**Algorithm**

**K-Means-**

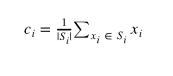
The K-Means algorithm is used for cluster analysis by dividing data points into k clusters. The K means algorithm will group the data into the cluster based on feature similarity.

1. The input of algorithm is Data points with n features and the number of clusters given by K. Initially K centroids are assigned randomly. The points in the dataset are assigned to a cluster based on Euclidean distance.



Where, dist(ci,x)2 is the Euclidean Distance

1. The Centroid is then computed again by taking mean of all points coming in the same cluster.



The Steps are repeated until the centroid does not change beyond a limit. The limit must be set while coding.

Source Code –

**import** numpy **as** num  
**import** matplotlib.pyplot **as** pty  
**from** matplotlib **import** style  
**import** pandas **as** pan

**import** numpy **as** num  
**import** matplotlib.pyplot **as** pty  
**from** matplotlib **import** style  
**import** csv  
**import** os  
**import** subprocess  
**from** collections **import** Counter  
style.use(**'seaborn'**)  
  
**class** KMean:  
 **def** \_\_init\_\_(self, k=11, limit=0.0001, counts=99999999): *# Parameters and Initial count* self.counts = counts  
 self.k = k  
 self.limit = limit  
 **def** dev(self, data):  
 self.cent = {}  
 **for** count **in** range(self.k): *# assign initial centroids* self.cent[count] = data[count\*15]  
 **for** count **in** range(self.counts):  
 self.classes = {}  
 **for** count **in** range(self.k):  
 self.classes[count] = []  
 **for** i **in** data:  
 dist = [num.linalg.norm(i - self.cent[c]) **for** c **in** self.cent]  
 cl = dist.index(min(dist)) *#find min. distance* self.classes[cl].append(i)  
 p = dict(self.cent)  
 **for** cl **in** self.classes:  
 self.cent[cl] = num.average(self.classes[cl], axis=0) *#take average value* Opt = **True  
 for** c **in** self.cent:  
 temp = p[c]  
 curr = self.cent[c]  
 **if** num.sum((curr - temp) / temp \* 100.0) > self.limit:  
 Opt = **False  
 if** Opt: *#check optimal value* **break  
  
 def** pred(self, data):  
 dist = [num.linalg.norm(data - self.cent[c]) **for** c **in** self.cent]  
 cl = dist.index(min(dist))  
 **return** cl  
  
**def** main():  
 df = pan.read\_csv(**r"F:\communication and networking\A3\driverlog.csv"**) *#input data* df = df[[**'Dist.'**, **'Speed'**]]  
 X = df.values  
 km = KMean(11)  
 km.dev(X)  
 colors = 10 \* [**"y"**, **"g"**, **"m"**, **"r"**, **"r"**, **"m"** , **"c"** ,**"w"**, **"w"**, **"b"**,**"b"**] *# define color* **for** cl **in** km.classes:  
 color = colors[cl]  
 **for** i **in** km.classes[cl]:  
 pty.scatter(i[0], i[1], color=color, s=20, edgecolor=**'k'**) *#plot points* **for** c **in** km.cent:  
 pty.scatter(km.cent[c][0], km.cent[c][1], s=50, marker=**"x"**, edgecolor=**'k'**, c=**'k'**) *#plot centroid* pty.title(**r'KMeans'**, fontsize=15)  
 pty.show()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

style.use(**'seaborn'**)  
  
**class** KMean:  
 **def** \_\_init\_\_(self, k=11, limit=0.0001, counts=99999999): *# Parameters and Initial count* self.counts = counts  
 self.k = k  
 self.limit = limit  
 **def** dev(self, data):  
 self.cent = {}  
 **for** count **in** range(self.k): *# assign initial centroids* self.cent[count] = data[count\*15]  
 **for** count **in** range(self.counts):  
 self.classes = {}  
 **for** count **in** range(self.k):  
 self.classes[count] = []  
 **for** i **in** data:  
 dist = [num.linalg.norm(i - self.cent[c]) **for** c **in** self.cent]  
 cl = dist.index(min(dist)) *#find min. distance* self.classes[cl].append(i)  
 p = dict(self.cent)  
 **for** cl **in** self.classes:  
 self.cent[cl] = num.average(self.classes[cl], axis=0)

