Clustering Report

K-means Algorithm

Algorithm Description

After loading and separating the dataset into ground truths, gene IDs and gene ID number (row numbers), for the initial centroids we choose top 5 k values. These centroid-mappings are considered in order to recalculate the new centroids. For recalculating the centroids, we iterate the cluster mappings and append a data point's index to the "points" list if that data point belongs to that cluster. These recalculated centroids are set as the current centroids ("new_centroid" list).
The breaking condition happens when the clusters have not changed in consecutive iterations of the algorithm. In such a case, the function returns it's most recent centroid values and exits. Otherwise, it will set the value of the function-wide "centroid" variable equal to the value of the new_centroid variable.

Result Visualization:

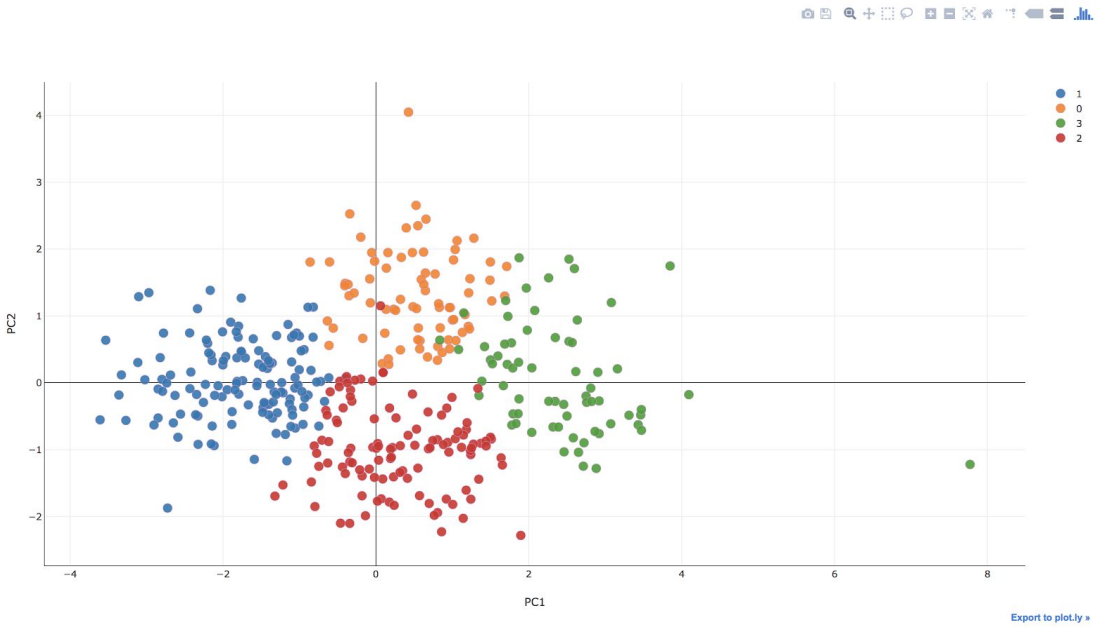
For cho.txt:

For k : 2 Jaccard is : 0.352850462641 Rand is : 0.650755724986
For k : 3 Jaccard is : 0.411914527653 Rand is : 0.761349297968
For k : 4 Jaccard is : 0.418035918991 Rand is : 0.79556498161
For k : 5 Jaccard is : 0.404033242743 Rand is : 0.796893876346
For k : 6 Jaccard is : 0.306951690207 Rand is : 0.780665252758
For k : 7 Jaccard is : 0.297298465528 Rand is : 0.781779376628
For k : 8 Jaccard is : 0.276937618147 Rand is : 0.784356627024
For k : 9 Jaccard is : 0.270836203334 Rand is : 0.786853338345
For k : 10 Jaccard is : 0.263823738451 Rand is : 0.791444065612
For k : 11 Jaccard is : 0.256852300242 Rand is : 0.794007892829
For k : 12 Jaccard is : 0.235463274358 Rand is : 0.78944401192
For k : 13 Jaccard is : 0.259944139459 Rand is : 0.793712582888
For k : 14 Jaccard is : 0.258329685296 Rand is : 0.795323364386
For k : 15 Jaccard is : 0.256220119036 Rand is : 0.795350210744

We computed the jaccard and rand value from cluster size of 2 to 15 to see which *k* will give the best results. As we can see above, we achieved the best jaccard value when k = 4. Therefore,
Jaccard Value: 0.418035918991
Rand Value: 0.79556498161

