University of Salford, MSc Data Science

Module: Machine Learning & Data Mining

Session: Workshop Week 4

Topic: Clustering

Tools: Jupyter Notebook

Objectives:

After completing this workshop, you will be able to:

- Use the scikit-learn implementation of the K-Means algorithm for clustering
- Identify the optimal number of clusters using the elbow method
- Use the scikit-learn implementation of Agglomerative clustering
- Use the scikit-learn implementation of DBSCAN clustering
- Plot and interpret the results of your clustering
- Perform PCA to reduce the dimensionality of the data so you can visualize clustering results for higher dimensional datasets

MLDM Week 4 Workshop: Clustering

Introduction

Clustering is widely used in many fields. Clustering is regarded as unsupervised learning for its lack of a class label or a quantitative response variable, which in contrast is present in supervised learning such as classification and regression.

Clustering is a division of data into groups, or clusters, of similar observations. A common example of where clustering is useful is in customer segmentation. Many businesses want to run targeted marketing campaigns, focused on specific subsets of their customers who have similar characteristics, as they can therefore tailor their marketing messages to each group. One approach a business can adopt to define these customer segments is to use a clustering algorithm to identify groups of customers based on their characteristics (e.g., their demographic or spending habits).

Clustering also plays a role in data mining applications such as scientific data exploration. It can be applied in information retrieval and text mining, spatial database applications, web analysis, marketing, medical diagnostics, and computational biology (Berkhin, 2006).

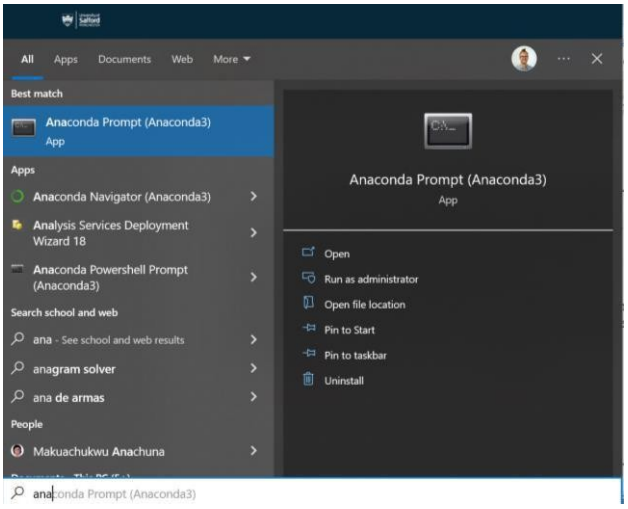
Some other applications of clustering are:

- Marketing: Help marketers discover distinct groups in their customer bases, and then use this knowledge to develop targeted marketing programs.
- Land use: Identification of areas of similar land use in an earth observation database.
- Insurance: Identifying groups of motor insurance policy holders
- City-planning: Identifying groups of houses according to their house type, value, and geographical location.
- The Office of National Statistics uses K-Means clustering to identify clusters of local authority regions which have similar characteristics from census data. You can read more here.

Important Note: Saving Your Notebook to F Drive

By default, when you launch Jupyter Notebook, the starting directory is in the C drive. If you are using a university desktop or laptop to complete this workshop, you should ensure that, at the end of the session, you save your notebook to the F drive so you can access it from other university devices.

Alternatively, you can launch Jupyter Notebook with the F drive as the starting directory. To do this, search for Anaconda Prompt in the Windows Search bar.



Once you have opened the Anaconda Prompt, enter the below command and press enter:

```
jupyter notebook --notebook-dir=F:
```

Part One: K-means Clustering

The first clustering algorithm we are going to explore is the K-means Clustering algorithm. With K-means clustering, the number of clusters (k) is a user-defined parameter. For some problems, you may know the ideal number of clusters from your domain knowledge – however, this workshop will also demonstrate a couple of approaches which can help with selecting a suitable value for *k.*

K-means clustering works like this:

- Step 1: The k centroids for the clusters are initialised randomly (this isn't quite true, but we will come back to this!)
- Step 2: Allocate each point to a cluster based on which centroid is closest
- Step 3: Calculate the mean value of all points in a cluster. This new point becomes the cluster centroid
- Steps 2 and 3 are repeated until the cluster centroids are no longer changing significantly.

You can see an interactive visualization of how the K-means algorithm works here.

Step 1, as described above, is an oversimplification. How the centroids are initialized can impact the running time and the final clustering result. Because of this, there are now better approaches to initialize the centroids which are not completely random – instead, the centroids are initialized to reduce the probability that they will start off close together. This is known as K-means++ and is implemented in Scikit Learn.

To calculate the closest centroid to a data point, we also need to have some metric to calculate the distance between two points. There are different distance metrics you can choose from, but the most common is *Euclidean distance.* This is given by:

$$\boldsymbol{distance}(p, q) = \sqrt{\sum_{i=1 \text{ to n}} (q_i - p_i)^2}$$

We're going to start by applying K-means clustering to just two numerical variables in a dataset. This makes it easy to visualise the clustering results in a scatterplot.
The dataset is saved on Blackboard as **Mall_Customers.csv**. You should download this file and save it in the same directory as your notebook.

