Chapter 6 - Clustering

Solution Workbook for Student Practical Classes

(c) Dr Rob Collins 2023

2024-07-12

Table of contents

List of Figures

Figure 1: Scholars of the Kensington Ladies Mathematical Society enjoying the game of 'Kluster-Puc' : A pastime reserved for the mathematically gifted and intended to enhance reasoning and inference skills. (1842)

6 Clustering

6.1 Introduction

In this workshop we will be introducing an un-supervised Machine Learning task - that of 'Clustering'. Clustering is intended to help organise data into groups based on the similarity of data records.

This particular activity uses the 'k-means' algorithm - which is one of the simpler un-supervised Machine Learning algorithms. In the last part of the practical these is 'bonus material' covering other clustering algorithms.

6.2 Instructions for Students

In this workbook there are regular 'callout' blocks indicating where you should add your own code. They look like this:

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided 'clues' towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

6.3 Load the required libraries

For this practical session we will need the following libraries:

- pandas : Conventionally given the reference name 'pd'
- 'preprocessing' from sklearn

- ‘metrics’ from sklearn
- numpy : Conventionally given the reference name ‘np’
- ‘kMeans’ from sklearn.cluster
- ‘DBSCAN’ from sklearn.cluster
- ‘OPTICS’ from sklearn.cluster
- ‘cluster_optics_dbscan’ from sklearn.cluster
- matplotlib.pyplot : Conventionally given the reference name ‘plt’
- seaborn : Conventionally given the reference name ‘sns’
- missingno : Conventionally given the reference name ‘msno’
- ‘scatter_matrix’ from pandas.plotting
- ’Axes3D’from mpl_toolkits.mplot3d

```
import pandas as pd
from sklearn import preprocessing
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
from sklearn.cluster import OPTICS
from sklearn.cluster import cluster_optics_dbscan
from sklearn import metrics
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
from pandas.plotting import scatter_matrix
from mpl_toolkits.mplot3d import Axes3D
```

6.4 Load the data into a Pandas Dataframe

For the first part of this practical activity we will be a ‘synthetic’ data set .. that is, one that I created for this activity rather than being real customer data from a business. We are using this data set because it provides a useful illustration of the techniques whilst also removing any issues of privacy or security.

The data is contained within a file called ‘Retail Data v4 - unclean.csv’. Load this data into a pandas dataframe called ‘customer_data’

```
customer_data = pd.read_csv("Retail Data v4 - unclean.csv")
```

Display the first 20 rows of the ‘customer_data’ dataframe.

```
customer_data[0:20]
```

	Age	Gender	Married	Salary	Annual Spend
0	28.2	Male	Single	26908.95	331.56
1	43.5	Female	Married	39366.44	3071.18
2	27.7	Female	Single	NaN	1357.19
3	18.9	Male	Single	26235.55	769.78
4	18.0	Male	Single	30822.14	100.00
5	22.9	Female	Single	18334.52	2854.59
6	43.5	Female	Married	36642.04	1926.37
7	28.0	Male	Single	34612.04	853.10
8	52.3	Female	Married	34779.79	NaN
9	27.2	Female	Single	21901.08	566.21
10	47.3	Female	Married	33535.09	2344.79
11	31.0	Female	Single	24944.10	1661.97
12	34.9	Female	Single	23136.51	1343.36
13	45.8	Female	Married	27398.68	3250.98
14	32.8	Female	Single	21680.78	2600.72
15	28.7	Female	Single	27930.64	3717.44
16	70.1	Female	Married	38683.06	3965.79
17	28.0	Female	Single	25404.68	786.29
18	NaN	Male	Married	48905.96	3105.87
19	30.4	Female	Single	30677.96	1120.23

6.5 Clean and tidy the Data

6.5.1 Deal with Missing Data

By now you should be familiar with the process of data cleaning. Perform the following data cleaning operations on the ‘customer_data’ dataframe.

