Rohit Jyotiba Chougule, 19200240

Assignment No. 6

Question 1.

The dataset in the question has 2D data points, which was imported into the data frame to perform K-means clustering algorithm. The KMeans method from sklearn.cluster is used implement K-means clustering algorithm. Clustering is a technique for grouping data those are very similar to each other. The features of data points in one cluster are very similar to those within 2 clusters.

The image *question_1.png* shows the output for KMeans clustering value using k=3 as suggested in the question. As seen the image, we can identify if that the KMeans clustering has created 3 clusters for the given dataset and can be distinguished from the above scatter plot.

The red markers in the above graph denotes the centroid for the Cluster formed. The clusters are plotted in the scatter plot using the Cluster label function:

ClusterClass = pd.DataFrame(cluster.predict(df))

In k-means clustering, random k data points (say cluster centre) are selected after which the nearest distant points to the cluster centre and clustered accordingly. Once all data points are grouped then a mean of each cluster is calculated, and the above process is repeated unless there are any data points left. The clusters in the dataset are namely: 0, 1, 2

Here the clusters formed can be either calculated using Euclidean distance or distance between two points. The class label 0 is the only class in the cluster, and hence overlaps with the centroid for the cluster. The cluster 1 and 2 seem to be well distinguishing from each other, however, the data points in cluster 1 seem to be more tightly clustered as compared to the data points in cluster 2.

The output of the graph and the new dataframe is exported using.

df.to_csv and plt.tosaveimage as suggested in the question.

Question 2:

We discard the columns NAME, MANUF, TYPE, and RATING using:

df2_new = df2.drop(['NAME', 'MANUF', 'TYPE', 'RATING'], axis=1)

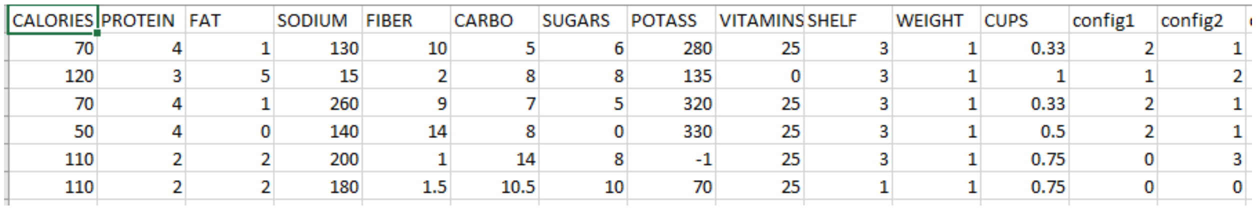
The KMeans can again be applied in this function, where the number of clusters to be created are defined. However, the *n_int,* which is the maximum number of iterations for which the kmeans algorithm will run with different centroid values. We have specified the *n_clusters* value to 5 which specifies number of clusters to be formed. Here is *n_int = 5* which denotes the number of times the k-means clustering algorithm should run independently, each time with different random centroids and then select the one which gives clusters that are tightly clustered as compared to other iterations. The *max_iter* value here defines the number of iterations the k means algorithm for a single run.

The two configurations that we used for K-means clustering are:

- *kmeans2 = KMeans(n_clusters=5, random_state=0, n_init=5, max_iter=100).fit(df2_new)*
- *kmeans2 = KMeans(n_clusters=5, random_state=0, n_init=100, max_iter=100).fit(df2_new)*

The results for both the configuration are indeed different, as in the second configuration we have specified *n_init=100,* that would run the k-means algorithm to run 100 times independently and optimize the cluster to find its cluster class.

The dataframe head output below shows a sample of the output difference in the cluster class from the column config1 and config2.

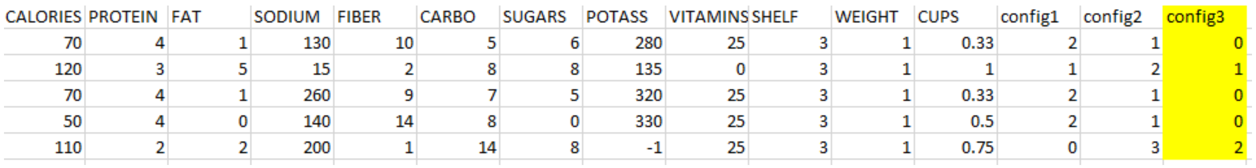


CALORIES	PROTEIN	FAT	SODIUM	FIBER	CARBO	SUGARS	POTASS	VITAMINS	SHELF	WEIGHT	CUPS	config1	config2
70	4	1	130	10	5	6	280	25	3	1	0.33	2	1
120	3	5	15	2	8	8	135	0	3	1	1	1	2
70	4	1	260	9	7	5	320	25	3	1	0.33	2	1
50	4	0	140	14	8	0	330	25	3	1	0.5	2	1
110	2	2	200	1	14	8	-1	25	3	1	0.75	0	3
110	2	2	180	1.5	10.5	10	70	25	1	1	0.75	0	0

Another configuration that we use is:

- *kmeans2 = KMeans(n_clusters=3, random_state=0, n_init=5, max_iter=100).fit(df2_new)*

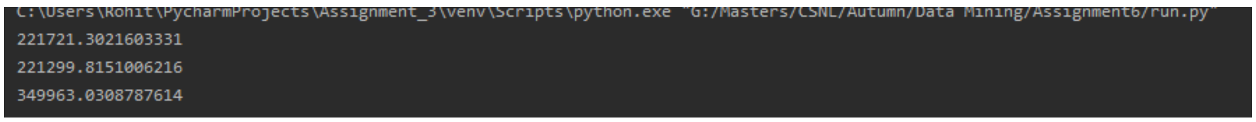
Here, we define the number of clusters to be three with the same above configuration to check the results:



CALORIES	PROTEIN	FAT	SODIUM	FIBER	CARBO	SUGARS	POTASS	VITAMINS	SHELF	WEIGHT	CUPS	config1	config2	config3
70	4	1	130	10	5	6	280	25	3	1	0.33	2	1	0
120	3	5	15	2	8	8	135	0	3	1	1	1	2	1
70	4	1	260	9	7	5	320	25	3	1	0.33	2	1	0
50	4	0	140	14	8	0	330	25	3	1	0.5	2	1	0
110	2	2	200	1	14	8	-1	25	3	1	0.75	0	3	2

To identify the best configuration from the above three, we can utilize the *inertia_* parameter which is the Sum of squared distances of samples to their closest cluster center.