1. Open Jupyter Notebook, either from the Anaconda Prompt as described on page 3 or using the Anaconda Navigator. Open a new notebook.
2. Run the below cell to import the necessary libraries for this workshop.

```
In [1]:
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

# Run before importing KMeans

import os
os.environ["OMP_NUM_THREADS"] = '1'
```

3. We are now going to use the read_csv() function in the pandas library to read our **Mall_Customers.csv** dataset into a pandas Data Frame.

```
In [2]:
# Importing the dataset
dataset = pd.read_csv('Mall_Customers.csv')
```

4. As we have done in previous workshops, start by exploring the Data Frame using head(), info() and describe() to get a better understanding of the dataset and the columns in our data.

```
In [3]:
dataset.head()
```

Out[3]:

	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

```
In [4]:
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   CustomerID              200 non-null    int64
 1   Genre                   200 non-null    object
 2   Age                     200 non-null    int64
 3   Annual Income (k$)      200 non-null    int64
 4   Spending Score (1-100)  200 non-null    int64
dtypes: int64(4), object(1)
memory usage: 7.9+ KB
```

```
In [5]: dataset.describe()
```

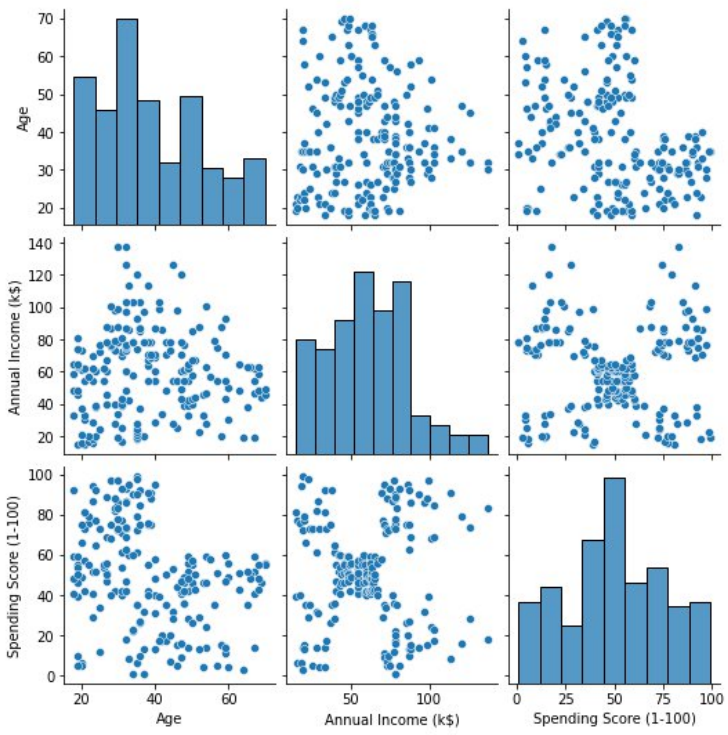
Out[5]:

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

5. We can see the third, fourth and fifth columns are all numerical. Seaborn comes with a useful function we can use for exploring numerical variables, pairplot(). This will provide us with a series of scatterplots for each pair of variables and a histogram for each variable too. We can see from the scatterplot of Spending Score against Annual Income, that the data does seem to roughly form five clusters.

```
In [6]: sns.pairplot(dataset.iloc[:,[2,3,4]])
```

Out[6]: <seaborn.axisgrid.PairGrid at 0x1dbb58c65b0>



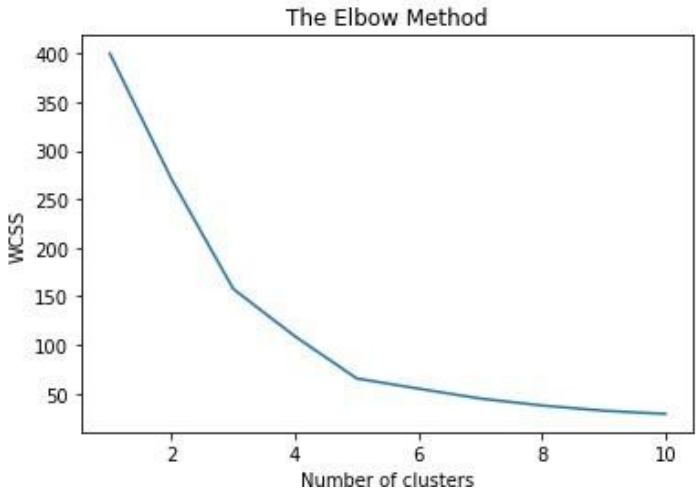
6. For this first exercise, we are going to work with just these two variables, Spending Score and Annual Income. As we are only working with two variables it is easy to plot our results so will help you visualise how clustering works. Later in this workshop we will look at datasets which have more features and how we can still visualise the results of our clustering in two-dimensions.

   Use the iloc() function to select the two columns we want, and then we apply scaling to the data. Because K-means relies on a distance metric, scaling can have a big impact on the results. For example, if one of our variables had values from 10,000 to 100,000 and the other had values from 0 to 1, the Euclidean distance will end up being dominated by the first of the variables. For this reason, we will normally perform Min-Max scaling or normalisation before carrying out clustering.

```
In [7]: from sklearn.preprocessing import StandardScaler
        X = dataset.iloc[:, [3,4]].values
        sc_X = StandardScaler()
        X = sc_X.fit_transform(X)
```

7. We will also use the elbow method to assess the optimal number of clusters using K-means. The elbow method involves calculating the *within cluster sum of squares (wcss)* for the data with various values of *k.* The aim of K-means clustering is to minimise this value. We can therefore plot how wcss changes as we increase the number of clusters. The wcss tends to reduce rapidly initially as we increase the number of clusters, and after a certain point decrease more slowly. The ideal number of clusters is the 'elbow' in the graph where the decrease in wcss becomes less rapid. The below code uses Scitkit Learn to perform K-means clustering with values of *k* from 1 to 10 and saves the resulting wcss to an array. We then plot this using matplotlib. The elbow here appears to be at $k = 5$.

