Clustering Algorithms

Areebuddin Aatif Mohammed Khaja (areebudd - 50289734)
Mohammed Abdul Kamran (m37 - 50290758)

1. K-Means Algorithm

1.1 Introduction

K-means is a center-based, iterative partition method for clustering where a cluster is defined as a set of objects such that an object in a cluster is closer (more similar) to the "center" of a cluster, than to the center of any other cluster. Let 'n' be the number of data points and 'k' be the number of desired clusters, the K-means clustering algorithm aims to partition all the n points into k clusters where each point belongs to the cluster with the nearest mean which represent each cluster. This process is repeated until all the points are assigned to clusters and further iteration will not result in change in assignment of the cluster.

1.2 Implementation

The overall implementation of the K-means algorithm is as follows:

- First, 'k' – the desired number of clusters – is taken as an input from the user.
- Next, k points are taken from the dataset to act as initial centroids of each cluster. These points can be taken at random or pseudo-randomly i.e., taking one random point from each cluster. The latter approach is used in the implementation to improve performance.

```
def initializeCentroids(self, dataset, k):
    centroids = list()
    cluster = 0
    print("Enter {} number of initial centroids: ".format(k))
    while cluster < k:
        inputs = list(map(float,input().split()))
        centroids.append(inputs)
        cluster+=1
    return centroids
```

- Then, the following steps are performed iteratively:
    - For each point in the dataset, its distance from all the centroids is calculated and the point is assigned to the cluster whose centroid is closest to the point.

```
def find_cluster(self, centroids, gene, clusters):
    min_dist = float('inf')
    cluster = 0
    for i,centroid in enumerate(centroids):
```

```
            dist = distance.euclidean(gene.point, centroid)
            if dist < min_dist:
                min_dist = dist
                cluster = i+1
        gene.cluster = int(cluster)
        clusters[cluster].append(gene.point)
        return clusters
```

- After all the points are assigned to a cluster, new centroids are calculated, by calculating the mean of each cluster.

```
def findClusterCentroid(self, centroids, clusters):
    for i,key in enumerate(clusters):
        centroids[i] = np.array(clusters[key], dtype=np.float64).mean(axis=0)
    return centroids
```

- The above steps are performed for user-specified maximum number of iterations.

1.3 Advantages:

- Easy to implement.
- For low values of k, where k is the number of clusters, the algorithm runs very efficiently with a time complexity of O(t * k * n) where t is the maximum number of iterations and n is the number of data points.
- Closely packed clusters are formed if the clusters are globular.

1.4 Disadvantages:

- It is difficult to predict the value of k and therefore should be passed as input.
- The algorithm depends heavily on the initial centroids that are selected and different initialization results in different output.
- Difficult to find clusters of non-globular shapes.
- Can give empty clusters.

2. Hierarchical Clustering (Min Approach)

2.1 Introduction

Hierarchical clustering is classified into two types:

a) Agglomerative
b) Divisive

We will be implementing agglomerative hierarchical clustering technique in our report. In this technique we start with assuming each data point as an individual cluster. In each iteration we find the two closest clusters and merge them until there is only one cluster left or k clusters are formed.

The input to the algorithm is a dataset and a value N defining the number of clusters.

2.2 Implementation

The following algorithm is followed:

- Compute the distance matrix.

  Below image shows the code to get the initial distance matrix which shows the similarity between each cluster. We used the scipy.spatial.distance.cdist() to calculate the distance matrix using Euclidean distances on the input dataset.

```
def getDistanceMatrix(self):
    # To calculate the initial distance matrix
    distanceMatrix = spatial.distance.cdist(self.dataMatrix, self.dataMatrix, metric='euclidean')
    distanceMatrix = pd.DataFrame(distanceMatrix, index=self.geneIds, columns=self.geneIds)
    return distanceMatrix
```

- Let each data point be a cluster.
- Repeat
  1) Merge the two closest clusters.

     Here we find the two closest clusters having the smallest distance in the distance matrix. Below code shows the implementation of merging two clusters.

```
# finding the row and column with min distance value
minclusterIndex1, minClusterIndex2 = clusterMatrix.stack().idxmin()
# Deleting the row and column
clusterMatrix.drop([minclusterIndex1, minClusterIndex2], axis = 1, inplace = True)
clusterMatrix.drop([minclusterIndex1, minClusterIndex2], inplace = True)

# new index
newClusterIndex = minclusterIndex1 + "-" + minClusterIndex2
# creating new row
rowDf = pd.DataFrame(np.full((1, clusterMatrix.shape[1]), np.inf), columns=clusterMatrix.columns)
rowDf.rename(index={0:newClusterIndex}, inplace=True)
# creating new column
colDf = pd.DataFrame(np.full((clusterMatrix.shape[0], 1), np.inf), index=clusterMatrix.index)
colDf.rename(columns={0:newClusterIndex}, inplace=True)
```

2) Update the distance matrix.

   After merging two clusters we need to update the distance matrix. We use the min approach to update the distances. The distance between the two clusters is represented by the distance of the closest pair of data objects belonging to different clusters.

   Below image shows the code to update the distance between clusters after merging.

```
for cluster in list(rowDf):
    minValue = float('Inf')
    for data in cluster.split('-'):
        for j in minclusterIndex1.split('-'):
            minValue = min(minValue, self.distanceMatrix.at[data, j])

        for j in minClusterIndex2.split('-'):
            minValue = min(minValue, self.distanceMatrix.at[data, j])

    rowDf[cluster][newClusterIndex] = minValue
    colDf[newClusterIndex][cluster] = minValue

clusterMatrix = pd.concat([clusterMatrix, rowDf])
clusterMatrix = pd.concat([clusterMatrix, colDf], axis=1)
clusterMatrix[newClusterIndex][newClusterIndex] = np.inf
```

- Until there are N clusters remaining.

