# CSE 601

# Data Mining & Bioinformatics

# Report

# Clustering Algorithms

**Submitted By:**

**Mohammed Kamran Syed (msyed3 )**

**Shreya Bhargava ( shreyabh )**

# Objective:

The purpose of the project is to implement five clustering algorithms to find clusters of genes that exhibit similar expression profiles: K-means, Hierarchical Agglomerative clustering with Min approach, density-based, mixture model, and spectral clustering.
Compare the algorithms, validate clustering results by the following methods:

- Using external index (Rand Index and Jaccard Coefficient) and compare the clustering results from different clustering algorithms. The ground truth clusters are provided in the datasets.
- Visualize data sets and clustering results by Principal Component Analysis (PCA).

## 1. K-Means:

### 1.1 Introduction:

K-means clustering is a type of unsupervised learning for clustering analysis, it aims to assign n observations to K clusters, such that an observation belongs to a particular cluster with the nearest mean, the algorithm works iteratively to assign each observation to one of the K clusters.

K-Means clustering involves 3 major steps:

1. K-means Initialization
   Initialization: Determine the K cluster centers
2. Cluster Assignment
   Assign each object to the cluster which has the closet distance from the centroid to the object
3. Update Cluster Centroid
   Compute cluster centroid as the center of the points in the cluster

### 1.2 Algorithm:

*1: Fetch the data from the input file.*
*2: n observations are assigned to the initial clusters*
*3: Once the cluster is assigned:*
*Compute the mean again*
*Update the mean/centroids*
*4: Repeat step until the mean converges.*
*5: When the final clusters are obtained calculate Jaccard Coefficient and randIndex of the clusters.*
Jaccard Coefficient and randIndex will provide the similarity between the clusters as defined by the size of the intersection divided by the size of the union of the sample set.

## 1.3 Implementation:

- Read the required parameters like initial centers, max iterations and number of clusters the dataset has to be clustered into.

```
k = int(input("Enter the number of clusters to cluster the datasets into:"))
max_iterations = int(input("Enter max no. of iterations:"))
```

```
def get_initial_points_by_index(self):
    initial_indices = []
    count = 0
    while count < self.k:
        idx = int(input("Enter the initial cluster starting point index:"))
        initial_indices.append(idx)
        count += 1

    data_dict = dict()
    for row in self.data:
        data_dict[row[0]] = row[2:]

    self.initial_points = []

    for idx in initial_indices:
        self.initial_points.append(data_dict[idx])

def get_initial_points_by_points(self):
    print("Now enter starting points for {} clusters for data file {}".format(self.k, self.file_name))
    self.initial_points = []
    count = 0
    while count < self.k:
        line = input("Enter the initial cluster starting points with {} dimenstions separated by tabs:".format(self.data.s
        self.initial_points.append([float(x) for x in re.split("\t+", line)])

        if len(self.initial_points[-1]) != self.data.shape[1] - 2:
            raise Exception("Incorrect dimensions for the starting point.")
        count += 1

    self.initial_points = np.array(self.initial_points, dtype=np.float64)
```

- Assign clusters for every point based on the minimum Euclidean distance to any of the initial centers

```
def assign_clusters(self):
    cluster_dict = dict()

    for row in self.data:
        min_dist = float("inf")
        min_cluster = -1

        for i, center in enumerate(self.initial_points):
            distance = abs(np.linalg.norm(row[2:] - center))
            if distance < min_dist:
                min_dist = distance
                min_cluster = i + 1

        if min_cluster not in cluster_dict:
            cluster_dict[min_cluster] = []

        cluster_dict[min_cluster].append(row)

    for k, v in cluster_dict.items():
        cluster_dict[k] = np.array(v, dtype=np.float64)

    return cluster_dict
```

- Adjust the centers by taking the mean of the assigned points in each of the cluster

```
def adjust_centers(self, cluster_dict):
    centers = [None] * self.k
    for k, v in cluster_dict.items():
        centers[k - 1] = list(v[:, 2:].mean(axis=0))

    for i, c in enumerate(centers):
        if c is None:
            centers[i] = self.initial_points[i]
    return np.array(centers, dtype=np.float64)
```

- Repeat this process until max iterations or until the centers do not change (which ever comes first):

```
def process_k_means(self):
    iter = 1

    cluster_dict = self.assign_clusters()
    centers = self.adjust_centers(cluster_dict)

    while not np.array_equal(centers, self.initial_points) and iter < self.max_iterations:
        self.initial_points = centers
        cluster_dict = self.assign_clusters()
        centers = self.adjust_centers(cluster_dict)
        iter += 1

    self.result = None
    for k, v in cluster_dict.items():
        for r in v:
            r[1] = k

        if self.result is not None:
            self.result = np.concatenate((self.result, v[:,0: 2]), axis=0)
        else:
            self.result = v[:,0: 2]

    self.result = self.result[self.result[:,0].argsort()]

    self.calculate_coefficients(cluster_dict)

    self.pca(cluster_dict)
```

- Run PCA on the dataset to reduce the dataset to two dimensions and visualize.

```
def pca(self, cluster_dict):
    pca = PCA(n_components=2, svd_solver='full')
    pca.fit(self.rows[:, 2:])
    principle_components_matrix = pca.transform(self.rows[:, 2:])
    df = pd.DataFrame(data = np.concatenate((principle_components_matrix, self.result[:, 1: 2]), axis=1), columns = ['PC1
    self.plot(df, "K-Means: {}".format(self.file_name))

@staticmethod
def plot(df, title):
    lm = sns.lmplot(x='PC1', y='PC2', data=df, fit_reg=False, hue='Cluster')
    lm.fig.suptitle(title)
    plt.show()
    path = os.path.abspath(os.path.join(os.path.abspath(os.path.dirname(__file__)), '..', 'Plots'))
    lm.savefig('{}/{}.png'.format(path, title))
```

- Also, Jaccard and Randindex values are calculated to verify the results with the ground truth values.

```
def calculate_coefficients(self, algo_result):
    P = np.zeros((self.rows.shape[0], self.rows.shape[0]), dtype=np.float64)
    C = np.zeros((self.rows.shape[0], self.rows.shape[0]), dtype=np.float64)

    ground_truth_dict = dict()

    for r in self.rows:
        ground_truth_dict[r[0]] = r

    algo_dict = dict()

    for r in self.result:
        algo_dict[r[0]] = r[1]

    i = 0
    while i < self.rows.shape[0]:
        j = i + 1

        while j < self.rows.shape[0]:
            if ground_truth_dict[self.rows[i][0]][1] ==  ground_truth_dict[self.rows[j][0]][1]:
                P[i][j] = 1
                P[j][i] = 1

            if algo_dict[self.rows[i][0]] == algo_dict[self.rows[j][0]]:
                C[i][j] = 1
                C[j][i] = 1

            j += 1

        i += 1

    M00 = M10 = M01 = M11 = 0
```

```
    M00 = M10 = M01 = M11 = 0

    for i in range(self.rows.shape[0]):
        for j in range(self.rows.shape[0]):
            if P[i][j] == 1 and C[i][j] == 1:
                M11 += 1
            elif P[i][j] == 1 and C[i][j] == 0:
                M10 += 1
            elif P[i][j] == 0 and C[i][j] == 1:
                M01 += 1
            else:
                M00 += 1

    rand_index = (M11 + M00) / (M11 + M00 + M10 + M01)
    jaccard = (M11) / (M11 + M10 + M01)

    print("For file {}:\n RandIndex: {}\n Jaccard Coefficient: {}\n".format(self.file_name, rand_index, jaccard))
```

## 1.4 Visualization:

Figure 1:

