George Dunnery, Katrina Truebenbach
Git repo: https://github.com/2020-F-CS6240/project-spotifykmeans

Project Overview

This project creates a Spotify radio application by implementing distributed K-means clustering for over 2.5 million Spotify songs. Given a song selected by a user, we can present other similar songs in its cluster that the user will likely also enjoy. We are clustering based on 13 numeric audio features such as time signature, energy, and danceability.

We implement distributed K-means using two methods: (1) distributed K-means that parallelizes the computation for each K and (2) sequential simultaneous K-means that computes each clustering locally while exploring different K values in parallel. We evaluate our clusters with Within Cluster Sum of Squares error (average distance between each song and its final cluster center) and choose the best K value by finding an “elbow” in the decreasing error over increasing values of K. This is the appropriate evaluation metric because for the radio application, we are more concerned about songs within a cluster being closely related than with clusters being distinct.

Our two implementations of K-means are both highly scalable and exhibit strong speedup. For distributed K-means, we utilize Spark’s ability to hold data in memory between iterations to increase efficiency, and for sequential simultaneous K-means, we implement load balancing by balancing how setup configurations are assigned to machines. We also perform a series of experiments to understand how various parameters affect the algorithms and present interesting findings. We believe that we have successfully implemented effective, scalable clustering to find related songs to support a radio application.

Input Data

We started with the Spotify Million Playlist Dataset (MPD), which stores playlist metadata and lists songs by their Spotify URI. It contains a total of approximately 2,260,000 unique songs. The URI can be used to query 13 numerical audio features from the Spotify Web API, which provides interesting dimensions to cluster with. We used a multistep process including a local Java program and a 4-job MapReduce program to parse the JSON objects in the MPD to create our final dataset. The first Map-only job parses the JSON playlist files, the second job drops duplicate songs within and between playlists, the third Map-only job queries the Spotify API in batches to get the numeric features, and the fourth job attaches the playlists each song appears in to its feature vector. We do not ultimately use this playlist feature, but future work could use common playlist membership as a similarity feature or a means of evaluating clusters.

Format:
SongURI   title, artist, duration (ms), key, major/minor, time signature, acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness, valence, tempo
Sample song (truncated features):
3Oup6y15oZCkG5GCWDaemC Lesser People,Spherix,74998,11,0,4,7.58E-5,0.708,0.816,0.72

For sequential simultaneous K-means, we also use a setup file as an input to specify the parameters for each sequential K-means run.
Sample Setup Line (K, Random Seed, Max Iters, Epsilon):
1000 42 25 0.0000001

Please see ReadMe in git repo for list of Resources.

Sample inputs:
https://github.com/2020-F-CS6240/project-spotifykmeans/tree/master/sample-input-files

Task 1: Distributed K-Means

Overview

This task implements distributed K-means clustering that parallelizes the computation for each K. It outputs each song with its cluster assignment. The program also enables us to loop through multiple values of K and calculate the average Within Cluster Sum of Squares for each.

Pseudo-code
https://github.com/2020-F-CS6240/project-spotifykmeans/blob/master/src/main/scalaCode/KMeansParallelK.scala
Note some of this code is modified from example code provided in the module.
K-Medoids is also implemented as an option. Pseudo-code only includes K-Means.

```
// find closest center to each point (song)
def findClosestCenter(p, centers)
        for (k <- centers.keys)
                if (tempDist < squaredDistance(p, centers(k))): closest = tempDist; bestKey = k

// assign points to closest center and calculate new centers
def clusterAssignment(scaledFeatures, centers)

        // find the closest center to each point (song): pair RDD where key = cluster
        // number and value = (point, 1). 1 so can count number of points in cluster
        val closest = scaledFeatures.map(p => (findClosestCenter(p, centers.value), (p, 1)))

        // find new centers: take average in each dimension in each cluster. Collect as map
        val pointStats = closest.reduceByKey { case ((p1, c1), (p2, c2)) => (p1 + p2, c1 + c2) }
        val newCenters = pointStats.map { pair =>  (pair._1, pair._2._1 * (1.0 / pair._2._2)) }
                .collectAsMap()

def main(inputDir, outputDir, wSoSDir, KArray, convergeDist, maxIters, seed, algorithm)
        val songs = read songs from input directory and create DF with one column per attribute
        val scaledFeatures =  scale features such that mean = 0 and std = 1, cache as RDD

        // clusterings for given values of K and calculate average wSoS for each clustering
        for (k <- KArray)
                val K = broadcast(k)   // broadcast k to all executors
                // K random initial centers, broadcast. Map where key = cluster number
                val centers = broadcast(scaledFeatures.takeSample(K, seed).collectAsMap())

                // iterate until change in distance from old to new centers is less than
                // convergence distance OR reach max iterations (user input)
                var tempDist = Double.PositiveInfinity; var iter = 0
                while ((tempDist > convergeDist) && (iter < maxIters))

                        // assign points to closest center and find new centers. Broadcast centers
                        val closest, newCenters = clusterAssignment(scaledFeatures, centers)
```

```
            val newCenters = broadcast(newCenters)

            tempDist = total sum squared distance between centers and newCenters
            centers = newCenters

    // Within SoS error: avg. dist between song and center using squaredDistance
    // zip cluster assignments for each point (song) with their title and artist
```

Algorithm and Program Analysis

We chose to use Spark for K-Means rather than MapReduce because Spark can hold data in memory between iterations. Thus we cache the scaledFeatures dataset, which is repeatedly used in each iteration, so that it does not need to be reloaded each time. We also broadcast the K value and the cluster centers to all machines so that each task can access them. Otherwise the driver would send the variables to each task even if there are multiple tasks per machine, which would cause unnecessary data transfer.

