**MACHINE LEARNING LAB**

|  |
| --- |
| Lab 9:  Implement K means to group the visitors to a website using just their age (a one-dimensional space) into k clusters |
| DataSet:  kmean.txt:- it is a text file consisting of different ages groups of website visitors |
| **Example:**  Ages: 15,15,16,19,19,20,20,21,22,28,35,40,41,42,43,44,60,61,65  **Initial clusters:**  Centroid (C1) = 16 [16] Centroid (C2) = 22 [22] I  **Iteration 1**:  C1 = 15.33 [15,15,16] C2 = 36.25 [19,19,20,20,21,22,28,35,40,41,42,43,44,60,61,65]  **Iteration 2**:  C1 = 18.56 [15,15,16,19,19,20,20,21,22] C2 = 45.90 [28,35,40,41,42,43,44,60,61,65]  **Iteration 3**:  C1 = 19.50 [15,15,16,19,19,20,20,21,22,28] C2 = 47.89 [35,40,41,42,43,44,60,61,65]  **Iteration 4**:  C1 = 19.50 [15,15,16,19,19,20,20,21,22,28] C2 = 47.89 [35,40,41,42,43,44,60,61,65]  No change between iterations 3 and 4 has been noted. By using clustering, 2 groups have been identified 15-28 and 35-65. |
| K-Means Clustering K-Means clustering intends to partition *n* objects into *k* clusters in which each object belongs to the cluster with the nearest mean. This method produces exactly *k* different clusters of greatest possible distinction. The best number of clusters *k* leading to the greatest separation (distance) is not known as a priori and must be computed from the data. The objective of K-Means clustering is to minimize total intra-cluster variance, or, the squared error function:    **Algorithm**   1. Clusters the data into *k* groups where *k*  is predefined. 2. Select *k* points at random as cluster centers. 3. Assign objects to their closest cluster center according to the *Euclidean distance* function. 4. Calculate the centroid or mean of all objects in each cluster. 5. Repeat steps 2, 3 and 4 until the same points are assigned to each cluster in consecutive rounds.     K-Means is relatively an efficient method. However, we need to specify the number of clusters, in advance and the final results are sensitive to initialization and often terminates at a local optimum. Unfortunately there is no global theoretical method to find the optimal number of clusters. A practical approach is to compare the outcomes of multiple runs with different *k* and choose the best one based on a predefined criterion. In general, a large *k* probably decreases the error but increases the risk of overfitting. |
| Code:  import random  def k\_means(data, k):  # 1. k initial "means" are randomly selected from the data set.  means = random.sample(data, k)  def kmeans1(data, means):  # 2. k clusters are created by associating every observation with the nearest mean.  clusters = [ [] for \_ in means ]  for d in data:  distances = [ (abs(d-m), i) for i,m in enumerate(means) ]  \_,closest = min(distances)  clusters[closest].append(d)  return sorted(clusters)  prev = []  curr = kmeans1(data, means)  while curr != prev:  prev = curr  # 3. The centroid of each of the k clusters becomes the new means.  means = [ sum(c) / max(1,len(c)) for c in curr ]  curr = kmeans1(data, means)  # 4. ...repeated until convergence has been reached.  return zip(means, curr)  if \_\_name\_\_ == '\_\_main\_\_':  Ages = [15,15,16,19,19,20,20,21,22,28,35,40,41,42,43,44,60,61,65]  print(list(k\_means(Ages, 10))) |
| Output: for k=10 clusters  [(15.333333333333334, [15, 15, 16]),  (19.0, [19, 19]),  (20.0, [20, 20]),  (21.5, [21, 22]),  (28.0, [28]),  (35.0, [35]),  (40.5, [40, 41]),  (42.0, [42]),  (43.5, [43, 44]),  (62.0, [60, 61, 65])] |
| To Do:  The initial choice of centroids can affect the output clusters ? Justify using varying inputs and vary the k value and find the best solution |
| Learning outcome:  1. K-Means clustering is very useful for applications which are one level based clustering techniques  2. Useful for smaller datasets  3. Always tries to form spherical shaped clusters. |

|  |
| --- |
| Implement hierarchical clustering for some Italian cities ,given distances in kilometers |
| DataSet:  cluster.txt:- it is a text file consisting of distances in kilometers between some Italian cities |
| Let’s now see a simple example: a hierarchical clustering of distances in kilometers between some Italian cities. The method used is single-linkage.  Input distance matrix (L = 0 for all the clusters):        The nearest pair of cities is MI and TO, at distance 138. These are merged into a single cluster called "MI/TO". The level of the new cluster is L(MI/TO) = 138 and the new sequence number is m = 1. Then we compute the distance from this new compound object to all other objects. In single link clustering the rule is that the distance from the compound object to another object is equal to the shortest distance from any member of the cluster to the outside object. So the distance from "MI/TO" to RM is chosen to be 564, which is the distance from MI to RM, and so on.  After merging MI with TO we obtain the following matrix:        min d(i,j) = d(NA,RM) = 219 => merge NA and RM into a new cluster called NA/RM L(NA/RM) = 219 m = 2      min d(i,j) = d(BA,NA/RM) = 255 => merge BA and NA/RM into a new cluster called BA/NA/RM L(BA/NA/RM) = 255 m = 3      min d(i,j) = d(BA/NA/RM,FI) = 268 => merge BA/NA/RM and FI into a new cluster called BA/FI/NA/RM L(BA/FI/NA/RM) = 268 m = 4      Finally, we merge the last two clusters at level 295.  The process is summarized by the following hierarchical tree: |