*#take average value* Opt = **True  
 for** c **in** self.cent:  
 temp = p[c]  
 curr = self.cent[c]  
 **if** num.sum((curr - temp) / temp \* 100.0) > self.limit:  
 Opt = **False  
 if** Opt: *#check optimal value* **break  
  
 def** pred(self, data):  
 dist = [num.linalg.norm(data - self.cent[c]) **for** c **in** self.cent]  
 cl = dist.index(min(dist))  
 **return** cl  
  
**def** main():  
 df = pan.read\_csv(**r"F:\communication and networking\A3\driverlog.csv"**)

*#input data* df = df[[**'Dist.'**, **'Speed'**]]  
 X = df.values  
 km = KMean(11)  
 km.dev(X)  
 colors = 10 \* [**"y"**, **"g"**, **"m"**, **"r"**, **"r"**, **"m"** , **"c"** ,**"w"**, **"w"**, **"b"**,**"b"**]

*# define color* **for** cl **in** km.classes:  
 color = colors[cl]  
 **for** i **in** km.classes[cl]:  
 pty.scatter(i[0], i[1], color=color, s=20, edgecolor=**'k'**) *#plot points* **for** c **in** km.cent:  
 pty.scatter(km.cent[c][0], km.cent[c][1], s=50, marker=**"x"**, edgecolor=**'k'**, c=**'k'**) *#plot centroid* pty.title(**r'KMeans'**, fontsize=15)  
 pty.show()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

1. limit=0.0001 – is the limit set for variation in centroid.
2. Counts is the limit of number of maximum iterations possible.
3. r"F:\communication and networking\A3\driverlog.csv") gives the location of source file.
4. Pred () function will find the new centroids
5. The code finds 11 clusters based on the data provided in driverlog.csv file by calling KMean(11) and this will help to form 11 clusters.
6. Pyplot is used for plotting the diagram

The other Python Packages imported are numpy, matplotlib.pyplot matplotlib-style  
pandas, csv, os,, subprocess, collections- Counter.

Advantages –

1. Can be used for unlabeled data
2. Easy to implement and interpret
3. Lower computational Cost

Disadvantages –

1. May converge to local optima

Reference :

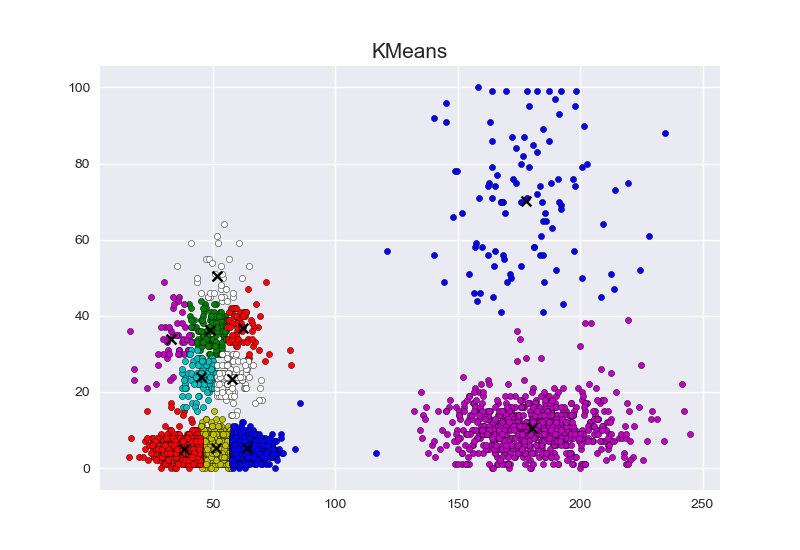
https://en.wikipedia.org/wiki/K-means\_clustering

<https://www.datascience.com/blog/k-means-clustering>

https://home.deib.polimi.it/matteucc/Clustering/tutorial\_html/kmeans.html

https://github.com/madhug-nadig/Machine-Learning-Algorithms-from-Scratch/blob/master/K%20Means%20Clustering.py

**Results :**

11 Clusters where found using Kmeans algorithm and the output is plotted below.

**DBSCAN-**

DBSCAN clustering algorithm will find all nearby points and pack them into a cluster. It finds the nearby points using Euclidean distance and for a minimum number of nearby points, a cluster is formed. The points which are lying outside the clusters and marked as outliers.