Experiments

Output and log files: https://github.com/2020-F-CS6240/project-spotifykmeans/tree/master/aws-runs-parallelk/aws-experiments

We ran experiments to understand how run time changes with the value of K (number of clusters), the number of iterations, and different random seeds. These experiments were run on a subset of 150,000 songs with 3 workers of type m4.xlarge. We set converge distance to 0 so that the full number of iterations specified ran. Unless otherwise specified, we set 10 iterations and 5,000 clusters. For the sake of experimentation, we only compute one clustering for one K.

The first experiments varied the number of iterations (Appendix 1). For small values, doubling the iterations doubled the run time. However, the additional time per iteration decreases as the number of iterations increases. We believe this is because the clusters are becoming more stable and less shuffling is required to assign points to new clusters (reduceByKey).

The second set of experiments varied the number of clusters (Appendix 2). As expected, the number of clusters and run time increase at the same rate.

Finally, we considered different seeds for setting the initial random centers. We tried 4 seeds and got a range of run time values from 13 to 22 minutes. This is because the “goodness” of the initial centers help determine how much shuffling is needed to assign points to clusters.

Scalability & Speedup

We achieved good speedup for our problem: doubling the number of workers from 3 to 6 decreased run time by about half (Appendix 3).

For scalability, the results are less interpretable due to the large effect different initial centers have on run time. In all prior experiments, the input data was the same, so by setting a seed we ensured the same centers were used. However, changing the input data will result in different initial centers, making the experiments less comparable. Decreasing the input from 150,000 to 70,000 songs approximately cut the run time in half. However, doubling the input from 150,000 to 300,000 songs only increased run time by 2 minutes, likely due to a “lucky” initial choice of random centers for the 300,000 song run. (Appendix 4)

Results Sample

Using the above experiments, we estimated that the full set of 2,260,000 songs would take approximately 12.6 hours to run, which was unreasonable to complete. Instead, we estimated that 850,000 songs would complete in about 2 hours using 6 m4.xlarge workers. This is assuming maximum 10 iterations and number of clusters = number of songs / 100, such that each playlist has approximately 100 songs. We set convergence distance to 1.0. The actual run took only 45 minutes, but the initial random centers add variability to this calculation. The next section will produce clusterings for multiple K values and choose the optimal K.

1. Cluster output: (cluster number, [song title, artist name])
```
(0,[Argumento,Paulinho Da Viola])
(0,[I Love You California,Rick Pickren])
(1,[Depression Party,Endwell])
```

2. Within Sum of Squares: (number of clusters K, average within cluster sum of squares error)
```
(1000, 2.414)
```

Output and log files: https://github.com/2020-F-CS6240/project-spotifykmeans/tree/master/aws-runs-parallelk/aws-final

Task 2: Simultaneous Sequential K-Means

Overview

The goal of this task is to try many different values of K simultaneously by running sequential KMeans algorithms on multiple machines. For each cluster, it outputs a single line prefaced by the setup details (K, random seed, max iterations, epsilon) with a list of lists containing song indices and a list of the dimensional values of the centroid for each cluster. A list of real centers is also included for K-Medoids and/or to check if K-Means happened to pick a real song as its center. KMedoids is also included as an option, although it will run more slowly compared to KMeans. For each setup, we calculate and output Within Cluster Sum of Squares Error.

