|  |
| --- |
| **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.
* 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.

**3. Density-based clustering**

**3.1 Introduction:**

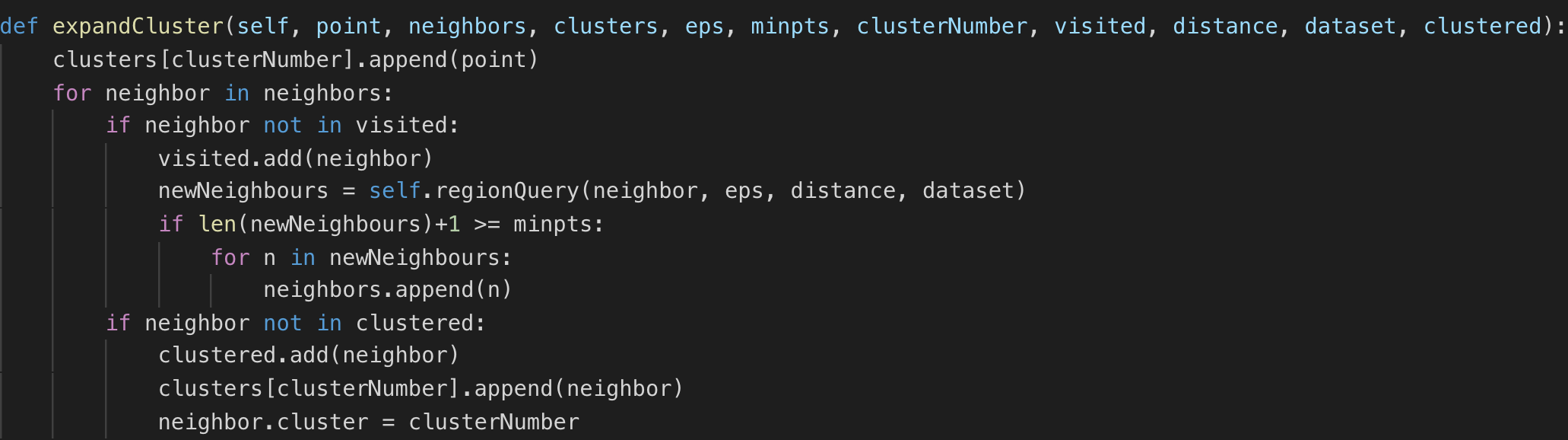
Density based clustering method defines a cluster as a maximal set of density-connected points. A high-density point is defined as any point with at least “MinPts” number of points in its ε-neighborhood where ε is the radius of neighborhood for each point. To understand the working of algorithms, the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

****

* 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.

**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:



For understanding 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 xi and xj.
  + 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 genes

        '''

        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:



* The ƛk – ƛk-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:**

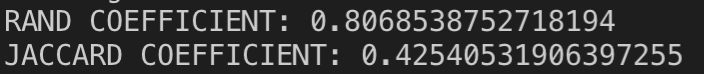
* The performance is greatly affected by noisy dataset.
* The process of computing eigen vectors gets very expensive when dealing with large datasets and consequently impacts the speed of the algorithm.
* Slower than k-means.

**Result Analysis**

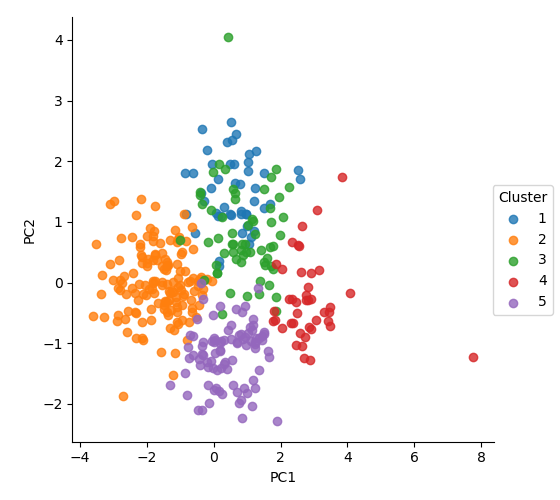
This section gives the results of each algorithm on two datasets: cho.txt and iyer.txt and compares each result with all the results of other algorithms.