The Parameters are

eps: The Euclidean distance between points so that they are in single cluster. Choosing optimal value of eps is need as low value will cause most of data not being classified and a high value will cause all data to converge

minPoints: The minimum number of points to form a Cluster or dense region. It is usually taken more than Dimension of data set

dim : dimension of Dataset

Core Points are having more than minimum point in vicinity defined by epsilon. Border point has less than minimum point in vicinity but is near a core point. If a point is not a core point or border point it will be classified as noise.

Direct-density reachable means the points which are within eps from the point. Indirect-density reachable means the point is reachable from a core point through other points which are in eps distance. For every point, if it is not classified and is a core point, then find all points which are reachable from the point.

Source Code

**import** re, csv, sys  
**from** mpl\_toolkits.mplot3d **import** Axes3D  
**import** matplotlib.pyplot **as** plt  
**from** scipy.spatial **import** distance  
**import** numpy **as** np  
  
  
Param = **'values\_parameters'**DATA = **'driverlog.csv'  
  
class** Cluster(object):  
 **def** \_\_init\_\_(self, name, dim):  
 self.name = name  
 self.dim = dim  
 self.points = []  
  
 **def** append(self, point):  
 self.points.append(point)  
  
 **def** finddata(self):  
 **return** self.points  
  
 **def** fun(self):  
 self.points = []  
  
 **def** findpx(self):  
 **return** [p[0] **for** p **in** self.points]  
  
 **def** findpy(self):  
 **return** [p[1] **for** p **in** self.points]  
  
 **def** has(self, point):  
 **return** point **in** self.points  
  
 **def** \_\_str\_\_(self):  
 **return "%s: %d points"** % (self.name, len(self.points))  
  
  
**class** DBScanner:  
  
 **def** \_\_init\_\_(self, config):  
 self.eps = config[**'eps'**]  
 self.min\_pts = config[**'min\_pts'**]  
 self.dim = config[**'dim'**]  
 self.clusters = set()  
 self.cluster\_count = 0  
 self.visited = []  
 self.color = [**'b'**, **'g'**, **'r'**, **'c'**, **'m'**, **'y'**, **'k'**, **'w'**]  
  
 **def** dbscan(self, data):  
 self.init\_params()  
 self.data = data  
  
 *## Setting up the plot* fig = plt.figure()  
  
 axis\_proj = **'rectilinear'  
 if** self.dim > 2:  
 axis\_proj = **'%dd'** % self.dim  
  
 ax = fig.add\_subplot(111, projection=axis\_proj)  
  
 *# default noise cluster* noise = Cluster(**'Noise'**, self.dim)  
 self.clusters.add(noise)  
  
 **for** point **in** data:  
 **if** point **not in** self.visited:  
 self.visited.append(point)  
 neighbour\_pts = self.region\_query(point)  
 **if** len(neighbour\_pts) < self.min\_pts:  
 noise.append(point)  
 **else**:  
 name = **'cluster-%d'** % self.cluster\_count  
 new\_cluster = Cluster(name, self.dim)  
  
 self.cluster\_count += 1  
 self.expand\_cluster(new\_cluster, point, neighbour\_pts)  
  
 **if** self.dim == 2:  
 ax.scatter(new\_cluster.findpx(), new\_cluster.findpy(),  
 c=self.color[self.cluster\_count % len(self.color)],  
 marker=**'o'**, label=name, s=20, edgecolor=**'k'**)  
  
 ax.hold(**True**)  
  
 **if** len(noise.finddata()) != 0:  
 ax.scatter(noise.findpx(), noise.findpy(), marker=**'x'**, label=noise.name, s=20)  
  
 print(**"Number of clusters found: %d"** % self.cluster\_count)  
  
 ax.hold(**False**)  
 ax.legend(loc=**'upper left'**)  
 ax.grid(**True**)  
 plt.title(**r'DBSCAN Clustering'**, fontsize=18)  
 plt.show()  
  