Pseudo-code

https://github.com/2020-F-CS6240/project-spotifykmeans/blob/master/src/main/scalaCode/KMeansMultiK.scala

```
// Driver function. KMedoids is also an option!
main(input, setup, output, algo)    // Also takes: balance, machines, nameCenters
    // Load & broadcast song data. Scale the features for clustering
    var songs = getSongsDF(spark, input)
    val scaledFeatures = broadcast(getScaledFeatures(songs).collect())
    // The setup file contains different K values for each KMeans run
    var runs = Read setup file into an RDD & give each entry a key using zipWithIndex()
    if (load balancing enabled)
        val partitioning = roundRobinAndExchange(runs.map(estimate workload))
        runs = parallelize(partitioning).partitionBy(custom partitioner)
    // Sequential KMeans algorithm is run on an individual machine for each K value
    val results = runs.map(setup => (setup, KMeansLocal( … ))
    // Output is saved by index for now
```

```
        results.saveAsTextFile(output)

// Sequential KMeans that uses only regular scala objects and runs on one machine
KMeansLocal(features, K, seed, maxIters, epsilon) returns Clustering
        // Initialize centroids to K random points
        centroids = randomCenters()
        init counter, delta, assignments
        // Run KMeans until convergence or maxIters
        while (delta > epsilon && counter < maxIters)
                assignments = reassign(features, centroids)
                newCentroids = recomputeMeans(features, assignments)
                delta = getDelta(centroids, newCentroids)
        return Clustering(iterations, getWsos(), assignments, centroids)

// Assigns all songs to their closest centroid
reassign(features, centroids) returns new assignments
        assignments = initAssignments()
        for (songID <- 0 until features.length)
                assignments(closestPoint(features(songID), centroids)) += songID
        return assignments

// Recomputes the centroids as a synthetic data point (average of cluster members)
recomputeMeans(features, assignments) returns list of new centroids
        for (cluster <- 0 until K)
                sum = new Vector(13 dimensions init to 0.0)
                for (songID <- assignments(cluster))
                        sum += features(songID)
                newCentroids(cluster) = sum / new Vector(13D = assignments(cluster).length)
        return newCentroids

// Sort by estimated work, Assign to machines cyclically, like dealing cards
// Do an outer-pairwise sliding window exchange to attempt to reduce the max workload
roundRobinAndExchange(work, machines)
        val sorted = work.sortByKey().collect()
        var assignments = for run in sorted, assign to machine by (index mod machines)
        return slidingWindowExchange(assignments)

// Pair workers (1 with n), (2 with n-1)..., walk back through lists & exchange runs to reduce max
slidingWindowExchange(assignments)
        for i = list length to 0 by -1             // starting index for both lists
                for j = 0 to floor(machines/2)   // these two lines do the "outer-pairwise" pairing
                        k = machines-j             // so we only go through each list once
                        if (exchange at index i reduces max(load) for machines (j,k)
                                do exchange (assignments)
        return assignments
```

Algorithm and Program Analysis

The sequential KMeans algorithm and its helper functions are the main bottleneck in this implementation, since the driver in Spark only uses a map() call. The features are broadcast to every machine and libraries for vector math operations are used to avoid the use of deeply nested for loops. Further analysis of this task revolved around the choice of partitioning. The intuition is that Spark doesn't know the implication of a given setup line (K value, max iterations, epsilon), which means it could accidentally give one machine all the longest runs, thus slowing down the whole job by leaving the other machines idle. Controlling this behavior turned out to be a key target in performance optimization.

The load balancing process uses a custom partitioner and manually assigns KMeans runs to machines by using keys in a PairRDD. First it estimates the amount of work each run will take using the heuristic (K * Max Iterations), then it sorts the runs in ascending order of difficulty, distributes them cyclically to all machines (like dealing cards), and lastly it steps backwards through pairs of lists to attempt to exchange runs to reduce the difference between the lightest and heaviest workloads (Appendix 10).

Experiments

As discussed above, load balancing is a key concern and was the focus of our experiments. For all of these experiments, we set the input size to 70,000 songs, used 42 as the random seed, and removed epsilon as a confounding factor by setting it to 0 (avoiding early convergence). The machine type was set to m4.xlarge, with the number of workers varying by experiment.

For a single machine, increasing the number of clusters or max iterations both result in a linear increase in runtime (Appendices 5 & 6). This result, in addition to the upper bound run time of KMeans, was the inspiration behind the work estimation heuristic used: K * max iterations.

Experiments were also run for clusters using 2, 4, and 6 workers to compare the performance of the unbalanced and balanced versions of the program (Appendix 7). The same setup file was used for each. It was discovered that the unbalanced program could receive both "lucky" and "unlucky" conditions based on the order of the runs in the setup file. This resulted in the 4 machine run outperforming the 6 machine run for the unbalanced program. However, the balanced program ran faster for every configuration and consistently demonstrated a significant improvement as more machines were added (the runtimes include the load balancing process).

Scalability and Speedup

The experiments comparing the balanced and unbalanced versions of the program are used for speedup and to quantify the improvement gained by load balancing (Appendices 7 & 8). The overall speedup from 2 to 6 workers showed a 63% improvement for the balanced program, while just a 9% improvement for the unbalanced program. The balanced program improved by 51% from 2 to 4 workers, and 21% from 4 workers. The unbalanced program shows a "lucky" improvement of 36% from 2 to 4 workers, but does not get lucky again and deteriorated by 41% from 4 to 6 workers. Using load balancing clearly results in better speedup.

Scalability was measured by examining the effect of increasing the input size on a standard cluster of 6 workers (Appendix 9). For this part of the project, we defined input size as the number of K-Means runs (i.e. lines in the setup file), with each line containing the same parameters to make them comparable to each other. Setup files with 12, 24, and 36 lines were tested, showing a clear linear increase overall. This pattern could be used to help estimate how long a large setup file would take to run.

Results Sample

The clusters are output onto a single line, along with their setup information, some run statistics, and the clustering results. The setup is preserved to help guide future clustering configurations, and the real centers are especially useful for running the K-Medoids option where the centers will be real songs. The centroids are included for convenience and the clustering itself is output in a readable fashion as a triple containing the cluster number, song title, and artist.

For the final AWS run, we explored 12 cluster setups from K = 100 to 2300 in steps of 200, using a constant seed, 20 max iterations, and epsilon at 0 on a cluster with 6 workers.