First, display a total count of the missing data.

```python
print(f"Count of missing data = {customer_data.isnull().sum().sum()}")
```

```
Count of missing data = 29
```

Visualise missing data using ‘missingno’

```python
msno.matrix(customer_data)
plt.show()
```

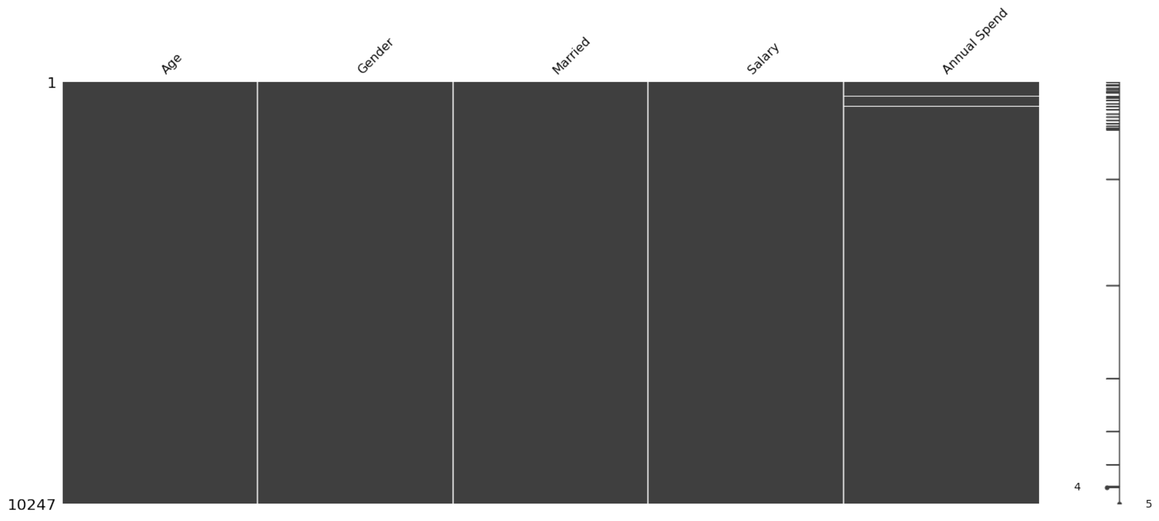


Figure 2: missingno matrix of the dataframe to identify missing data

Draw a bar-chart to indicate the amount of missing data in each feature:

```
msno.bar(customer_data)
plt.show()
```

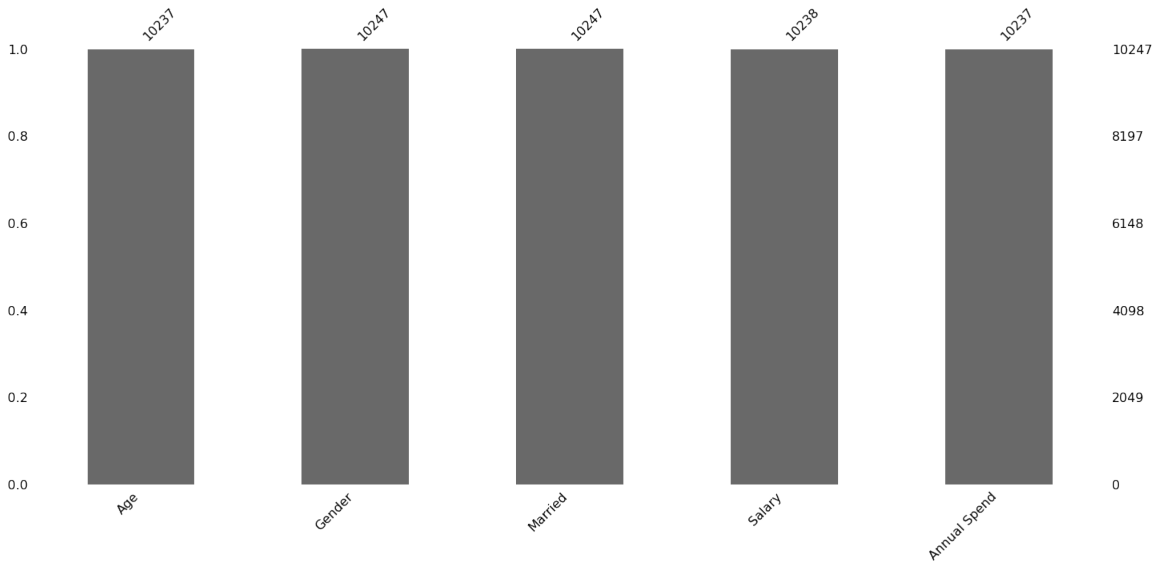


Figure 3: Bar chart showing volume of missing data for each feature

Impute missing data using the mean of other data from the same feature

```python
customer_data['Age'].fillna(customer_data['Age'].mean(), inplace = True)
customer_data['Salary'].fillna(customer_data['Salary'].mean(),
             inplace = True)
customer_data['Annual Spend'].fillna(customer_data['Annual Spend'].mean(),
             inplace = True)
```