2.3 Advantages

- Any desired number of clusters can be obtained.
- The min approach can handle non-elliptical shapes.

2.4 Disadvantages:

- Once a decision is made to combine two clusters than it cannot be undone.
- The algorithm becomes slow for a large dataset.
- The min approach is sensitive to noise and outliers.

3. Density-based clustering

3.1 Introduction:

Density based clustering method defines a cluster as a maximal set of density-connected points. A high-density point is defined as any point with at least "MinPts" number of points in its ε-neighborhood where ε is the radius of neighborhood for each point. To understand the working of algorithms, the following terminology is required:

- **Core point:** A point is a core point if it has more than a specified number of points (MinPts) within Eps—These are points that are at the interior of a cluster.
- **Border point:** A border point has fewer than MinPts within Eps, but is in the neighborhood of a core point.
- **Noise point:** A noise point is any point that is not a core point nor a border point.

3.2 Implementation:

The overall implementation of the DBSCAN algorithm is as follows:

- Initially, the input for parameter ε and MinPts is taken from the user and is passed as a parameter to the DBSCAN method.
- Next, each point in the dataset is expanded and all the points that are directly and indirectly density-reachable from it are calculated.
    - The expansion of the cluster is done by the following method:

```
def expandCluster(self, point, neighbors, clusters, eps, minpts, clusterNumber, visited, distance, dataset, clustered):
    clusters[clusterNumber].append(point)
    for neighbor in neighbors:
        if neighbor not in visited:
            visited.add(neighbor)
            newNeighbours = self.regionQuery(neighbor, eps, distance, dataset)
            if len(newNeighbours)+1 >= minpts:
                for n in newNeighbours:
                    neighbors.append(n)
        if neighbor not in clustered:
            clustered.add(neighbor)
            clusters[clusterNumber].append(neighbor)
            neighbor.cluster = clusterNumber
```

- The ε-neighborhood of each point is calculated as follows:

```
def regionQuery(self, neighbor, eps, distance, points):
    result = list()
    for point in points:
        if distance[neighbor.id-1][point.id-1] < eps:
            result.append(point)
    return result
```

- Finally, the number of points in the ε-neighborhood of the point is compared with MinPts and if the ε-neighborhood consists of at least MinPts number of points then it is added to the current cluster else, it is marked as a noise point and the process continues for the next point.
- This continues until all the points are assigned to a cluster or marked as noise.

The entire process is wrapped in the following method:

```
def dbScan(self, dataset, eps=1, minpts=5, distance=None, points=None):
    clusterNumber = 0
    clusters = defaultdict(list)
```

```
visited = set()
clustered = set()
for point in dataset:
    if point not in visited:
        visited.add(point)
        neigbors = self.regionQuery(point, eps, distance, dataset)
        if len(neigbors)+1 < minpts:
            clustered.add(point)
        else:
            clusterNumber+=1
            self.expandCluster(point, neigbors, clusters, eps, minpts, clusterNumber, visited, distance, dataset, clustered)
return clusters
```

3.3 Advantages

- DBSCAN algorithm is resistant noise.
- Can handle clusters of different shapes and sizes.
- It does not require number of clusters unlike the k-means algorithm.

3.4 Disadvantages:

- DBSCAN cannot handle clusters of varying densities as determining a meaningful epsilon value will be difficult.
- This algorithm is highly sensitive to parameters and it is difficult to estimate the correct set of parameters.
- It is not entirely deterministic, as a border point can lie in either of its neighboring clusters and is assigned depending on the order of data points given.

4. Gaussian Mixture Model

4.1 Introduction:

Gaussian mixture model is a clustering algorithm which assigns clusters to dataset in a similar fashion to k-means. However, the Gaussian mixture model takes into the account the covariance (to determine the shape of the distribution). This gives an advantage of finding clusters of varying sizes and variance.

In Gaussian mixture model, each cluster is assumed as a Gaussian distribution. The goal of the algorithm is to assign data points to different distributions. The Gaussian mixture model uses soft clustering approach to cluster each dataset i.e. it gives the likelihood that the sample i came from Gaussian k rather than assigning each data point to a cluster.

4.2 Implementation:

Expectation – Maximization Algorithm:
This algorithm has two steps, E-step and the M-Step.

1) Repeat

a. E-Step

Here we compute rik i.e the probability that the sample i belongs to cluster k. Below code shows the steps to find the expected values of each data point. We used scipy.stats.multivariate_normal.pdf to get the probability density function of the dataset.

```
def eStep(self):
    # probMatrix (rik) = n x k where n = number of data points, k = number of clusters
    probMatrix = np.zeros((self.dataMatrix.shape[0], self.numClusters))
    for mu, sig, pi, idx in zip(self.mu, self.sigma, self.pi, range(self.numClusters)):
            sig += self.reg_sigma
            probMatrix[:, idx] = self.rik(mu, sig, pi)
    return probMatrix

def rik(self, mu, sigma, pi):
    numerator = pi*multivariate_normal.pdf(self.dataMatrix, mean=mu, cov=sigma, allow_singular=True)
    denominator = np.sum([p*multivariate_normal.pdf(self.dataMatrix, mean=m, cov=s, allow_singular=True)
                          for p, m, s in zip(self.pi, self.mu, self.sigma+self.reg_sigma)], axis=0)
    return numerator / denominator
```

b. M-Step

In this step we modify the mean, covariance and weight such that the likelihood of each sample belonging to a cluster increases.

```
def mStep(self):
    self.mu = []
    self.sigma = []
    self.pi = []
    for k in range(len(self.probMatrix[0])):
        sigma_rik = np.sum(self.probMatrix[:,k], axis=0)
        new_mu = (1/sigma_rik)*np.sum(self.probMatrix[:,k].reshape(len(self.probMatrix), 1)*self.dataMatrix,
        self.mu.append(new_mu)
        new_pi = sigma_rik / len(self.dataMatrix)
        self.pi.append(new_pi)
        new_sigma = (np.dot((np.array(self.probMatrix[:,k]).reshape(len(self.dataMatrix), 1)
                    *(self.dataMatrix - new_mu)).T, (self.dataMatrix - new_mu))) / sigma_rik
        self.sigma.append(new_sigma + self.reg_sigma)
```