- Setup Line: K value, random seed, max iterations, epsilon
- Clustering: Iterations taken, wSoS, [(cluster number, song name, artist), ...]
- Centroids: Cluster number [13 dimensions of the center point]
- Real Centers: [reports as synthetic or can lookup real song names]

(setup line), (run data, (song assignments)), (centroid vectors), (center names)

```
(100,123,75,0.0), (46,3.5869420061351542, (0,Life Goes On,Vybz Kartel),...),
(0 (-0.009763907369938726,...) …), ((0,<Synthetic>,<Synthetic>), ...)
```

https://github.com/2020-F-CS6240/project-spotifykmeans/tree/master/aws-runs-multik/aws-16-final

We graphed the resulting wSoS error against each K in the final run in order to find the optimal K (Appendix 11). As expected, the error decreases as K increases, but at a decreasing rate. We determined the optimal K to be 1,100 where the error's rate of change begins to flatten and we find an "elbow" in the graph. Thus increasing K more gives only marginal improvements in error.

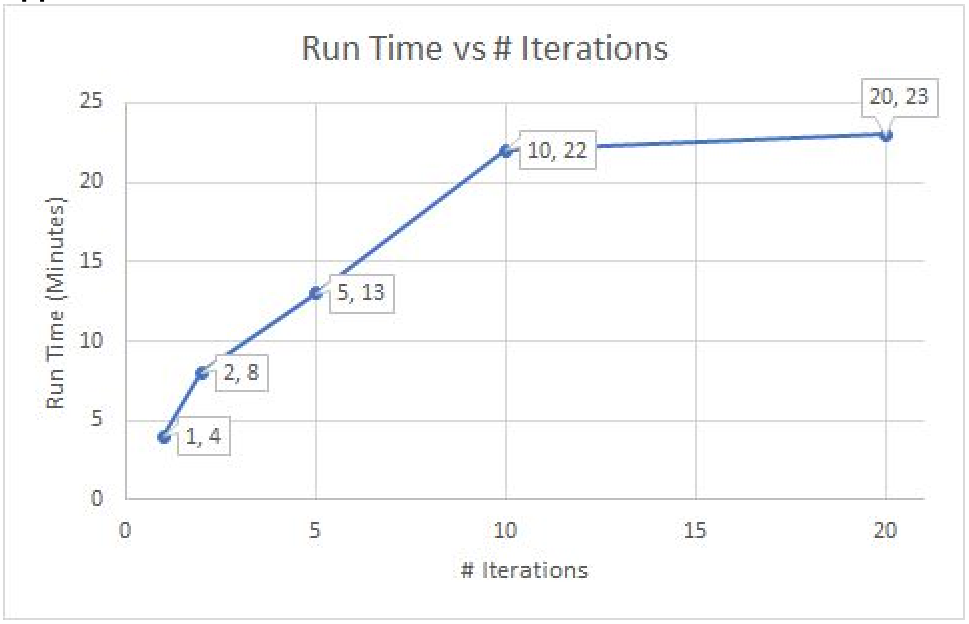
Conclusion

Potential extensions of this project include using the playlist data to enhance the songs' features such that songs in a similar set of playlists are considered more similar or to evaluate clusterings by calculating the fraction of songs that are in the same playlists within a cluster. Another extension is to explore other clustering algorithms. For example, the Lingo algorithm uses dimensionality reduction techniques to allow songs to belong to multiple clusters and thus radio playlists. Other load balancing techniques could also be investigated, and the current method could be extended to assign uneven numbers of runs. For example, in Appendix 10, we would ideally want to just give the first run on M1 (weight 1) over to M2 and make the total weight for each machine 15, where each machine has a different number of runs.

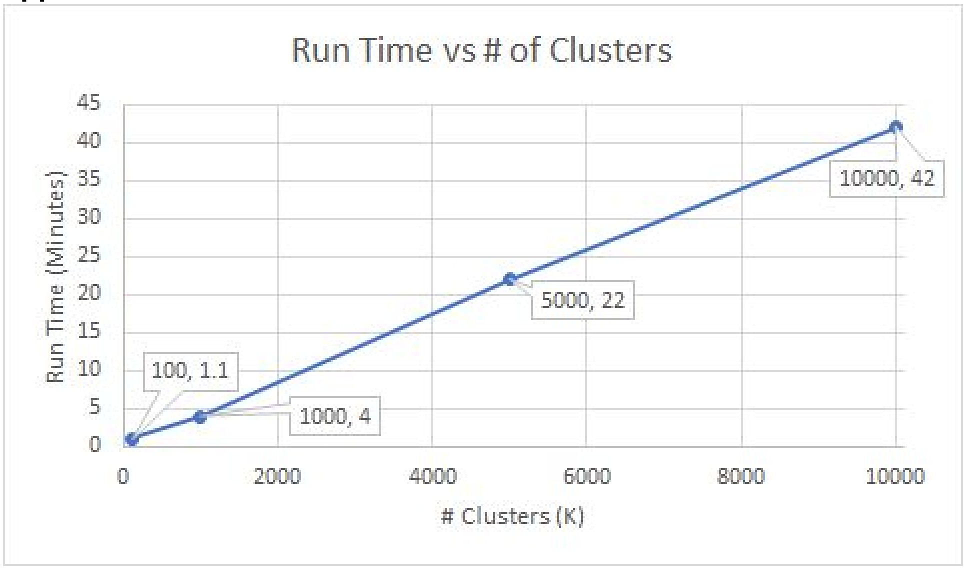
The Distributed algorithm is best for training clusterings for a couple K values with a large songs list, while the Simultaneous Sequential algorithm allows for many different setups to be trained at once with a smaller songs list. In order to improve efficiency, we utilized broadcasting and caching in Distributed K-Means to take advantage of Spark's ability to hold data in memory between iterations, which resulted in significant performance gains. We also implemented explicit load balancing in Simultaneous Sequential K-Means, which dramatically reduced runtime by ensuring that a good partitioning was always chosen (with a fast approximation of a combinatorially large problem). Ultimately, we successfully built two highly scalable parallel programs that trained K-Means clustering to generate song clusters which could be used in a Spotify radio application to find similar songs based on cluster assignments.