Do a final check by re-counting the amount of missing data in the 'customer_data' data set

```python
print(f"Count of missing data = {customer_data.isnull().sum().sum()}")
```

```
Count of missing data = 0
```

6.5.2 Improve Layout by Renaming Features and Reducing decimal places

I want to rename the two categorical features at this point simply to make them shorter and improve page layout for this workbook. This is not strictly necessary, but is a useful thing to know how to do when using Pandas.

Fist, add code that changes the category feature names as follows:

- 'Gender' into 'G'
- 'Married'into 'M'

Then set Pandas to display only a single decimal place for float values.

```python
customer_data.rename(columns={'Gender': 'G', 'Married' : 'M'}, inplace=True)

pd.set_option('display.float_format', '{:.1f}'.format)

customer_data
```

	Age	G	M	Salary	Annual Spend
0	28.2	Male	Single	26909.0	331.6
1	43.5	Female	Married	39366.4	3071.2
2	27.7	Female	Single	36629.1	1357.2
3	18.9	Male	Single	26235.5	769.8
4	18.0	Male	Single	30822.1	100.0
...	...	...	...	...	...
10242	23.8	Female	Single	24218.5	1779.7
10243	26.6	Female	Single	26793.3	2352.4
10244	22.1	Male	Single	29096.5	410.2
10245	23.0	Female	Single	23438.0	1945.9

	Age	G	M	Salary	Annual Spend
10246	29.8	Male	Single	28754.0	2673.3

6.5.3 Deal with Categorical Data

Two of the features are category data (strings). Convert these into a one-hot-encoded form

```
categorical_features = ['G', 'M']
customer_data = pd.get_dummies(customer_data,
                               columns = categorical_features)
```

Review the final data set by again displaying the first 20 rows.

```
customer_data[0:20]
```

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
0	28.2	26909.0	331.6	0	1	0	1
1	43.5	39366.4	3071.2	1	0	1	0
2	27.7	36629.1	1357.2	1	0	0	1
3	18.9	26235.5	769.8	0	1	0	1
4	18.0	30822.1	100.0	0	1	0	1
5	22.9	18334.5	2854.6	1	0	0	1
6	43.5	36642.0	1926.4	1	0	1	0
7	28.0	34612.0	853.1	0	1	0	1
8	52.3	34779.8	2159.1	1	0	1	0
9	27.2	21901.1	566.2	1	0	0	1
10	47.3	33535.1	2344.8	1	0	1	0
11	31.0	24944.1	1662.0	1	0	0	1
12	34.9	23136.5	1343.4	1	0	0	1
13	45.8	27398.7	3251.0	1	0	1	0
14	32.8	21680.8	2600.7	1	0	0	1
15	28.7	27930.6	3717.4	1	0	0	1
16	70.1	38683.1	3965.8	1	0	1	0
17	28.0	25404.7	786.3	1	0	0	1
18	39.6	48906.0	3105.9	0	1	1	0
19	30.4	30678.0	1120.2	1	0	0	1

6.6 Visualise the data as a chart to help understand it better

Add code to help you visualise the data. The task is to visualise the numerical columns (‘Age’, ‘Salary’ and ‘Annual Spend’). We want to see a grid that shows the relationships between each of these three variables and also provides a histogram that shows the distribution for each variable.

```
%matplotlib inline

scatter_matrix(customer_data.iloc[:,0:3], figsize=(7, 7));

plt.show()
```

