Clustering Algorithms

Team No : 16

Members:

Jay Shah – 50205647 (jaynaren)

Yash Jain – 50206851 (yashnavi)

Akshay Shah – 50206543 (akshaybh)

K-Means

Implementation

Reading a file

File path is accepted as standard input. Values are extracted and stored as gene_id to gene_data dictionary, cluster to list of gene_ids dictionary and gene_id to cluster dictionary.

Initializing clusters with initial centroids

Initial centroids (gene_id) are accepted as input. change_clusters() is a method which assigns cluster with nearest centroid for each point.

Performing k_means

Loop is executed until convergence or for fixed number of iteration

- centroids are calculated from existing clusters using method calculate_centroids()
- reassign points to clusters according to new centroids using the same method change_clusters()
- calculate value of jaccard coefficient for keeping track of maximum value of jaccard coefficeint using find_jaccard()

Final Steps

Calculate final value of jaccard coefficient using find_jaccard().
Print file to create input pca_and_plot() of program implemented in PA1. Graph is plotted after performing PCA.

Method change_clusters(data, cluster_to_point_list, centroids)

for all points, Euclidean distance from all centroids all calculated, and point is then assigned a cluster with shortest distance to centroid.

Method calculate_centroids(data, cluster_to_point_list)

finding sum of all features of points in same cluster and dividing by total number of points in that cluster i.e. calculating average of value features of points in same clusters.

Method find_jaccard(point_to_cluster, actual_cluster)

It calculates the external index (Jaccard) using the given ground truth and resultant clusters obtained in the flow. For any two points, if both lies in same cluster as per ground truth and in same clusters as per result, we mark it as matched pair, else if one of the points lies in same cluster as per ground truth and result, but other point lies in same cluster in one case and different in other, we mark it as unmatched pair. Finally, we calculate Jaccard coefficient as ratio of match pairs to total pairs.

Pros

- Implementation of K-means algorithm is easy.
- Algorithm is fast.
- Algorithm is scalable.

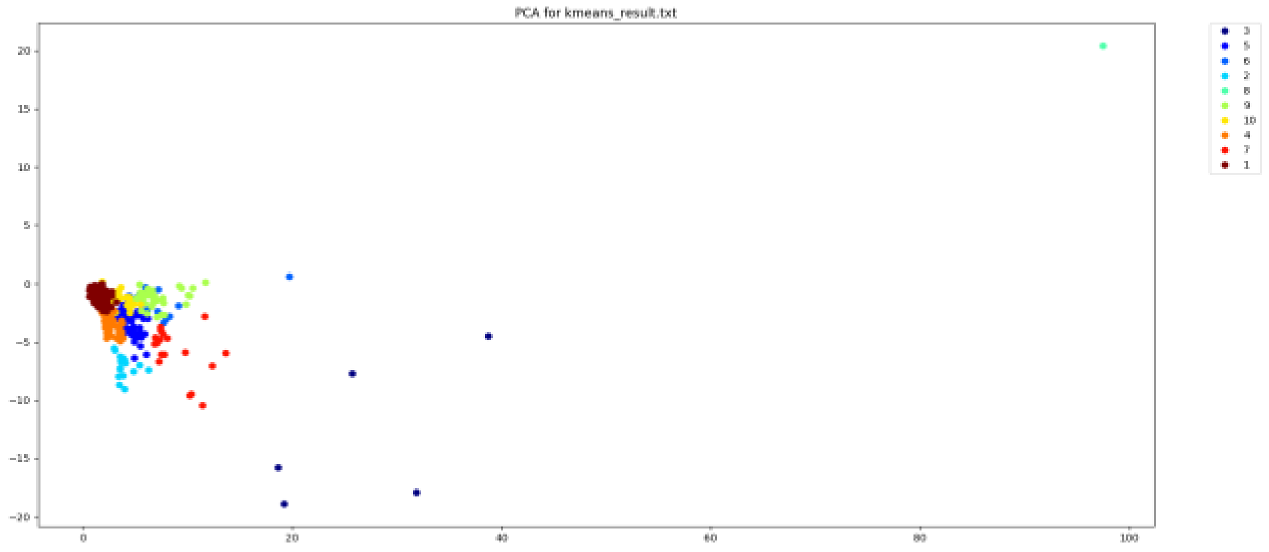
Cons

- Need to know number of clusters priori.
- Works good only for certain shape (hyper-spherical) and size of clusters.
- Algorithm output depends on initial centroids.
- Outliers and noise is not handled.

Evaluation and Results

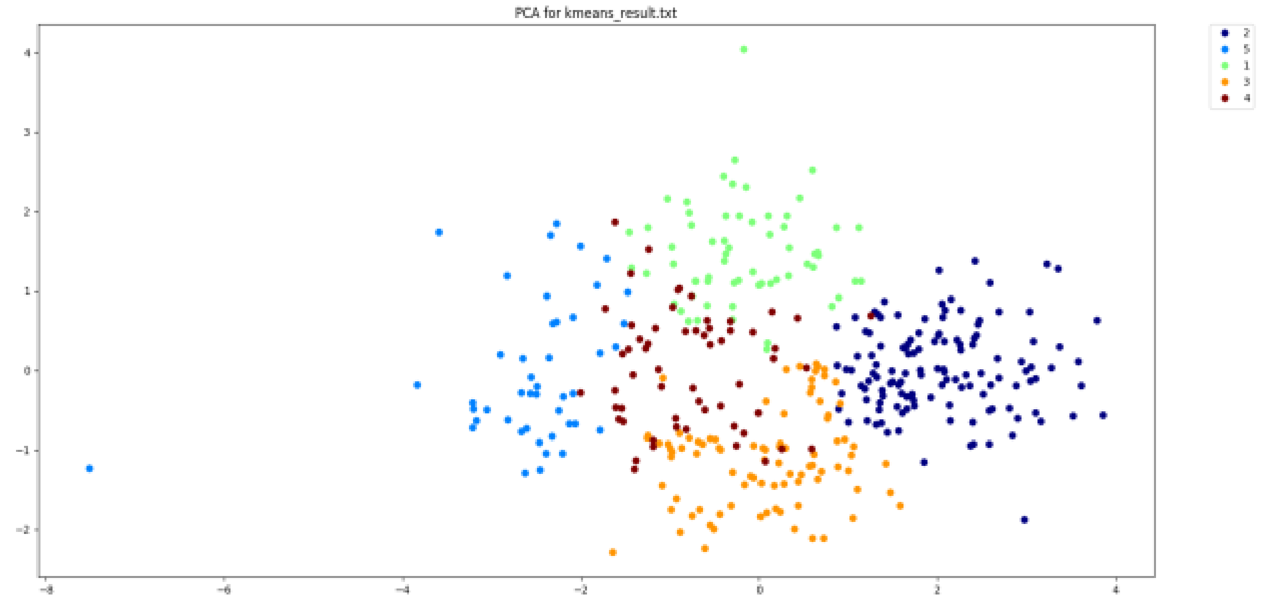
For iyer.txt k = 10, initial centroids: 1,2,3,4,5,6,7,8,9,10