Appendix

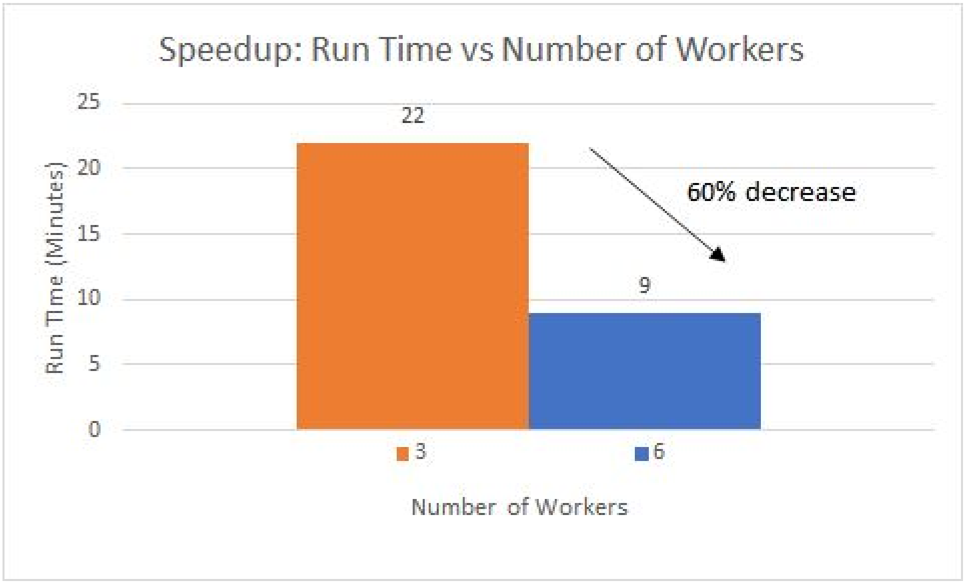
Appendix 1



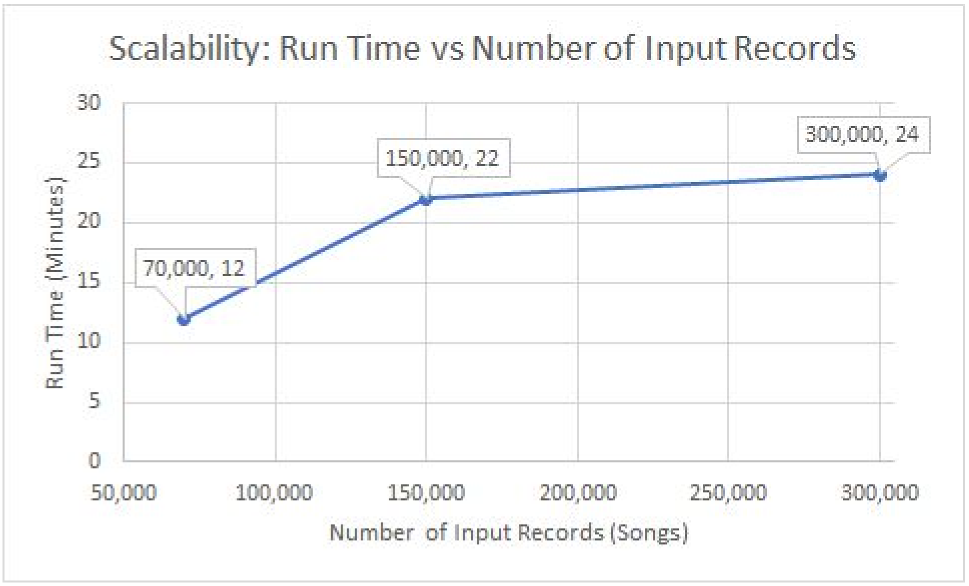
Appendix 2



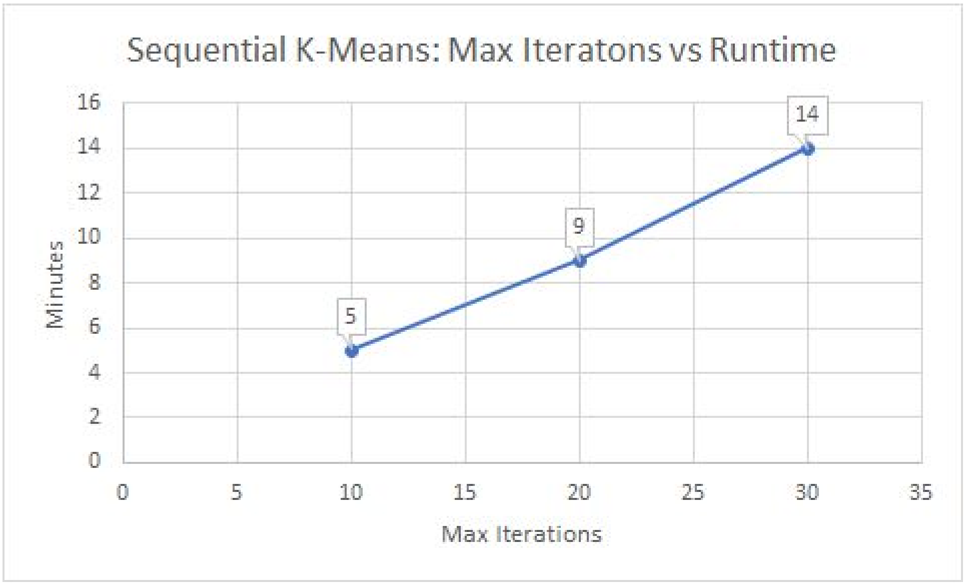
Appendix 3