Time to run is when k = 4 is 2.13940191269 seconds

The PCA below is K-Means algorithm on cho.txt with cluster size = 4



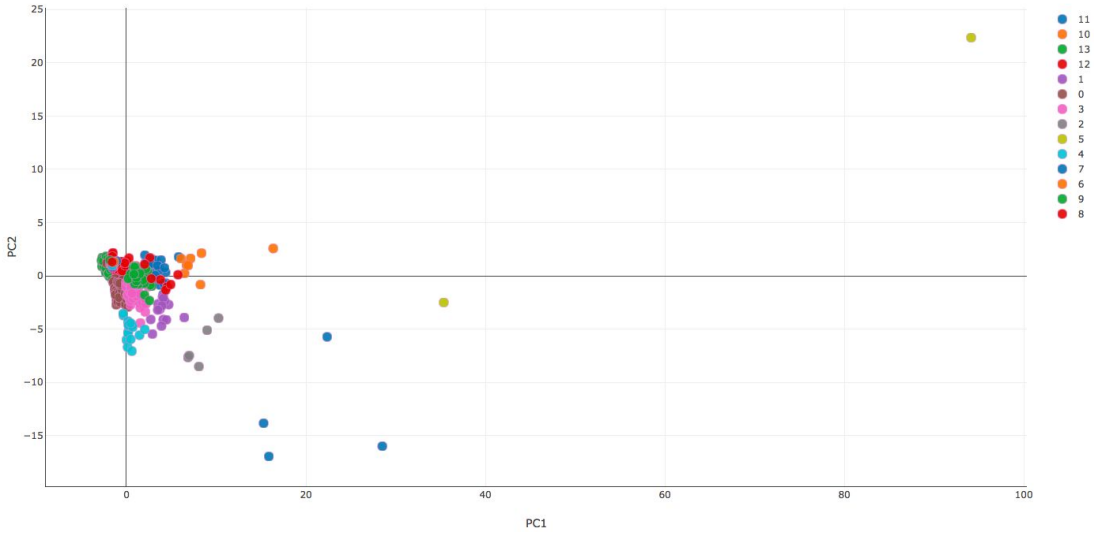
For iyer.txt

For k : 2 Jaccard is : 0.155997409177 Rand is : 0.176105264339
For k : 3 Jaccard is : 0.181225867566 Rand is : 0.362944228906
For k : 4 Jaccard is : 0.209741768847 Rand is : 0.47459117285
For k : 5 Jaccard is : 0.266213309579 Rand is : 0.620927161237
For k : 6 Jaccard is : 0.270497192476 Rand is : 0.638855321394
For k : 7 Jaccard is : 0.281892987788 Rand is : 0.661422654879
For k : 8 Jaccard is : 0.303305178422 Rand is : 0.691098399111
For k : 9 Jaccard is : 0.291563411849 Rand is : 0.68662384161
For k : 10 Jaccard is : 0.291986818217 Rand is : 0.692145954379
For k : 11 Jaccard is : 0.302761902182 Rand is : 0.711084257115
For k : 12 Jaccard is : 0.306904964433 Rand is : 0.722592399987
For k : 13 Jaccard is : 0.285994158724 Rand is : 0.777746184841
For k : 14 Jaccard is : 0.353720460536 Rand is : 0.788102016918
For k : 15 Jaccard is : 0.307314602473 Rand is : 0.815802371216

We computed the jaccard and rand value from cluster size of 2 to 15 to see which k will give the best results. As we can see above, we achieved the best jaccard value when k = 14. Therefore,
Jaccard Value: 0.353720460536
Rand Value: 0.788102016918

Time to run when k = 14 : 19.2089071274 seconds

The PCA below is K-Means algorithm on iyer.txt with cluster size = 14



Pros and Cons:

Pros:

1. The algorithm is very easy to implement.
2. For it's ease of implementation, it gives good clustering results as shown by the PCA plots.

Cons:

1. We need to specify K, the number of clusters every time we run the algorithm.
2. K-means is very sensitive to outliers.
3. Empty clusters might appear in the output.
4. K-means also has problems when clusters are of differing sizes, densities, or irregular shapes.
5. Clustering results depend largely on the initial centroid values. Having a good set of initial cluster centroids and not, can mean the difference between very good and dissatisfactory results. A solution to bringing more certainty in choosing initial centroids is to use the KMeans++ algorithm which chooses one centroid randomly and probabilistically tries to choose a better next centroid with uniform distribution.

Result Evaluation:

K-Means is very efficient as it runs in O(tkn), where *n* is the number of objects, *k* is the number of clusters, and *t* is the number of iterations. We can observe from the results above that for *cho.txt* dataset, as we increase the size of *k*, the jaccard coefficient relatively decreases. However, when we compare this result with *iyer.txt* dataset, as we increase the size of k, the jaccard coefficient increases.

Hierarchical Agglomerative Clustering(HAC) with Single Link(Min)

Algorithm Description

The first step in the algorithm is to create a distance matrix. To find the distances between each of the points, we are using Euclidean distance. Initially, each and every point is a cluster. Then to merge the clusters, we check the entire distance matrix for the minimum distance between two clusters because we are doing HAC with Min link. Once we find the minimum distance between two clusters, we merge these clusters and update the distance matrix by finding out the new distance between the cluster we formed and all the other clusters in the distance matrix. Everytime we merge two clusters, we create a dendogram which shows the link between two merged clusters. We repeat this same process of merging minimum clusters until we get the desired number of cluster which depends on the k-value or until until we merge all objects into a single cluster.

