A7 Report

Shreysa Sharma, Jashangeet Singh

11/2/2017

Code Repository: https://github.ccs.neu.edu/pdpmr-f17/a7-jashangeet-shreysa.git

Specifications of host execution Environment

Attribute	Value
Java Version	1.8.0_102
Java(TM) SE Runtime Environment	(build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM	(build 25.102-b14, mixed mode)
Model Identifier	MacBookPro11,2
Processor Name	Intel Core i7
Processor Speed	2.2 GHz
Number of Processors	1
Total Number of Cores	4
L2 Cache (per Core)	256 KB
L3 Cache	6 MB
Memory	16 GB
Driver Memory	2 GB
Executor Memory	2 GB

Specifications of AWS execution Environment

Attribute	Value
Instance Type	M4 Extra Large m4.xlarge
Memory	16.0 GiB
vCPUs	4

Summary of the design of evaluated program

The implementation involves reading the data and putting it in SparkContext. We have implemented K-Means and Agglomerative Hierarchical alogorithm.

K-Means :

We have created 4 seperate classes for loudness, length, tempo and hotness. These classes check whether the respective values are valid and returns them to the clustering object. To get the 3 different categories we have done the following:

Loudness

Segregated values into 3 different groups by using the random function in which we do the following:

- loudness score > (rand.nextDouble - rand.nextInt(10))
- loudness score > (rand.nextDouble - rand.nextInt(20)) and the rest in 1 category. (here rand is a random generator function in scala)

Length

Segregated values into 3 different groups by using the random function in which we do the following:

- duration > (rand.nextDouble + rand.nextInt(2800))
- duration > (rand.nextDouble + rand.nextInt(800)) and the rest in 1 category. (here rand is a random generator function in scala)

Tempo

Segregated values into 3 different groups by using the random function in which we do the following:

- Tempo > (rand.nextDouble + rand.nextInt(250))
- Tempo > (rand.nextDouble + rand.nextInt(150)) and the rest in 1 category. (here rand is a random generator function in scala)

Hotness

Segregated values into 3 different groups by using the random function in which we do the following:

- Song Hotness score > rand.nextDouble
- Song Hotness score > rand.nextDouble and the rest in 1 category. (here rand is a random generator function in scala)

As we are creating 3 clusters, 1 for each category we start with 3 random centroids and then compare the distance of each score(loudness, duration, tempo, hotness) from these centroids and the one which has the minimum distance from the centroid is moved into that cluster. We compute the new centroid value by taking the mean of the points in the cluster and repeat this till either the number of iterations reach 11 or the change in the centroid position is < 10

Hierarchical Agglomerative Clustering

Approaches used:

We first prototyped the Hierarchical Agglomerative Clustering (HAC) in python to have a better grip over the algorithm and how exactly it is implemented as we were not very familiar with scala. This was done to analyze where can it be potentially parallelized and get a rough estimate of the time taken on the large dataset. In the Python version that was implemented, code was not optimized and lacked parallelism due to which it failed to run even on the small dataset. We then took around 500 datapoints to check the correctness of the implemented algorithm. The clusters generated are shown in plot 1 and verified that the results looked reasonable.

We then ported the python code to Scala. Following describes the steps:

- 1. First we take the feature vector
- 2. The feature vector is transformed using zipwithindex
- 3. We extract cartesian pairs from the transformed RDD and compute the distances between each pair
- 4. Then we emit the sorted distance and pair data to the next step.
- 5. Using this RDD we iterate taking the first pair, computing its centroid, computing its pair-wise distances to the remaining clusters and added to the cluster list.
- 6. We then iterate through the clusters till we reach the required number of clusters.