Max Jaccard co-efficient	0.3595759607140073
Final Jaccard co-efficient	0.33329534554640683
Number of iterations to converge	29



For cho.txt k = 5, initial centroids: 1,2,3,4,5

Max Jaccard co-efficient	0.37609587388401833
Final Jaccard co-efficient	0.37609587388401833
Number of iterations to converge	23



HAC (Hierarchical Agglomerative Clustering)

Implementation

Code flow

- Take Filename and required number of cluster as input from user.
- We start with considering each point as a cluster and continue till number of cluster becomes equal to required number of cluster.
- We calculate initial distance matrix by calculate Euclidean distance between each point.
- We pick the minimum and merge two to form a single cluster and update distance by picking the minimum.
- Repeat above step till number of cluster becomes equal to required number of cluster.
- Calculate the jacard coefficient of the resultant clusters using given ground truth.
- Plot the result.

Method fileRead (file name)

Read the input file from users.
Takes epsilon and minpoints as input from user and calls the DBScan algo.
Finally, it prints the clusters and plots it.

Method makeInitialDistanceMatrix()

Creates the initial distance matrix by calculating euclidean matrix between each points

Method mergeNext()

Find the minimum distance from the matrix and merges those two clusters and updates corresponding distance by taking the minimum distance between two points from either clusters update the number of clusters.

Method updateMinMatrix(cluster I, cluster j , distance)

It checks for current minimum distance for cluster i and j. If given distance is less than current stored distance, it updates.

Method getMinGeneIds()

It finds minimum from minmatrix where key is each cluster and value is min distance to any cluster

Method getDistance(point i, point j)

It calculates the euclidean distance between the given two points using the feature given.

Method getJaccard()

It calculates the external index (Jaccard) using the given ground truth and resultant clusters obtained in the flow. For any two points, if both lies in same cluster as per ground truth and in same clusters as per result, we mark it as matched pair, else if one of the points lies in same cluster as per ground truth and result, but other point lies in same cluster in one case and different in other, we mark it as unmatched pair. Finally, we calculate Jaccard coefficient as ratio of match pairs to total pairs.

Method printClusters()

It creates a matrix for each cluster, and plots the output using imported PCA implementation

Pros

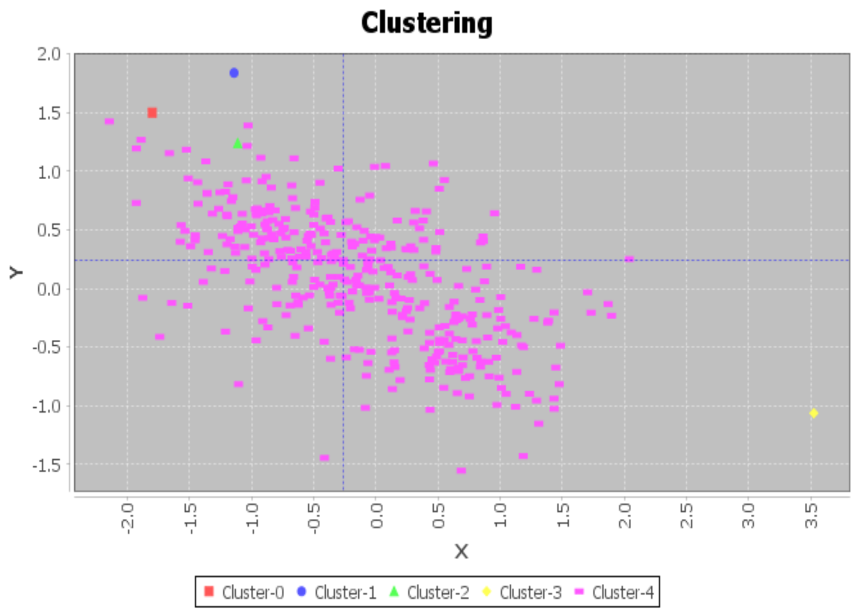
- Algorithm can be run for any desired number of clusters
- We can get similar results on multiple runs as opposed to K means.
- It can produce an ordering of the objects, which may be informative for data display.
- Smaller clusters are generated, which may be helpful for discovery.

Cons

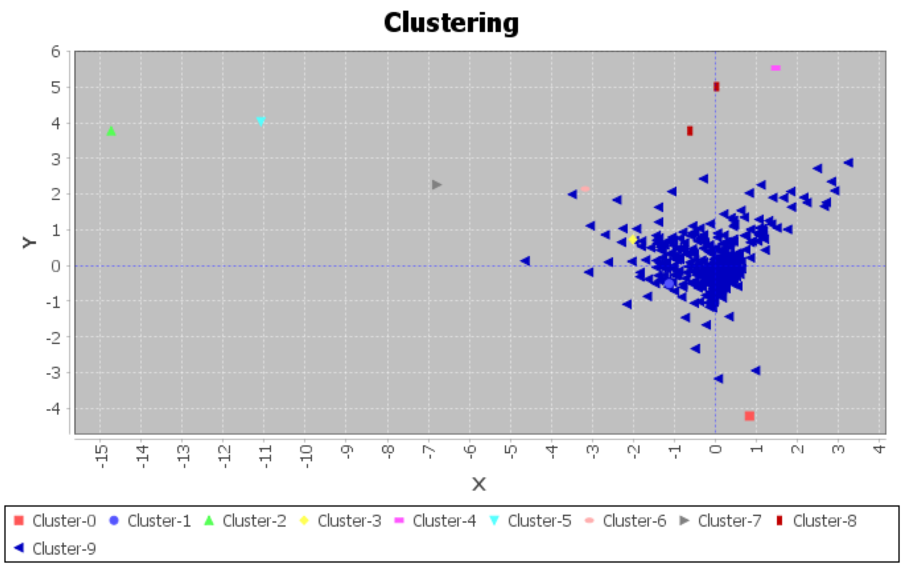
- Once we combine two clusters, it cannot be undone.
- No objective function is directly minimized
- Sensitivity to noise and outliers
- Difficulty handling different sized clusters and irregular shapes
- sometimes difficulty breaking large clusters

Evaluation and Results

For cho.txt, number of clusters = 5
Jaccard coefficient = 0.22839497757358454



For iyer.txt, number of clusters = 10
Jaccard coefficient = 0.15824309696642858



Density Based Clustering (DBScan)

Implementation

Code flow

- For each unvisited point, find neighbours for that point at epsilon radius.
- Mark point as noise if no of neighbours less than minpoints
- else find neighbours of each neighbour point and form these as a clusters.
- Calculate the jacard coefficient of the resultant clusters using given ground truth.
- Plot the result.

