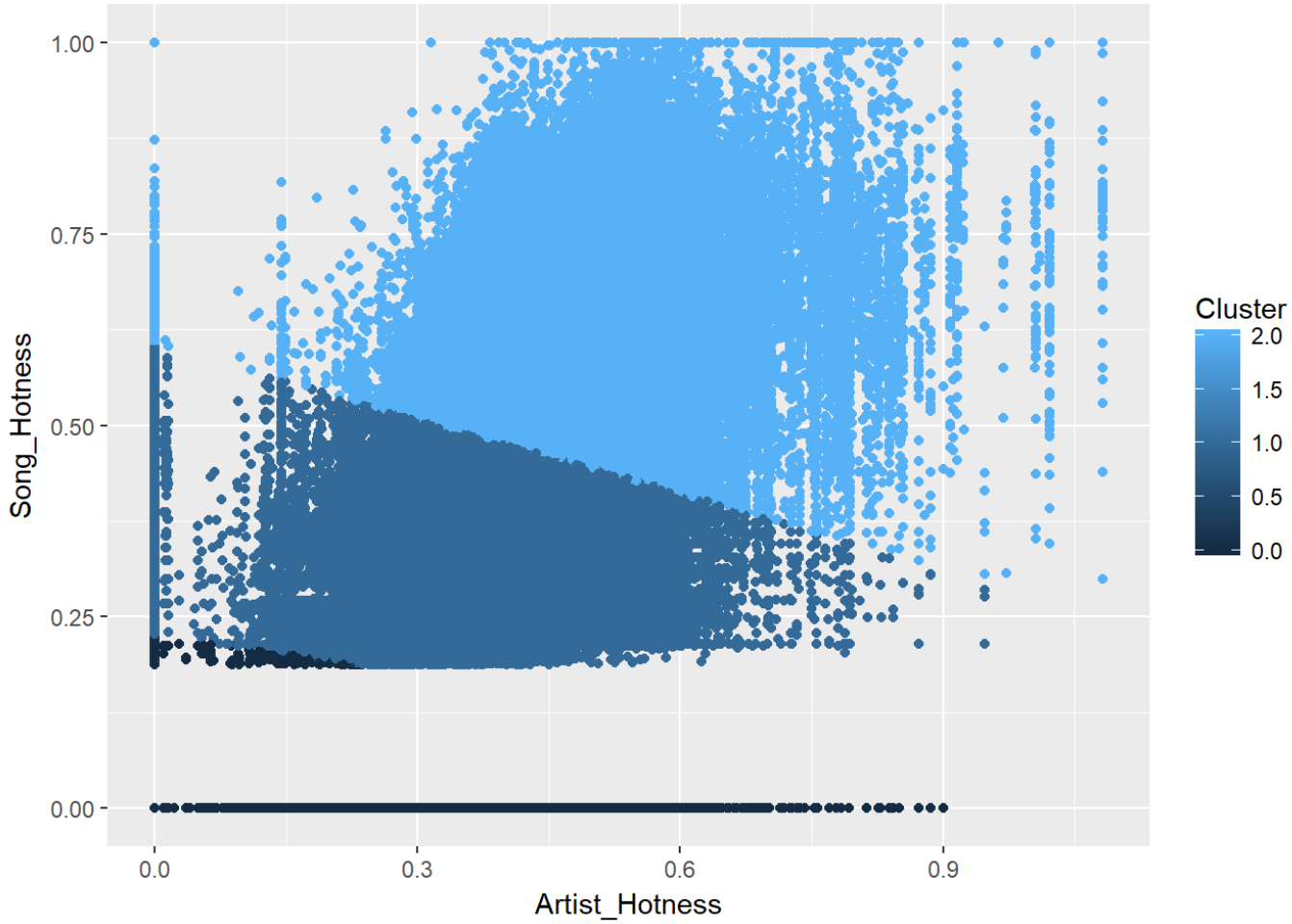
The graph above has 3 clusters on the x-axis and song tempo on the y axis. The tempo ranges from 0 to 300. Cluster 0 is for slow songs, Cluster 1 is for medium songs and Cluster 2 is for fast songs. The tempo of songs is densely populated from 50 to 250.



The graph above has 3 clusters on the x-axis and song loudness on the y axis. The loudness ranges from 5 to -60. Cluster 0 is for loud songs, Cluster 1 is for medium songs and Cluster 2 is for quiet songs. There are few songs with loudness less than -40 and they are densely populated from -40 to 0.