Subproblem 2

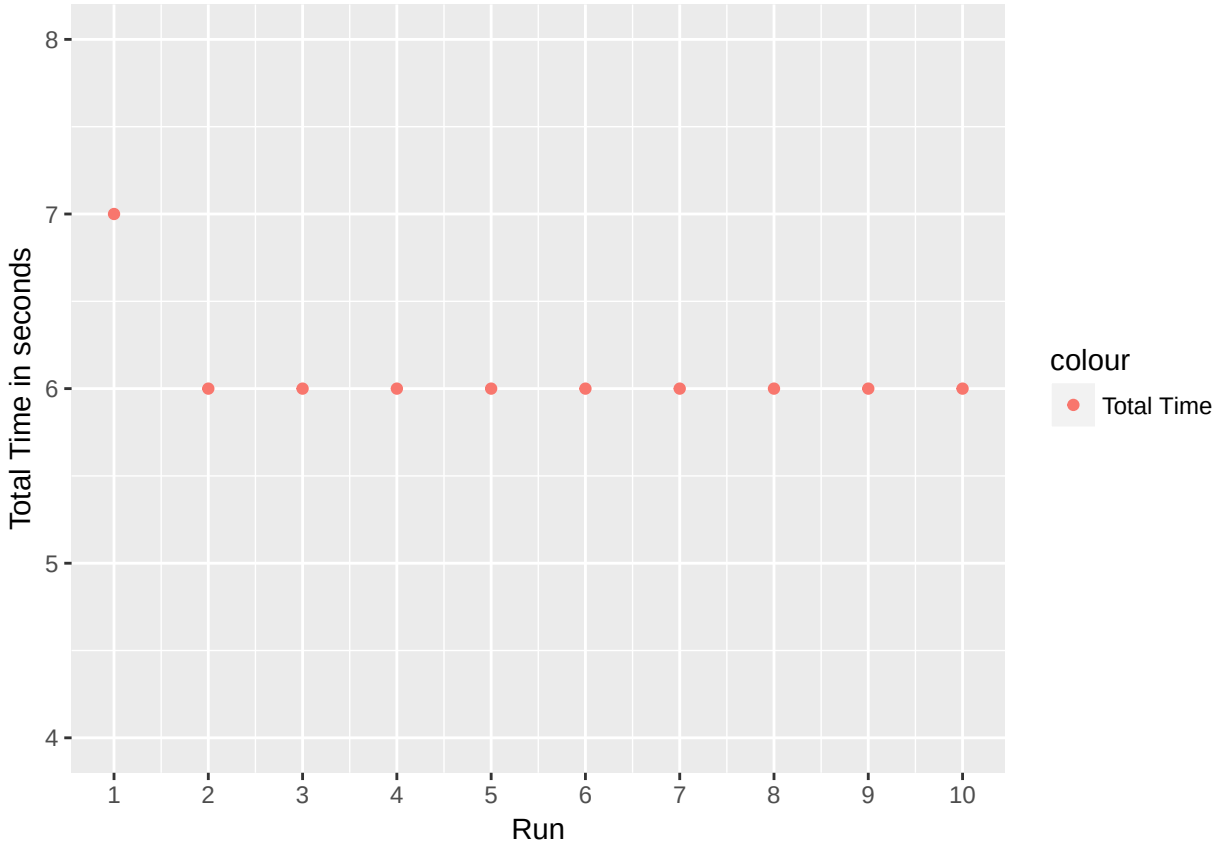
Data Analysis

The first subproblem says to observe

- If a song's loudness, length, or tempo predict its hotness
  - In order to do this we modified the code in the previous assignment to get the top 40 loudness, duration, tempo and hottest songs but could not find any commonality between them to come to a conclusion that any of the 3 factors i.e loudness, duration or tempo predict the hotness of the song.
- If a song's loudness, length, tempo, or hotness predict its combined hotness
- In order to do this we modified the code in the previous assignment to get the top 40 loudness, duration, tempo and hottest songs

Performance Analysis

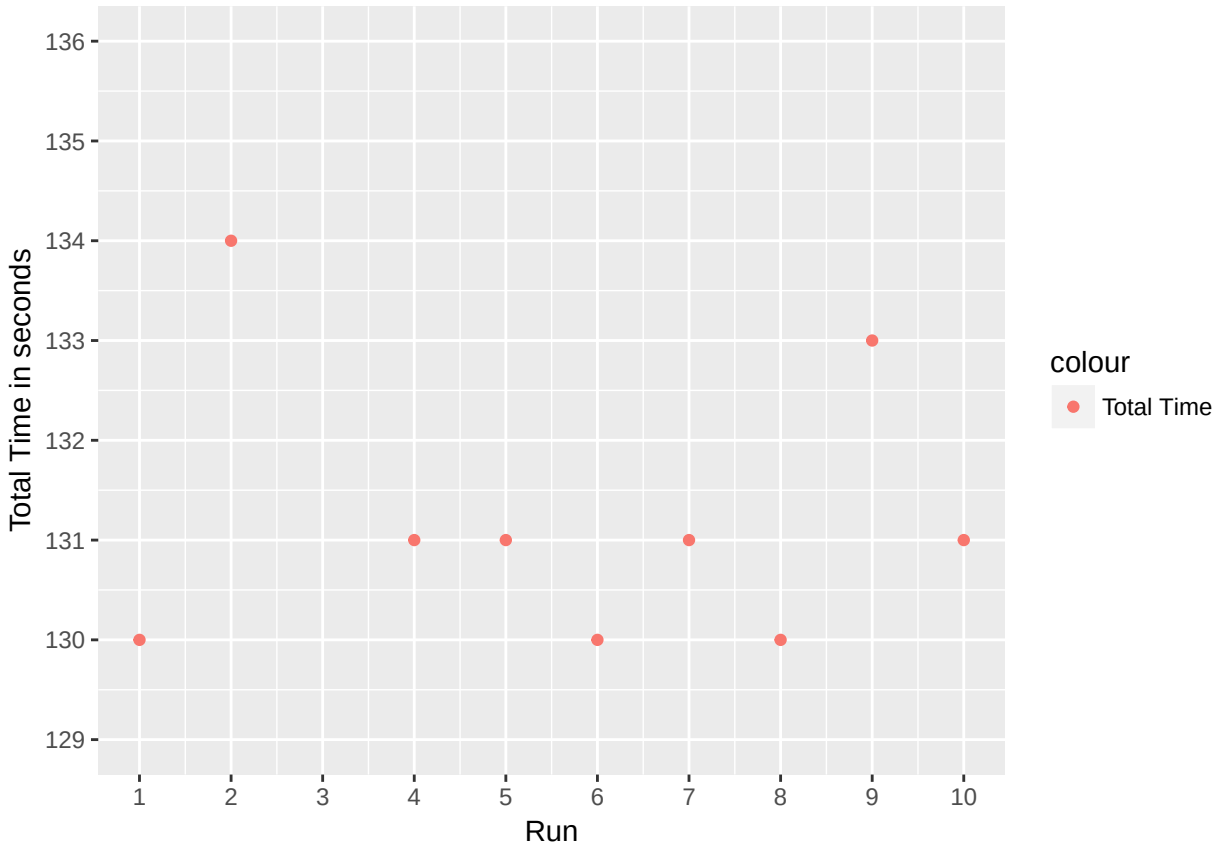
Plot 1: Total time in seconds vs Run for the subset by K-Means