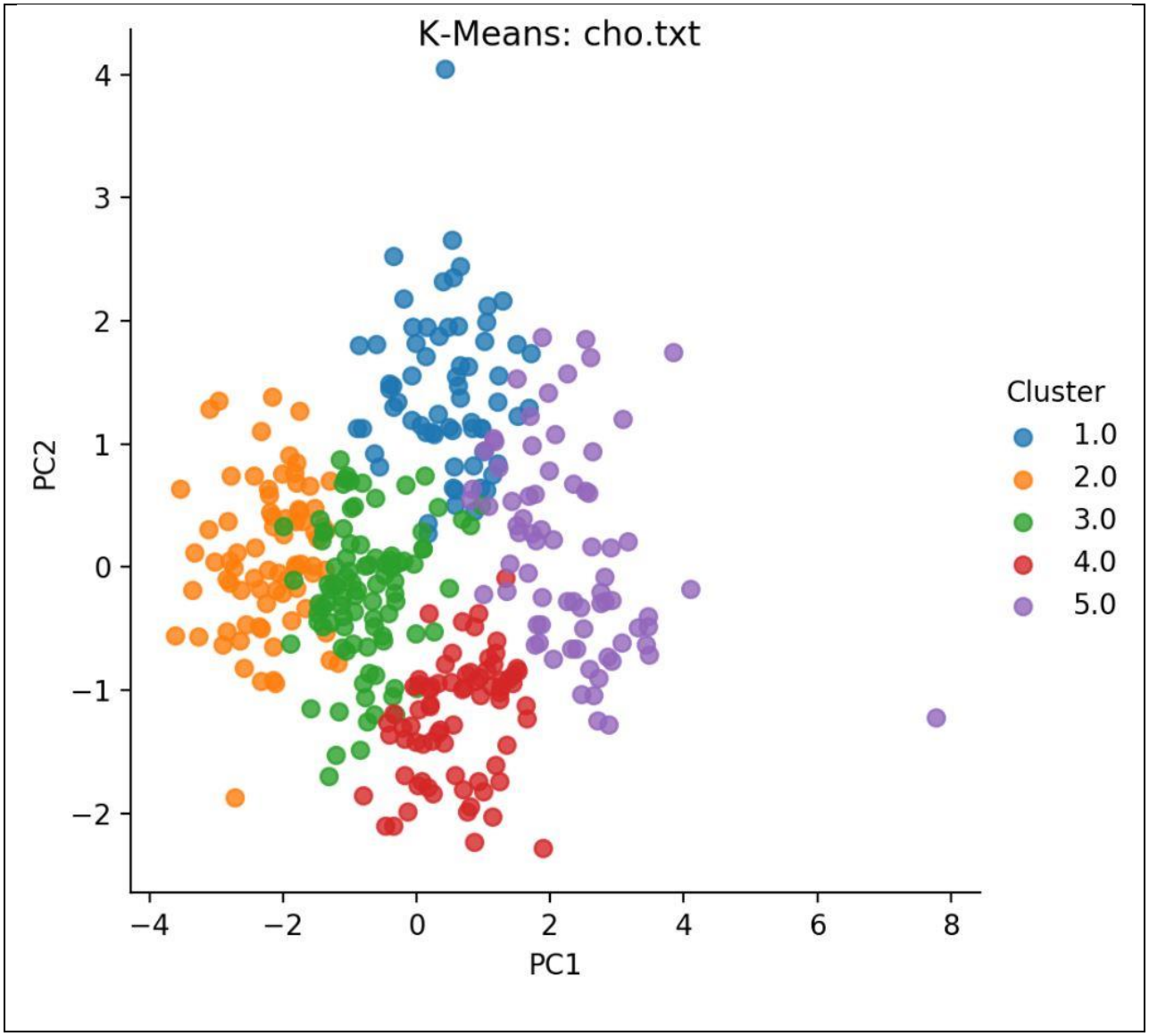



Figure 1, shows that there are 5 clusters at the end of the algorithm for the file cho.txt which has been visualized as shown above.

Number of clusters: 5

Max number of iterations: 100

Enter the initial cluster starting point index:1

Enter the initial cluster starting point index:85

Enter the initial cluster starting point index:217

Enter the initial cluster starting point index:284

Enter the initial cluster starting point index:355

RandIndex: 0.8068538752718194

Jaccard Coefficient: 0.42094249265564004

Figure 2:

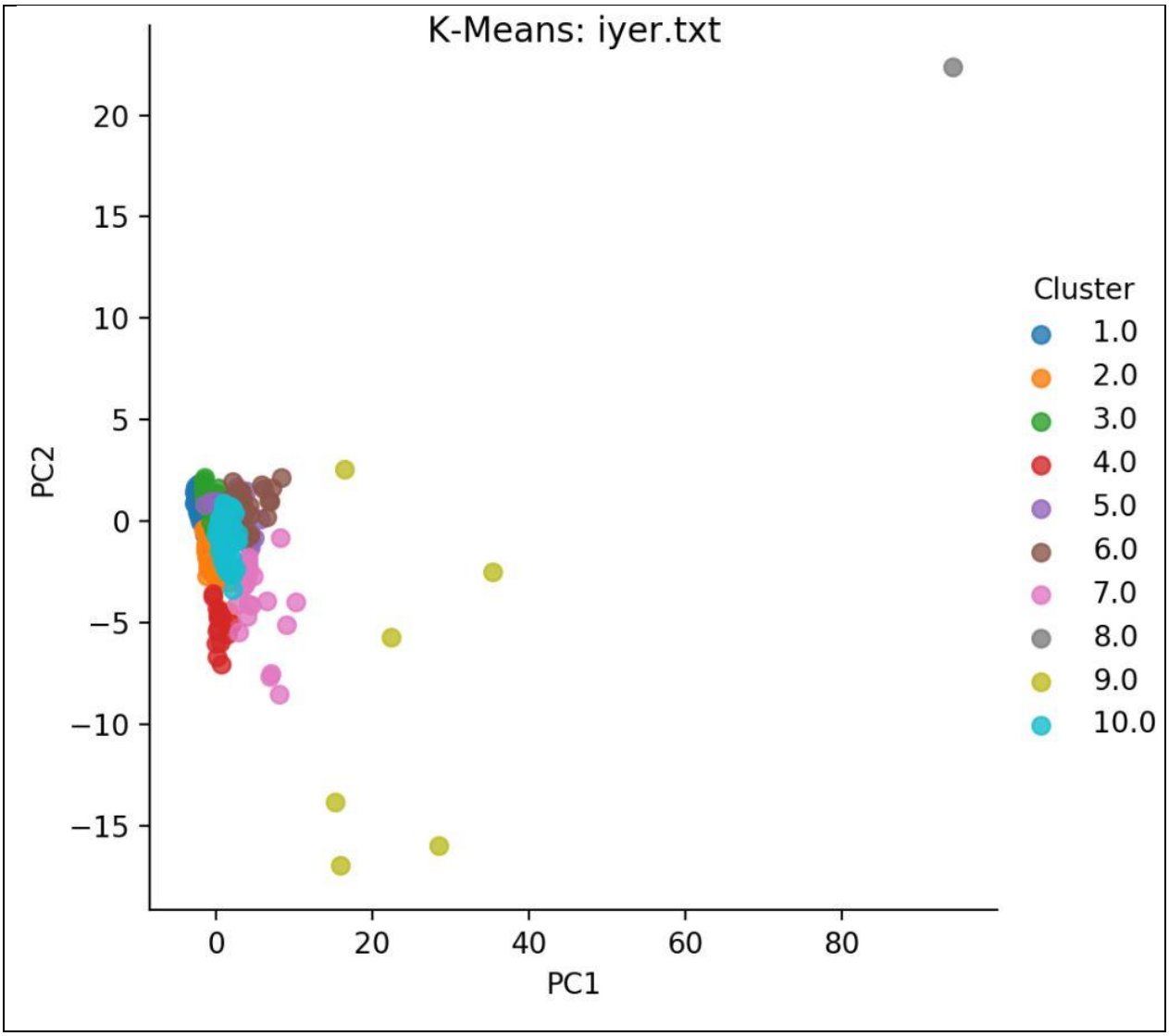



Figure 2, shows that there are 10 clusters at the end of the algorithm for the file iyer.txt which has been visualized as shown above.

Number of clusters: 10

Max number of iterations: 100

Enter the initial cluster starting point index:2

Enter the initial cluster starting point index:120

Enter the initial cluster starting point index:280

Enter the initial cluster starting point index:310

Enter the initial cluster starting point index:350

Enter the initial cluster starting point index:358

Enter the initial cluster starting point index:395

Enter the initial cluster starting point index:440

Enter the initial cluster starting point index:490

Enter the initial cluster starting point index:503

RandIndex: 0.7198463086771247

Jaccard Coefficient: 0.3154834817266029

### 1.5 Pros and Cons of K-Means:

Pros:

1. Easy to implement
2. Efficient if the number of iterations are sufficiently large.
3. K-means algorithm can be easily implemented on variety of large data sets.

Cons:

1. Number of clusters K to be specified.
2. Clusters might be empty.
3. Sensitive to different sizes, densities and irregular shapes.

## 2. Hierarchical Agglomerative Clustering

### 2.1 Introduction:

Hierarchical Clustering builds hierarchy of clusters in a greedy manner, if the hierarchy is formed in bottom-up approach then it is called Agglomerative and if hierarchy is formed from top-down approach then it is Divisive.
Hierarchical clustering is represented by a tree like structure known as Dendrogram, here each cluster is being merged with their nearest neighbor (based on minimum Euclidean distance) until all the points form a single cluster. The proximity between the clusters can be done using three approaches MIN, MAX and group Average, in this project we are asked to implement MIN approach.
Hence, our greedy implementation merges those pairs which have smallest inter-cluster Euclidean distance.

