COMP40370 Practical 6

Ryan Jennings 19205824

Question 1:

1) The first question involved calling KMeans from sklearn.cluster after loading the csv file.
```
kmeans = KMeans(n_clusters=3, random_state=0)
df_1_kmeans = kmeans.fit(df_1)
```
2) After generating the clusters the field of the data can be added based on the cluster labels and then written out to a csv file.
```
df_1['cluster'] = df_1_kmeans.labels_
df_1.to_csv('./output/question_1.csv')
```
3) Now that we have each point identified to a cluster we can use matplotlib to get a look at our data and saved to a viewable format.
```
plt.scatter(df_1['x'], df_1['y'], c=df_1['cluster'], cmap='brg')
plt.savefig('./output/question_1.pdf', dpi=199)
```

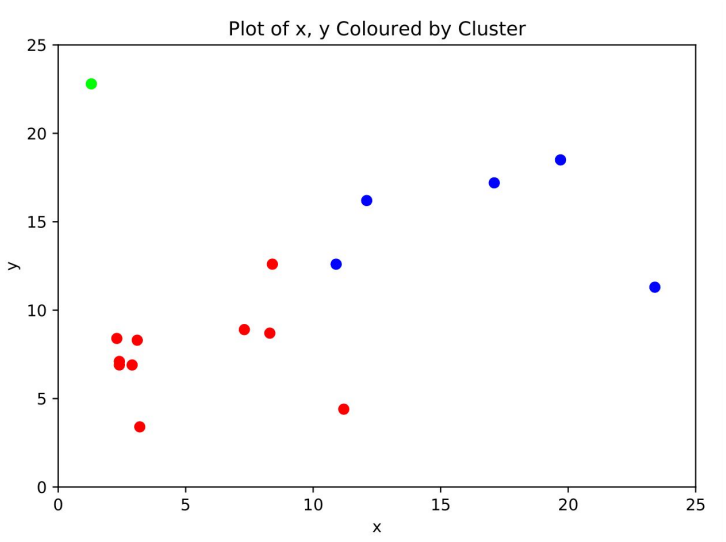

Figure 1. Plot for question 1 showing the 3 coloured clusters.
From Figure 1 we can make out the 3 clustered classes identified by the 3 colours Red, Blue and Green. A clear line of demarcation is visible dividing up the red and blue points. The green point is the outlier. The green point on its own is to be expected while visibly inspecting the data but without more x, y coordinates to add to our dataset we don't know if the green point is an outlier, noise or part of a larger cluster.

Question 2:

1) To start question 2 we want to remove the meta qualities of the cereal data and leave only the physical attributes to determine our clustering of the data.
```
df_2 = df_2.drop(['NAME', 'MANUF', 'TYPE', 'RATING'], axis=1)
```
2) Similar to question 1.1 and 1.2 we call our KMeans algorithm again.
```
df_2_kmeans1 = KMeans(n_clusters=5, random_state=0, n_init=5,
max_iter=100).fit(df_2)
```

`df_2['config1'] = df_2_kmeans1.labels_`

3) For 2.3 we repeat the steps above but with n_init=100 and max_iter=100. Saving the data to a new column config2.

4) How does config1 results compare to config2 results?

	config1	config2
0	3	1
1	2	3
2	3	1
3	3	1
4	1	2
...	...	...
72	4	2
73	1	0
74	4	4
75	1	4
76	1	2

77 rows × 2 columns

Figure 2. The `config1` values vs the `config2` values.

We can see that the config values don't generally match. In fact if we run the command on our dataframe `df_2[df_2['config1'] == df_2['config2']`
We only get one match and that is row **74** seen in Figure 2.
For data such as Figure 1 we are only dealing with 2D data but adding all the cereal attributes gives our data multiple dimensions to be represented in. These extra dimensions allow for greater and more in detail partitioning of the data. More intense computation may also allow us to better select which cluster our row would be in. Increasing the maximum runs such as for `config2` could allow greater granualisation of the data.

5) Once again we run our KMeans algorithm and save to config3
`df_2_kmeans3 = KMeans(n_clusters=3, random_state=0, n_init=100, max_iter=100).fit(df_2)`
`df_2['config3'] = df_2_kmeans3.labels_`

6) As discussed above in question 2.4, our dataset has many dimensions above what we would expect from a simple 2D plot. Unfortunately that means it's not feasible to plot it or even for our limited human minds to comprehend data on that many planes.
Thankfully there are some metrics we can use to quantify their effectiveness without visual inspection. One useful method is to find the sum of squared distances of the data points to their closest cluster centre. We can see the results of each of the KMeans models below in Figure 3.

```
print(f"KMeans 1: {df_2_kmeans.inertia_}")
print(f"KMeans 2: {df_2_kmeans2.inertia_}")
print(f"KMeans 3: {df_2_kmeans3.inertia_}")
```

```
KMeans 1: 227204.10020632186
KMeans 2: 221254.82753263402
KMeans 3: 349804.93817359424
```

Figure 3. Sum of square distances for each model.

Going by the values in Figure 3 I would choose our **second** model as the best. The n_init=100 and max_iter=100 values allowed it to run for long enough to do more effective computation than model 1 and the use of 5 clusters instead of 3 meant that each point was more likely to be nearer to a centre due to better division of the plane.