Result Visualization:

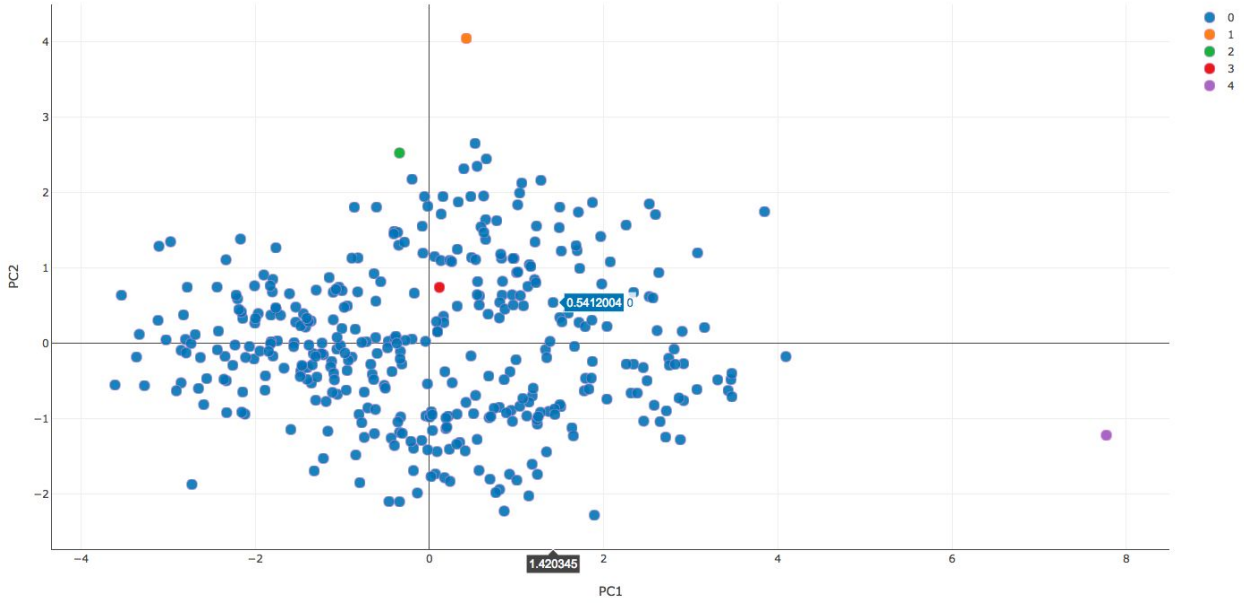
Cho.txt

For k : 1 Jacc is : 0.230073290558 Rand is : 0.230073290558
For k : 2 Jacc is : 0.230371998328 Rand is : 0.233791511182
For k : 3 Jacc is : 0.22933831189 Rand is : 0.235348599962
For k : 4 Jacc is : 0.229429347826 Rand is : 0.238717817928
For k : 5 Jacc is : 0.228394977574 Rand is : 0.240274906709
For k : 6 Jacc is : 0.228493188197 Rand is : 0.243644124674
For k : 7 Jacc is : 0.228791879848 Rand is : 0.24732207576
For k : 8 Jacc is : 0.22873361601 Rand is : 0.250422830143
For k : 9 Jacc is : 0.228842125706 Rand is : 0.25377862493

We computed the jaccard and rand value from cluster size of 1 to 9 to see which *k* will give the best results. As we can see above, we achieved the best jaccard value when k = 5. Therefore,
Jaccard Value: 0.228394977574
Rand Value: 0.240274906709