In [8]:
```
# Using the elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

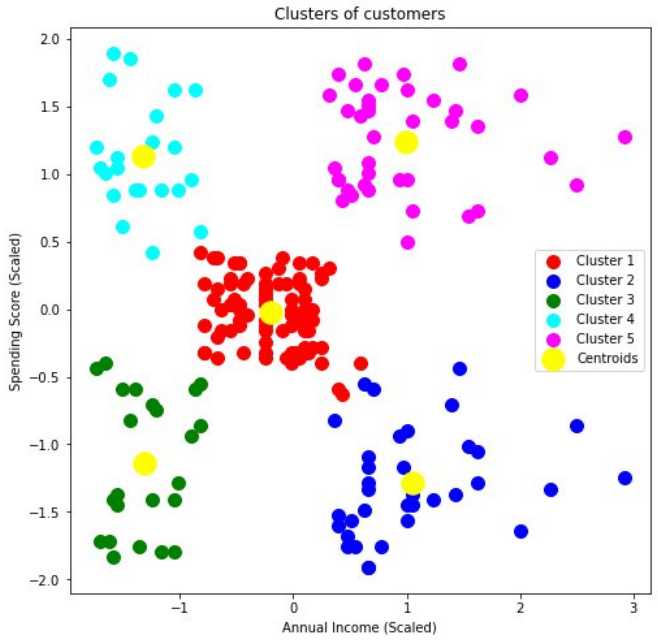


8. Now we have identified the optimal number of clusters we can use the fit_predict() method to train a KMeans() method on the dataset and return an array y_kmeans which is the same length as our Data Frame and tells us which cluster each row has been assigned.

In [9]:
```
# Fitting K-Means to the dataset
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
y_kmeans = kmeans.fit_predict(X)
```

9. Using the below code to display each cluster in a different colour, we can plot a scatterplot of the resulting data with the cluster assignments and centroids shown. Are the clusters roughly what you expected given the data?

```
# Visualising the clusters
plt.figure(figsize=(8,8))
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 300, c = 'yellow', label = 'Centroids')
plt.title('Clusters of customers')
plt.xlabel('Annual Income (Scaled)')
plt.ylabel('Spending Score (Scaled)')
plt.legend()
plt.show()
```



Part Two: Hierarchical Clustering

Hierarchical clustering is also a distance-based clustering method. However, while K-means clustering starts by assigning all data points to a cluster and then iteratively updating the cluster centroids and cluster assignments, hierarchical clustering works very differently. Agglomerative algorithm works from the ground up as follows:

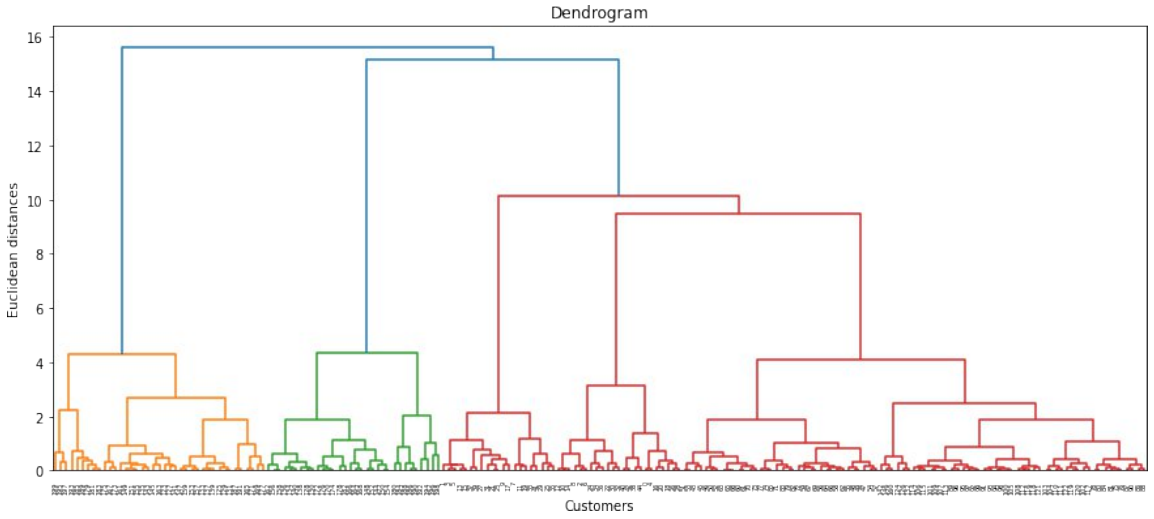
- Step One: Each point starts as its own cluster
- Step Two: Find the closest pair of clusters and merge them
- Step Three: Repeat the previous step until there is only one cluster

Using this method, we can construct a tree, or dendrogram which shows the point at which each cluster merges with another. This is therefore particularly useful in cases where we're not just interested in the clusters but in constructing a full dendrogram. For example, one application is in evolutionary biology. We can look at the DNA sequences for different species and use hierarchical clustering based on how similar the sequences are to construct a *phylogenetic tree*, which is used to identify which species are most closely related to one another from an evolutionary perspective.

We can use a function from the Scipy package to visualise the dendrogram. We can use this to select the number of clusters we would like to use and then perform clustering with our chosen number of clusters, and then visualise this as before.

10. Run the cell below to import the required function, and then to perform hierarchical clustering and plot the resulting dendrogram. If you draw a horizontal line across the dendrogram, the number of vertical lines crossing the line equates to the number of clusters. For example, from the below we could end the clustering process at the point where the Euclidean distance is 12 and this would give us three clusters (one formed of the orange data points, one formed of the green data points, and one formed of the red data points shown on the dendrogram). However, typically we take a horizontal line around halfway down the dendrogram to identify the optimal number of clusters, which in this case gives us five.