### 2.2 Algorithm:

1. *Assign each item as a cluster, therefore if we have n observations initially then we have n clusters.*
2. *Compute the distance between the clusters.*
3. *Find the clusters which have minimum distance and merge them.*
4. *Update the proximity matrix.*
5. *Repeat steps 2 to 4 until all items clustered into a single one*
6. *When the final clusters are obtained calculate Jaccard Coefficient and randIndex of the clusters.*

*Jaccard Coefficient and randIndex will provide the similarity between the clusters as defined by the size of the intersection divided by the size of the union of the sample set.*

## 2.3 Implementation:

- Read the number of clusters the clustering algorithm has to cluster dataset into, like K-means algorithm did.
- Start by assigning each point as its own cluster in a dictionary.

```
def process_hierarchical_clustering(self):
    count = 1
    cluster_dict = dict()

    for row in self.data:
        if count not in cluster_dict:
            cluster_dict[count] = []

        cluster_dict[count].append(row)
        count += 1

    for k, v in cluster_dict.items():
        cluster_dict[k] = np.array(v, dtype=np.float64)
```

- Now find the minimum Euclidean distance between each cluster (in case of clusters with multiple points, distance is equal to the distance between the nearest points in the two clusters)
  And merge the two clusters which are nearest to each other using the minimum distance obtained.

```
def reduce_clusters(self, cluster_dict):
    global_min = float("inf")
    clusters = None

    keys = list(cluster_dict.keys())

    i = 0
    while i < len(keys):
        j = i + 1

        while j < len(keys):
            local_min = float("inf")

            for row1 in cluster_dict[keys[i]]:
                for row2 in cluster_dict[keys[j]]:
                    d = abs(np.linalg.norm(row1[2:] - row2[2:]))
                    if  d < local_min:
                        local_min = d

            if local_min < global_min:
                global_min = local_min
                clusters = sorted([keys[i], keys[j]])

            j += 1

        i += 1

    cluster_dict[clusters[0]] = np.concatenate((cluster_dict[clusters[0]], cluster_dict[clusters[1]]), axis=0)
    del cluster_dict[clusters[1]]

    return cluster_dict
```

- Repeat this until the dictionary contains exactly ‘k’ number of merged clusters where k is the number of clusters we want.

```
def process_hierarchical_clustering(self):
    count = 1
    cluster_dict = dict()

    for row in self.data:
        if count not in cluster_dict:
            cluster_dict[count] = []

        cluster_dict[count].append(row)
        count += 1

    for k, v in cluster_dict.items():
        cluster_dict[k] = np.array(v, dtype=np.float64)


    while len(cluster_dict) > self.k:
        cluster_dict = self.reduce_clusters(cluster_dict)

    final_dict = dict()

    keys = sorted(list(cluster_dict.keys()))

    count = 1
    for key in keys:
        final_dict[count] = cluster_dict[key]
        count += 1

    cluster_dict = final_dict

    self.result = None
    for k, v in cluster_dict.items():
        for r in v:
            r[1] = k

        if self.result is not None:
            self.result = np.concatenate((self.result, v[:,0: 2]), axis=0)
        else:
            self.result = v[:,0: 2]
```

- Run PCA on the result to reduce the dataset to two dimensions and visualize.

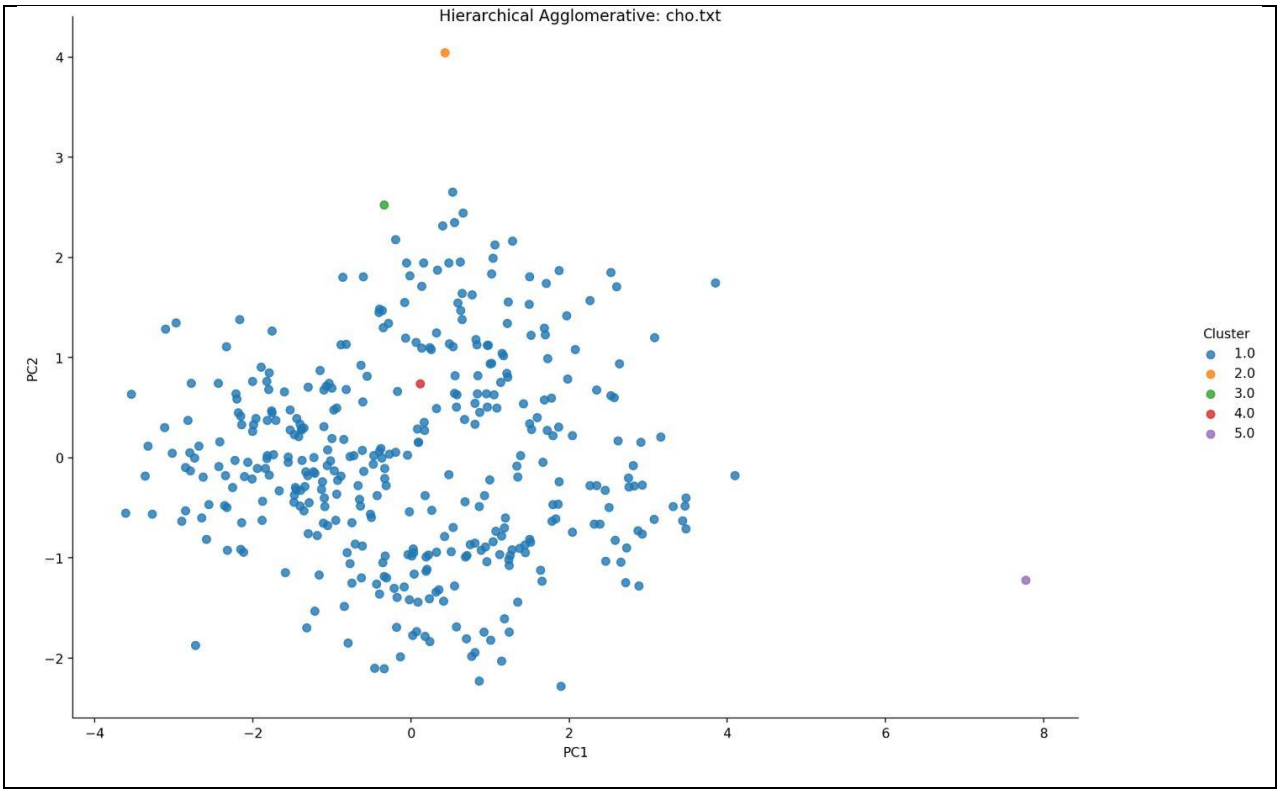
```
def pca(self, cluster_dict):
    pca = PCA(n_components=2, svd_solver='full')
    pca.fit(self.rows[:, 2:])
    principle_components_matrix = pca.transform(self.rows[:, 2:])
    df = pd.DataFrame(data = np.concatenate((principle_components_matrix, self.result[:, 1: 2]), axis=1), columns = ['PC1'
    self.plot(df, "Hierarchical Agglomerative: {}".format(self.file_name))

@staticmethod
def plot(df, title):
    lm = sns.lmplot(x='PC1', y='PC2', data=df, fit_reg=False, hue='Cluster')
    lm.fig.suptitle(title)
    plt.show()
    path = os.path.abspath(os.path.join(os.path.abspath(os.path.dirname(__file__)), '..', 'Plots'))
    lm.savefig('{}/{}.png'.format(path, title))
```

- Also, Jaccard and Randindex values are calculated as in to verify the results with the ground truth values.