Method fileRead()

Read the input file from users.
Takes epsilon and minpoints as input from user and calls the DBScan algo.
Finally, it prints the clusters and plots it.

Method DBScan(filename, epsilon radius, minpoints)

For each point i.e. not visited, it marks it as visited and finds all neighbors for that point at epsilon radius.
If number of neighbors is less than minpoints, then we classify it as outlier(noise)
else we included it in a cluster and expandCluster for that point.

Method expandCluster(point, neighbors, epsilon, minpoints)

For each neighbor point, if it is not visited we calculate its neighbor points.
And if these neighbors are more than minpoints, we add them to current cluster and expand the clusters even more.

Method findNeighbors(point I, epsilon)

For each point we calculate the Euclidean distance from the given point.
And if the distance is less than epsilon radius, we add it to the collection of neighbors.
If this neighbor point is not assigned to any cluster before, we add it to current cluster.

Method getDistance(point i, point j)

It calculates the Euclidean distance between the given two points using the feature given.

Method getJaccard()

It calculates the external index (Jaccard) using the given ground truth and resultant clusters obtained in the flow. For any two points, if both lies in same cluster as per ground truth and in same clusters as per result, we mark it as matched pair, else if one of the points lies in same cluster as per ground truth and result, but other point lies in same cluster in one case and different in other, we mark it as unmatched pair. Finally, we calculate Jaccard coefficient as ratio of match pairs to total pairs.

Method printClusters()

It creates a matrix for each cluster, and plots the output using imported PCA implementation
(https://github.com/CSE601-DataMining/Clustering/blob/master/src/edu/buffalo/cse/clustering/PCA.java)

Pros

- DBScan handles the outliers (i.e. noisy data)
- No need to give number of clusters as input
- DBScan works well if the cluster to be formed are of different shapes and size.
- Insensitive to order of data to great extent.

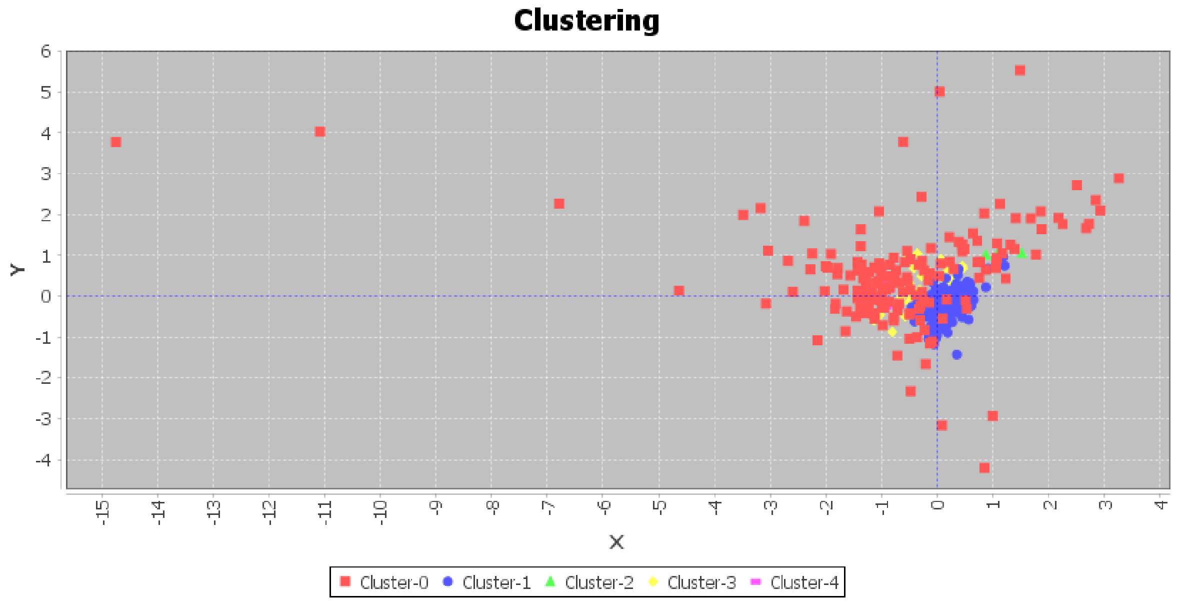
Cons

- It is very sensitive to parameters.
- Sometimes, it doesn't work properly in separating nearby clusters accurately.
- Cannot handle varying densities
- Requires connected regions of sufficiently high density.