Below is the code snippet to find the log likelihood.

```
log_likelihood = np.log(np.sum([pi*multivariate_normal.pdf(self.dataMatrix, mean=self.mu[m], cov=self.sigma[s]+self.reg_sigma,
                    for pi,m,s in zip(self.pi,range(len(self.mu)),range(len(self.sigma)))]))
```

2) Until the log likelihood reaches the local optimal or some predefined maximum number of iterations.

4.3 Advantages

- Gives the probabilistic clustering.
- It can handle clusters of varying size and variance.

4.4 Disadvantages:

- This model depends on the initialization parameters.
- There might be some overfitting issues.

5. Spectral clustering

5.1 Introduction

Spectral clustering takes a graph-based approach to encode the information about the local neighborhood. The similarity graph is used to store the similarity vis-à-vis distance between points where each point is considered as a vertex of the graph and the weight of the edge represents similarity. In our implementation of the project, this is defined by the gaussian kernel as follows:

$$w_{ij} = \exp(- \| x_i - x_j \|^2 / s^2)$$

For und erstanding the algorithm, the following concepts are necessary:

- Similarity Matrix: It is an n*n matrix, n representing the number of data points, and the value in each cell i, j is the distance given by the gaussian kernel between the point x_i and x_j.
    - One important point to note is that this is a symmetric matrix.
- Degree Matrix: It is an n*n diagonal matrix, n representing the number of data points, where the value of each cell i, i is the sum of row i.
- Laplacian Matrix: It is an n*n symmetric matrix which is derived by subtracting the Similarity matrix from the Degree matrix.

5.2 Implementation:

The overall implementation is as follows:

Step 1: Preprocessing

- The σ parameter is taken as input from the user. This hyper value needs to be finetuned to give the optimal result.
- The similarity matrix (W) is computed using the following code:

```
    def computeSimilarityMatrix(self, dataset, sigma=3):
        '''
            input:  dataset - a list of Point objects
                    sigma - parameter of calculating gaussian kernel
            output: similarityMatrix -
 a NxN matrix, where N is the size of dataset,                    consisting of gaussian weights between ge
nes
        '''
        similarityMatrix = [[0 for x in range(len(dataset))] for y in range(len(dataset))]
        for point in dataset:
            for p in dataset:
                dist = np.linalg.norm(point.point - p.point)
                similarityMatrix[point.id-1][p.id-1] = np.exp(-dist**2/(sigma**2.))
        return similarityMatrix
```

- Next, the Degree Matrix (D) is computed using the following method:

```
    def computeDegreeMatrix(self, W):
        '''
        input:  W - similarityMatrix
        output: D - a NxN matrix, where N is the size of similarityMatrix, defining the degree.
```

```
    '''
    res = np.sum(W,axis=1).tolist()
    D = [[0 for _ in range(len(W))] for _ in range(len(W))]
    for i in range(len(D)):
        D[i][i] = res[i]
    return D
```

Step 2: Decomposition

- This step comprises of finding eigen vectors and eigen values of the aforementioned Laplacian matrix.
- The Laplacian Matrix is calculated as explained above. i.e., L=D-W

```
def computeLaplacianMatrix(self, D, W):
    '''
    input:  D, W - NxN matrices
    output: L   -  NxN laplacian matrix
    '''
    a = np.array(D)
    b = np.array(W)
    return a-b
```

- Then the embedded space is computed from the eigen vectors corresponding to the k smallest eigen values, where k is the value which maximizes the expression:

$$D_k = \left| \lambda_k - \lambda_{k-1} \right|$$

- The $\lambda_k - \lambda_{k-1}$ is defined as the absolute difference between two consecutive eigen values after sorting them in ascending order.

```
def sort(self, eigenValues, eigenVectors):
        '''
        input:  eigenValues, eigenVectors
        output: eigen vectors corresponding to the sorted eigen values in ascending order
        '''
        eigenValues = eigenValues.argsort()
        k = self.findEigenGap(eigenValues)
        idx = eigenValues[:k]
        eigenVectors = eigenVectors[:,idx]
        return eigenVectors
def findEigenGap(self, eigenValues):
    delta = 0
    k = 0
    for i in range(1, len(eigenValues)):
        tmp = abs(eigenValues[i] - eigenValues[i-1])
        if tmp > delta:
            k = i
            delta = tmp
```

```
return k
```

Step 3: Clustering

- Finally, this embedded space is passed as an input to the k-means algorithm which performs the final clustering.

5.3 Advantages:

- Not as sensitive to algorithms when compared to other algorithms such as DBSCAN.
- Can find clusters of arbitrary shapes.
- Unlike k-means, spectral clustering works well even for anisotropic data.

5.4 Disadvantages:

- The performance is greatly affected by noisy dataset.
- The process of computing eigen vectors gets very expensive when dealing with large datasets and consequently impacts the speed of the algorithm.
- Slower than k-means.