## 2.4 Visualization:

Figure 1:

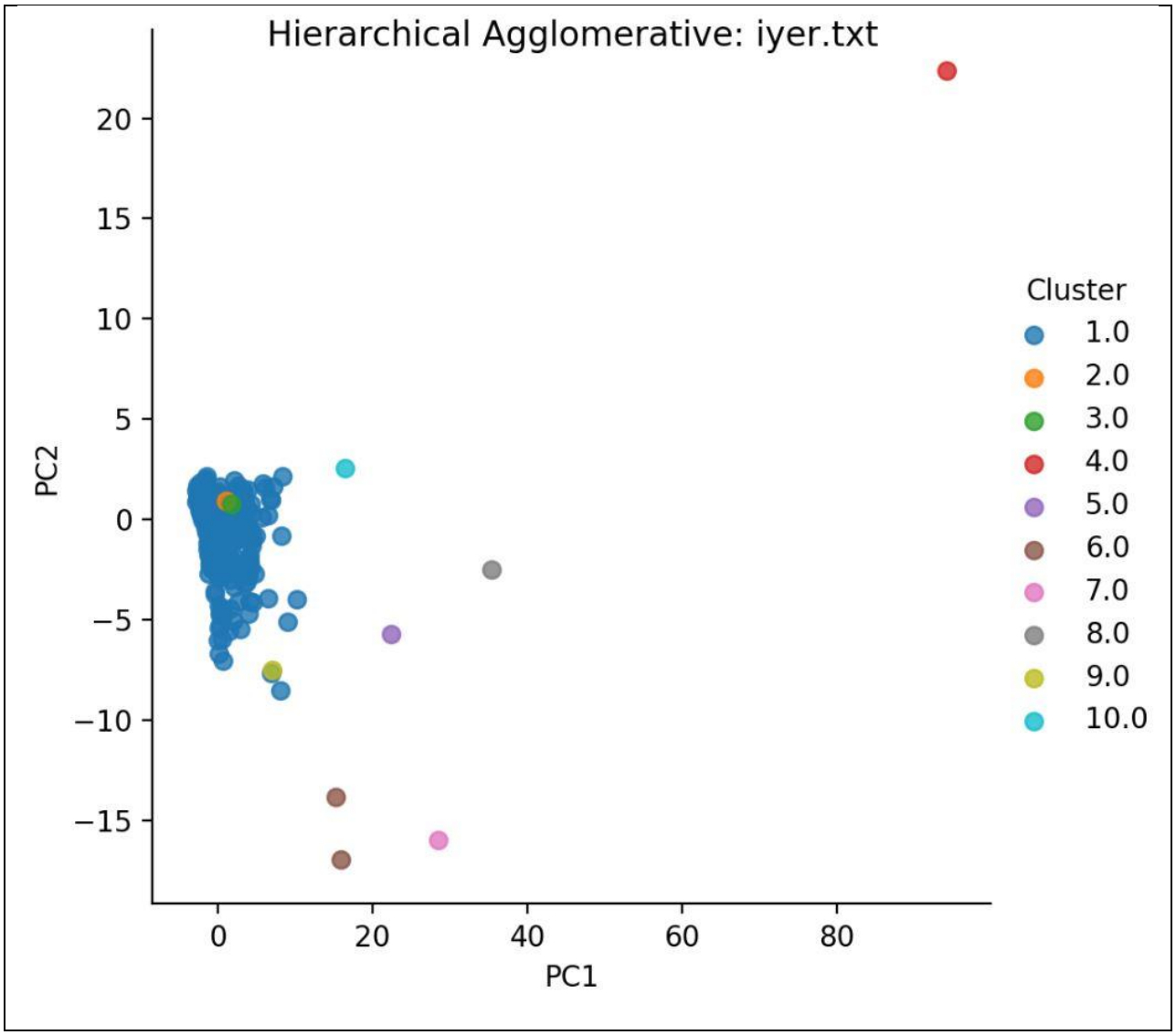



Number of clusters: 5

RandIndex: 0.24027490670890495

Jaccard Coefficient: 0.2263593865332568

Figure 2:




Number of clusters: 10

RandIndex: 0.1882868355974245

Jaccard Coefficient: 0.1565512844436151

## 2.5 Result Evaluation:

- Hierarchical clustering is the slowest among all the clustering algorithms as It has to compute Euclidean distances for every pair of points multiple times.
- As evident from the plots and coefficients, hierarchical clustering algorithm performs poorly for both the datasets cho.txt and iyer.txt.
- However this algorithm can find clusters of any shape or sizes.

### 2.6 Pros and Cons of Hierarchical Clustering:

Pros:

1. No prior information about the observations are required.
2. Used in industry since the output produced are in order which is informative for plotting the data.
3. Produces the hierarchy of clusters even for the singular ones.

Cons:

1. Complexity of algorithm is higher.
2. Cannot handle outliers in data.
3. Chaining issue: single linkage where minimum distance is considered, does not consider other points in the data.
4. Once the cluster is assigned, it cannot be undone.

## 3. Density-based spatial Clustering

### 3.1 Introduction:

Density Based Spatial Clustering commonly known as DBSCAN is a density based clustering algorithm which groups together points that are closely related and nearby neighbors thus makes the outliers lie in the low density region.

## 3.2 Algorithm:

(adapted from lecture slides)

```
DBSCAN(D, eps, MinPts)
  C = 0
  for each unvisited point P in dataset D
    mark P as visited
    NeighborPts = regionQuery(P, eps)
    if sizeof(NeighborPts) < MinPts
      mark P as NOISE
    else
      C = next cluster
      expandCluster(P, NeighborPts, C, eps, MinPts)

expandCluster(P, NeighborPts, C, eps, MinPts)
  add P to cluster C
  for each point P' in NeighborPts
    if P' is not visited
      mark P' as visited
      NeighborPts' = regionQuery(P', eps)
      if sizeof(NeighborPts') >= MinPts
        NeighborPts = NeighborPts joined with NeighborPts'
    if P' is not yet member of any cluster
      add P' to cluster C

regionQuery(P, eps)
  return all points within P's eps-neighborhood (including P)
```

### 3.3 Implementation:

- Read the parameters epsilon and minimum number of points from the command line

```
epsilon = float(input("Enter the max radius value (epsilon):"))
min_points = int(input("Enter the min number of points:"))
```

- For every point which has not be processed yet, check if it is a core point. If so then create a new cluster in cluster dict and find all points which are density reachable from this point and assign them all to this new cluster. Otherwise, mark it as a noise point.

```
def process_dbscan(self):
    cluster_number = 0
    cluster_dict = dict()

    visited = set()
    clustered = set()
    noise = []

    for point in self.data:
        if point[0] not in visited:
            visited.add(point[0])

            neighbours = self.region_query(point)

            if len(neighbours) + 1 < self.min_points:
                clustered.add(point[0])
                noise.append(point)
            else:
                cluster_number += 1

                if cluster_number not in cluster_dict:
                    cluster_dict[cluster_number] = []

                self.expand_cluster(cluster_dict, cluster_number, point, neighbours, visited, clustered)
```

```
def region_query(self, point):
    neighbours = deque()

    for neighbour in self.data:
        if point[0] != neighbour[0]:
            distance = abs(np.linalg.norm(neighbour[2:] - point[2:]))

            if distance < self.epsilon:
                neighbours.append(neighbour)

    return neighbours
```

```
def expand_cluster(self, cluster_dict, cluster_number, point, neighbours, visited, clustered):
    cluster_dict[cluster_number].append(point)
    clustered.add(point[0])

    while neighbours:
        neighbour = neighbours.popleft()
        if neighbour[0] not in visited:
            visited.add(neighbour[0])
            neighbours2 = self.region_query(neighbour)

            if len(neighbours2) + 1 >= self.min_points:
                for n in neighbours2:
                    neighbours.append(n)

        if neighbour[0] not in clustered:
            cluster_dict[cluster_number].append(neighbour)
            clustered.add(neighbour[0])
```

- Run PCA on the result to reduce the dataset to two dimensions and visualize.