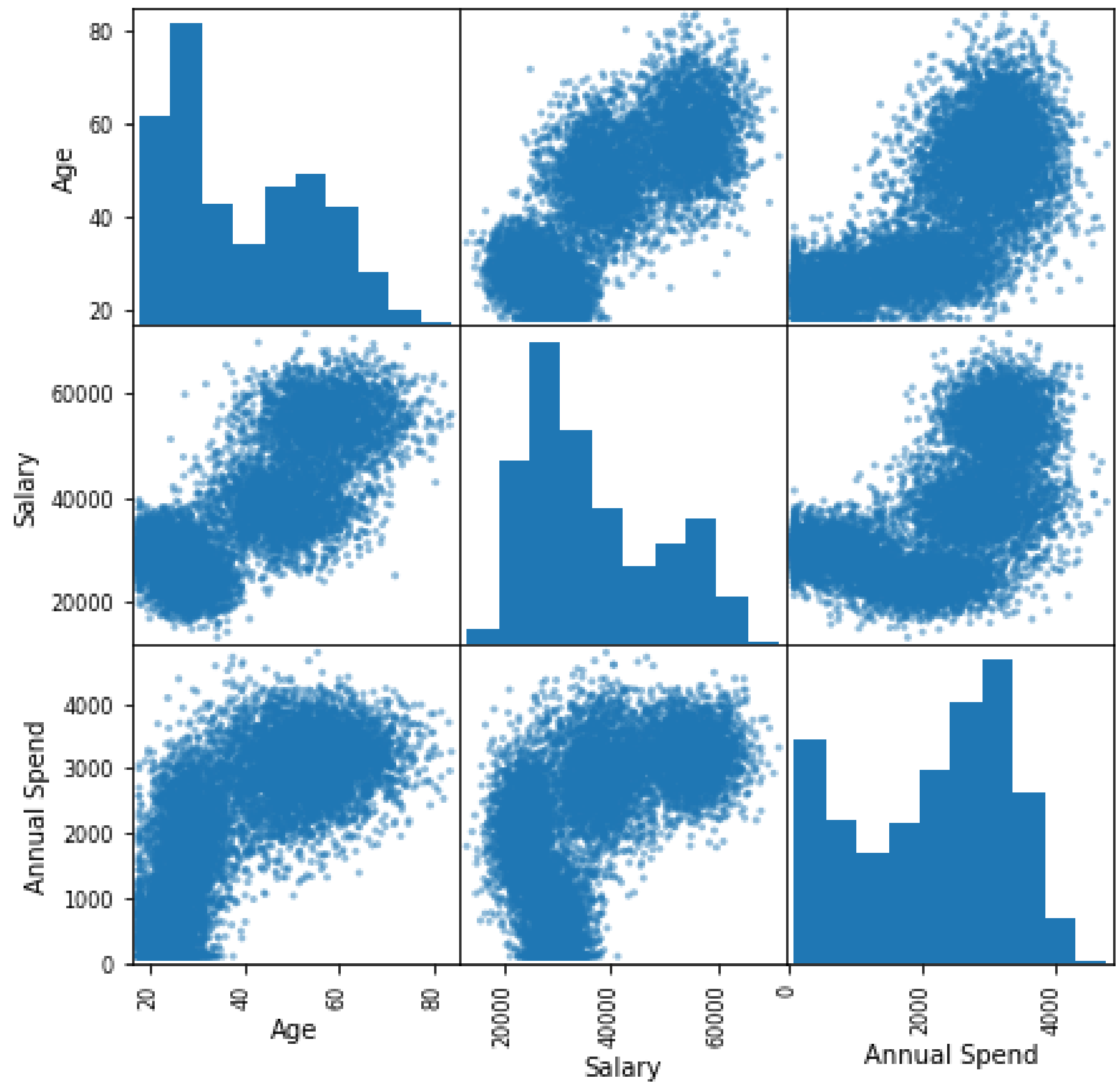


Figure 4: Scatter Matrix of Numerical features

It is already fairly easy to see that there is some natural clustering of data here.

6.7 Scale (standardise) the data into a standard size

Scaling is often important pre-processing step for Machine Learning. In particular, standardisation can often make algorithms run more quickly. There are several approaches to scaling. In this case we are going to shift the data so that its mean is around zero. The data is also scaled to a width which matches a 'Normal' (Gaussian) distribution.

This concept will be explored in one of the later lectures in the course. However, for now, we will just go ahead and re-scale the data.