Result Analysis

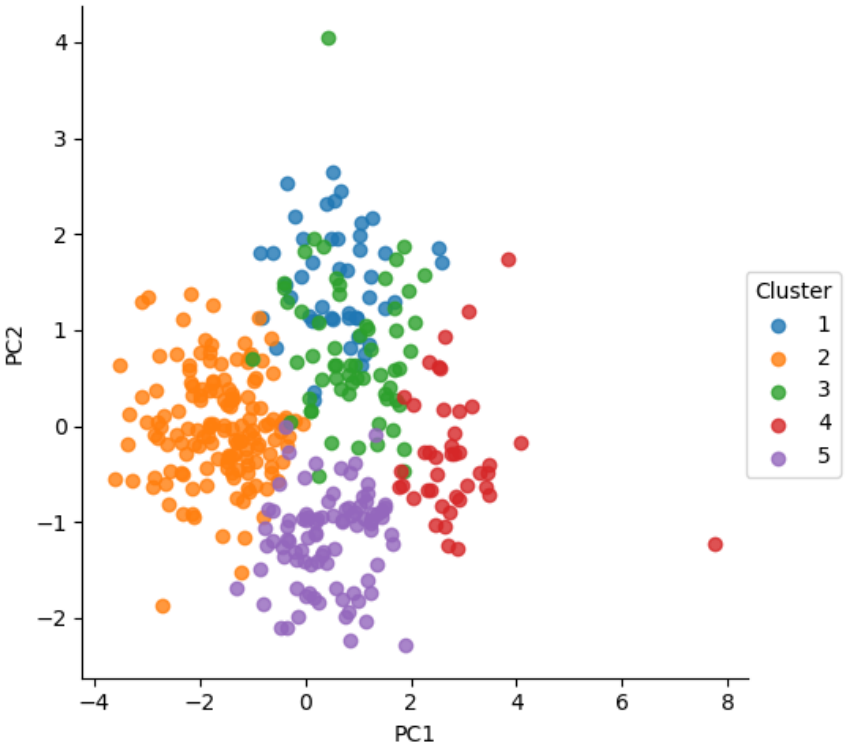
This section gives the results of each algorithm on two datasets: cho.txt and iyer.txt and compares each result with all the results of other algorithms.

K-means:

CHO.TXT:

```
RAND COEFFICIENT: 0.8068538752718194
JACCARD COEFFICIENT: 0.42540531906397255
```

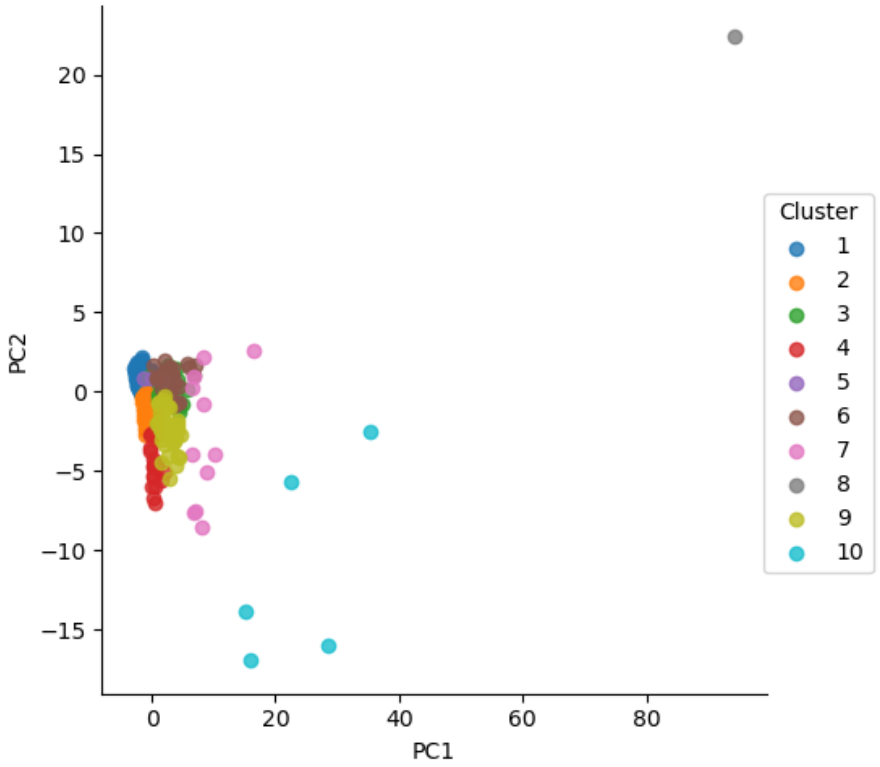
K = 5 and max_iterations = 20



IYER.TXT:

```
RAND COEFFICIENT: 0.7157009828313174
JACCARD COEFFICIENT: 0.3115787757172753
```

K = 10 and max_iterations = 10



Analysis:

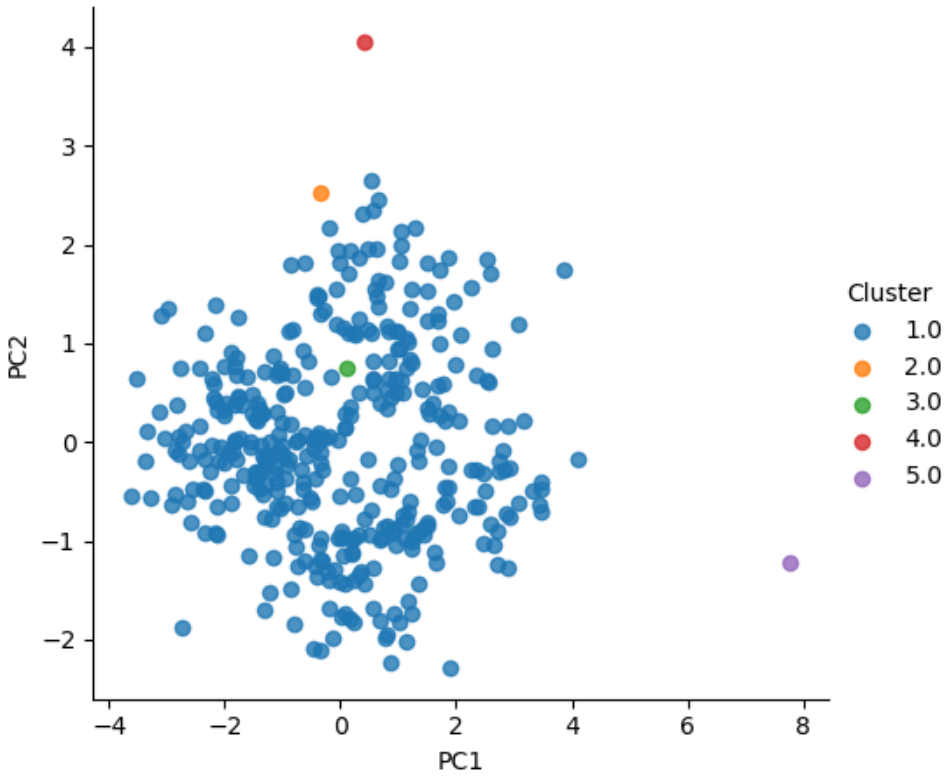
- As it is evident from the Jaccard and Rand Index metrics, K-means algorithm has clustered the points in cho.txt better than that in iyer.txt.
- This asserts that K-means does not perform well when the clusters are non-globular in shape.
- Also, compared to other algorithms, K-means seemed to perform faster.