Time to run when k = 5: 8.12586688995 seconds

The PCA below is HAC (min link) algorithm on cho.txt with cluster size = 5



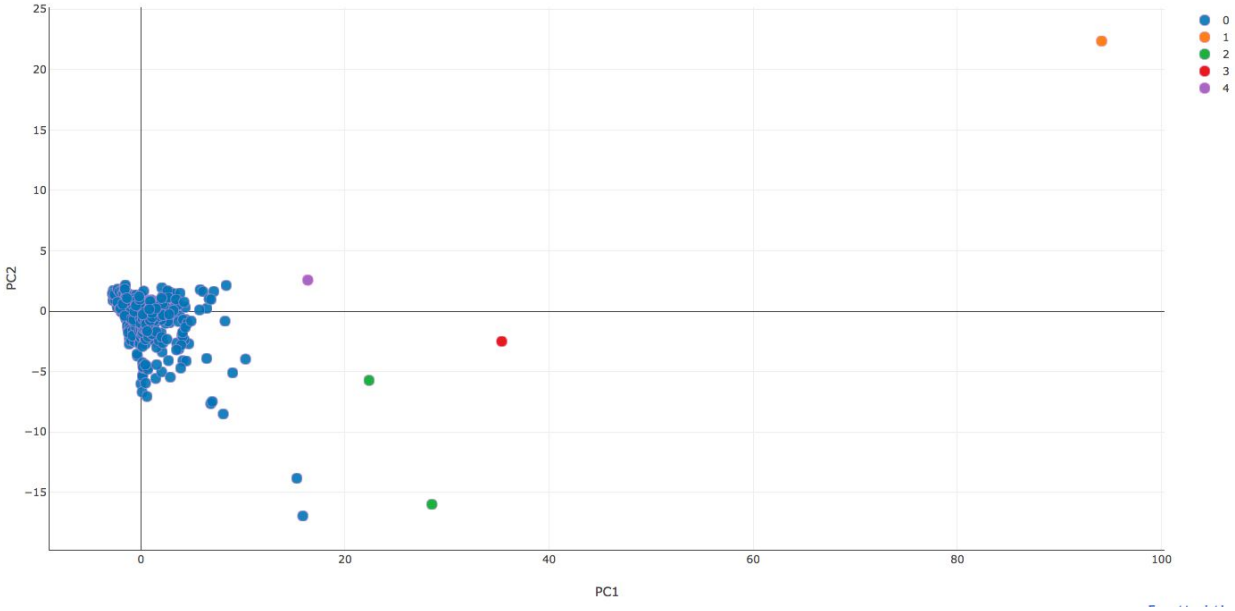
lyer.txt

For k : 1 Jacc is : 0.155169124057 Rand is : 0.155169124057
For k : 2 Jacc is : 0.155484130173 Rand is : 0.158536265989
For k : 3 Jacc is : 0.155930938571 Rand is : 0.162120401513
For k : 4 Jacc is : 0.156327430783 Rand is : 0.165607264048
For k : 5 Jacc is : 0.156472223809 Rand is : 0.171443643397
For k : 6 Jacc is : 0.156464606068 Rand is : 0.17143616086
For k : 7 Jacc is : 0.156642031748 Rand is : 0.17730247036
For k : 8 Jacc is : 0.157104524613 Rand is : 0.180856675733
For k : 9 Jacc is : 0.158250856453 Rand is : 0.188294318135

We computed the jaccard and rand value from cluster size of 1 to 9 to see which *k* will give the best results. As we can see above, we achieved the best jaccard value when k = 5. Therefore,
Jaccard Value: 0.156472223809
Rand Value: 0.171443643397

Time to run when k = 5 is 19.4827349186 seconds

The PCA below is HAC (min link) algorithm on iyer.txt with cluster size = 5



Pros and Cons:

Pros:

1. We do not have to assume any particular number of clusters, any desired number of clusters can be obtained by cutting the dendrogram at the correct level.
2. They are also correspond to many meaningful taxonomies like shopping websites.
3. Min approach of HAC can handle non elliptical shapes.

Cons:

1. Once a decision is made to combine two clusters, it is very hard to undo that. The algorithm is also sensitive to noise and outliers.
2. It can have difficulty handling different sized clusters and irregular shapes.
3. For a cho.txt and iyer.txt, HAC does a very bad job of clustering compared to other clustering algorithms.

Result Evaluation :

The running time of HAC is O(N^3) because they are N steps and at each step the size, N^2, distance matrix needs to be updated. This is very slow compared to K-Means.

Density Based (DBSCAN):

Algorithm Description

The implementation of this was based on DBSCAN. The algorithm we implemented is based on the pseudo-code that was provided in the slides. The *DBSCAN* function take in the set of data points which are ID's, distance matrix, eps value, and MinPts. The eps value is used to check all the points that are within a radius of *eps* from an object. In the DBSCAN function, we go through each unvisited point P in the dataset and first mark each point as visited. Then, we called a function called *regionQuery* that takes in the point, eps value, and the distance matrix as input and returns all the points that are within P's eps neighbourhood. After that we check to see if the size of the points that we got from the regionQurey function is less than MinPts, if it is then we mark this point as Noise, else we put the point the cluster.

Lastly, we have an expandCluster function that takes in the cluster that we just added to, the neighboring points, the point, the eps value, and the minpts. The expandCluster function first adds the point to the cluster. For each of the point in the neighboring points, if that particular point is not visited, then we mark that as visited. We now create call the regionQuery function again with the new point, and the eps value. This will return a new set of neighbouring points and now we check to see if the size of new neighbouring points is greater than equal to minpts, and if it is then this new neighbouring points will be joined with the old neighboring points. After that, we check if the point in the old neighbouring points is any member of any cluster, if its not then we add this point to the cluster C that we passed in as the input to the expand cluster function. We repeat this pattern until all points are assigned to a cluster.