Plot 1 represnts the total time taken by each run in seconds on the small dataset. It can be seen from the graph that the values are 6 and 7 seconds averaging out to 6.1 seconds for 10 runs.

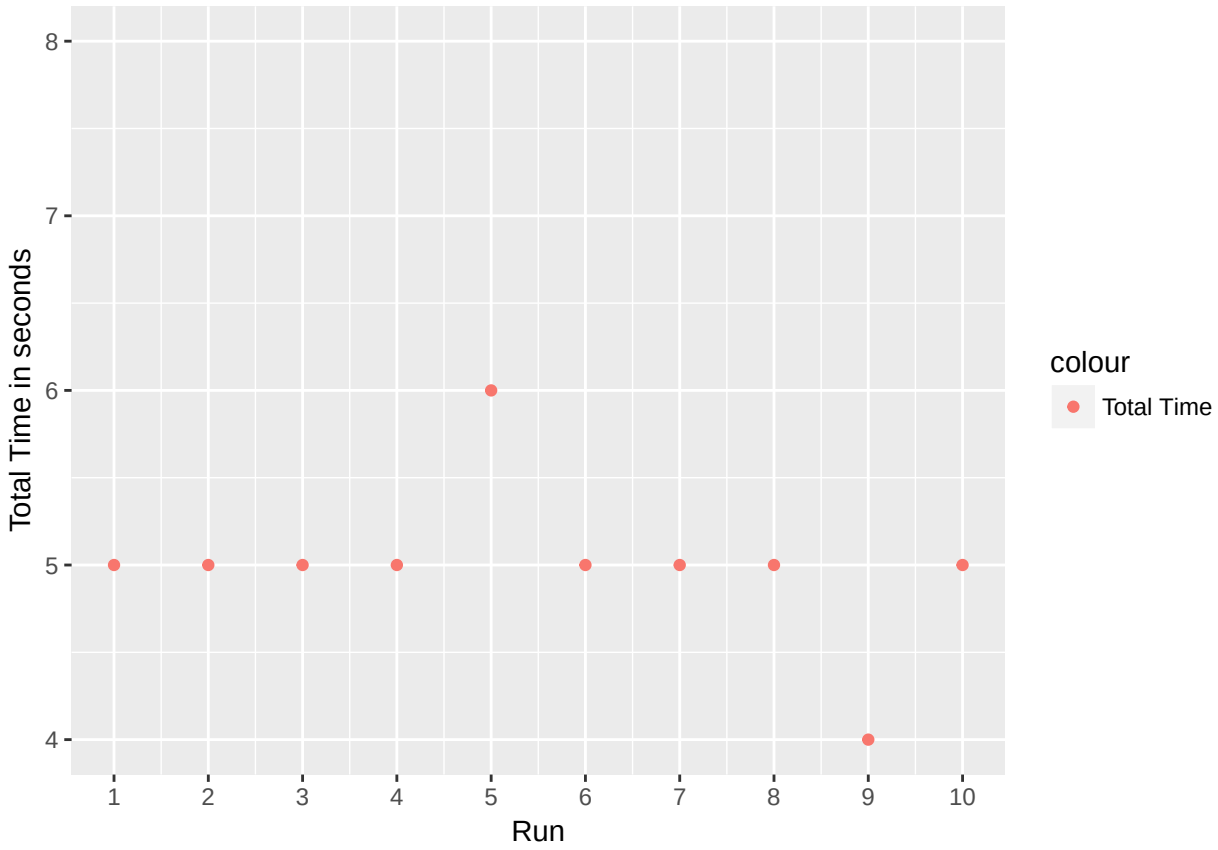
Plot 2: Total time in seconds vs Run for the big dataset for K-Means