From comparing the *inertia_* for each of the configuration we get the below values:



```
C:\Users\Rohit\PycharmProjects\Assignment_3\venv\Scripts\python.exe "G:/Masters/CSNL/Autumn/Data Mining/Assignment6/run.py"
221721.3021603331
221299.8151006216
349963.0308787614
```

The configuration with lowest inertia value is the one which gives the better clustering solution.

From the above *inertia_* results we can conclude that the second configuration gives better clustering solution.

kmeans2 = KMeans(n_clusters=5, random_state=0, n_init=100, max_iter=100).fit(df2_new)

Question 3:

The data points in this dataset are highly dense and the Kmeans clustering has produced 7 clusters as given the function parameter:

kmeans3 = KMeans(n_clusters=7, random_state=0, n_init=5, max_iter=100).fit(df3_new)

Here, we are setting the *n_init* to 5 which runs the algorithm for 5 iterations independent of each run. As the data being highly dense the k-means clustering might have provided better results in case we had a large value for *n_init* as it might have tried to find the best centroid value for each of the cluster. As seen in the image some of the data points seem to be scattered a bit far from the centroid which indicates that a single cluster in graph is both tightly and loosely clustered.

The data is then normalized using minmax normalization as the data points are dense and some of them are scattered as well.

We use the DBSCAN method from sklearn to find the samples of high density that form good clusters.

DBSCAN is Density Based Spatial Clustering of Applications with Noise.

We use the function with parameters as below:

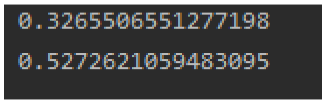
clustering_df3 = DBSCAN(eps=0.04, min_samples=4).fit(df4)

clustering_df4 = DBSCAN(eps=0.08, min_samples=4).fit(df4)

Here *eps* is the parameter epsilon, which is the most important parameter in the DBSCAN function, it is the maximum distance between two samples to be considered in neighborhood of other. DBSCAN forms an n-dimensional shape around each point to identify how many points fall within that shape. In this scenario, the minimum number of points has been set to 4. When going through each data point, if DBSCAN finds 4 points within epsilon distance of each other, a cluster is formed.

We can find many of the cluster labels in the scatter plot below are marked as -1. The DBSCAN method categorizes it as noise. The noisy data is not included in any of the clusters and hence separately marked.

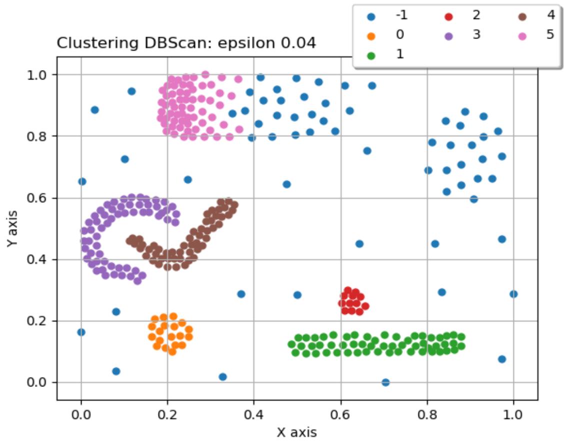
We can use the silhouette_score coefficient function from the sklearn.metrics which evaluates the mean intra cluster distance between data points and the mean nearest cluster distance. A silhouette score ranges from -1 to 1, with 1 being the best score. When evaluating both the configuration of the functions to get their silhouette score we get the results as below for eps=0.04 and eps=0.08 respectively:

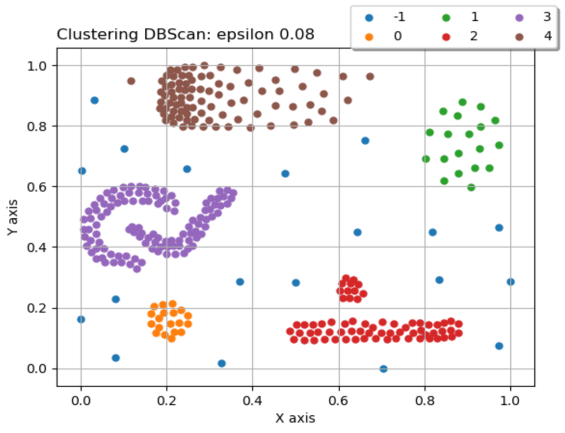


```
0.3265506551277198
0.5272621059483095
```

From the above silhouette score results we can conclude that the second configuration with eps=0.08 is a better configuration and gives a better clustering solution.

The reason that the second configuration is better is that the low eps values in the first configuration leads to marking many of the data points as noise.





As seen in the above image the blue data points are shown as noise in both the scatter plots, however, the second scatter plot has less noise as compared to the first one. The reason for more noise in the first configuration is setting of low eps value which leads to finding only the closest data points.

In addition to noise, silhouette score of the second configuration is better than the configuration1. Hence, configuration 2 gives a better solution.