Result Visualization:

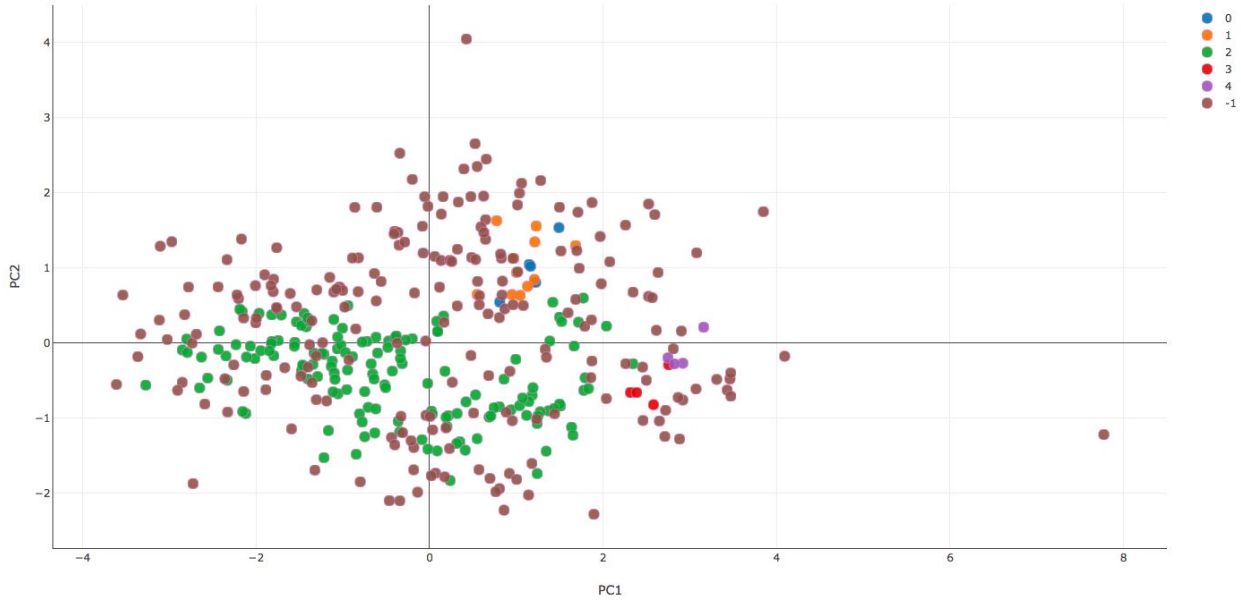
Cho.txt

For eps : 1.03 and for minPoints : 3 Jaccard is : 0.201645100774 Rand is : 0.554430991436
For eps : 1.03 and for minPoints : 4 Jaccard is : 0.203187062606 Rand is : 0.54761201643
For eps : 1.03 and for minPoints : 5 Jaccard is : 0.202069043068 Rand is : 0.523745603909
For eps : 1.03 and for minPoints : 6 Jaccard is : 0.209723404255 Rand is : 0.501422856989
For eps : 1.03 and for minPoints : 7 Jaccard is : 0.203350722311 Rand is : 0.467032672018
For eps : 1.04 and for minPoints : 3 Jaccard is : 0.20100587978 Rand is : 0.558578753792
For eps : 1.04 and for minPoints : 4 Jaccard is : 0.201572551823 Rand is : 0.550189266826
For eps : 1.04 and for minPoints : 5 Jaccard is : 0.200390075747 Rand is : 0.526725549679
For eps : 1.04 and for minPoints : 6 Jaccard is : 0.208744153363 Rand is : 0.516322585841
For eps : 1.04 and for minPoints : 7 Jaccard is : 0.204987066736 Rand is : 0.484294880399

We computed the jaccard and rand value for when eps = 1.03, and minPoints were from 3 to 7. We also computed jaccard and rand value for when eps = 1.04 and minPoints were from 3-7. We obtained the best jaccard and rand value when eps = 1.03 and minPoints = 4. Therefore,
Jaccard Value: 0.203187062606
Rand Value: 0.54761201643

Time to run for eps = 1.03 and minPoints = 4 is 0.0544278621674 seconds.