```
In [11]: # Using the dendrogram to find the optimal number of clusters
         import scipy.cluster.hierarchy as sch

         plt.figure(figsize=(15,6))
         dendrogram = sch.dendrogram(sch.linkage(X, method = 'ward'))
         plt.title('Dendrogram')
         plt.xlabel('Customers')
         plt.ylabel('Euclidean distances')
         plt.show()
```

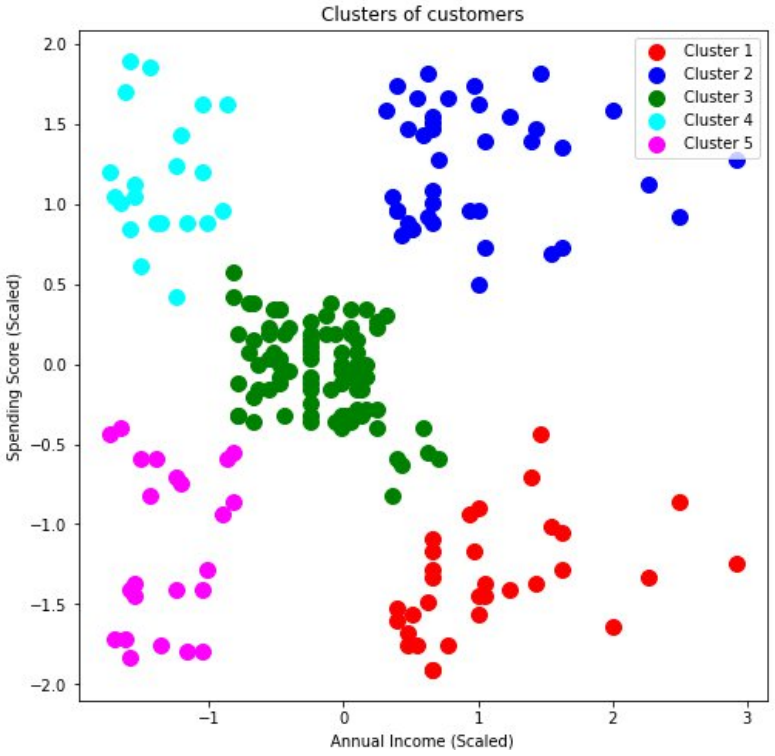


11. With our selected number of clusters, we can now instantiate an AgglomerativeClustering object and use the fit_predict() method on this to generate an array of cluster assignments, in the same way as we did when using the K-means clustering algorithm.

```
# Fitting Hierarchical Clustering to the dataset
from sklearn.cluster import AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters = 5, metric = 'euclidean', linkage = 'ward')
y_hc = hc.fit_predict(X)
```

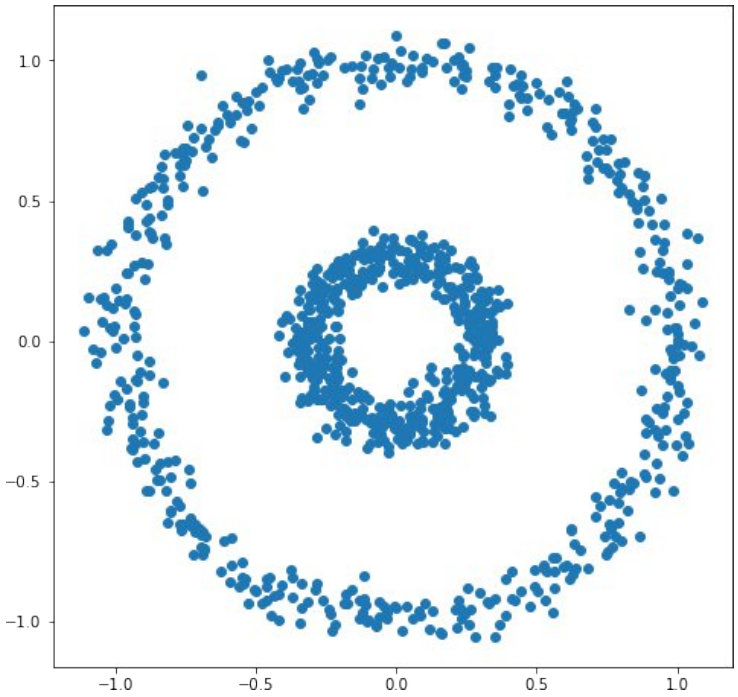
12. We can also write a similar block of code to plot the data on a scatterplot again. As you can see, the cluster assignments are very close to those from K-means clustering, with some minor differences with some of the points on the edges of the cluster in the middle.