Create code that uses the sklearn preprocessing.scale' method to standardise the dataframe. Store the standardised data into a new dataframe called 'standardized_customer_data'

```
standardized_customer_data = preprocessing.scale(customer_data)
numeric_columns = ['Age', 'Salary', 'Annual Spend']
standardized_customer_data_df = pd.DataFrame(standardized_customer_data,
            columns = customer_data.columns)
standardized_customer_data_df[1:20]
```

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
1	0.3	0.2	0.8	1.0	-1.0	1.0	-1.0
2	-0.8	0.0	-0.7	1.0	-1.0	-1.0	1.0
3	-1.4	-0.8	-1.2	-1.0	1.0	-1.0	1.0
4	-1.4	-0.5	-1.8	-1.0	1.0	-1.0	1.0
5	-1.1	-1.5	0.6	1.0	-1.0	-1.0	1.0
6	0.3	0.0	-0.2	1.0	-1.0	1.0	-1.0
7	-0.8	-0.2	-1.1	-1.0	1.0	-1.0	1.0
8	0.8	-0.2	0.0	1.0	-1.0	1.0	-1.0
9	-0.8	-1.2	-1.4	1.0	-1.0	-1.0	1.0
10	0.5	-0.3	0.2	1.0	-1.0	1.0	-1.0
11	-0.6	-0.9	-0.4	1.0	-1.0	-1.0	1.0
12	-0.3	-1.1	-0.7	1.0	-1.0	-1.0	1.0
13	0.4	-0.7	1.0	1.0	-1.0	1.0	-1.0
14	-0.4	-1.2	0.4	1.0	-1.0	-1.0	1.0
15	-0.7	-0.7	1.4	1.0	-1.0	-1.0	1.0
16	2.0	0.2	1.6	1.0	-1.0	1.0	-1.0
17	-0.8	-0.9	-1.2	1.0	-1.0	-1.0	1.0
18	0.0	1.0	0.8	-1.0	1.0	1.0	-1.0
19	-0.6	-0.5	-0.9	1.0	-1.0	-1.0	1.0

6.8 Decide the number of clusters

There are several approaches to determining the number of clusters. It may be that we determine the number of clusters arbitrarily. For example, this might be because we wish to manage a certain number of clusters for business operational reasons. For example, we may simple decide that 4 different customer groups is all we can practically managed.

In some cases, scatter plots will show that there are visibly distinct clusters in the data.

Alternatively, we can use statistics from the clustering to help determine a 'natural' number of clusters. Two methods for doing this are presented in the following sections.

6.8.1 Silhouette Method

One example of a metric that with help in this regard is a 'Silhouette' chart, as shown below.

The silhouette function returns a value between -1 and 1 and measures the extent to which clusters overlap. The higher the score the better separated are the clusters.

The silhouette library function is documented at:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

Add the following code to your notebook to display the silhouette score for various numbers of clusters.

```
%matplotlib inline

def k_silhouette(X, clusters):
    K = range(2, clusters+1)
    score = []
    for k in K:
        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        labels = kmeans.labels_
        score.append(metrics.silhouette_score(X, labels,
                        metric='euclidean'))

    plt.plot(K, score, 'b*-')
    plt.xlabel('k')
    plt.ylabel('silhouette_score')

    plt.show();


k_silhouette(standardized_customer_data_df, 15)
```