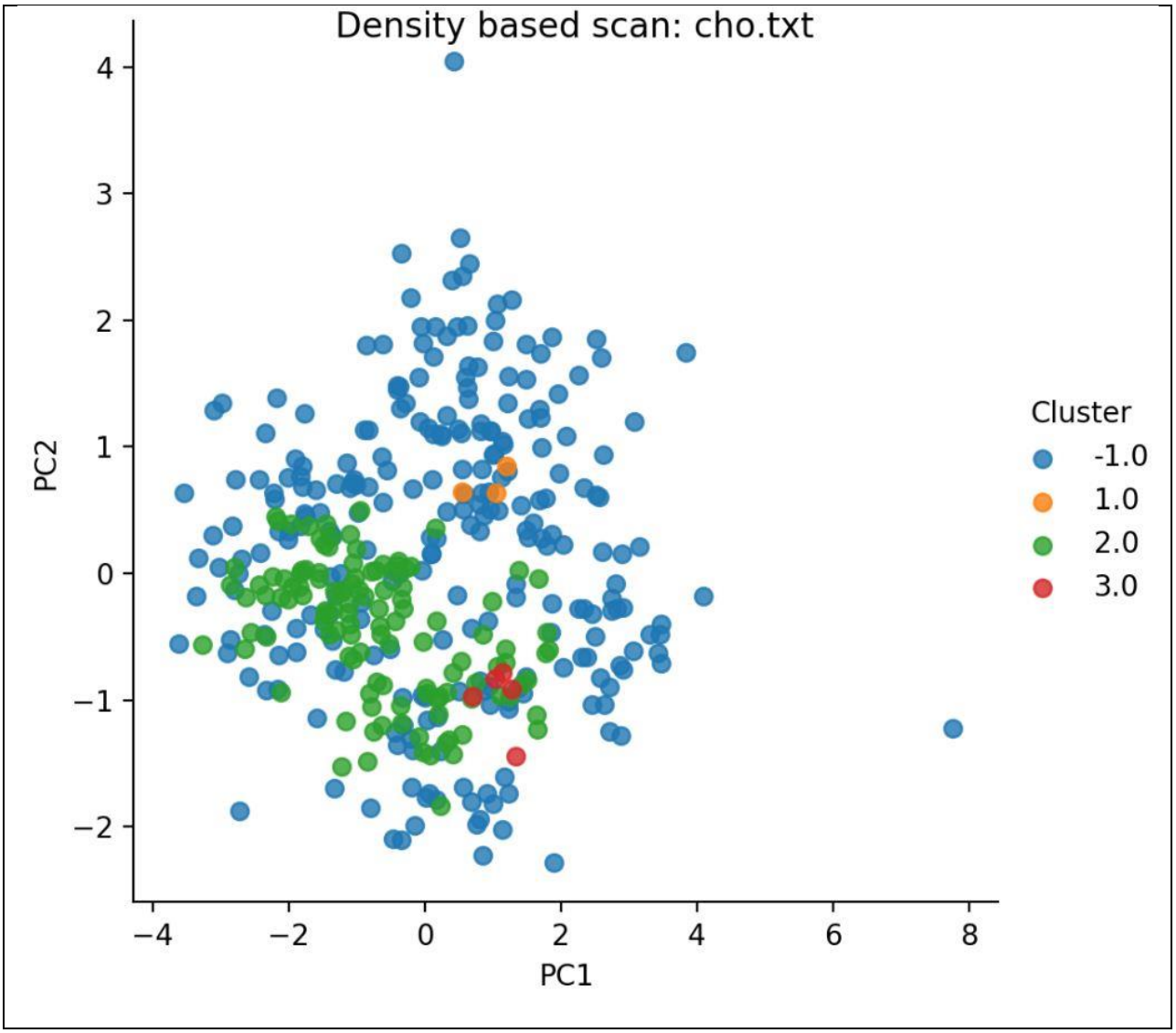
```
def pca(self, cluster_dict):
    pca = PCA(n_components=2, svd_solver='full')
    pca.fit(self.rows[:, 2:])
    principle_components_matrix = pca.transform(self.rows[:, 2:])
    df = pd.DataFrame(data = np.concatenate((principle_components_matrix, self.result[:, 1: 2]), axis=1), columns = ['PC1'
    self.plot(df, "Density based scan: {}".format(self.file_name))

@staticmethod
def plot(df, title):
    lm = sns.lmplot(x='PC1', y='PC2', data=df, fit_reg=False, hue='Cluster')
    lm.fig.suptitle(title)
    plt.show()
    path = os.path.abspath(os.path.join(os.path.abspath(os.path.dirname(__file__)), '..', 'Plots'))
    lm.savefig('{}/{}.png'.format(path, title))
```

- Also, Jaccard and Randindex values are calculated as in to verify the results with the ground truth values.

## 3.4 Visualization:

Figure1:



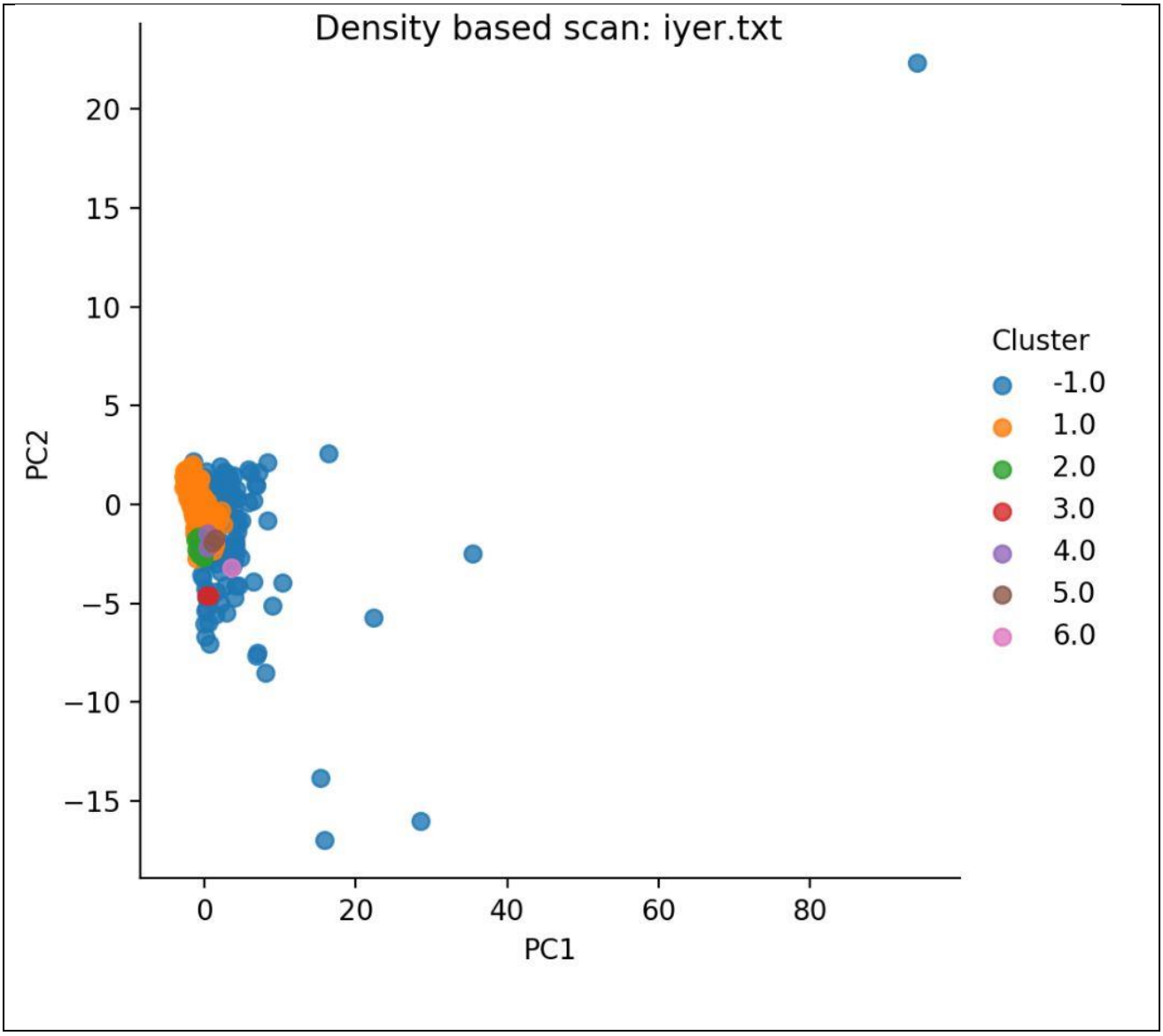

Epsilon: 1

Min points: 5

RandIndex: 0.48969099841606484

Jaccard Coefficient: 0.20191036002939017

Figure2:




Epsilon: 1

Min points: 5

RandIndex: 0.6400300798012638

Jaccard Coefficient: 0.2776685034759238

### 3.5 Result Evaluation:

- Density based scan algorithm is highly sensitive to the epsilon and minimum number of points parameters.
- It can efficiently find outliers in the dataset.

### 3.6 Pros and Cons of Density-based spatial Clustering:

Pros:

1. No prior information about the observations are required.
2. Can handle outliers and arbitrarily shaped clusters.
3. Resistant to noise.

Cons:

1. Cannot handle if the observations have varying densities.
2. Algorithm is sensitive to parameters.

## 4. Gaussian Mixture Model

### 4.1 Introduction:

Gaussian mixture model is a clustering algorithm which does the cluster assignment for the input dataset similar to the K-means algorithm but it differs from it by using the covariance matrix instead in doing so. The covariance matrix helps to determine the shape of the distribution and helps find clusters of different shapes and variances.

The Gaussian mixture model uses soft clustering approach to assign points to clusters i.e. it calculates the log likelihood of a sample 'i' belonging to a gaussian 'k' rather than assigning the point to a cluster. In GMM, each cluster is assumed a gaussian distribution and the goal is to assign each data point to a distribution.