```
In [13]: # Visualising the clusters
plt.figure(figsize=(8,8))
plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(X[y_hc == 3, 0], X[y_hc == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
plt.scatter(X[y_hc == 4, 0], X[y_hc == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
plt.title('Clusters of customers')
plt.xlabel('Annual Income (Scaled)')
plt.ylabel('Spending Score (Scaled)')
plt.legend()
plt.show()
```



Part Three: DBSCAN

Both of the previous clustering methods rely on *distance* metrics to group data into clusters. However, distance will not always be the most appropriate way to assign points to a cluster. For example, take a look at the dataset below:



It seems intuitive that we would treat each of the two rings as a separate cluster. However, we cannot identify these clusters using the K-means clustering algorithm, since the points at the bottom of the outer ring are closer to the inner ring than they are to the points at the top of the outer ring! However, what we can do instead is use a *density-based* method. These start from the intuition we've developed from the above image that clusters should be denser regions of data points, separated by less dense regions. DBSCAN (Density Based Spatial Clustering of Applications with Noise) is one such algorithm.

With DBSCAN, we have to set two parameters, epsilon and the minimum number of points. This algorithm works as follows:

- Search for a data point which has at least the pre-set minimum number of points within a radius of epsilon from it. This is a *core* point. All the points that are in the circle of radius epsilon of this point are added to the same cluster as it.
- For each of these points added to the cluster, check how many points they have within a circle of radius epsilon. If this is more than the minimum number of points, then this is also a core point, and we continue this process of adding the surrounding points to this cluster.
- If we come across a point less than the minimum number of points in a circle of radius epsilon around it, this is an *edge point.* We still add this point to the cluster, but the process ends there.
- Once we have continued to 'grow' our cluster as much as we can, we then look for another core point elsewhere, and start to grow our next cluster.

- At the end we may have some points which are unallocated to a cluster, because they have less than the minimum number of points within distance epsilon, and because none of those points have been assigned to a cluster. These are *noise* or *outliers* and are not allocated to a cluster.

You can view an animation of DBSCAN in action here.

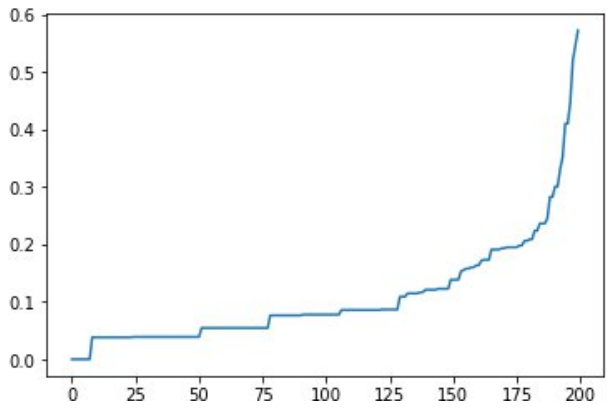
DBSCAN is therefore different to the two methods we have described above, as some points may not be allocated to a cluster at all. However, this feature of the algorithm can be useful if we know the data may include outliers.

With DBSCAN we don't have to pre-define the number of clusters. However, we do need to identify the value for epsilon and the minimum number of points. One method for selecting epsilon is to calculate the distance of each point to its nearest neighbour and plot these distances in rank order from smallest to largest. The point at which the nearest neighbour distances start increasing more rapidly is the distance we choose as epsilon, as the points with the longest distances to their nearest neighbours are those outliers which don't fit within a dense cluster.

13. Run the below code to calculate the Nearest Neighbours to each point, sort the distances in ascending order and then plot this on a graph using matplotlib. We can see the distances start increasing rapidly at around ~0.25 so we will select this as our value for epsilon.

```
In [15]: from sklearn.neighbors import NearestNeighbors

         neighbours = NearestNeighbors(n_neighbors=2)
         distances, indices = neighbours.fit(X).kneighbors(X)

         distances = distances[:,1]
         distances = np.sort(distances, axis=0)
         plt.plot(distances)
```

```
Out[15]: [<matplotlib.lines.Line2D at 0x1dbba776e20>]
```



14. We can now instantiate a DBSCAN object and use the fit_predict() method on this to generate an array of cluster assignments, in the same way as we did when using the K-means and Agglomerative clustering algorithms. We will use the default value of 5 for the min_samples argument.