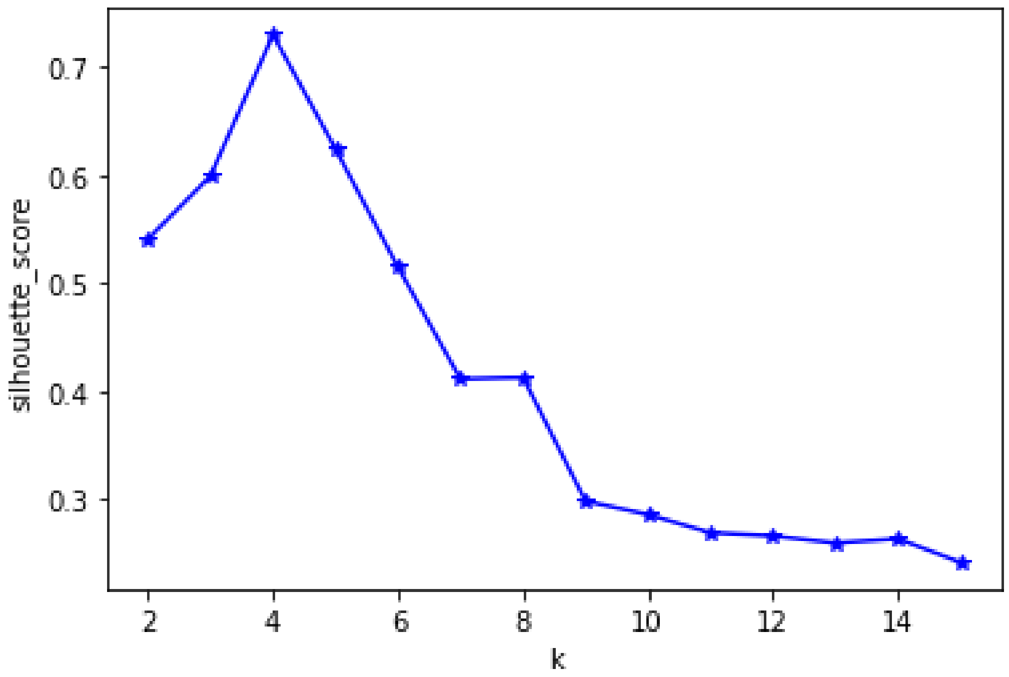
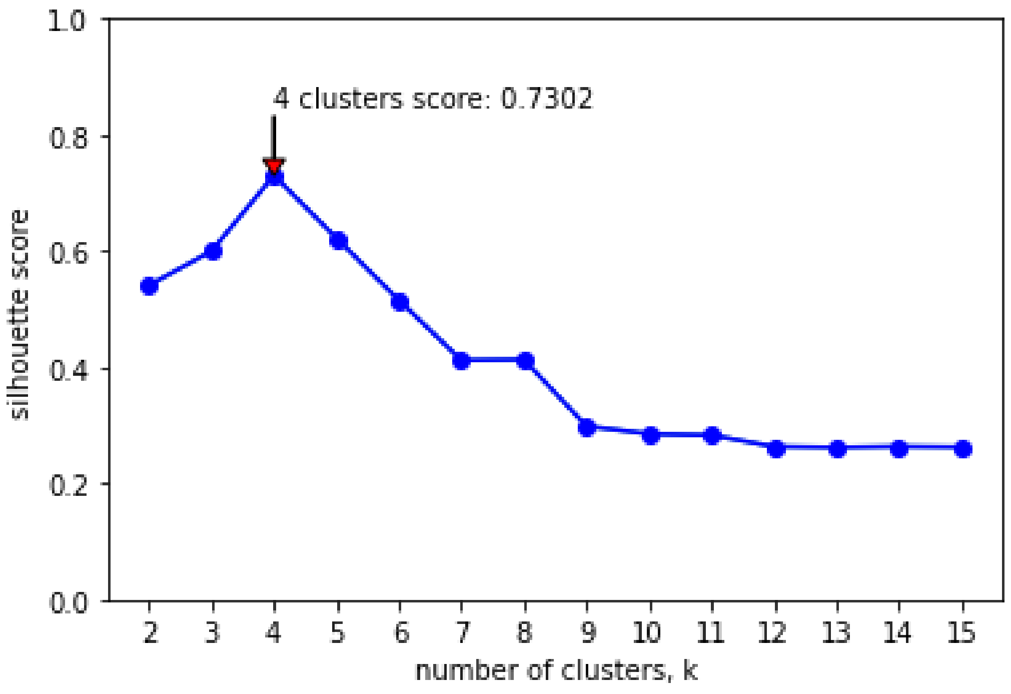


Figure 5: Silhouette score for various numbers of clusters

If you wish, you can annotate diagrams like this for reports and presentations

```
%matplotlib inline

#| fig-cap: Silhouette score with Annotation

def k_silhouette_annotated(X, clusters):
    K = range(2, clusters+1)
    scores = []
    for k in K:
        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        labels = kmeans.labels_
        scores.append(metrics.silhouette_score(X, labels,
                                    metric='euclidean'))


    fig = plt.figure()
    ax = fig.add_subplot(111)

    line = ax.plot(K, scores)

    ymax = max(scores)
    i = scores.index(ymax)
    xmax = K[i]

    ax.annotate(("%s clusters score: %s" % (xmax, round(ymax,4))),
                xy=(xmax, ymax), xytext=(xmax, ymax+0.12),
                arrowprops=dict(facecolor='red', shrink=0.0005,
                width=0.5,
                headwidth=8,
                headlength=6),
            )

    ax.set_ylim(0,1)

    plt.plot(K, scores, '-o', color='blue');
    plt.xlabel('number of clusters, k');
    plt.ylabel('silhouette score');
    plt.xticks(K);
    plt.show();


k_silhouette_annotated(standardized_customer_data_df, 15)
plt.show()
```



6.8.2 Elbow Method

A second, commonly applied way of determining an ‘optimum’ number of clusters is the ‘Elbow’ method.