| **hierarchical clustering** (also called **hierarchical cluster analysis** or **HCA**) is a method of cluster analysis which seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two types:   * **Agglomerative**: This is a "bottom up" approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. * **Divisive**: This is a "top down" approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.   In general, the merges and splits are determined in a greedy manner. The results of hierarchical clustering are usually presented in a dendrogram.  Given a set of N items to be clustered, and an NxN distance (or similarity) matrix, the basic process of Johnson's (1967) hierarchical clustering is this:   1. Start by assigning each item to its own cluster, so that if you have N items, you now have N clusters, each containing just one item. Let the distances (similarities) between the clusters equal the distances (similarities) between the items they contain. 2. Find the closest (most similar) pair of clusters and merge them into a single cluster, so that now you have one less cluster. 3. Compute distances (similarities) between the new cluster and each of the old clusters. 4. Repeat steps 2 and 3 until all items are clustered into a single cluster of size N.   Step 3 can be done in different ways, which is what distinguishes single-link from complete-link and average-link clustering. In single-link clustering (also called the connectedness or minimum method), we consider the distance between one cluster and another cluster to be equal to the shortest distance from any member of one cluster to any member of the other cluster. If the data consist of similarities, we consider the similarity between one cluster and another cluster to be equal to the greatest similarity from any member of one cluster to any member of the other cluster. In complete-link clustering (also called the diameter or maximum method), we consider the distance between one cluster and another cluster to be equal to the longest distance from any member of one cluster to any member of the other cluster. In average-link clustering, we consider the distance between one cluster and another cluster to be equal to the average distance from any member of one cluster to any member of the other cluster. A variation on average-link clustering is the UCLUS method of D'Andrade (1978) which uses the median distance. |
| class Cluster:  def \_\_init\_\_(self):  pass  def \_\_repr\_\_(self):  return '(%s,%s)' % (self.left, self.right)  def add(self, clusters, grid, lefti, righti):  self.left = clusters[lefti]  self.right = clusters[righti]  # merge columns grid[row][righti] and row grid[righti] into corresponding lefti  for r in grid:  r[lefti] = min(r[lefti], r.pop(righti))  grid[lefti] = list(map(min, zip(grid[lefti], grid.pop(righti))))  clusters.pop(righti)  return (clusters, grid)  def agglomerate(labels, grid):  """  given a list of labels and a 2-D grid of distances, iteratively agglomerate  hierarchical Cluster  """  clusters = labels  while len(clusters) > 1:  # find 2 closest clusters  print(clusters)  distances = [(1, 0, grid[1][0])]  for i,row in enumerate(grid[2:]):  distances += [(i+2, j, c) for j,c in enumerate(row[:i+2])]  j,i,\_ = min(distances, key=lambda x:x[2])  # merge i<-j  c = Cluster()  clusters, grid = c.add(clusters, grid, i, j)  clusters[i] = c  return clusters.pop()  if \_\_name\_\_ == '\_\_main\_\_':  # Ref #1  ItalyCities = ['BA','FI','MI','NA','RM','TO']  ItalyDistances = [  [ 0, 662, 877, 255, 412, 996],  [662, 0, 295, 468, 268, 400],  [877, 295, 0, 754, 564, 138],  [255, 468, 754, 0, 219, 869],  [412, 268, 564, 219, 0, 669],  [996, 400, 138, 869, 669, 0]]  print(agglomerate(ItalyCities, ItalyDistances)) |
| Outcome:  ['BA', 'FI', 'MI', 'NA', 'RM', 'TO']  ['BA', 'FI', (MI,TO), 'NA', 'RM']  ['BA', 'FI', (MI,TO), (NA,RM)]  [(BA,(NA,RM)), 'FI', (MI,TO)]  [((BA,(NA,RM)),FI), (MI,TO)]  (((BA,(NA,RM)),FI),(MI,TO)) |
| To Do:  1.modify the code to get dendogram of output  2.apply the program for Metropolitan cities of India with approximate distance |
| Learning Outcome:  1. Used for appications which are based on multilevel clustering techniques  2. It is a bottom up clustering technique |
|  |