```
In [16]: from sklearn.cluster import DBSCAN

         dbscan = DBSCAN(eps=0.25, min_samples=5)
         y_dbscan = dbscan.fit_predict(X)
```

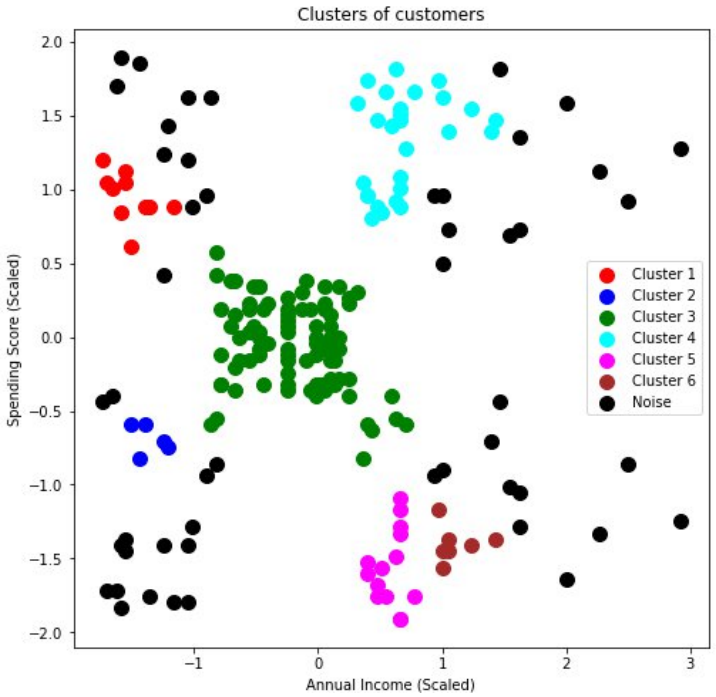
15. With DBSCAN, we didn't predefine the number of clusters. Let's inspect the array of cluster assignments to see how many clusters have been generated. (Values of -1 indicate those points which haven't been assigned a cluster as they are considered noise.)

```
In [17]: # inspect the array to identify number of clusters

         y_dbscan
```

```
Out[17]: array([-1,  0, -1,  0, -1,  0, -1, -1, -1,  0, -1, -1, -1,  0, -1,  0,  1,
                 0,  1, -1,  1,  0, -1,  0, -1, -1,  1, -1,  1, -1, -1,  0, -1, -1,
                -1, -1, -1, -1, -1, -1,  2, -1,  2,  2, -1,  2,  2,  2,  2,  2,  2,
                 2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
                 2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
                 2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
                 2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
                 2,  2,  2,  2,  3,  2,  3,  2,  3,  4,  3,  4,  3,  2,  3,  4,  3,
                 4,  3,  4,  3,  4,  3,  2,  3,  4,  3,  2,  3,  4,  3,  4,  3,  4,
                 3,  4,  3,  4,  3,  4,  3,  2,  3,  4,  3, -1, -1,  5,  3, -1, -1,
                 5, -1,  5,  3,  5,  3,  5, -1,  5,  3, -1,  3,  5,  3, -1, -1, -1,
                -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1], dtype=int64)
```

16. We can now use code similar to the code that we used above to generate the scatterplot to visualise the K-means clusters. We can see here that a lot of the points in the outer regions of the plot have not been assigned to a cluster and are labelled as noise {-1). This is because the outer regions have a much lower density, and the points are relatively 'spread out,' suggesting that a density-based method may not be the most appropriate to use in this instance.

```
In [18]: # Visualising the clusters
plt.figure(figsize=(8,8))
plt.scatter(X[y_dbscan == 0, 0], X[y_dbscan == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
plt.scatter(X[y_dbscan == 1, 0], X[y_dbscan == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
plt.scatter(X[y_dbscan == 2, 0], X[y_dbscan == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(X[y_dbscan == 3, 0], X[y_dbscan == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
plt.scatter(X[y_dbscan == 4, 0], X[y_dbscan == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
plt.scatter(X[y_dbscan == 5, 0], X[y_dbscan == 5, 1], s = 100, c = 'brown', label = 'Cluster 6')
plt.scatter(X[y_dbscan == -1, 0], X[y_dbscan == -1, 1], s = 100, c = 'black', label = 'Noise')
plt.title('Clusters of customers')
plt.xlabel('Annual Income (Scaled)')
plt.ylabel('Spending Score (Scaled)')
plt.legend()
plt.show()
```



Part Four: Clustering Data with Higher Dimensionality

So far, we have been clustering data with only two features (or *dimensions*). However, we can apply the same approach to data with many more features. For this part of the workshop, we will use the **costpercompany.csv** file which is also provided on Blackboard.

17. Use the read_csv() function in the pandas library to read our **costpercompany.csv** dataset into a pandas Data Frame. Then explore the DataFrame using head(), info() and describe() to get a better understanding of the dataset and the columns in our data.

```
In [19]: financials = pd.read_csv('costpercompany.csv')
```

```
In [20]: financials.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29 entries, 0 to 28
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Company       29 non-null     object
 1   surcharges    29 non-null     float64
 2   RoR           29 non-null     float64
 3   dailycost     29 non-null     int64
 4   costwithload  29 non-null     float64
 5   costofDemand  29 non-null     float64
 6   Sales         29 non-null     int64
 7   WearandTear   29 non-null     float64
 8   Fcost         29 non-null     float64
dtypes: float64(6), int64(2), object(1)
memory usage: 2.2+ KB
```