To calculate the Elbow Method, we use the ‘.inertia_’ property of the sklearn kmeans function. This gives us the sum of the squared distances of each point from the cluster centre. As you will see from the graph below, as the number of clusters increases, the Standard Square Error (SSE) rapidly falls to a value. After this, there is very little improvement in terms of moving points closer to cluster centres. The ‘elbow’ value on the graph indicates the optimum number of clusters.

```
sse = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k,)
    kmeans.fit(standardized_customer_data_df)
    sse.append(kmeans.inertia_)
x = range(1, 11)
plt.xlabel('K')
plt.ylabel('SSE')
plt.plot(x, sse, 'o-')
plt.show()
```

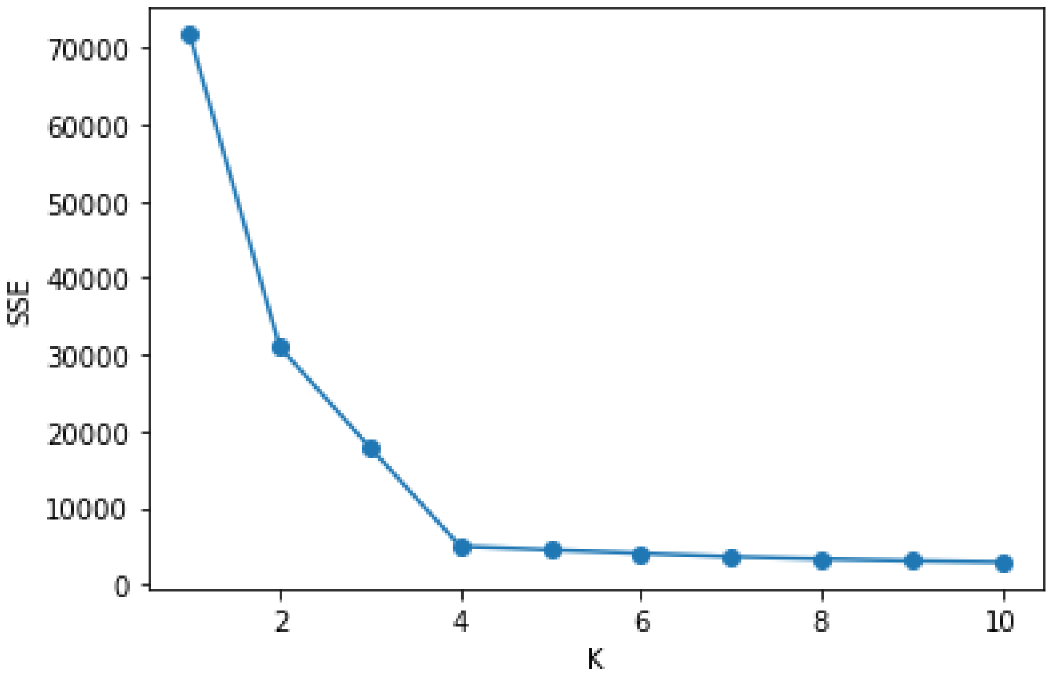


Figure 6: Elbow method plot

In this case we see good correspondence between the Silhouette and Elbow methods .. the number of clusters here is 4.

6.9 Cluster the data (Build the model)

In this section we will use the sklean 'kmeans' algorithm to cluster the (now standardised) customer data into clusters.

First we create the object ('machine') that we will use to build the model. In this case I arbitrarily decided that I want the data grouped into 4 clusters. One of the limitations of the KMeans algorithm is that the user has to decide how many clusters that the data should be arranged into.

Create an object called 'kmeans' as in instance of the sklearn 'KMeans' class. Set the number of cluster ('n_clusters') to 4.

```
kmeans = KMeans(n_clusters=4)
```

Apply the .fit() method of the KMeans class to the standardised data frame.

```
kmeans.fit(standardized_customer_data_df)
```

```
KMeans(n_clusters=4)
```

The '.fit' method determines the centres of each cluster using the k-means algorithm.

Having identified in centres of each cluster, we now assign each member of the data-set to one of the clusters. Members are assigned to the cluster which has it centre closest to it.