### 4.2 Algorithm:

The algorithm runs on two steps:

- E-step: For a given set of paramters, calculate rik which is the probability that a point 'I' belongs to cluster 'k':

$$\begin{aligned} r_{ik} \equiv E(z_{ik}) &= p(z_{ik}=1|x_i,\pi,\mu,\Sigma) \\ &= \frac{p(z_{ik}=1)p(x_i|z_{ik}=1,\pi,\mu,\Sigma)}{\sum_{k=1}^{K} p(z_{ik}=1)p(x_i|z_{ik}=1,\pi,\mu,\Sigma)} \\ &= \frac{\pi_k \mathcal{N}(x_i|u_k,\Sigma_k)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i|u_k,\Sigma_k)} \end{aligned}$$

- M-step: Update the parameters muew, sigma and pi to maximize the log likelihood:

$$E[\ln p(x,z|\pi,\mu,\Sigma)] = \sum_{i=1}^{n}\sum_{k=1}^{K} r_{ik}\{\ln \pi_k + \ln \mathcal{N}(x_i|\mu_k,\Sigma_k)\}$$

- **Parameter update:**

$$\pi_k = \frac{\sum_i r_{ik}}{n} \qquad \mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}$$

$$\Sigma_k = \frac{\sum_i r_{ik}(x_i-\mu_k)(x_i-\mu_k)^T}{\sum_i r_{ik}}$$

- Iterate the above two steps until the log likelihood converges to a local optimal.

## 4.3 Implementation:

- Read all the parameters required to run the GMM algorithm like number of iterations, convergence threshold, number of clusters etc.
- You can either run K-mean algorithm to determine the initial guess values of pi, sigma and mew or can read it as input.
- Now calculate the $r_{ik}$ value using the scipy's multivariate normal function and the other parameters. (E-step)

```
    log_likelyhoods = []

    for i in range(self.max_iterations):
        prob_matrix = np.zeros((self.data.shape[0], self.k))
        for mu, sig, pi, idx in zip(self.mu, self.sigma, self.pi, range(self.k)):
            sig += self.reg_sigma
            mn = multivariate_normal(mean=mu, cov=sig, allow_singular=True)
            prob_matrix[:, idx] = pi * mn.pdf(self.data[:, 2:]) / np.sum([pi_c * multivariate_normal(mean=mu_c,cov=cov_c,
            for pi_c, mu_c, cov_c in zip(self.pi, self.mu, self.sigma + self.reg_sigma)], axis=0)
```

- Now calculate the log likelihood values from the obtained $r_{ik}$ value and then also update the parameters pi, mew and sigm. (M-step)

```
        for c in range(len(prob_matrix[0])):
            m_c = np.sum(prob_matrix[:,c], axis=0)
            mu_c = (1/m_c) * np.sum(self.data[:, 2:] * prob_matrix[:, c].reshape(len(self.data), 1), axis=0)
            self.mu.append(mu_c)
            self.sigma.append(((1 / m_c) * np.dot((np.array(prob_matrix[:,c]).reshape(len(self.data), 1)*(self.data[:, 2:]
            self.pi.append(m_c / np.sum(prob_matrix))

        log_likelyhoods.append(np.log(np.sum([k*multivariate_normal(mean=self.mu[i],cov=self.sigma[j], allow_singular=True
```

- Repeat this process until we run out of the iterations or increase in log likelihood value is less than the convergence threshold. Also print the values of updated parameters.

```
        if len(log_likelyhoods) > 1 and abs(log_likelyhoods[-1] - log_likelyhoods[-2]) < self.conv_threshold:
            break

    print("Stopped in {} iterations".format(len(log_likelyhoods)))
    print("Mew: {}".format(self.mu))
    print("Sigma: {}".format(self.sigma))
    print("Pi: {}".format(self.pi))
```

- Run PCA on the result to reduce the dataset to two dimensions and visualize.

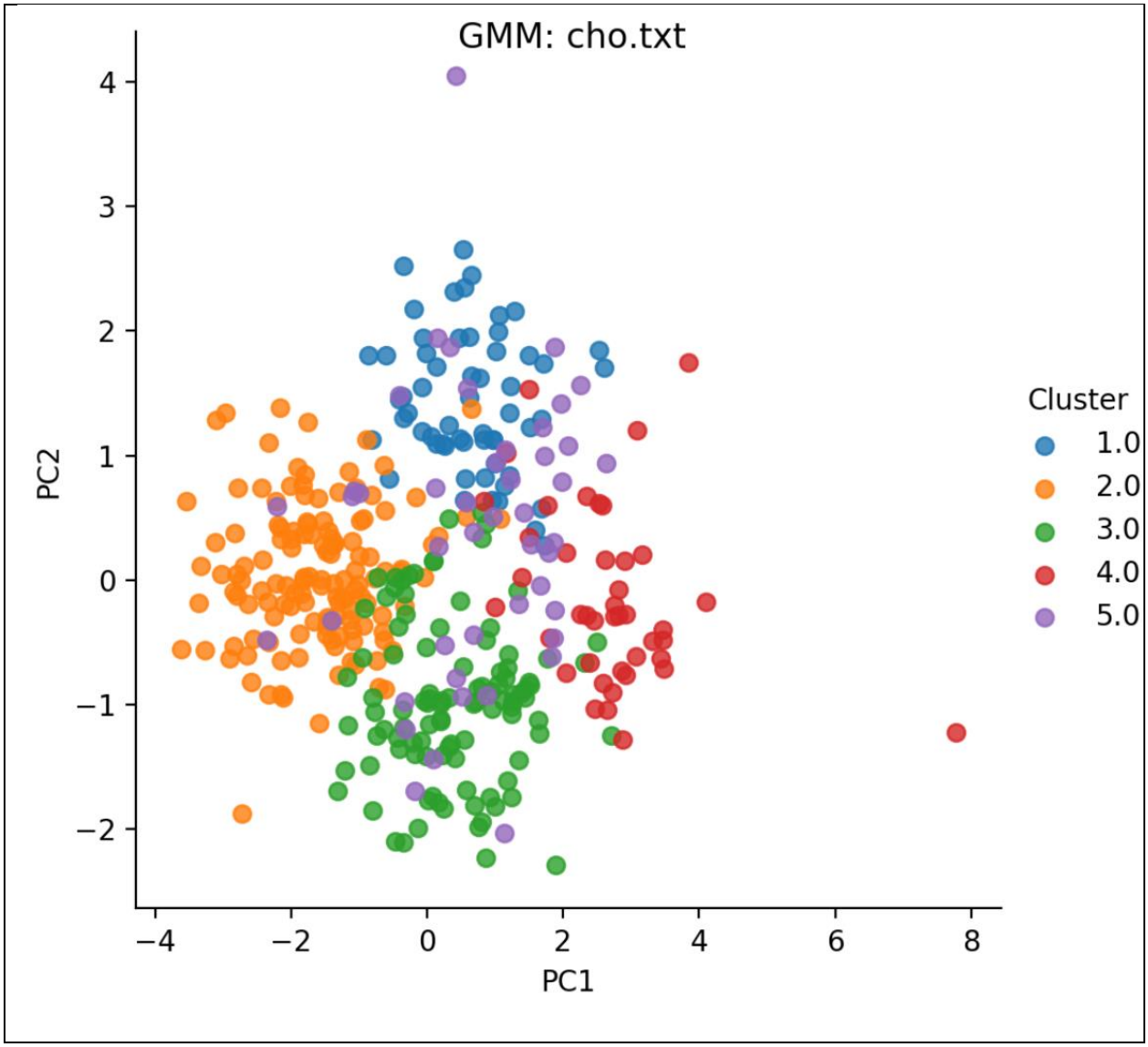
```
def pca(self, cluster_dict):
    pca = PCA(n_components=2, svd_solver='full')
    pca.fit(self.rows[:, 2:])
    principle_components_matrix = pca.transform(self.rows[:, 2:])
    df = pd.DataFrame(data = np.concatenate((principle_components_matrix, self.result[:, 1: 2]), axis=1), columns = ['PC1
    self.plot(df, "GMM: {}".format(self.file_name))

@staticmethod
def plot(df, title):
    lm = sns.lmplot(x='PC1', y='PC2', data=df, fit_reg=False, hue='Cluster')
    lm.fig.suptitle(title)
    plt.show()
    path = os.path.abspath(os.path.join(os.path.abspath(os.path.dirname(__file__)), '..', 'Plots'))
    lm.savefig('{}/{}.png'.format(path, title))
```

- Also, Jaccard and Randindex values are calculated as in to verify the results with the ground truth values.