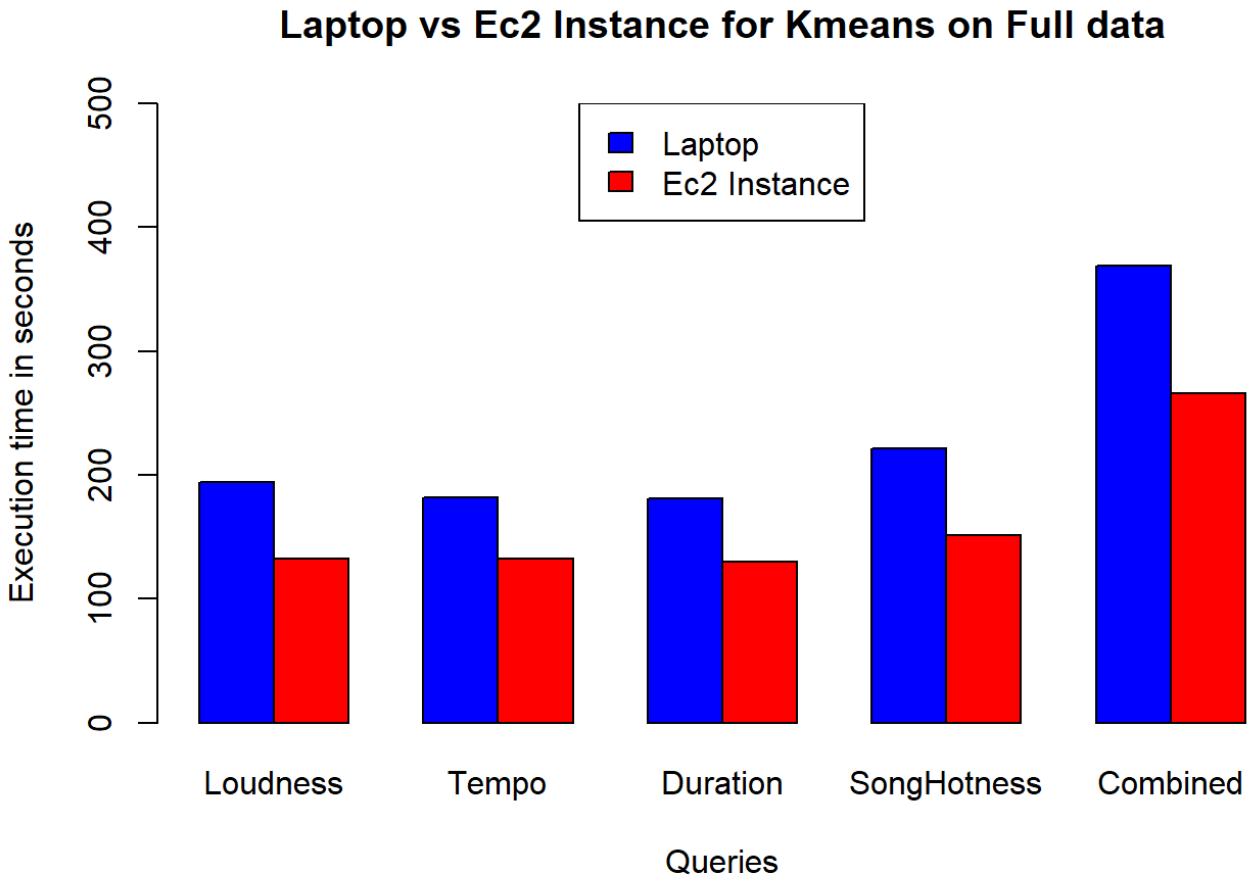


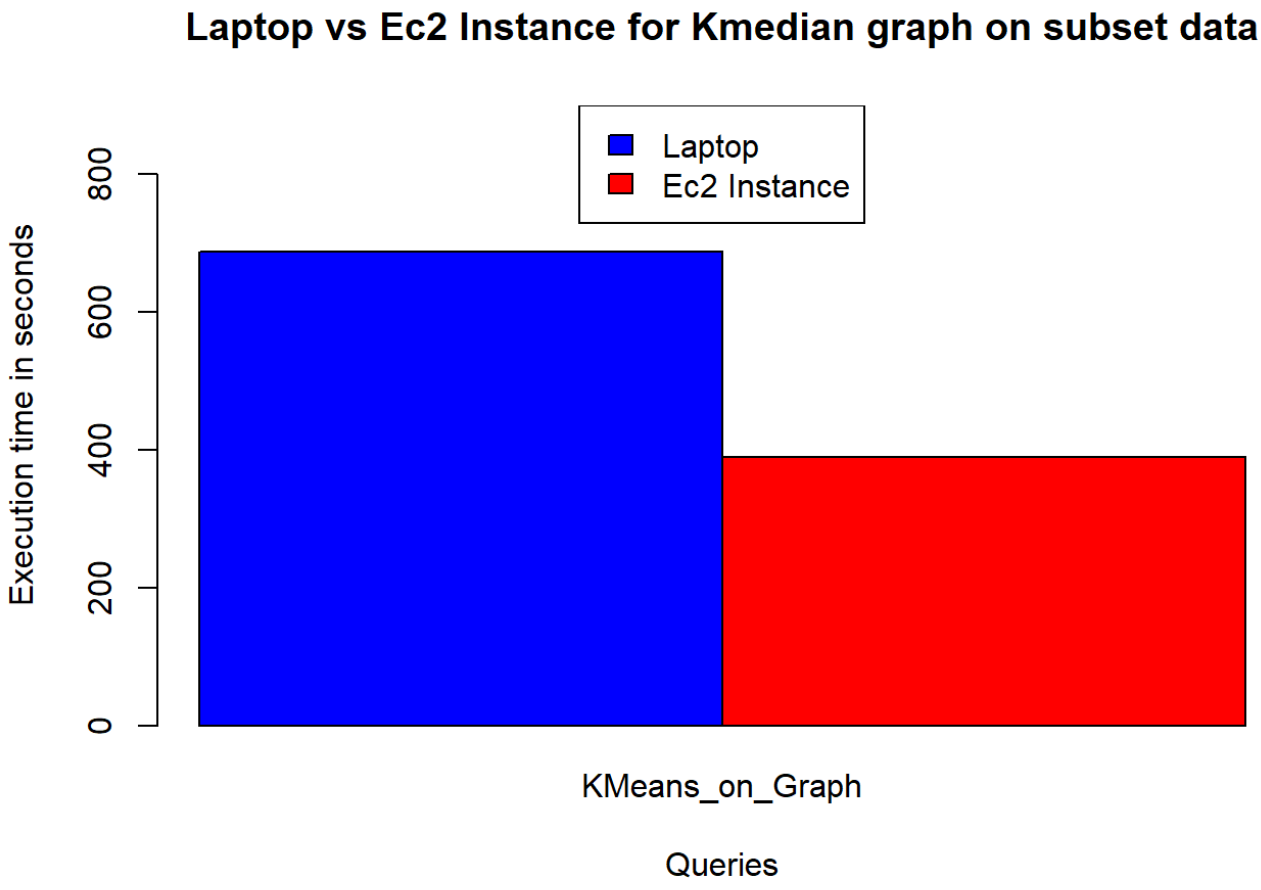
The graph above has 3 clusters on the x-axis and song hotness on the y axis. The hotness ranges from 0 to 1. Cluster 0 is for cool songs, Cluster 1 is for mild songs and Cluster 2 is for hot songs. There are very few songs with hotness near 0 and they are densely populated from 0.25 to 1.



The graph above has Artist_Hotness on the x-axis and Song_Hotness on the y-axis. Both ranges are from 0 to 1. It is observed that most songs are mild songs and are produced by mild artists.

Performance





The AWS run is actually not on a cluster you can think of it as a better laptop , because we ran it on local mode.As expected a machine with higher ram and more vcores performs slightly better.

Challenges faced:

Realized (to a certain extent) when to use cartesian product. While doing agglomerative clustering we were trying to do a cartesian product between all clusters and then sorting it to obtain the closest clusters. It seemed like a good idea when we were doing it on dummy data however when we tried it on the subset and got stuck for a while on the cartesian product line, we ran the numbers in our head and realized that we needed a new plan. However, for KMedian as we had 30 centroids and less than 50,000 unique artists a cartesian product was producing a reasonable number of rows. Accessing successive row while keeping the whole thing parallel. We had to create a whole new shifted rdd index it and perform a join just to access the next row in an rdd.

However the K-Means Clustering on Graph was only successful on the smaller dataset and and not on the large dataset

Conclusion

Although algorithms like KMean and Agglomerative are written in a couple of lines of code, it takes hours to come up with the logic behind it. Had to dig deep into scala and spark syntax to obtain transformations like zipWithIndex. Realised the expense of joins and cartesian product and when to use them. Regarding the data set we concluded that there are very few songs with hotness near 0 and they are densely populated from 0.25 to 1. There are few songs with loudness less than -40 and they are densely populated from -40 to 0. There are few songs with song length greater than 1500. The tempo of songs is densely populated from 50 to 250.

Another interesting thing we found out about kmeans which is not backed up here is that ,the way the points cluster around a centroid depends on the initial centroids. And we thought that 10 iterations are sufficient as the delta of centroids after that was less,but I guess if we run the Kmeans long enough it will converge properly even if the initial centroids are far away from the final one.