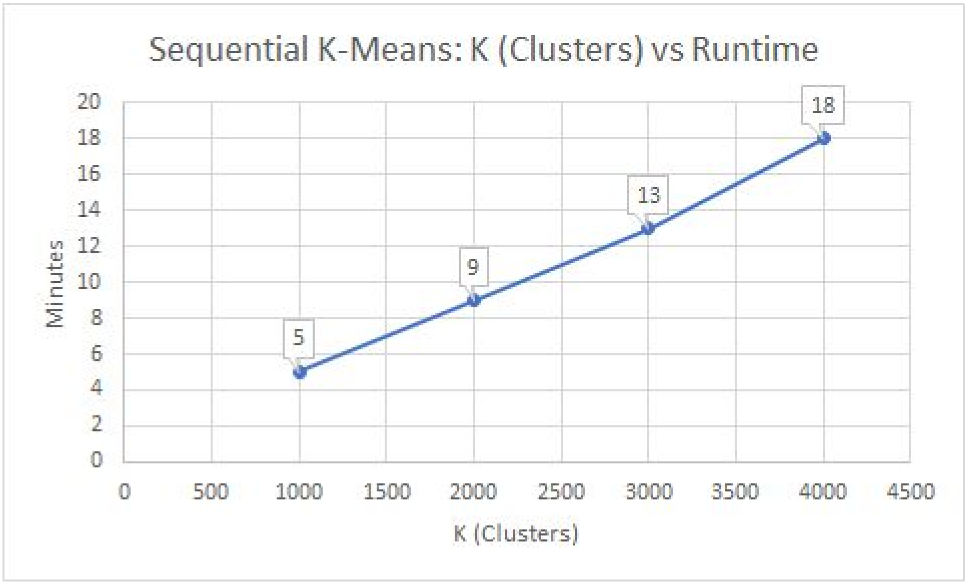
Appendix 4



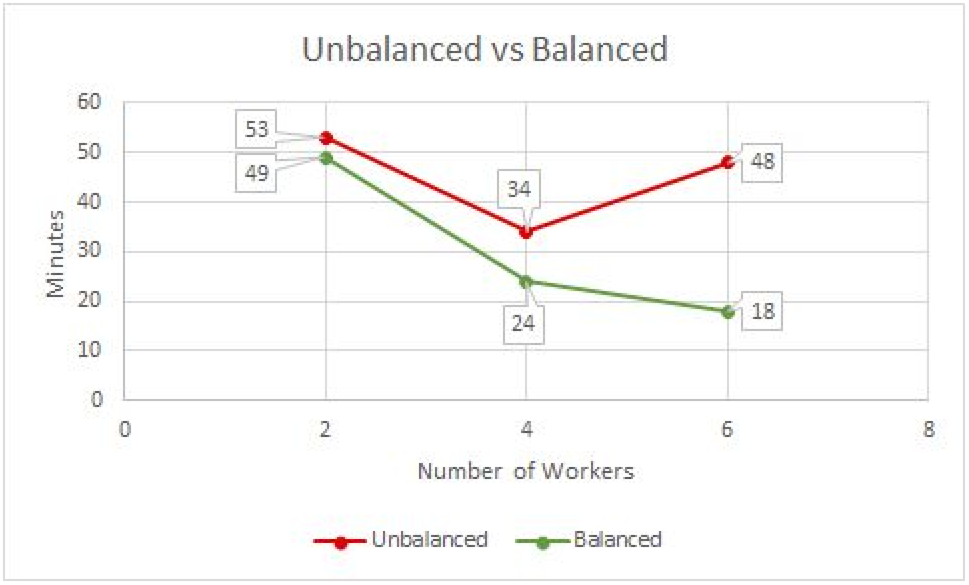
Appendix 5



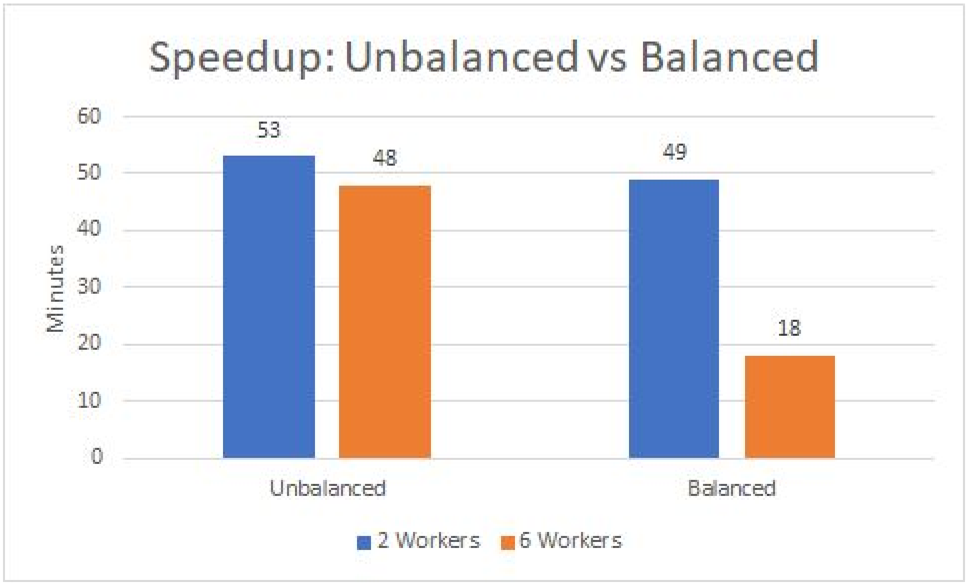
Appendix 6



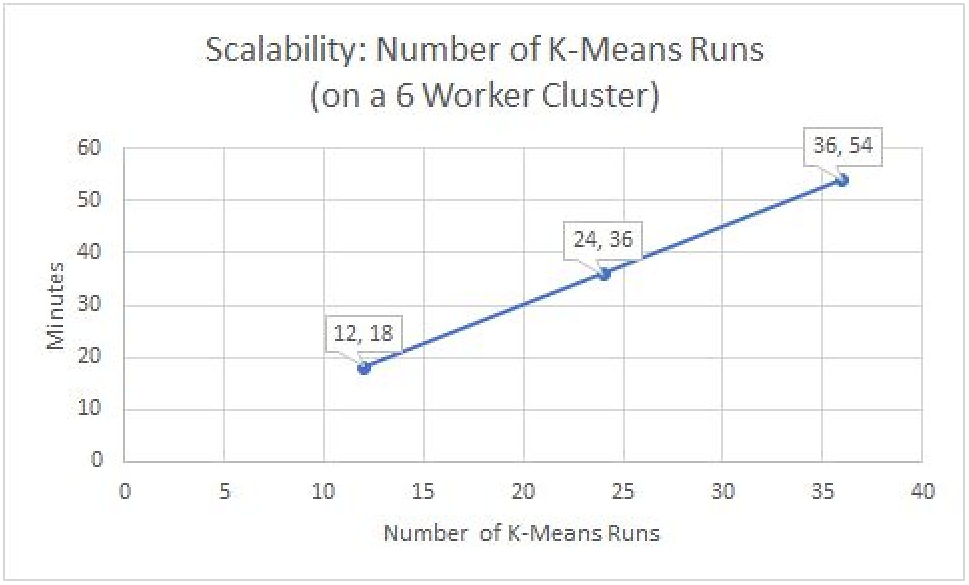