Evaluation and Results

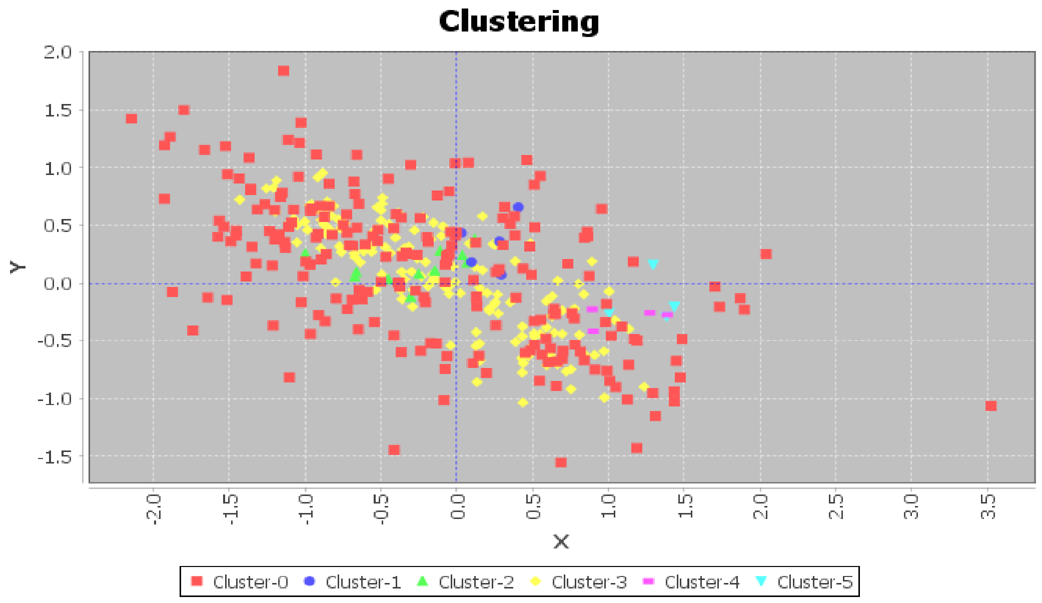
For iyer.txt, Epsilon radius = 1.05, min points = 5,

No of clusters formed = 4 Jaccard coefficient = 0.2828229375177354



For cho.txt, Epsilon radius = 1.03, min points = 4,

No of clusters formed = 5 Jaccard coefficient = 0.20318706260639305



K-MEANS IMPLEMENTATIONS USING HADOOP AND MAP-REDUCE

Implementation

Setup: We have setup a single node Hadoop cluster on a 32-Bit linux virtual machine.

Step 1: Reading File:

We read the file initially to populate the genepool map and create the ground truth cluster.

Step 2: Mapper Phase:

- We picked random initial clusters (Iyer:10 and Cho: 5) if not previously initialized.
- We have the dataset line by line and parse each line to the corresponding data point.
- For each data point, we find out the nearest centroid buy calculating distance with each centroid. And send the closest centroid and the data point as a pair.

Step 3: Reducer Phase:

- Hadoop combines these pairs and send it to the reducer in form of Key(Centroid) and a List of data points in the cluster.
- For each list, we compute the new centroid by finding the average of all coordinates.
- Similarly, we repeat the same process for all the clusters in the map function.

Step 4: Cleanup and Convergence:

- After every reduce function we update the old centroid list with the new centroids.
- Whenever we find that the list of centroids is same we know that we have converged to a result and we stop.

Step 5: Print results:

- We print out the results in 2 files.
- FinalOutput.txt: Primary output containing the number of iterations, time taken, cluster information and Jaccard Index values.
- ClusterOutput.txt: Consists of the gene & the cluster its allocated to for visualization.

Method setup(Context)

It checks if the centroid list is initialized or not. If not, it picks random centroid points.

Method map(Object, Text)

Input: Key(Object), Value(Text)
Output: Key-Centroid(Text), Value-Data Point(IntWritable).
This function read the line from the file and generates the data point and computes the nearest centroid and allocates itself to the cluster and returns out a pair of
(centroid, data point[RowID]) for the reducer function.

Method reduce(Text, Iterable<IntWritable>)

Input: Key(Text), Value-List of Data Points(Iterable<IntWritables>)
Output: Key-Cluster No(Text), Value-Data Point List(Text).
This function collects the aggregated list of data points in the cluster and computes the new centroid and adds it to the new Centroid List. It also returns the new clusters onto the output file.

Method cleanup(Context) implementation
After every iteration, this function checks for the convergence condition. If the current centroid list is same as the old centroid list, we stop.

Method calculateDistance(Centroid, Data) implementation
This function computes the Euclidean distance between the data point and the centroid and returns a double value.

Method getJaccard()
It calculates the external index (Jaccard) using the given ground truth and resultant clusters obtained in the flow. For any two points, if both lies in same cluster as per ground truth and in same clusters as per result, we mark it as matched pair, else if one of the points lies in same cluster as per ground truth and result, but other point lies in same cluster in one case and different in other, we mark it as unmatched pair. Finally, we calculate Jaccard coefficient as ratio of match pairs to total pairs.

Pros
- Map Reduce and Hadoop is scalable, and can accommodate large number of files and can be distributed over multiple clusters.
- Task Independency hence easy to handle partial failures.

Cons
- Slower than normal K-means: Due to the single node setup, and data being of small size, Hadoop provides no advantages and actually slows down the algorithm due to a generation of a lot of pairs.

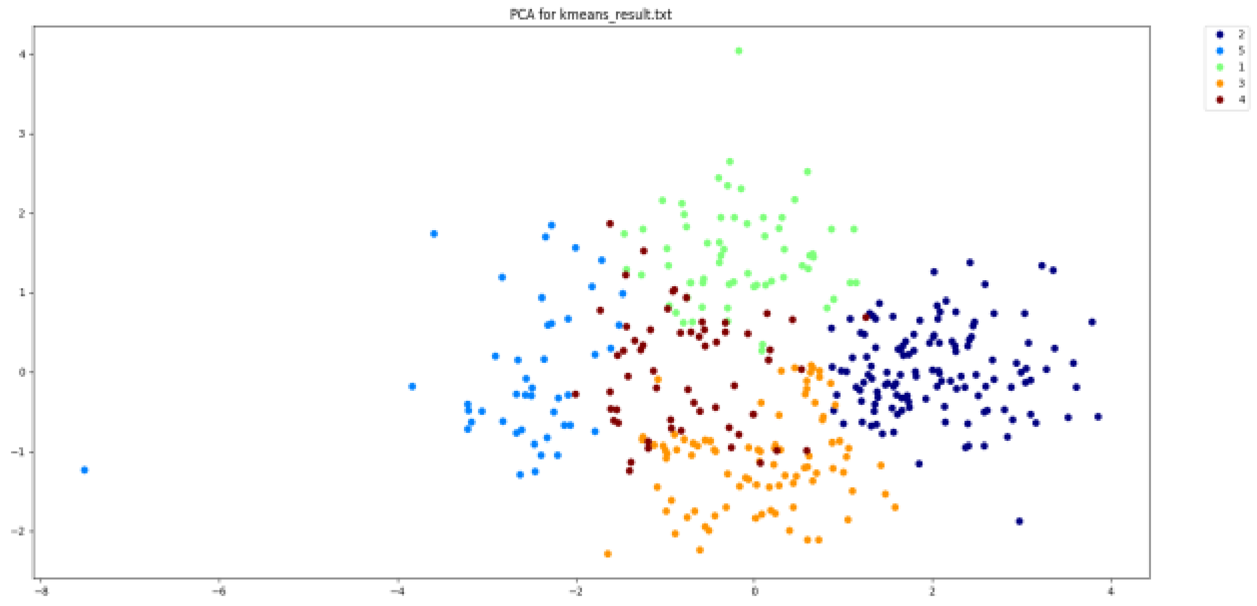
Efforts to Improve Running Time of Hadoop
- Created a HashMap of all the genes, mapping their RowIDs to the corresponding attributes. This HashMap allowed us to just write the pair (Centroid, RowID) in the map phase. The RowID of gene is sent instead of the whole feature set, which will reduce the memory overhead.
- Because the intermediate centroid list was small enough to fit in the memory we considered storing it in a list rather than writing into intermediate file again and again.