**K-means:**

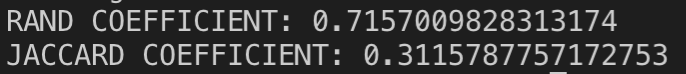
**CHO.TXT:**

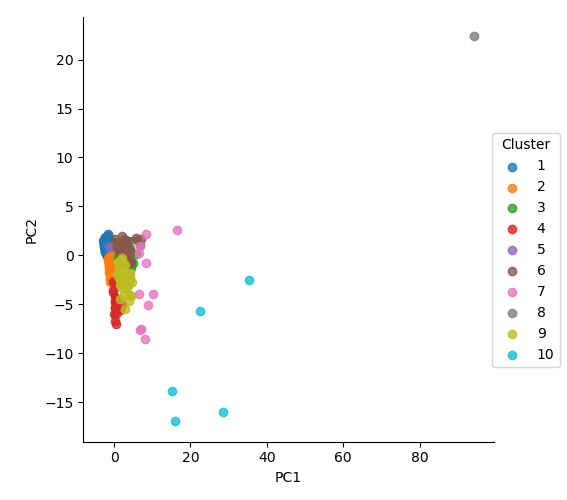
****

**K = 5 and max\_iterations = 20**

****

**IYER.TXT:**

****

**K = 10 and max\_iterations = 10**

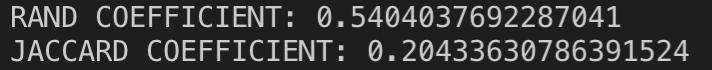
**Hierarchical:**

**CHO.TXT**

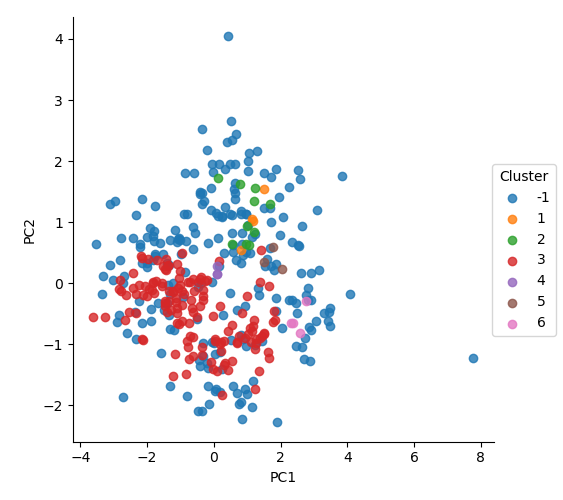
**IYER.TXT**

**Density based clustering (DBSCAN):**

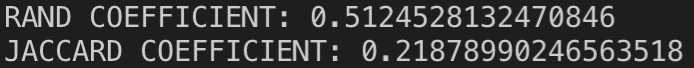
**CHO.TXT:**

****

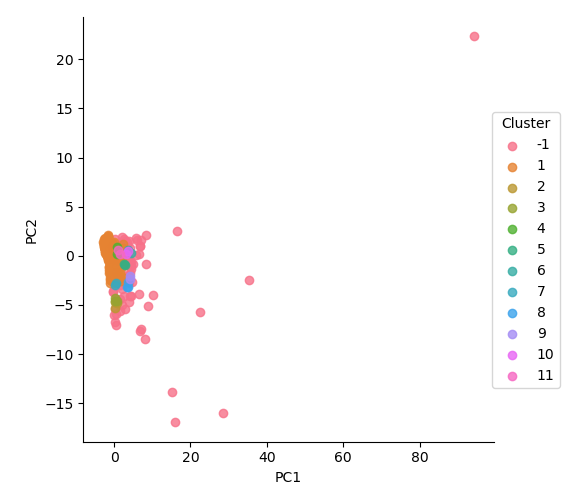
**Epsilon = 1 MinPts = 4**

****

**IYER.TXT:**

****

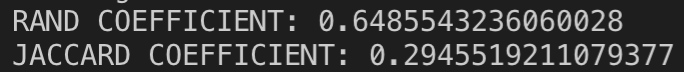
**EPSILON = 1.20 MinPts = 3**

****

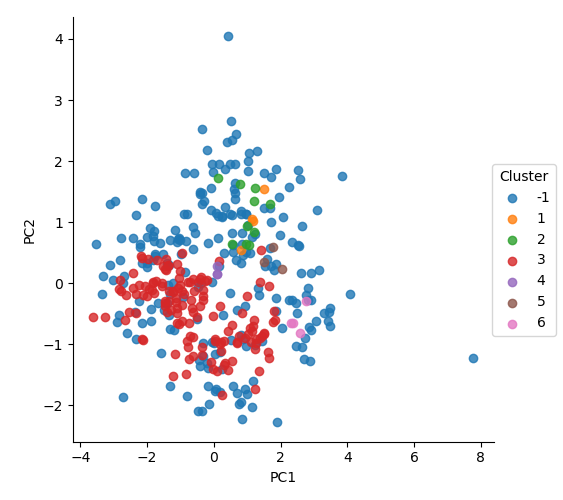
**4. GMM**

**5. Spectral Clustering:**

**CHO.TXT:**

****

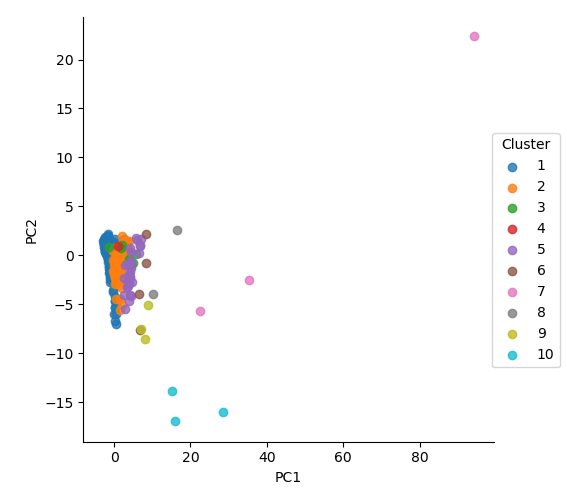
**Epsilon = 1.5**

****

**IYER.TXT:**

****

**Epsilon = 1.8**

****