Hierarchical:

CHO.TXT

```
Finding Rand and Jaccard Coefficient .....................
RAND COEFFICIENT: 0.24027490670890495
JACCARD COEFFICIENT: 0.22839497757358454
```

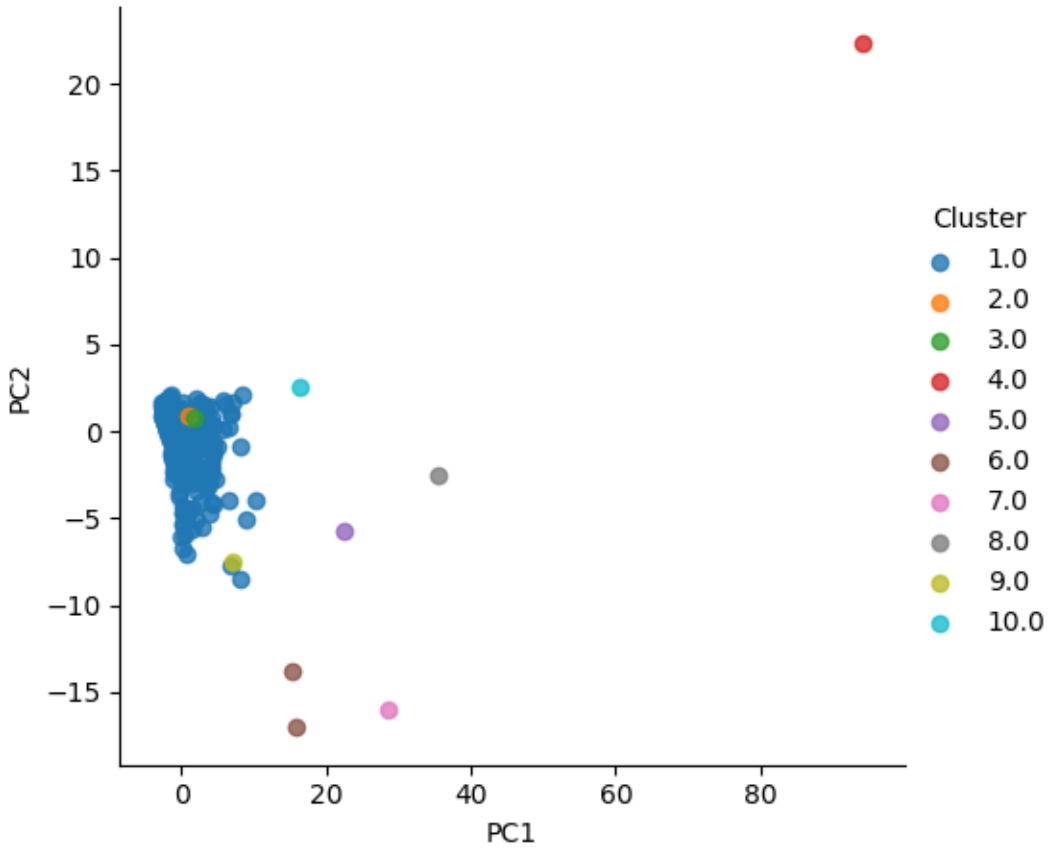
K = 10



IYER.TXT

```
Finding Rand and Jaccard Coefficient .....................
RAND COEFFICIENT: 0.1882868355974245
JACCARD COEFFICIENT: 0.15824309696642858
```