 **def** expand\_cluster(self, cluster, point, neighbour\_pts):  
 cluster.append(point)  
 **for** p **in** neighbour\_pts:  
 **if** p **not in** self.visited:  
 self.visited.append(p)  
 np = self.region\_query(p)  
 **if** len(np) >= self.min\_pts:  
 **for** n **in** np:  
 **if** n **not in** neighbour\_pts:  
 neighbour\_pts.append(n)  
  
 **for** other\_cluster **in** self.clusters:  
 **if not** other\_cluster.has(p):  
 **if not** cluster.has(p):  
 cluster.append(p)  
  
 **if** self.cluster\_count == 0:  
 **if not** cluster.has(p):  
 cluster.append(p)  
  
 self.clusters.add(cluster)  
  
 **def** get\_distance(self, from\_point, to\_point):  
 p1 = [from\_point[k] **for** k **in** range(self.dim)]  
 p2 = [to\_point[k] **for** k **in** range(self.dim)]  
 **return** distance.euclidean(p1, p2)  
  
 **def** region\_query(self, point):  
 result = []  
 **for** d\_point **in** self.data:  
 **if** d\_point != point:  
 **if** self.get\_distance(d\_point, point) <= self.eps:  
 result.append(d\_point)  
 **return** result  
  
 **def** init\_params(self):  
 self.clusters = set()  
 self.cluster\_count = 0  
 self.visited = []  
  
**def** get\_data(config):  
 data = []  
 **with** open(DATA, **'r'**) **as** file\_obj:  
 csv\_reader = csv.reader(file\_obj)  
 **for** id\_, row **in** enumerate(csv\_reader):  
 **if** len(row) < config[**'dim'**]:  
 print (**"Dimension dont match"** % (config[**'dim'**], len(row)))  
 sys.exit()  
 **else**:  
 point = {**'id'**:id\_}  
 **for** dim **in** range(0, config[**'dim'**]):  
 point[dim] = float(row[dim])  
 data.append(point)  
 **return** data  
  
  
**def** read\_config():  
 config = {}  
 **try**:  
 **with** open(Param, **'r'**) **as** file\_obj:  
 **for** line **in** file\_obj:  
 **if** line[0] != **'#' and** line.strip() !=**''**:  
 key, value = line.split(**'='**)  
 **if '.' in** value.strip():  
 config[key.strip()] = float(value.strip())  
 **else**:  
 config[key.strip()] = int(value.strip())  
 **except**:  
 print (**"Unable to get Parameters"**)  
 sys.exit()  
 **return** config  
  
**def** main():  
 config = read\_config()  
 dbc = DBScanner(config)  
 data = get\_data(config)  
 dbc.dbscan(data)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

1. Eps and min\_pts are the input parameters for the algorithm. They are defined in value\_parameters file. It is read using read\_config() file. Data is accepted from driverlog.csv from get\_data() file.
2. Dimension of dataset is set in dim variable which is set as 2 in value\_parameters file.
3. They algorithm runs as given above. It will take a point and check whether it is visited. If it is not visited, and near a core point, it is added to a cluster. Distance is found using get\_distance() function.
4. If it does not have a nearby core or boundary point and cannot be a core point, it is classified as noise.
5. expand\_cluster() is for adding new points to cluster and dbscan() helps to find the DBSCAN of the dataset
6. findpx helps to find x coordinate and findpy helps to find y coordinate of the points in dataset

**Advantages**

1. Can be used with data having Noise
2. Cluster can have odd shapes

**Disadvantage**

1. Parameters should be properly selected.
2. Difficulties with varying densities.

Reference :

https://en.wikipedia.org/wiki/DBSCAN

<https://www.cse.buffalo.edu/~jing/cse601/fa12/materials/clustering_density.pdf>

<https://towardsdatascience.com/how-dbscan-works-and-why-should-i-use-it-443b4a191c80>

<https://github.com/SushantKafle/DBSCAN>

**Result**

3 cluster are found with DBSCAN with eps=8 and min\_points = 11 and is shown below. A lot of data points didn’t fall under any cluster and was classified as noise.

**Conclusion**

K-Means and DBSCAN algorithm are implemented on driverlog.csv file and the output were plotted. 11 clusters were plotted in K-Means with k = 11 and 3 clusters were found in DBSCAN with eps = 8 and Minimum points = 11. DBSCAN was found to have more noise data due to parameters selected.