```
In [21]: financials.describe()
```

Out[21]:

	surcharges	RoR	dailycost	costwithload	costofDemand	Sales	WearandTear	Fcost
count	29.000000	29.000000	29.000000	29.000000	29.000000	29.000000	29.000000	29.000000
mean	1.402759	10.488966	172.448276	46.862069	4.906897	13024.517241	16.110345	1.173621
std	0.699065	3.564261	60.520709	28.265936	4.572428	9120.761558	18.885356	0.637931
min	0.750000	1.860000	49.000000	-49.800000	-2.200000	3300.000000	0.000000	-0.012000
25%	1.050000	9.200000	148.000000	51.500000	2.200000	6650.000000	0.000000	0.636000
50%	1.150000	10.580000	173.000000	56.000000	3.500000	9673.000000	8.300000	1.108000
75%	1.430000	12.200000	199.000000	60.000000	7.200000	15651.000000	26.700000	1.652000
max	3.900000	21.160000	370.000000	72.000000	16.400000	40008.000000	53.400000	2.610000

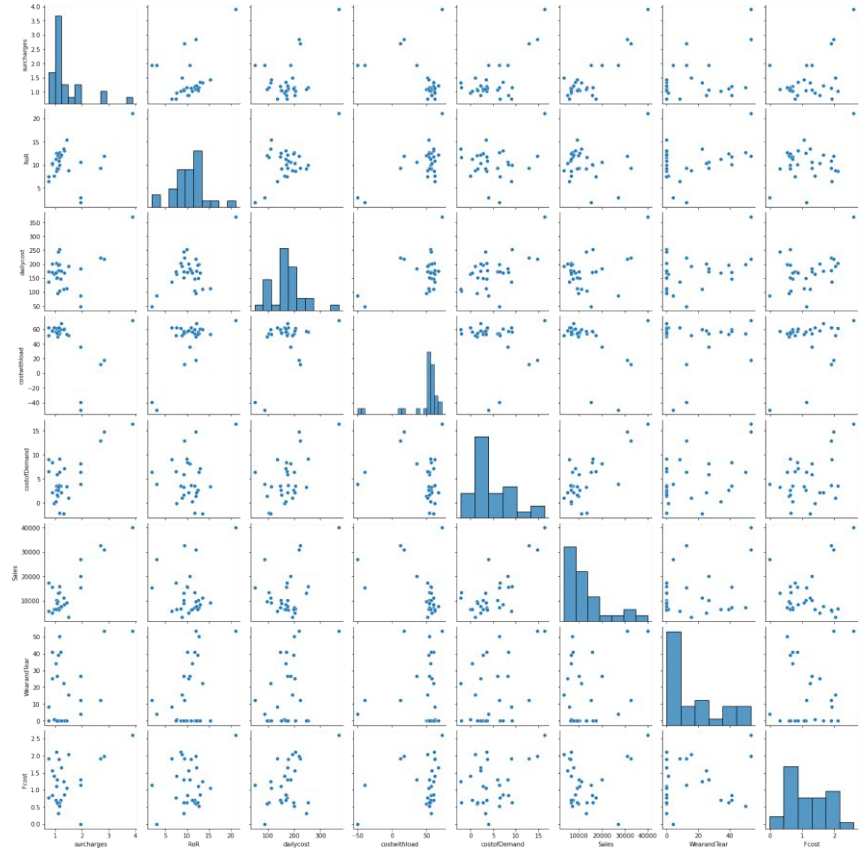
```
In [22]: financials.head()
```

Out[22]:

	Company	surcharges	RoR	dailycost	costwithload	costofDemand	Sales	WearandTear	Fcost
0	Yashida	2.70	9.36	222	12.1	12.9	32721	12.3	1.908
1	Wisconsin	1.20	11.80	148	59.9	3.5	7287	41.1	0.702
2	Virginia	1.07	9.30	174	54.3	5.9	10093	26.6	1.306
3	United	1.04	8.60	204	61.0	3.5	6650	0.0	2.116
4	Texas	1.16	11.70	104	54.0	-2.1	13507	0.0	0.636

18. As before, we can visualise the numerical columns using the pairplot function in Seaborn.

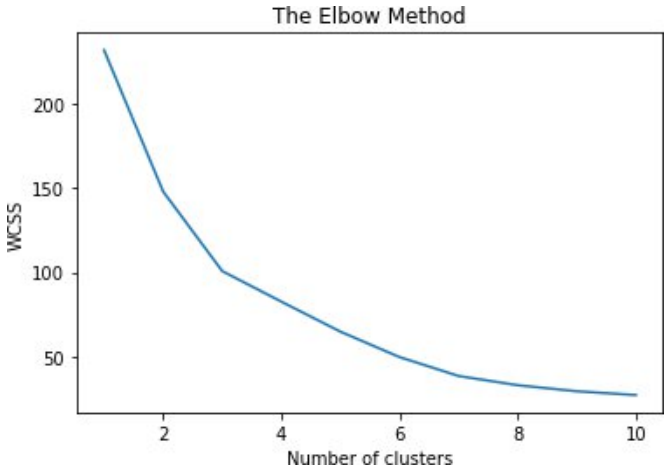
```
In [23]: X = financials.iloc[:,1:9]

         sns.pairplot(X)
```



19. We will use the StandardScaler as before to standardize the dataset. Then use the same code as we used above to visualise wcss against the number of clusters so we can select an appropriate number for *k*.

In [24]:
```
# Scale the data

sc_X = StandardScaler()
X = sc_X.fit_transform(X)
```

In [25]:
```
# Using the elbow method to find the optimal number of clusters

wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

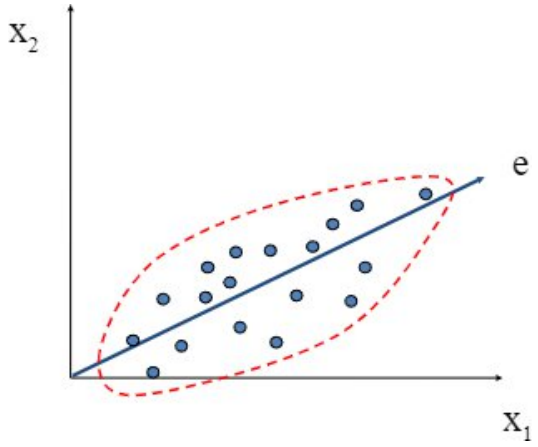


20. Selecting k=3, we can now run similar code as before to instantiate a KMeans object and call the fit_predict() method to generate the cluster assignments.

```
In [26]: # Fitting K-Means to the dataset
         kmeans = KMeans(n_clusters = 3, init = 'k-means++', random_state = 42)
         y_kmeans = kmeans.fit_predict(X)
```