Evaluation and Results

For cho.txt

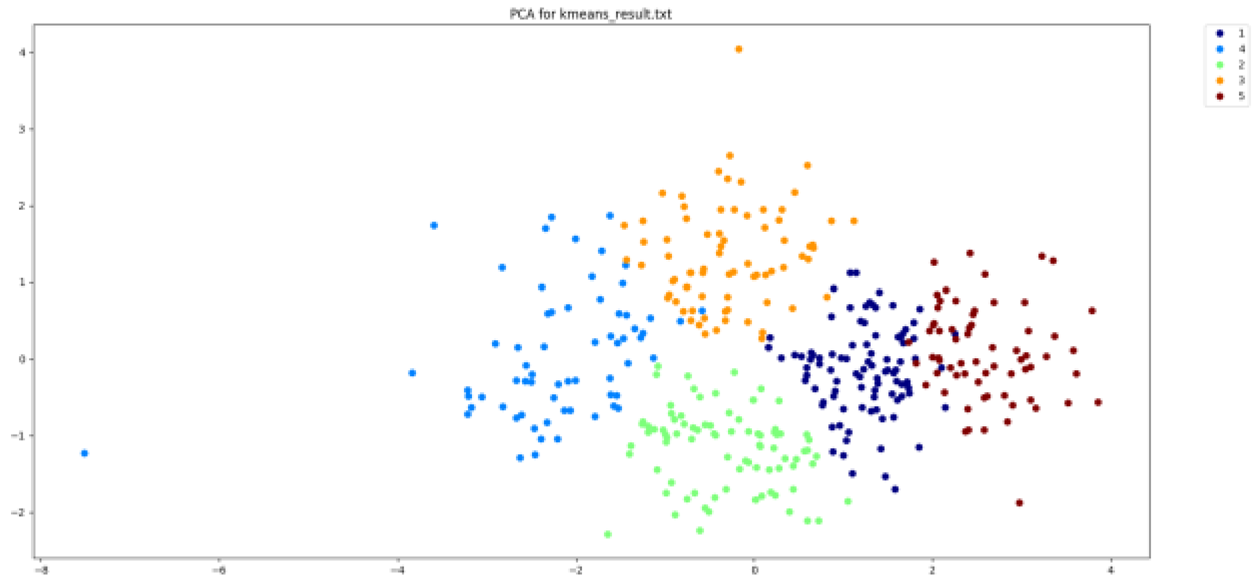
k = 5, initial centroids: 1,2,3,4,5

Max Jaccard co-efficient	0.37609587388401833
Final Jaccard co-efficient	0.37609587388401833
Number of iterations to converge	23



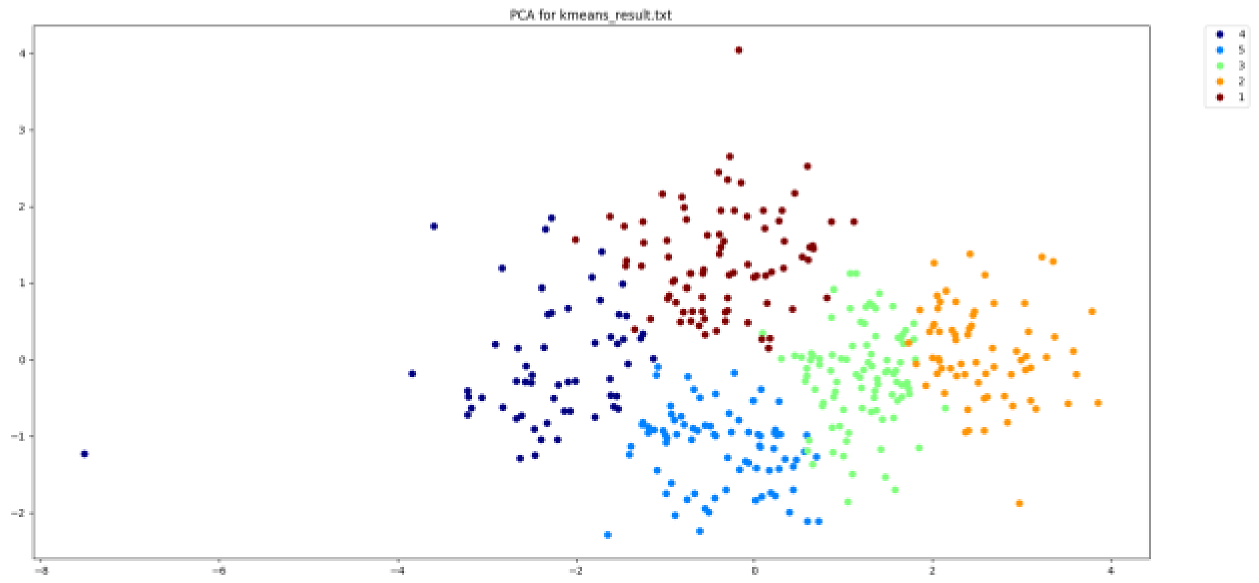
k = 5, initial centroids: 113, 327, 40, 59, 195

Max Jaccard co-efficient	0.41097241719467315
Final Jaccard co-efficient	0.33384140061791967
Number of iterations to converge	12