The PCA for cho.txt when eps= 1.03 and minPoints = 4



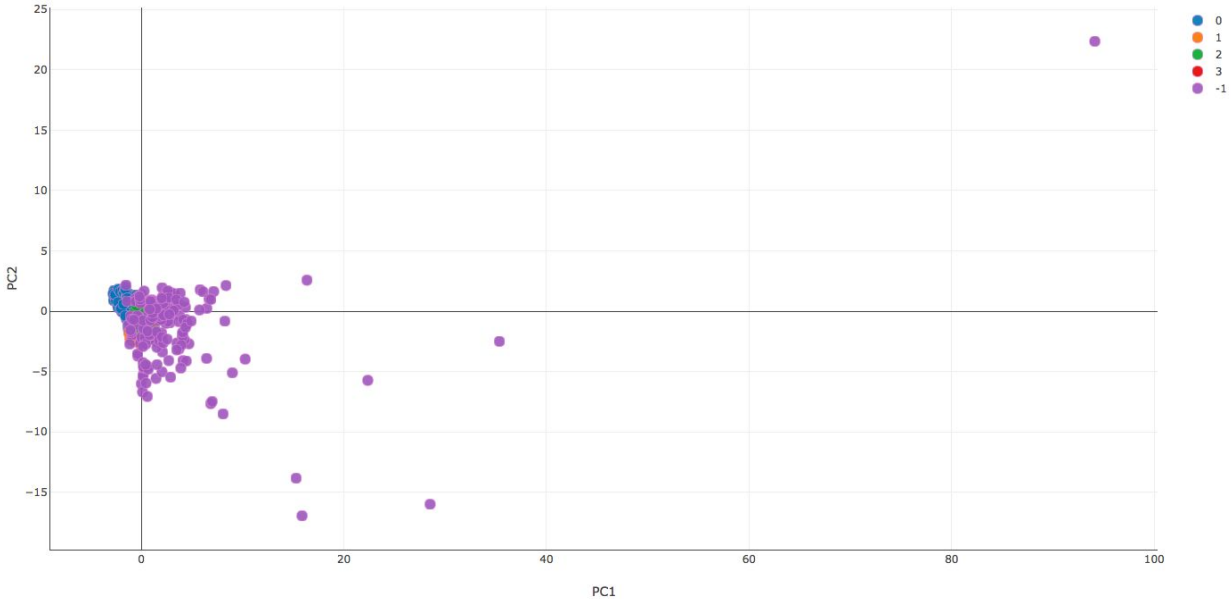
lyer.txt

For eps : 1.03 and for minPoints : 3 Jaccard is : 0.286011732759 Rand is : 0.662589930749
For eps : 1.03 and for minPoints : 4 Jaccard is : 0.284068935233 Rand is : 0.652323889124
For eps : 1.03 and for minPoints : 5 Jaccard is : 0.283833677472 Rand is : 0.650939619663
For eps : 1.03 and for minPoints : 6 Jaccard is : 0.283110534642 Rand is : 0.642267358552
For eps : 1.03 and for minPoints : 7 Jaccard is : 0.282564473223 Rand is : 0.63978315606
For eps : 1.04 and for minPoints : 3 Jaccard is : 0.248860628974 Rand is : 0.591184074167
For eps : 1.04 and for minPoints : 4 Jaccard is : 0.247902292866 Rand is : 0.581157473746
For eps : 1.04 and for minPoints : 5 Jaccard is : 0.282637820292 Rand is : 0.649428147062
For eps : 1.04 and for minPoints : 6 Jaccard is : 0.281932757396 Rand is : 0.640830711327
For eps : 1.04 and for minPoints : 7 Jaccard is : 0.281331539314 Rand is : 0.638309096147

We computed the jaccard and rand value for when eps = 1.03, and minPoints were from 3 to 7. We also computed jaccard and rand value for when eps = 1.04 and minPoints were from 3-7. We obtained the best jaccard and rand value when eps = 1.03 and minPoints = 4. Therefore,
Jaccard Value: 0.284068935233
Rand Value: 0.652323889124

Time to run for eps = 1.03 and minPoints = 4 is 0.134688138962 seconds.

The PCA for iyer.txt when eps = 1.03 and minPoints = 4



Pros and Cons:

Pros:

1. It is resistant to noise and can handle clusters of different shapes of sizes which the other two algorithms cannot handle.
2. It can also successfully identify outliers and this is shown in the PCA plots with -1.

Cons:

1. It cannot handle varying densities.
2. it is very sensitive to parameters. It is difficult to find the correct of Minpts and eps to obtain a good clustering. Even when we changed the eps value or the minpts value slightly, the results were very different. It is important to find the correct minpts and the eps value to output a good clustering.

Result Evaluation:The runtime of DBSCAN is O(n log n) if eps value is chosen in a meaningful value. However, the worst case runtime is O(n^2) if we don't pick the correct eps value.
Reference: https://en.wikipedia.org/wiki/DBSCAN

MapReduce K-Means

Implementation Details:

Initially we stored k centroids in a text file called “centroid.txt”. We wrote 3 python scripts to run our hadoop implementation :-

1) Mapper.py - Mapper takes in the input data line by line and for each row it emits centroid id, the point which belong to this centroid and the point attributes.

    Mapper -> emits(centroid_id, gen_id, attributes 1 ...2….3..)

2) Reducer.py - Reducer gets the sorted data from the mapper. It goes through each line and sum the attributes of each line which belongs to same centroid. After that, when it parses all rows of a centroid it emits the centroid_id and the list of all gen_ids which belong to this cluster. Reducer also updates the centroid.txt file with the mean of data. This updated text file is now used by the mapper to process second iteration.