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



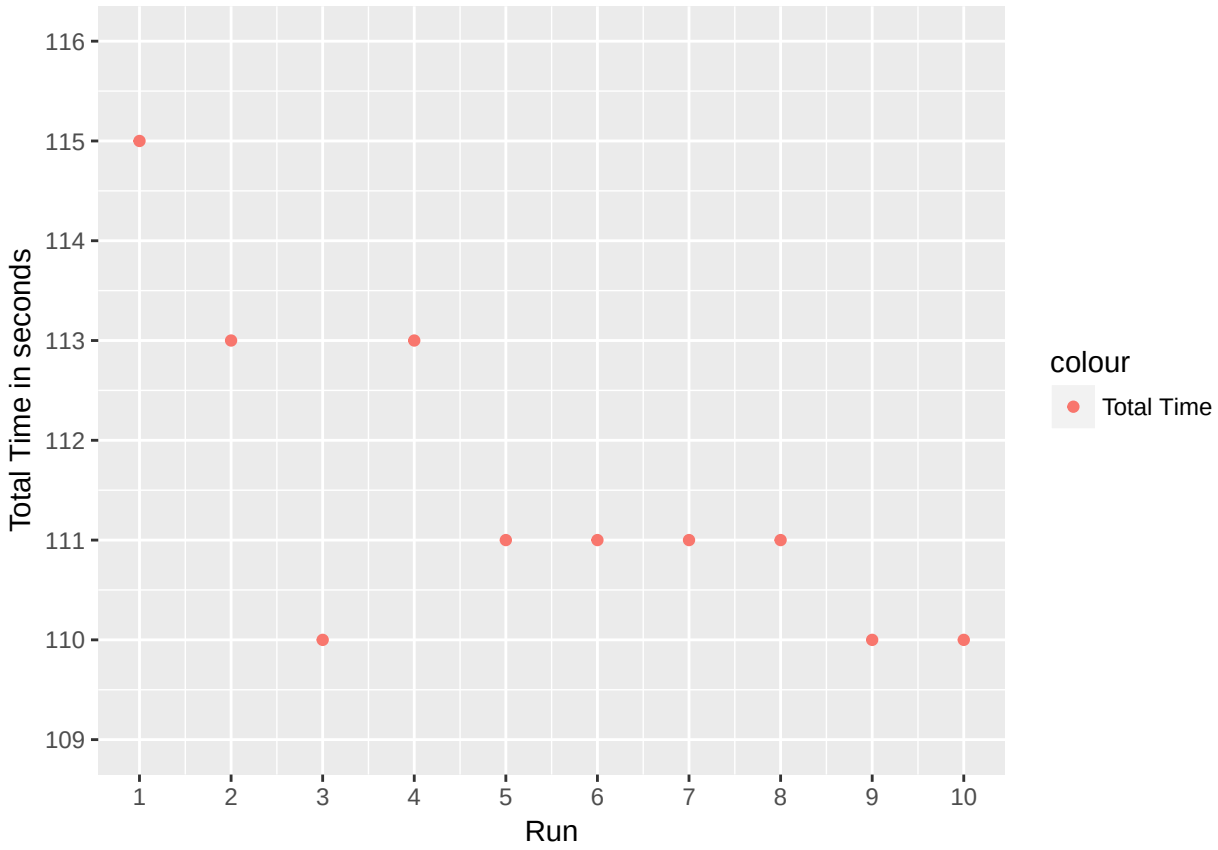
Plot 2 represents the total time taken by each run in seconds on the complete dataset. It can be seen from the graph that the values range between 130 to 135 seconds averaging out to 131.9 seconds for 10 runs.

Plot 3: Total Time in seconds vs Run for the big dataset on AWS for K-Means



Plot 3 represents the total time taken by each run in seconds on the small dataset when it was run on aws. It can be seen from the graph that the values are 4 and 5 seconds averaging out to 4.9 seconds for 10 runs.

Plot 4: Total Time in seconds vs Run for the big dataset on AWS for K-Means



Plot 4 represents the total time taken by each run in seconds on the complete dataset when run on AWS. It can be seen from the graph that the values range between 110 to 115 seconds averaging out to 111.5 seconds for 10 runs.