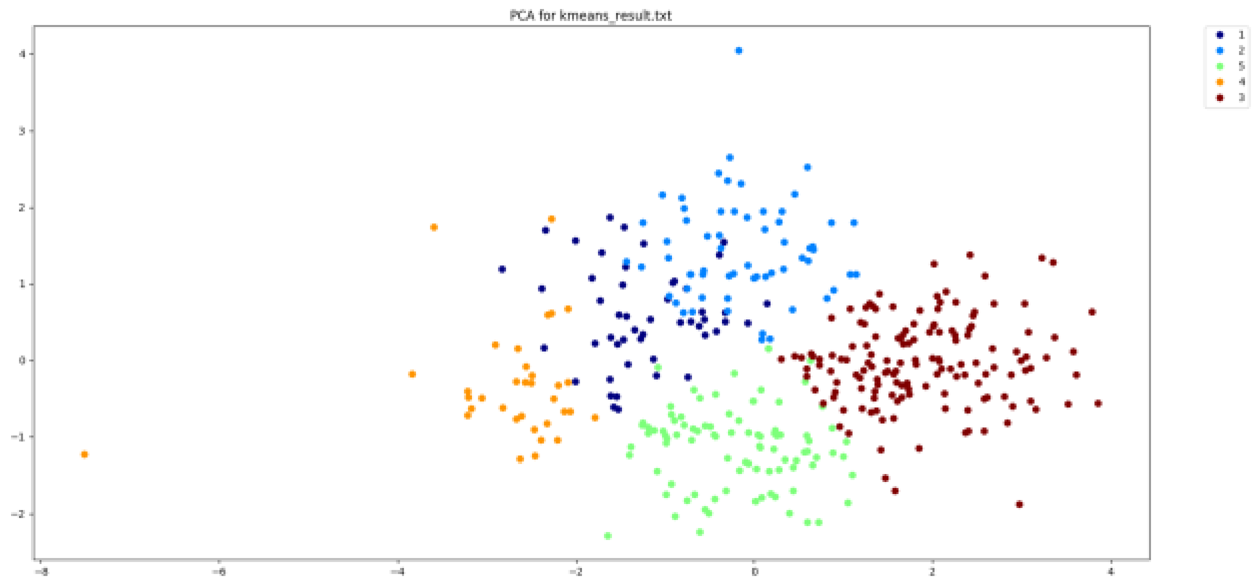
k = 5, initial centroids: 67, 77, 217, 256, 330

Max Jaccard co-efficient	0.39779596200501666
Final Jaccard co-efficient	0.33791084168419316
Number of iterations to converge	12



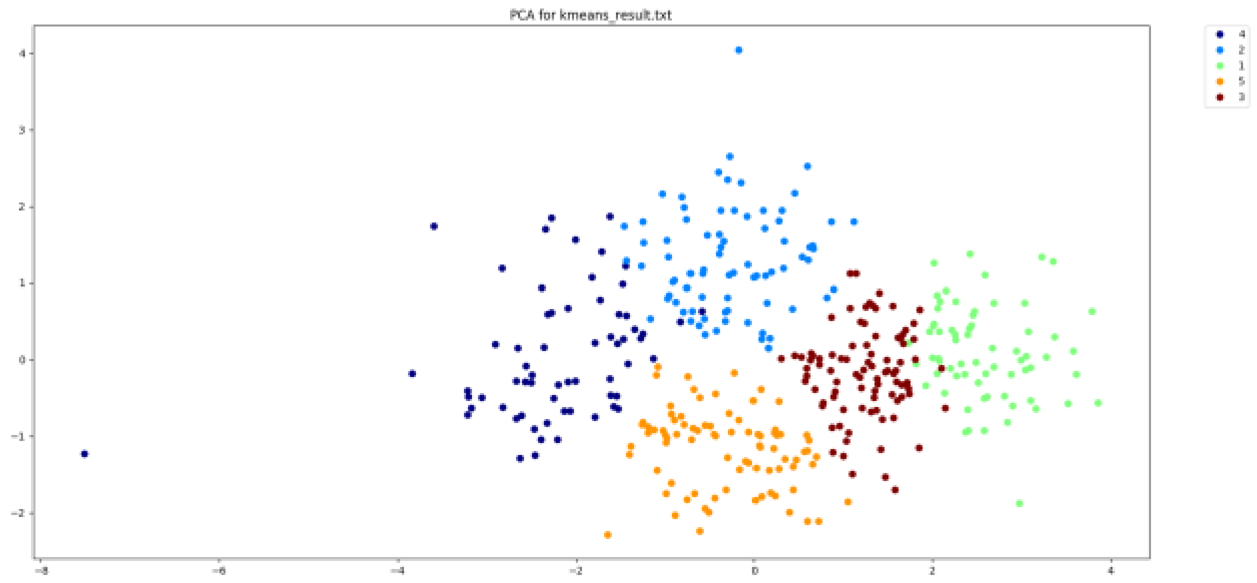
k = 5, initial centroids: 45, 50, 187, 368, 382

Max Jaccard co-efficient	0.41900715915348913
Final Jaccard co-efficient	0.41900715915348913
Number of iterations to converge	12



k = 5, initial centroids: 91, 154, 179, 237, 251

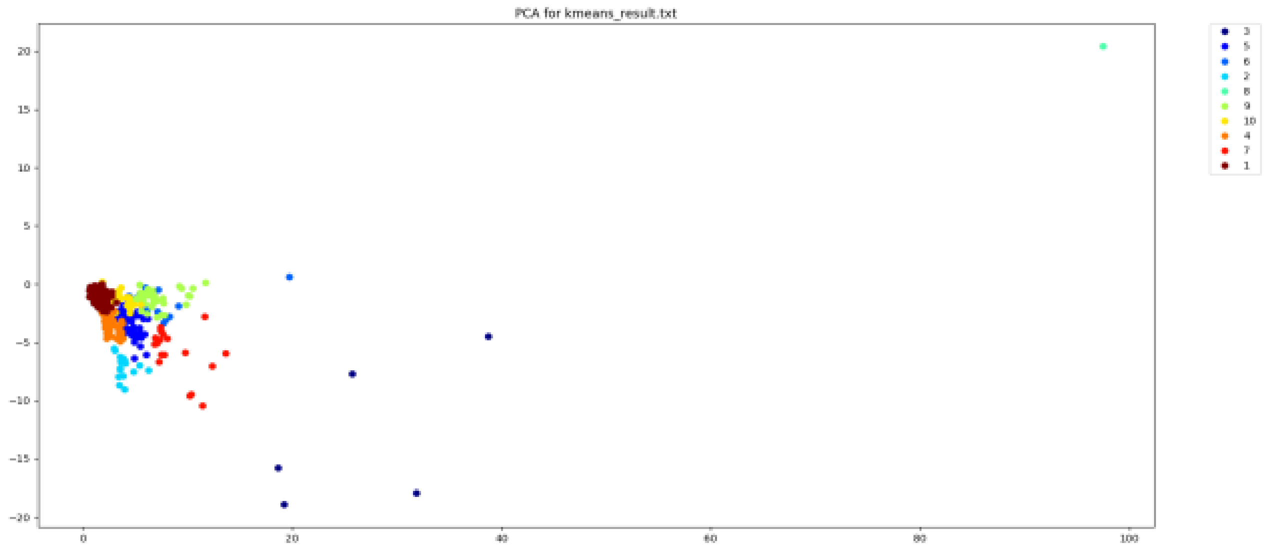
Max Jaccard co-efficient	0.3339798613403764
Final Jaccard co-efficient	0.330135068357766
Number of iterations to converge	15