## 4.4 Visualization:

Figure1:




Enter the convergence threshold:1e-9

Enter max number of iterations:100

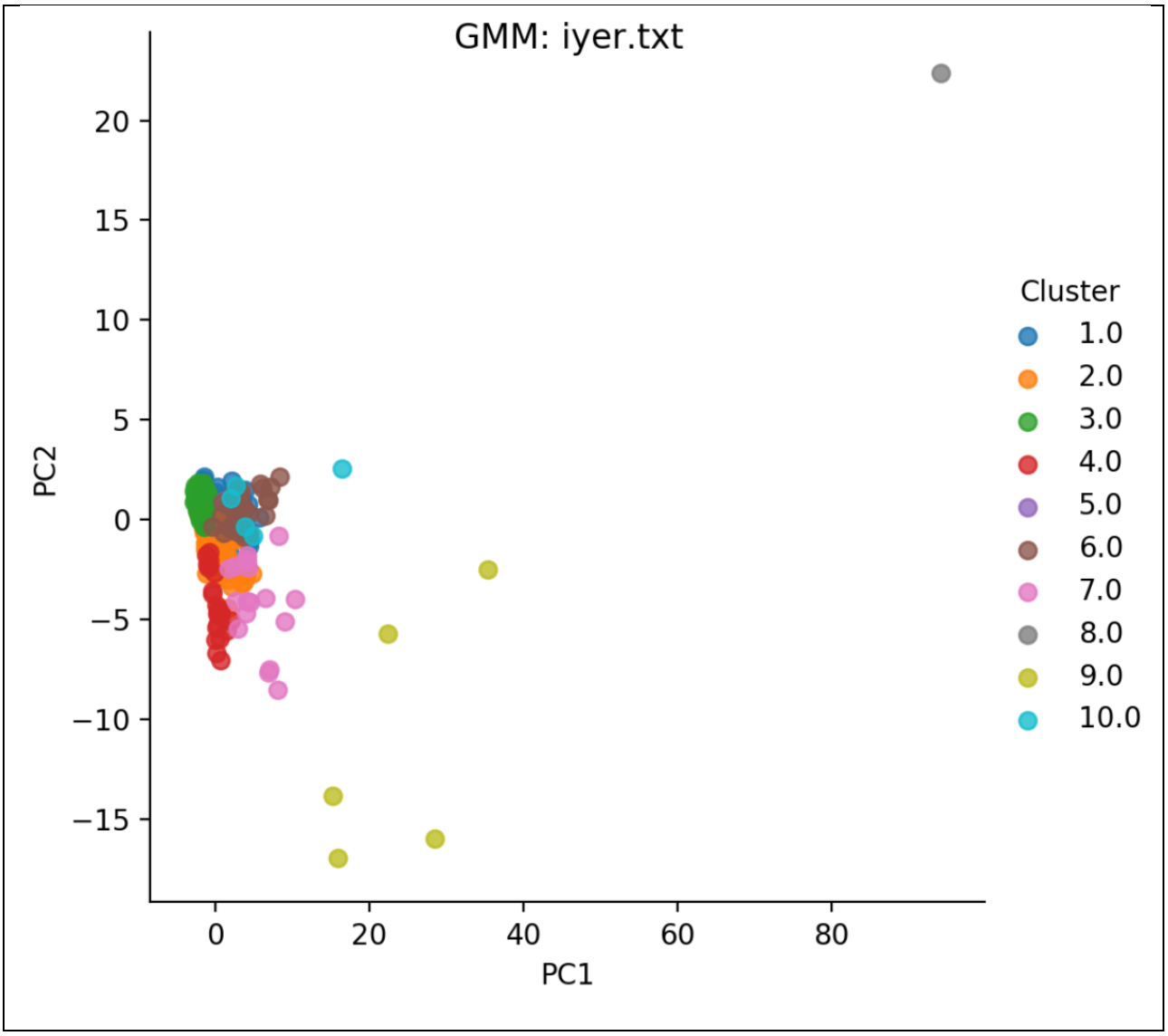
Enter the smoothing value:1e-9

Pi: [0.14916384642721395, 0.34992693290017635, 0.2719789578013504, 0.10903814158229778, 0.11989212128896143]

RandIndex: 0.7700340948750302

Jaccard Coefficient: 0.3431485315543287

Figure2:




Enter the convergence threshold:1e-9

Enter max number of iterations:100

Enter the smoothing value:1e-9

Enter number of clusters:10

Pi: [0.10650398183801077, 0.1761668878431402, 0.5178283919606413, 0.04105785779693818 5, 0.028347285582747093, 0.06371001939786938, 0.033786393694787815, 0.017359104122724296, 0.007619925329472179, 0.007620152433668754]

RandIndex: 0.7595187231797792

Jaccard Coefficient: 0.3456910767727356

### 4.5 Result Evaluation:

- GMM is also sensitive to parameters as different initializations of mean and covariance will change jaccard and randindex coefficient values resulting in different clusters
- It needs smoothing value while computation of probability distribution function value to avoid divide by zero errors.

### 4.6 Pros and Cons of Mixture Model:

Pros:

1. Gaussian Mixture Model gives probabilistic cluster assignments and probabilistic interpretation.
2. This algorithm works well for non-spherical clusters, can handle clusters with different probabilities.
3. Algorithm can handle clusters with varying sizes and variance.

Cons:

1. Prior initialization is critical might affects the results.
2. Distributions needs to be chosen appropriately.
3. This algorithm suffers from overfitting.

## 5. Spectral Clustering

### 5.1 Introduction:

This algorithm finds neighbors of nodes in a graph based which are based on the edges that connects them. Spectral Clustering generates clusters based on the eigen values matrices from the given number of observations.

In other words, a graph is formed between the given observations, where edges represents the similarities between those points, eigenvalues of the graph Laplacian is used to find the best possible number of clusters and eigenvectors used to find the actual cluster labels.

### 5.2 Algorithm:

1. *Compute Laplacian Matrix L*
2. *Find the eigenvalues and eigenvectors of L*
3. *Compute the first(smallest) K eigen vectors of L*
4. *Build embedded space from the eigenvectors corresponding to the k smallest eigenvalues.*
5. *Let U be the n*K matrix with eigenvectors as the columns.*
6. *Apply k-means clustering on the rows of U.*

### 5.3 Implementation:

- Read the parameters number of clusters and sigma value from the command line

```
k = int(input("Enter the number of clusters to cluster the datasets into:"))
sigma = float(input("Enter the sigma value:"))
```

- Create a weight (or similarity) matrix W of size N * N (where N is the size of the dataset) and fill it with gaussian distance between every pair of points calculated using the gaussian kernel:

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

- Then create a degree matrix of the same size and fill its diagonals with the total weight of edges incident on the point (here weight is taken from the previous matrix)
- Then calculate Laplacian Matrix L = D – W

```
def process_spectral_clustering(self):
    W = np.zeros((self.rows.shape[0], self.rows.shape[0]), dtype=np.float64)
    D = np.zeros((self.rows.shape[0], self.rows.shape[0]), dtype=np.float64)

    i = 0
    while i < self.data.shape[0]:
        j = i + 1
        while j < self.data.shape[0]:
            wt = e ** (-1 * (np.linalg.norm(self.data[i][2:] - self.data[j][2:])) ** 2 / (2 * self.sigma ** 2))

            W[i][j] = wt
            W[j][i] = wt

            D[i][i] += wt
            D[j][j] += wt

            j += 1

        i += 1
    L = D - W
```

- Now calculate eigen values and eigen vectors of the Laplacian matrix.
- Find the value of k for which the difference between consecutive eigen values is maximum.

```
def find_eigen_gap(self, eigenValues, ev):
    delta = 0
    idx = 0
    for i in range(1, len(ev)):
        tmp = abs(eigenValues[ev[i]] - eigenValues[ev[i-1]])
        if tmp > delta:
            delta = tmp
            idx = ev[i]
    return idx
```

- Using the value of k obtained from max eigen gap, compute the reduced embedded space.

```
w, v = np.linalg.eig(L)
ev = w.argsort()
k = self.find_eigen_gap(w, ev)
indices= ev[:k]
v = v[:,indices]

self.data = np.concatenate((self.data[:, 0:2], v), axis=1)
```

- Now run K-means algorithm on this reduced embedded space using the K means algorithm developed previously and cluster the points.

```
data_dict = dict()

for row in self.data:
    data_dict[row[0]] = row[2:]

self.initial_points = []

for idx in self.initial_indices:
    self.initial_points.append(data_dict[idx])

self.process_k_means()
```

- Run PCA on the result to reduce the dataset to two dimensions and visualize.