There are eight numerical variables in the data, and unfortunately, we can't visualise eight dimensions in a scatterplot! However, we can use a technique called *PCA* to reduce the dimensionality of the data. You don't need to worry too much about how this works, but you can think of it as a way of rotating the axes we use so that we can drop some of our dimensions without losing too much of the variance.

For example, look at the image below. In this example dataset we have two features (or dimensions), x_1 and x_2. However, most of the points lie on the diagonal line, 'e'. If we therefore 'rotate' the axes we are using so that they are aligned with the diagonal on which our data lies, most of the variance will be along that one dimension. We can then drop the second dimension without losing too much information about the data. PCA, therefore, allows us to reduce our eight dimensions to fewer dimensions (e.g., two dimensions), which we can then plot on a scatterplot.



21. Using the code below, we can instantiate a PCA object. When instantiating it, we define how many components (or dimensions) we want to retain. We then use the fit_transform() method to perform PCA and return the fitted data with only two dimensions.

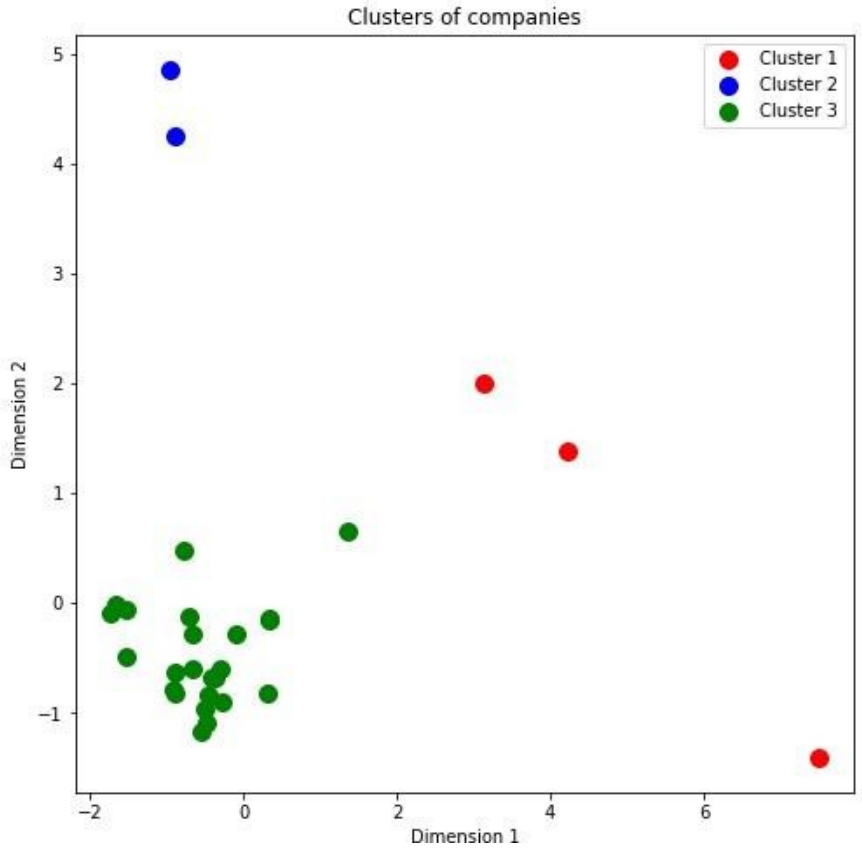
```
In [27]: # We need to reduce dimensionality before we can visualise

         from sklearn.decomposition import PCA

         pca = PCA(n_components=2)
         X_reduced = pca.fit_transform(X)

         pca.explained_variance_ratio_

Out[27]: array([0.44839294, 0.25725108])
```

22. We can find out how much of the original variance the two new dimensions explain by returning the explained_variance_ratio attribute and summing the values. We can see that over 70% of the original variance is explained by the two dimensions we have retained in our data.

```
In [28]: sum(pca.explained_variance_ratio_)

Out[28]: 0.7056440181560042
```

23. Let's now plot the scatterplot using similar code to that we have used above. You can simply adapt the code from one of the cells above. However, below we have shown you how to achieve the same result using a "for" loop instead.

In [29]:
```
# Visualising the clusters

colours = ['red', 'blue', 'green']

plt.figure(figsize=(8,8))
for i in range(3):
    plt.scatter(X_reduced[y_kmeans == i, 0], X_reduced[y_kmeans == i, 1],
                s = 100, c = colours[i] , label = 'Cluster '+str(i+1))
plt.title('Clusters of companies')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.legend()
plt.show()
```



We can see our clusters quite distinctly in the scatterplot. We can see most of the points form one cluster, with two smaller clusters, one with only two points and one with three.

As a further exercise, try performing both Agglomerative clustering and DBSCAN on this dataset too.