    Reducer -> emits(centroid_id, list of gen_ids)

3) bash.py - This is a driver file which runs hadoop mapreduce multiple times until we reach a convergence. This convergence is found when the text file centroid.txt does not change in any iteration. We stop our program and then write the final results into a text file called “part-00000”.

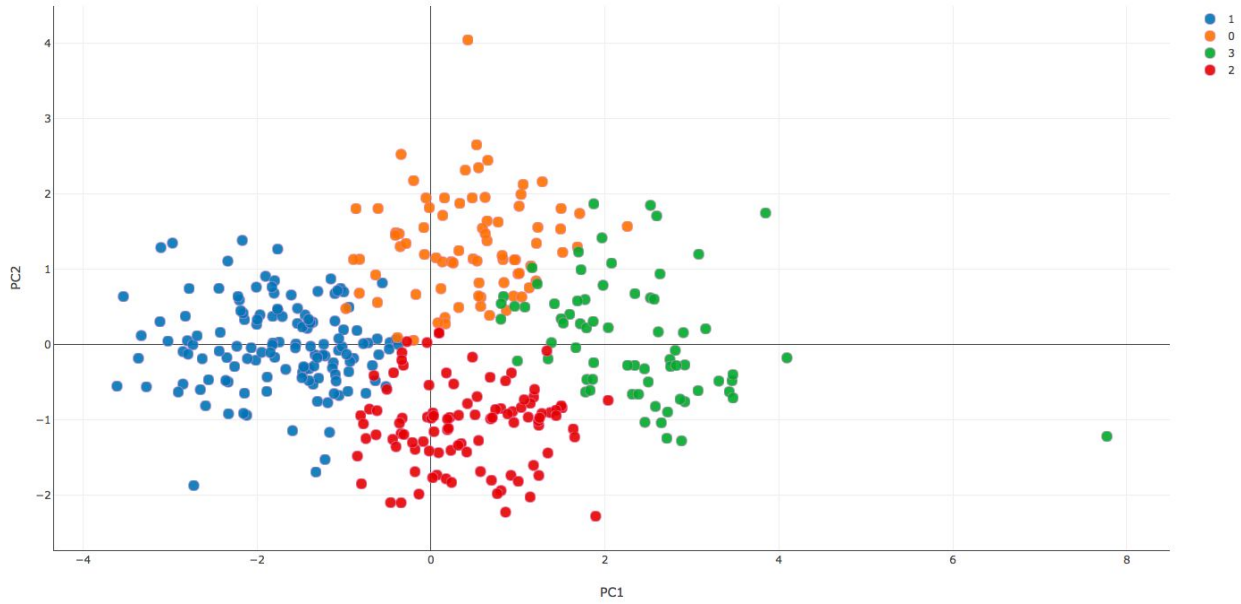
Result Visualizations:

Cho.txt

After some analysis on the MapReduce K-Means, we found that we get the best jaccard and rand value when we set **k = 4**.
Therefore, the jaccard value for when k = 4 is:
Jaccard: 0.423009053245.
Rand is: 0.798101962469

Time to run when k = 4 is: 59.27 seconds

The PCA for MapReduce K-Means for cho.txt when k = 4.