K = 10

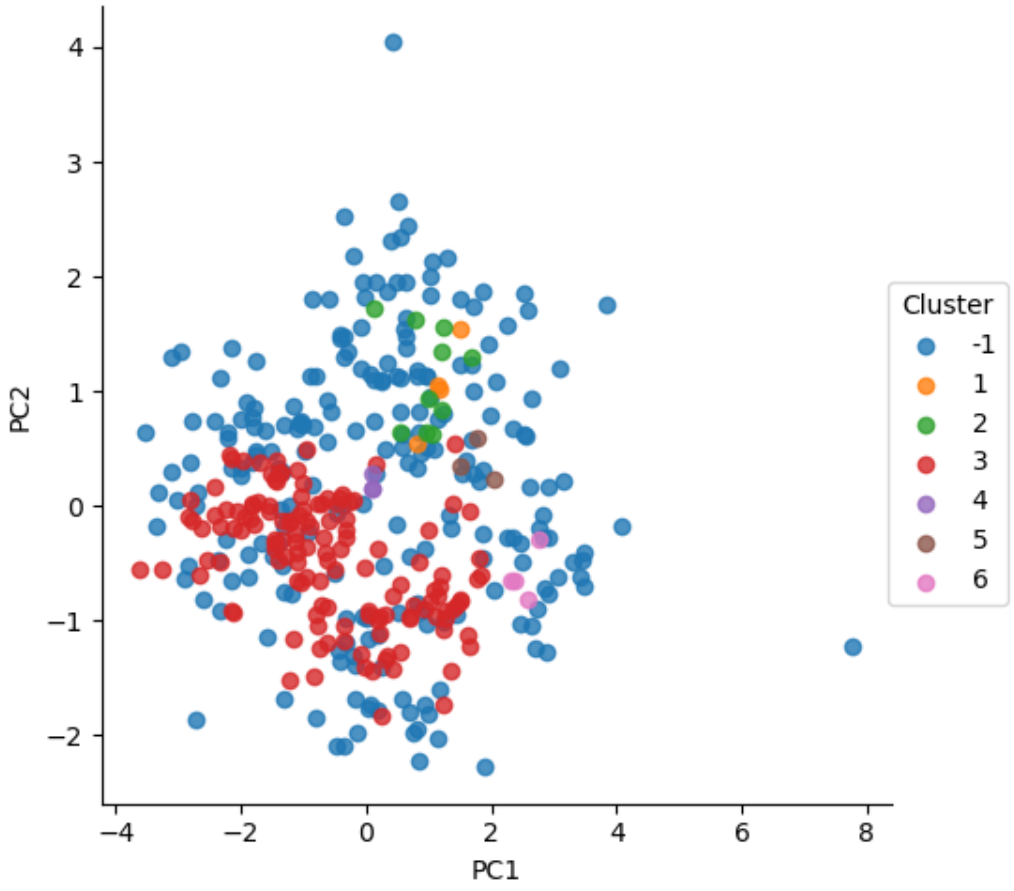


Density based clustering (DBSCAN):

CHO.TXT:

```
RAND COEFFICIENT: 0.5404037692287041
JACCARD COEFFICIENT: 0.20433630786391524
```

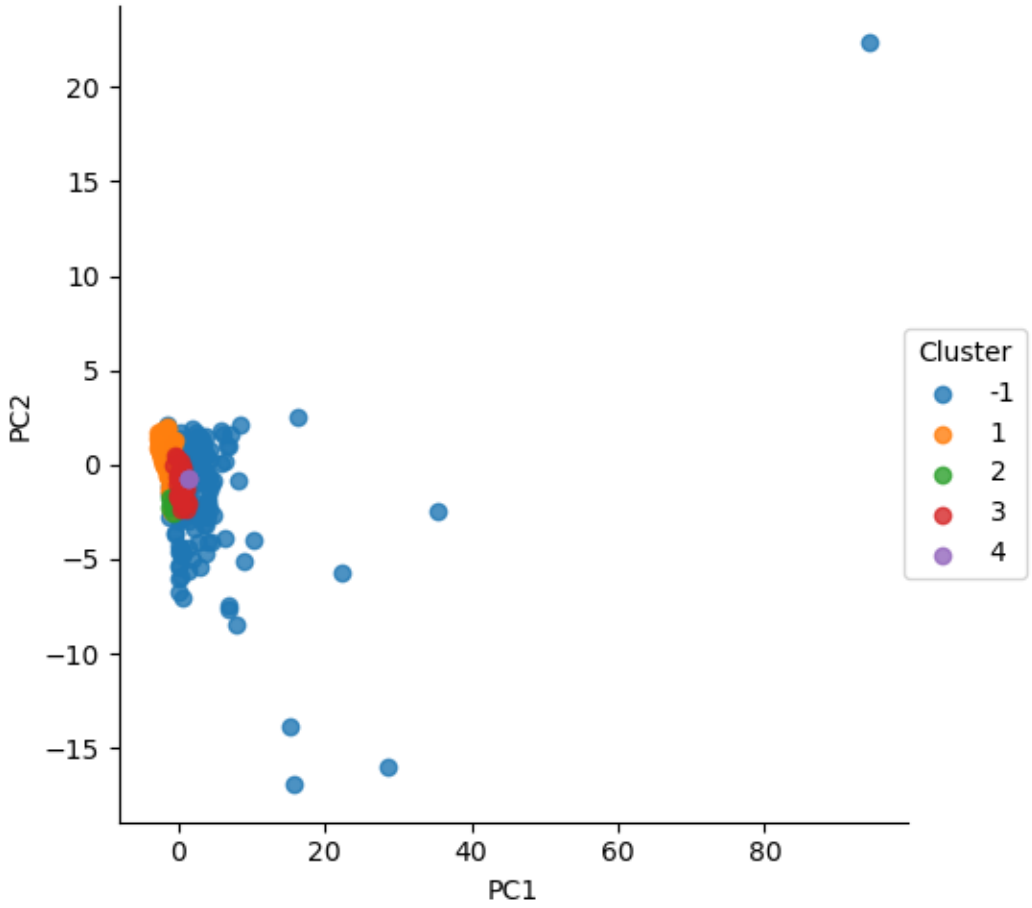
Epsilon = 1 MinPts = 4



IYER.TXT:

```
RAND COEFFICIENT: 0.5124528132470846
JACCARD COEFFICIENT: 0.21878990246563518
```

EPSILON = 1 MinPts = 5



Analysis:

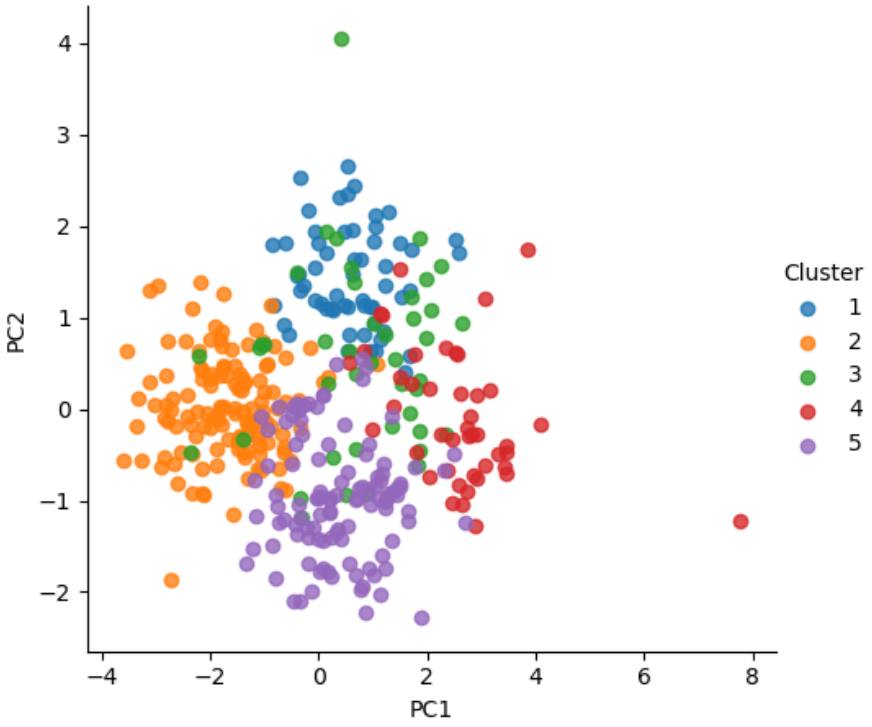
- One primary observation in the DBSCAN algorithm was that it is highly sensitive to parameters Epsilon and Minpts.
- Also, this algorithm performs rather poorly on the given dataset.
- One other observation is that DBSCAN removes the points it considers to be noise unlike k-means.

4. GMM

CHO.TXT:

```
Finding Rand and Jaccard Coefficient ..................
RAND COEFFICIENT: 0.7664501060431153
JACCARD COEFFICIENT: 0.33999696532888246
```

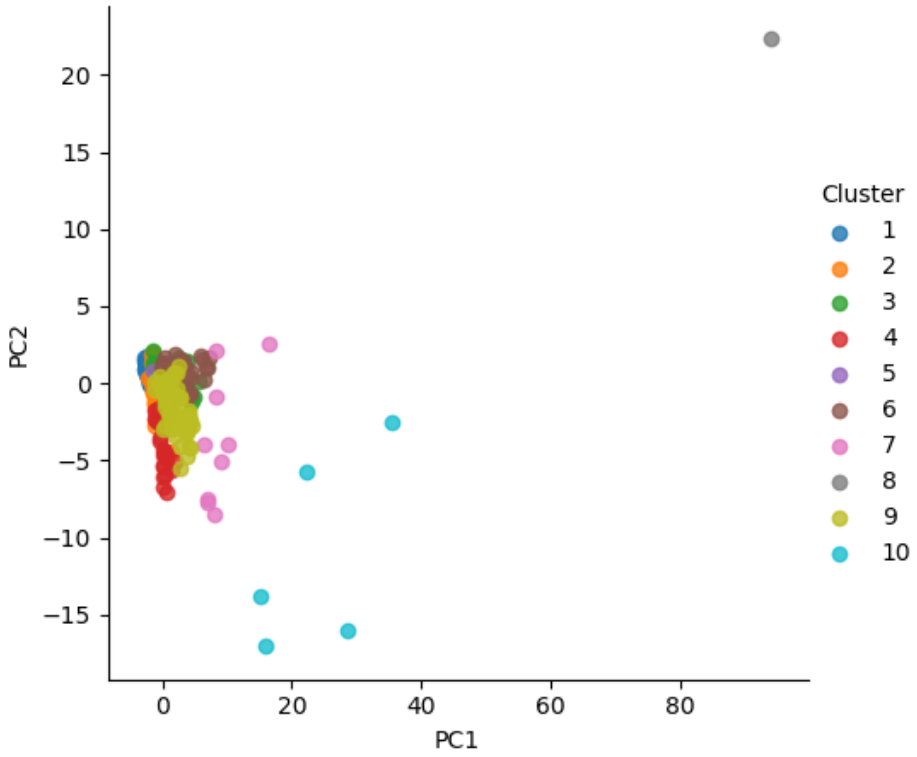
K = 5
Convergence Threshold = 1e-9
Maximum Iterations = 100



IYER.TXT:

```
Finding Rand and Jaccard Coefficient ..................
RAND COEFFICIENT: 0.7625416683814149
JACCARD COEFFICIENT: 0.3467409092312601
```

K = 10
Convergence Threshold = 1e-9
Maximum Iterations = 100



Analysis:

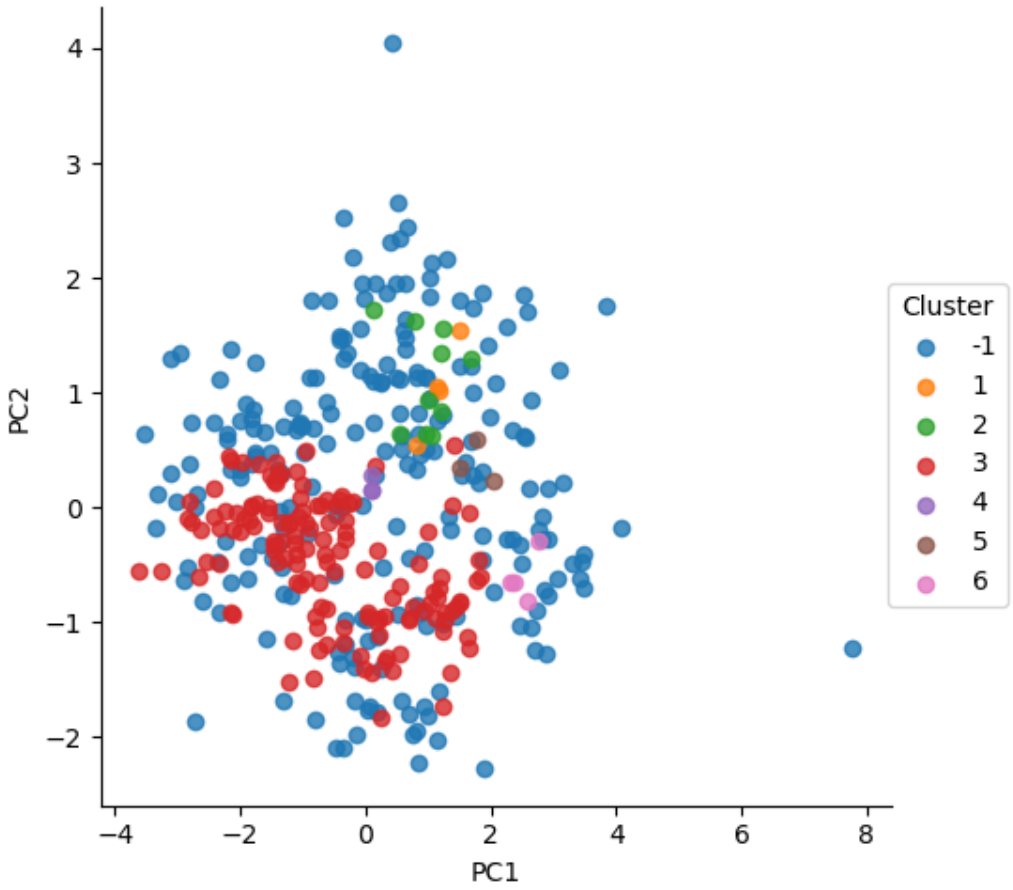
- GMM is sensitive to parameter as different initializations of mean and covariance will result in a highly varying rand and Jaccard coefficient.
- It needs smoothing value to calculate the probability distribution function.

5. Spectral Clustering:

CHO.TXT:

```
RAND COEFFICIENT: 0.6485543236060028
JACCARD COEFFICIENT: 0.2945519211079377
```

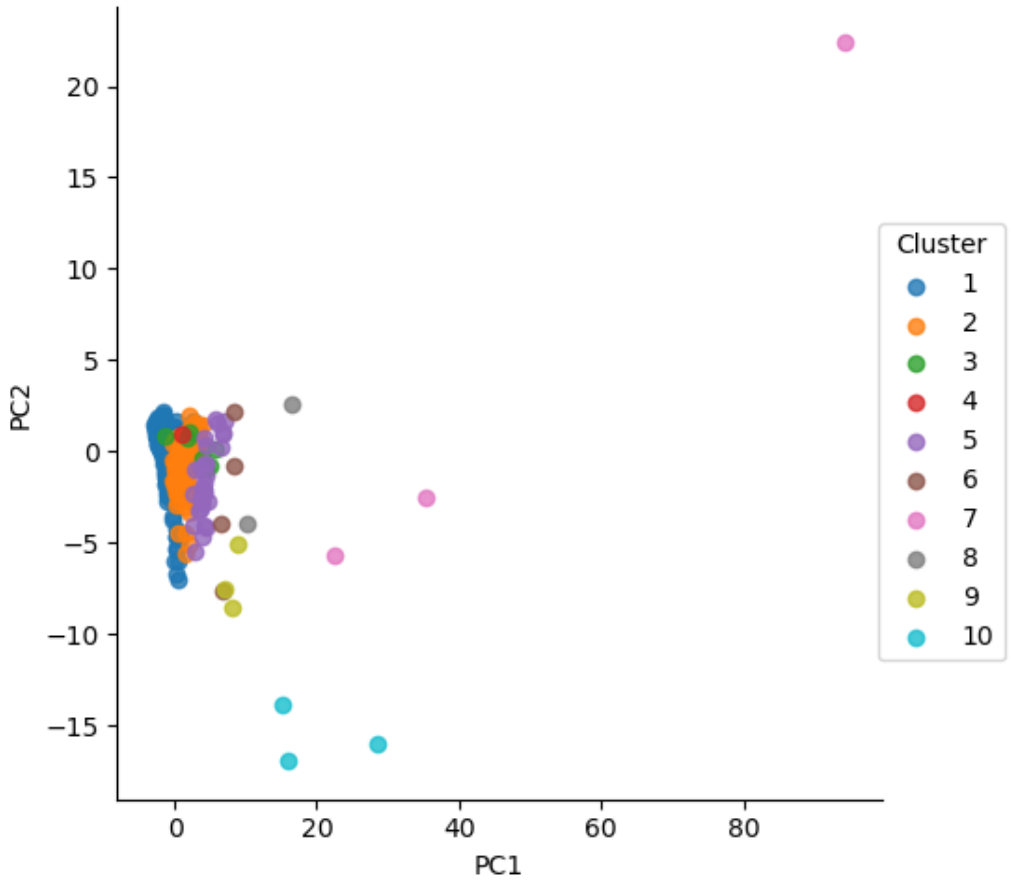
Sigma = 1.5



IYER.TXT:

```
RAND COEFFICIENT: 0.5987414371710021
JACCARD COEFFICIENT: 0.2576123597449972
```

Sigma = 1.8



Analysis:

- The value of sigma in the Gaussian kernel affected the weights of the edges in the similarity matrix thereby resulting in a varying Jaccard and Rand index values.
- The calculation process of embedded space from eigen vectors of Laplacian matrix makes the algorithm slower.