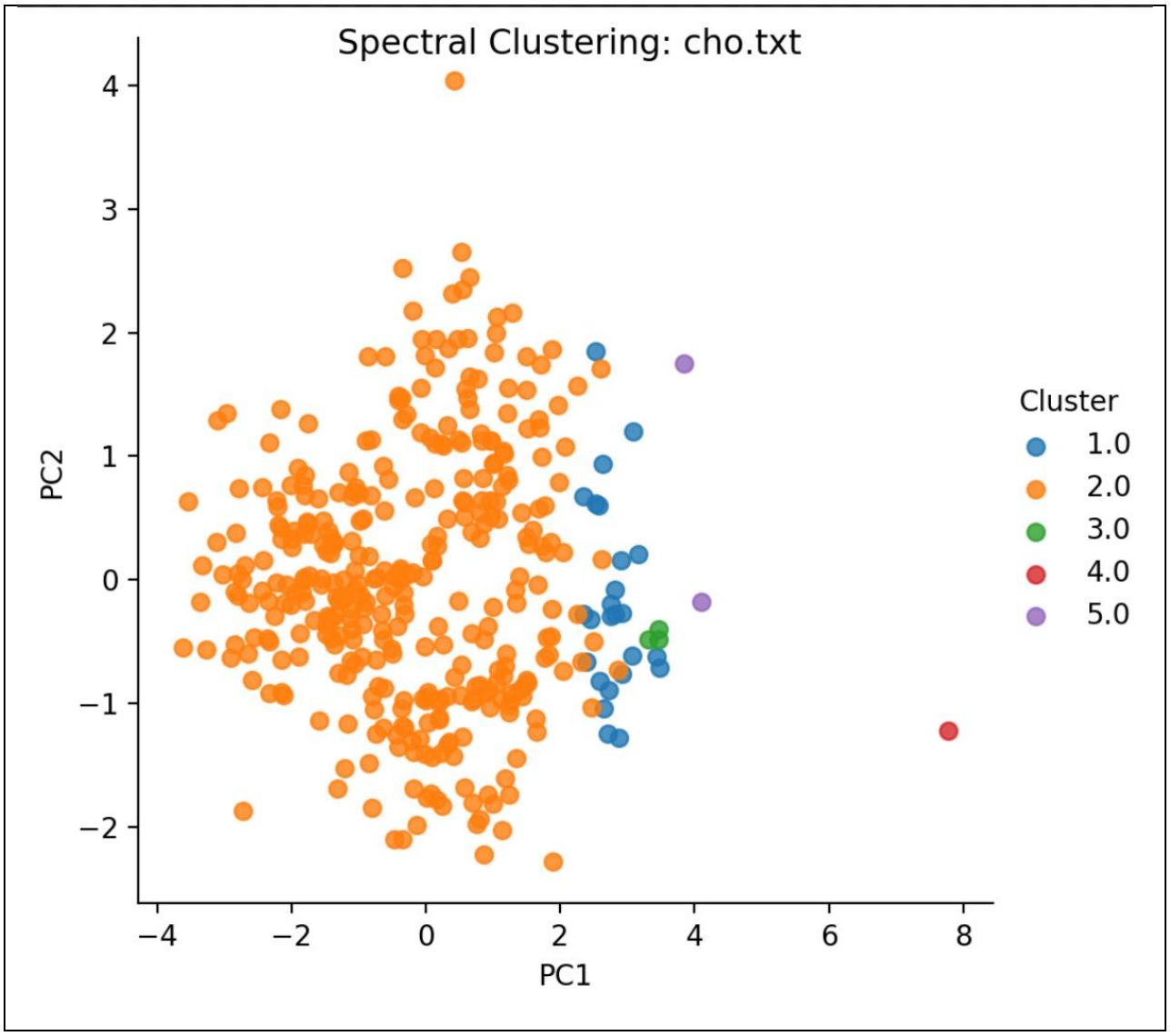
```
def pca(self, cluster_dict):
    pca = PCA(n_components=2, svd_solver='full')
    pca.fit(self.rows[:, 2:])
    principle_components_matrix = pca.transform(self.rows[:, 2:])
    df = pd.DataFrame(data = np.concatenate((principle_components_matrix, self.result[:, 1: 2]), axis=1), columns = ['PC1'
    self.plot(df, "Spectral Clustering: {}".format(self.file_name))

@staticmethod
def plot(df, title):
    lm = sns.lmplot(x='PC1', y='PC2', data=df, fit_reg=False, hue='Cluster')
    lm.fig.suptitle(title)
    plt.show()
    path = os.path.abspath(os.path.join(os.path.abspath(os.path.dirname(__file__)), '..', 'Plots'))
    lm.savefig('{}/{}.png'.format(path, title))
```

- Also, Jaccard and Randindex values are calculated as in to verify the results with the ground truth values.

## 5.4 Visualization:

Figure1:

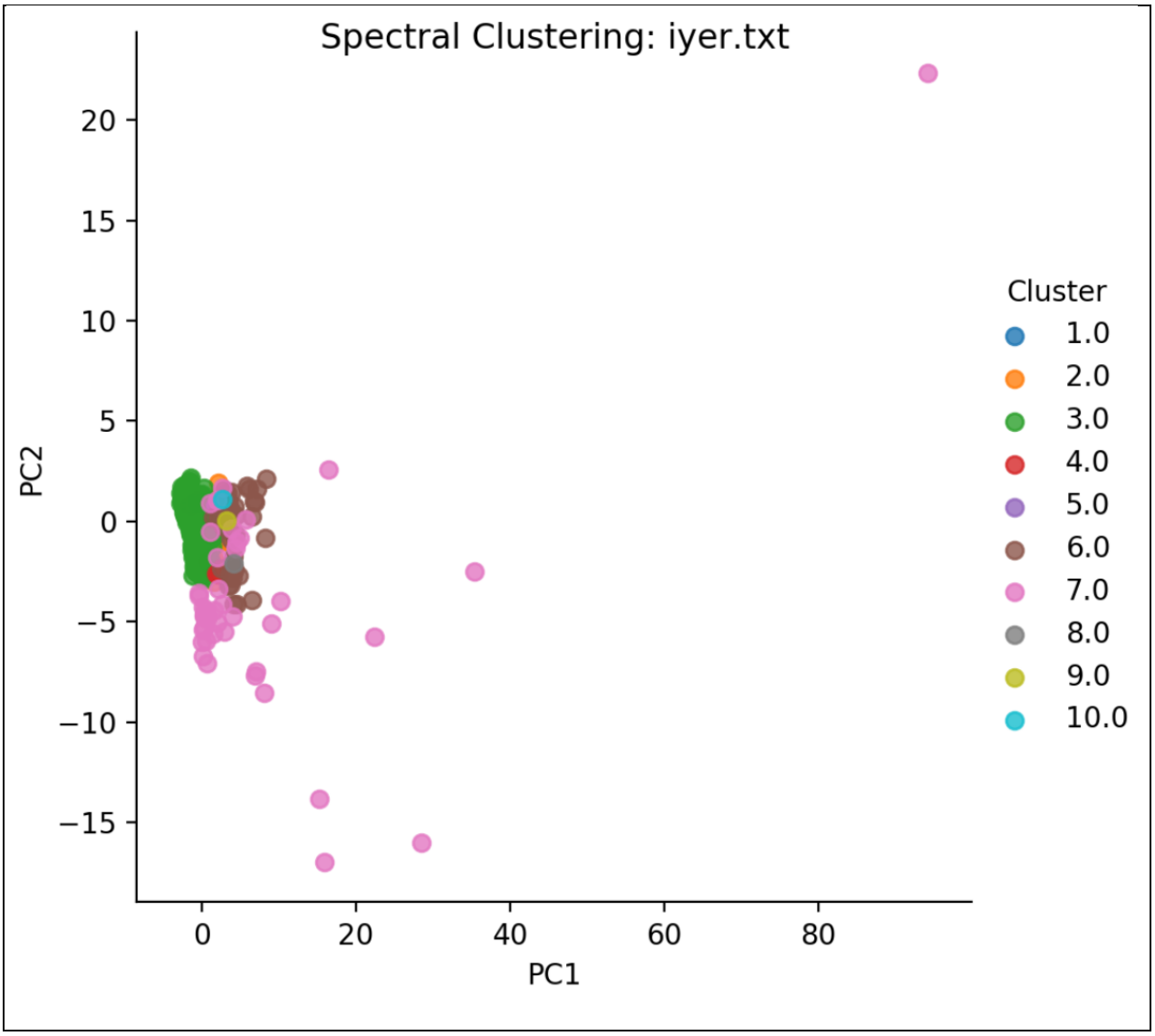



Sigma Value: 3

RandIndex: 0.6024993959569385

Jaccard Coefficient: 0.30552754391313525

Figure2:




Sigma Value: 5

RandIndex: 0.5129167305800089

Jaccard Coefficient: 0.21316072572554423

### 5.5 Result Evaluation:

- Spectral clustering is sensitive to sigma parameter value because of the use of gaussian kernel to find the similarity between points.

### 5.6 Pros and Cons of Spectral Clustering:

Pros:

1. This algorithm can be used for graph data and for arbitrary data.
2. Spectral Clustering is a flexible approach for finding clusters, when the other algorithms fails to cluster them in the desired output.

Cons:

1. This algorithm when applied on clusters with different scales, this clustering will fail when it finds k eigenvectors to bifurcate them into k clusters.
2. When applying recursive bi-partitioning there is a need of prominence in order to return more than two clusters.

## References and Citations:

Wikipedia:

https://www.python-course.eu/expectation_maximization_and_gaussian_mixture_models.php