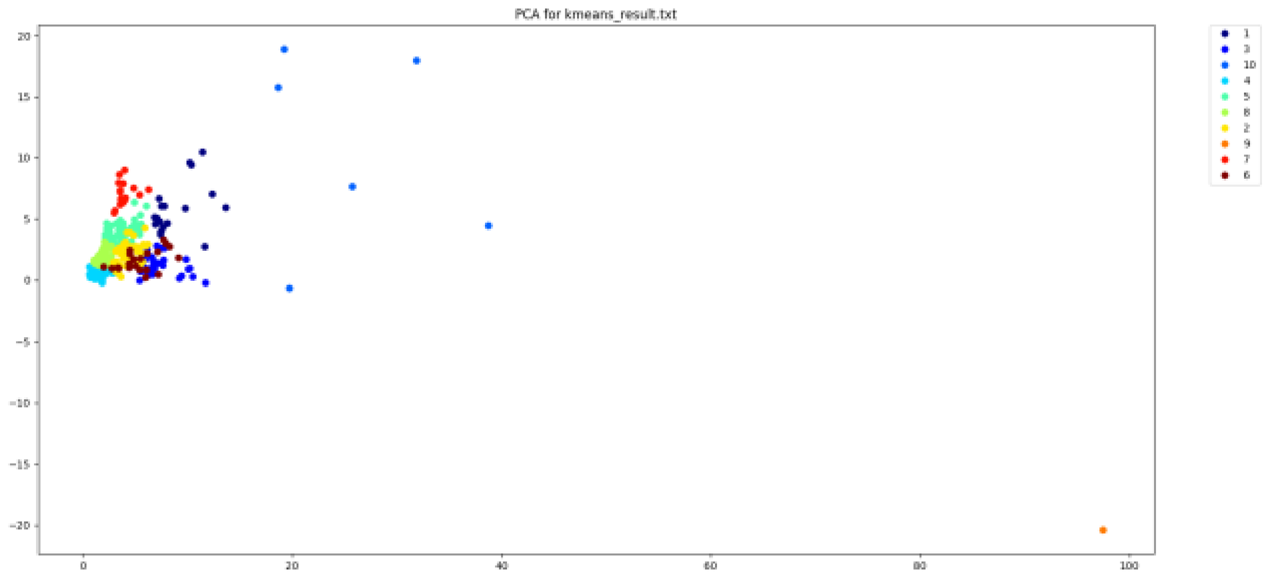
For iyer.txt

k = 10, initial centroids: 1,2,3,4,5,6,7,8,9,10

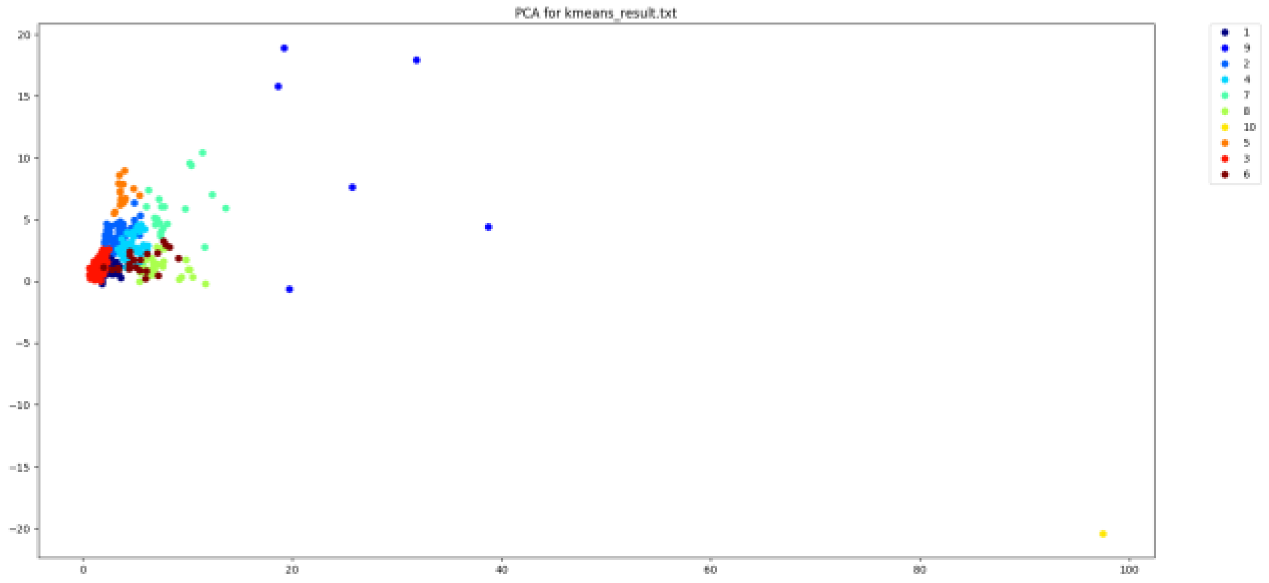
Max Jaccard co-efficient	0.3595759607140073
Final Jaccard co-efficient	0.33329534554640683
Number of iterations to converge	29



k = 10,	initial centroids: 89, 139, 144, 159, 177, 219, 222, 316, 362, 452
Max Jaccard co-efficient	0.3166128966941591
Final Jaccard co-efficient	0.29475720780068604
Number of iterations to converge	43