7) Finally for question 2 we save the output
`df_2.to_csv('./output/question_2.csv')`

Question 3:

1) Similar to other times we can drop the unused columns, here that is the ID column.
   `df_3 = df_3.drop(['ID'], axis=1)`
   For the rest of question 3.1 we can set up another KMeans model.
   `df_3_kmeans = KMeans(n_clusters=7, n_init=5, max_iter=100, random_state=0).fit(df_3)`
   `df_3['kmeans'] = df_3_kmeans.labels_`
   The plotted data can be seen below in Figure 4. Our data is split into 7 segments. You might notice that some points look off, that you won't cluster manually such as the bends on the left. An increase in our n_init value (maximum runs) could produce cleaner results.
2) We can see the result of our KMeans clustering on dataset 3 below

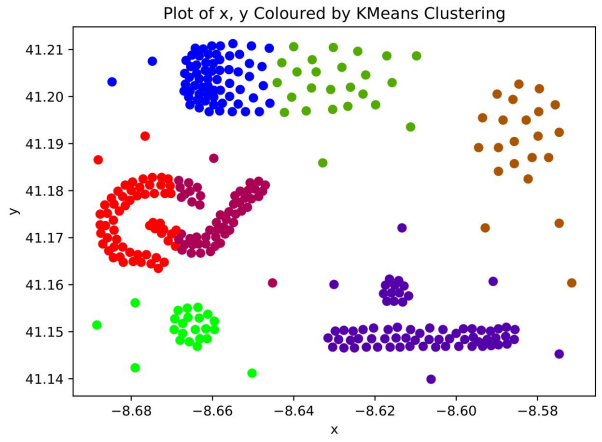

   Figure 4. We can see the effective clustering.
   For Figure 4 we have a lot of interesting shapes to test the limits of the clustering. Notice that our data is in a very narrow window around (8, 41).
3) To fix the problem with Figure 4 where our data is only contained in a small area that is offset from the origin we can normalise the data to view it between 0 and 1.
   `df_3_norm = MinMaxScaler().fit_transform(df_3[['x', 'y']])`
   Using our normalised data we start to use a different clustering algorithm, DBSCAN.
   `df_3_dbscan = DBSCAN(eps=0.04, min_samples=4).fit(df_3_norm)`
   `df_3['dbscan1'] = df_3_dbscan.labels_`
   Let's have a look at the difference the DBSCAN model makes from the KMeans alternative.

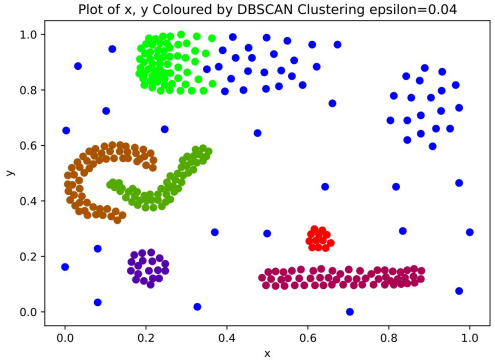

   Figure 5. The clustering of dataset 3 but with normalised data
4) Changing our epsilon value to 0.08 gives us a different plot that is observable in Figure 6 compared to Figure 5 above.

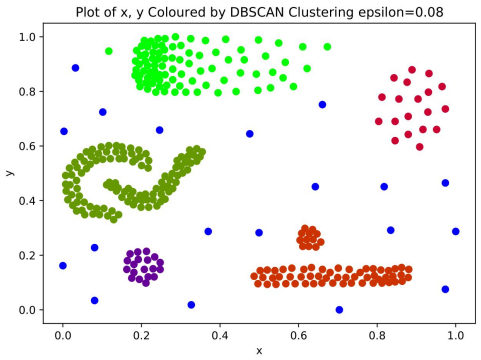

Figure 6.

5) Now for question 3.5 we simply save our dataframe with the cluster labels
`df_3.to_csv('./output/question_3.csv')`
6) Comparing our 3 clusterings, Figure X, Figure Y, and Figure Z, the different techniques perform differently. To ideally identify the best technique we would have to have a use case, some may require different criteria or different common datasets that might be dealt with better by the different models. So judging purely visually I would have to go with our last clustering model, the DBSCAN with an epsilon of 0.08. It had the lowest number of noise than the others while effectively grouping the clusters, the spread-out values and the convex shapes. The first DBSCAN with epsilon = 0.04 was able to distinguish between the two convex shapes and the line and ball clusterings in the bottom right but produced more noise particularly from spread-out data points.
The lower epsilon store ends up marking more data points as noise and that overall leads to worse analysis. We can attempt to verify this with some methods from sklearn's metrics library. Using the Silhouette score that measures the mean intra cluster distance between our data points I got the following results:

```
print(
f"""Silhouette score DBSCAN1:
{metrics.silhouette_score(df_3,
                          df_3_dbscan.labels_,
                          metric='sqeuclidean')}
Silhouette score DBSCAN2:
{metrics.silhouette_score(df_3,
                          df_3_dbscan2.labels_,
                          metric='sqeuclidean')}
""")
```

```
Silhouette score DBSCAN1:
0.7034306606613777
Silhouette score DBSCAN2:
0.5015200294202056
```

Figure 7.
With a best score being 1, the silhouette score identified the DBSCAN 1 model as performing better than our second. But again these measurements are subjective to the particular use case that they could be needed for. If say these were x, y coordinates for battleships you would not want to identify the two convex shapes as part of the same group, or in the bottom right the cluster and the line. Both methods have their merits and can help us better understand and analyse data to extract knowledge from it.