Appendix 7



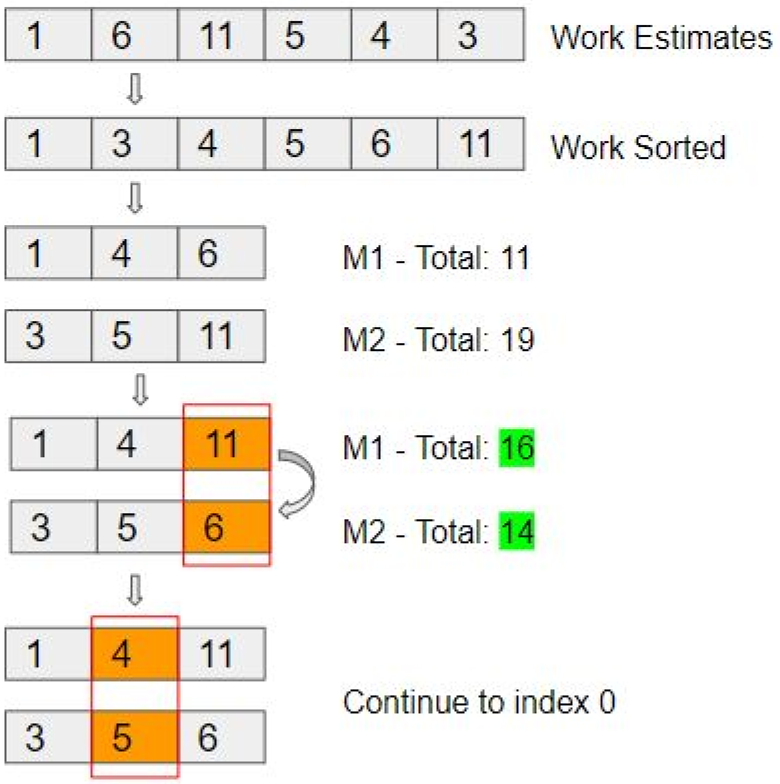
Appendix 8



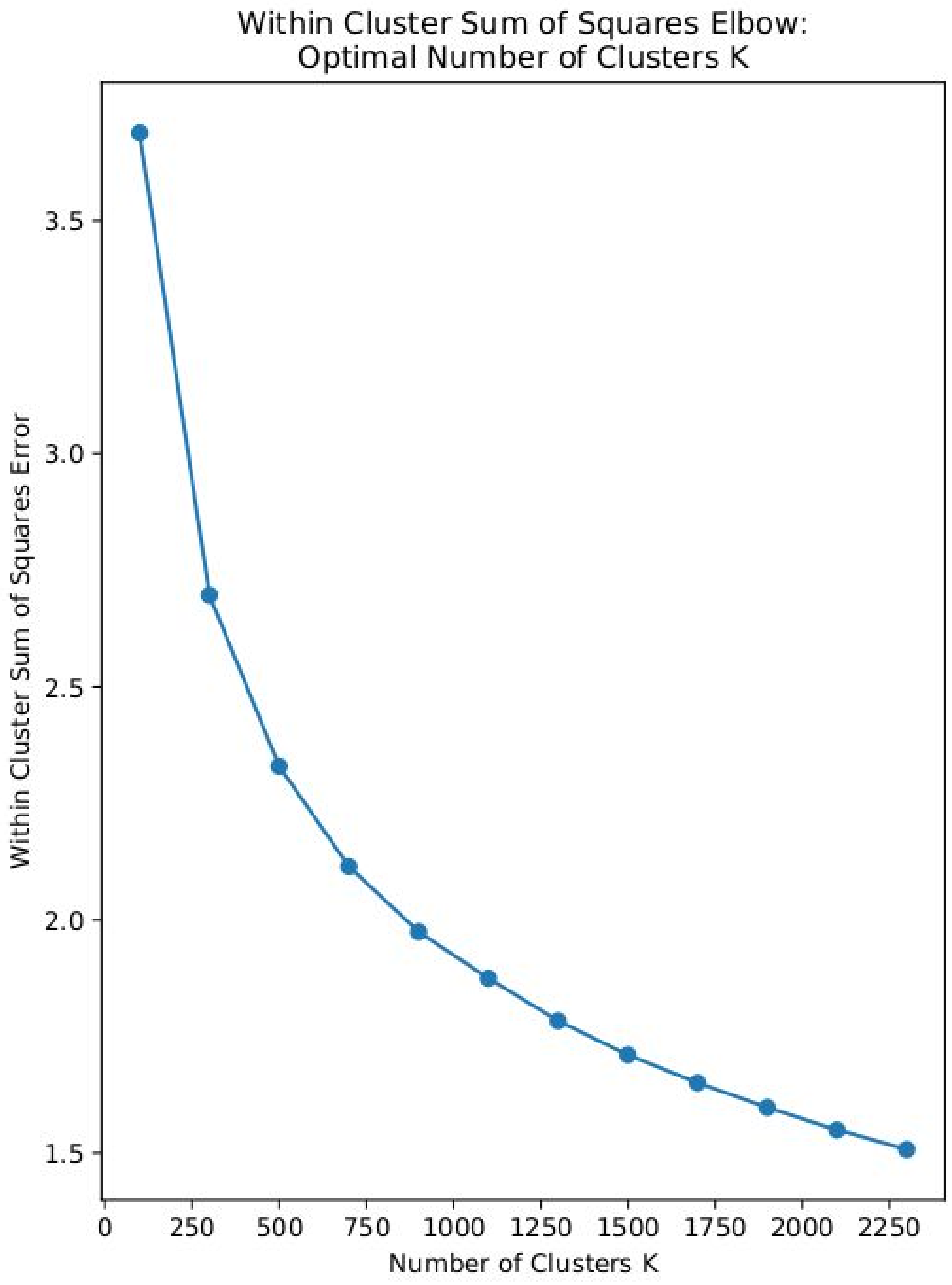
Appendix 9



Appendix 10: Load Balancing Strategy



Appendix 11: Plotting wSoS from Final AWS Run of Simultaneous Sequential K (Multi K)


Within Cluster Sum of Squares Elbow:
Optimal Number of Clusters K
Within Cluster Sum of Squares Error
Number of Clusters K