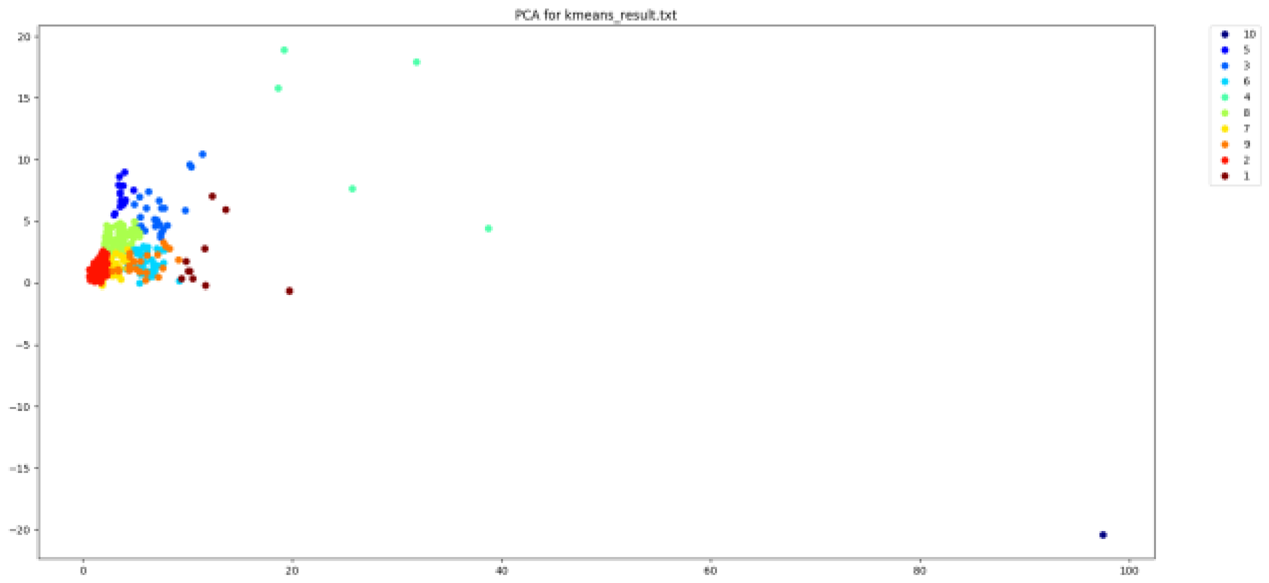


k = 10,	initial centroids: 48, 210, 230, 292, 338, 345, 453, 474, 489, 491
Max Jaccard co-efficient	0.43707921015769735
Final Jaccard co-efficient	0.35296757428067504
Number of iterations to converge	50



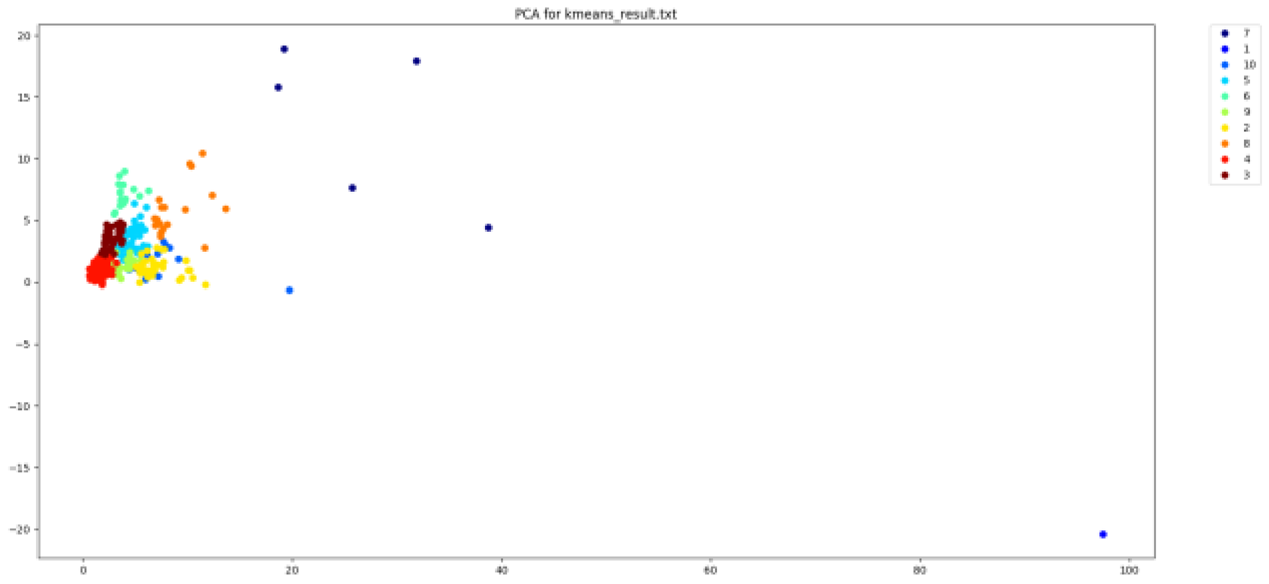
k = 10, initial centroids: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Max Jaccard co-efficient	0.42566266949937975
Final Jaccard co-efficient	0.3442832806514825
Number of iterations to converge	46



k = 10, initial centroids: 5, 10, 15, 20, 25, 30, 35, 40, 45, 50

Max Jaccard co-efficient	0.33328766721139824
Final Jaccard co-efficient	0.33216132172663776
Number of iterations to converge	39



From the above multiple results, we can see KMeans output varies with initial centroid selection.

Comparing K-Means, HAC and DBScan

- For any given K, we can see from above result that K-means performs better than HAC, but it might depend on initial centroid selection.
- From what we experimented, we could see K-means performs better than HAC for every random initial centroid selection.
- While DBScan does not depend on the shape of clusters, K-Means and HAC works good only for certain shape and size of cluster.
- K-Means algorithm is scalable and easy to implement as compared to HAC.
- DBScan does not require number of clusters as opposed to K-Means and HAC.
- Outliers and noise are best handled in DBScan than in HAC or K-Means.
- DBScan is more sensitive to parameters than HAC and K-Means.
- DBScan does not work for varying density of data.

Comparing K-Means with Python and K-Means with Hadoop

- From results displayed above, both implementations result in same value of external index (Jaccard coefficient) and same cluster visualization.
- Execution takes more time in Hadoop for given datasets, but Hadoop is scalable and will be fast if data is large and multiple nodes are utilized for execution.