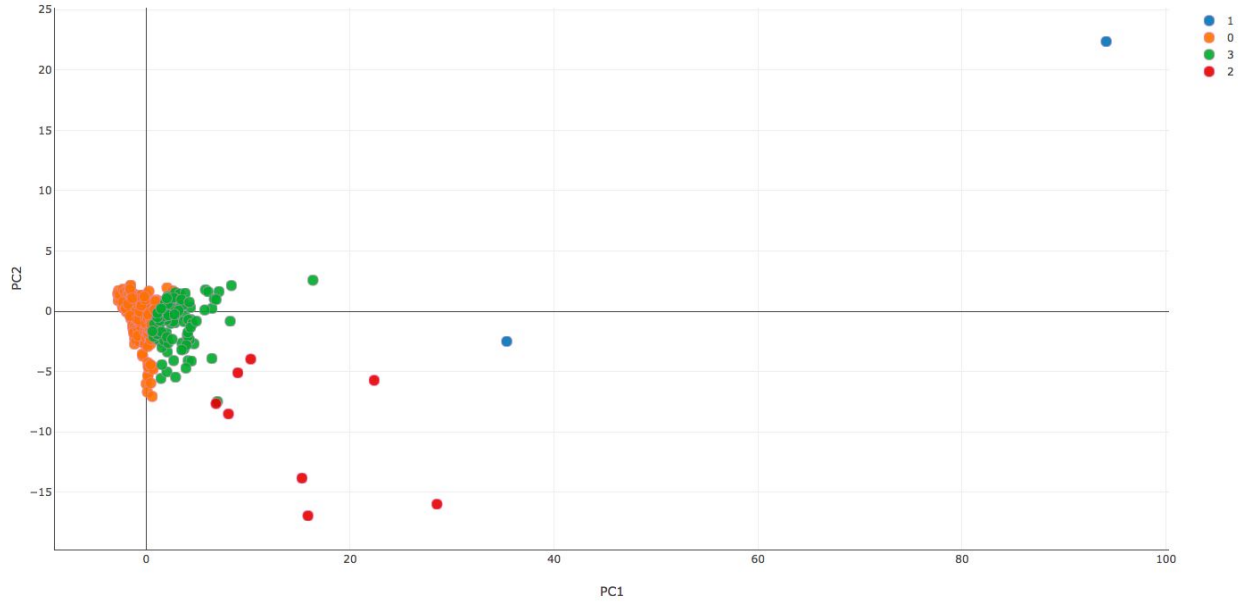


lyer.txt

After some analysis on the MapReduce K-Means, we found that we get the best jaccard and rand value when we set **k = 4**.
Therefore, the jaccard value for when k = 4 is:
Jaccard: 0.211123564805
Rand: 0.470722700897

Time to run when k = 4 is 171.5 seconds

The PCA for MapReduce K-Means for iyer.txt when k = 4.



Pros and Cons:

Pros:

1. The algorithm can run parallely compared to the normal K-Means it has to run non-parallely.

Cons:

1. MapReduce has no standard implementation of an iterative algorithm, but K-means is an iterative algorithm so MapReduce is not a great choice.
2. Since we are running MapReduce on a single node, we are not taking advantage of parallel processing. This compared with normal K-Means take a long time to run.

Result Evaluation :

The results of the normal k-Means and the MapReduce k-Means are very similar, as expected. The performance of parallel MapReduce K-means was worse than the the non-parallel K-Means as shown above with the time to took to run the algorithms. Since we are using text files to save the intermediate centroids and these text files have data in the form of string, we are converting these centroids from string to float and vice versa at each iteration. This makes it loose precision, and 3-4 points are getting incorrectly classified when compared with regular K-means.

Ideas for improving the performance of MapReduce K-means:

Combiners summarizes the map output records with the same key and the output of combiner is sent to actual reduce task as input. When the output from the mapper is very large and the transfer between mapper and reducer is very high combiner can limit the volume of data

transfer between map and reduce tasks. We tried to improve the performance by adding a combiner. However, it was a little complicated and we couldn't get the correct results but we tried to use a combiner.

MapReduce K-means can also be improved by using an iterative MapReduce framework that natively supports iterative computations, for example twister.

Result Evaluation of all Algorithms/Our Findings

On the basis of Performance:

Cho.txt
K-means: 2.13 sec
HAC(Min): 8.12 sec
Density based(DBSCAN): 0.05 sec
MapReduce K-means: 59.27 sec

Iyer.txt
K-means: 19.2 sec
HAC(min): 19.48 sec
Density based(DBSCAN): 0.13 sec
MapReduce K-means: 171.5 sec

When we compare the performance all the clustering algorithms, it is clear that density based (DBSCAN) algorithm is the fastest among all of them. This is expected because the runtime of DBSCAN is O(n log n), if we choose a good eps value. The slowest of all the algorithms is MapReduce K-means because MapReduce does not perform well on iterative algorithms and we are using a single node which does not make use of the parallel structure of MapReduce.

On the basis of Cluster Classification:

Jaccard values:

Cho.txt
K-means: 0.41
HAC(Min): 0.22
Density based (DBSCAN): 0.20
MapReduce K-means: 0.42

Iyer.txt
K-means: 0.35
HAC(Min): 0.15

Density based (DBSCAN): 0.28
MapReduce K-means: 0.21

When we compare all of the algorithms, the results for HAC were the worst on cho.txt and iyer.dataset based on the visualization using PCA and comparing their jaccard coefficients as stated above. HAC is very sensitive to outlier and for this input data for k = 5, we were getting 4 clusters of size 1 and the just one cluster that contains the result of the points. K-means did the best job at clustering as it has the highest jaccard coefficient and it produces the best visualization using PCA, but it is still sensitive to outliers. The DBSCAN is still better in identifying outliers, but it gave us a lower jaccard coefficient as compared to K-means. The jaccard coefficient of the MapReduce K-means and the normal K-means are similar and this is expected because the underlying algorithm is still the same.

<u>On the basis of ease of implementation:</u>

K-means was the easiest algorithm to implement among all of them and it doesn't require space because we only store the last centroid and we don't maintain a distance matrix.

For HAC we need to maintain a distance matrix which takes some space and we need to update it at iteration when we are merging two clusters. It was also a bit complicated to implement than K-means.

For DBSCAN we don't need a distance matrix, and we follow a complex algorithm of three functions to implement it.

For MapReduce K-means it was the most difficult to implement because Hadoop does not supportive iterative algorithms and we had to find out a way to make it work in an iterative manner.