Add code to apply the 'fit_predict' method to the standardised data frame. Store the result in a variable 'y_km'

```
y_km = kmeans.fit_predict(standardized_customer_data_df)
```

6.10 Review the Results

The 'y_km' variable stores a list of numbers. Each number represents the cluster assignment for the corresponding row of data in the standardised data frame.

Add code to print the first twenty rows of y_km

```
print (f"y_km[0:20] = {y_km[0:20]}")
```

```
y_km[0:20] = [3 2 1 3 3 1 2 3 2 1 2 1 1 2 1 1 2 1 0 1]
```

Alternatively, we could display the members of particular clusters .. starting with those in group 2

```
print(customer_data[y_km==2][0:10])
```

```
    Age  Salary  Annual Spend  G_Female  G_Male  M_Married  M_Single
1  43.5 39366.4        3071.2         1       0          1         0
6  43.5 36642.0        1926.4         1       0          1         0
8  52.3 34779.8        2159.1         1       0          1         0
10 47.3 33535.1        2344.8         1       0          1         0
13 45.8 27398.7        3251.0         1       0          1         0
16 70.1 38683.1        3965.8         1       0          1         0
23 26.3 38994.8        4377.9         1       0          1         0
24 46.5 34270.0        3361.1         1       0          1         0
27 43.3 37264.7        2919.6         1       0          1         0
30 37.4 34761.2        1954.9         1       0          1         0
```

6.10.1 Identifying cluster centres

It is often useful to identify the centre of clusters as they are in some way 'representative' of each cluster. They can also be used to measure the distance of any individual from the 'centre' or 'ideal' case.

First, create a new dataframe called 'customer_clusters' which is a copy of 'standardized_customer_data_df.

To this dataframe, append a new column called 'y_km' which is a copy of the y_km cluster numbers from the above activities.

```
customer_clusters = standardized_customer_data_df.copy()
customer_clusters["y_km"] = y_km
customer_clusters
```

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single	y_km
0	-0.8	-0.8	-1.6	-1.0	1.0	-1.0	1.0	3
1	0.3	0.2	0.8	1.0	-1.0	1.0	-1.0	2
2	-0.8	0.0	-0.7	1.0	-1.0	-1.0	1.0	1
3	-1.4	-0.8	-1.2	-1.0	1.0	-1.0	1.0	3
4	-1.4	-0.5	-1.8	-1.0	1.0	-1.0	1.0	3
...	...	...	...	...	...	...	...	...
10242	-1.0	-1.0	-0.3	1.0	-1.0	-1.0	1.0	1
10243	-0.9	-0.8	0.2	1.0	-1.0	-1.0	1.0	1
10244	-1.2	-0.6	-1.5	-1.0	1.0	-1.0	1.0	3
10245	-1.1	-1.1	-0.2	1.0	-1.0	-1.0	1.0	1

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single	y_km
10246	-0.6	-0.6	0.4	-1.0	1.0	-1.0	1.0	3

We can then use the 'groupby' method of pandas to collect together customers in each cluster and find the mean values for each feature.

```
Cluster_Means = customer_clusters.groupby(['y_km']).mean().round(1)
Cluster_Means
```

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
y_km							
0	1.2	1.5	0.9	-1.0	1.0	1.0	-1.0
1	-0.7	-1.0	-0.1	1.0	-1.0	-1.0	1.0
2	0.5	0.1	0.6	1.0	-1.0	1.0	-1.0
3	-1.0	-0.5	-1.4	-1.0	1.0	-1.0	1.0

6.10.2 Visualising Clusters

We can also review the clusters in chart form.

In the following charts, the cluster that a particular point belongs to is indicated by a colour.

.. First plotting Age Vs Salary

```
plt.figure(figsize=(7,7))
plt.scatter(customer_data[y_km ==0]['Age'],
                customer_data[y_km == 0]['Salary'],
                s=15, c='red', alpha=.5)
plt.scatter(customer_data[y_km ==1]['Age'],
                customer_data[y_km == 1]['Salary'],
                s=15, c='black', alpha=.5)
plt.scatter(customer_data[y_km ==2]['Age'],
                customer_data[y_km == 2]['Salary'],
                s=15, c='blue', alpha=.5)
plt.scatter(customer_data[y_km ==3]['Age'],
                customer_data[y_km == 3]['Salary'],
                s=15, c='cyan', alpha=.5)
plt.show()
```

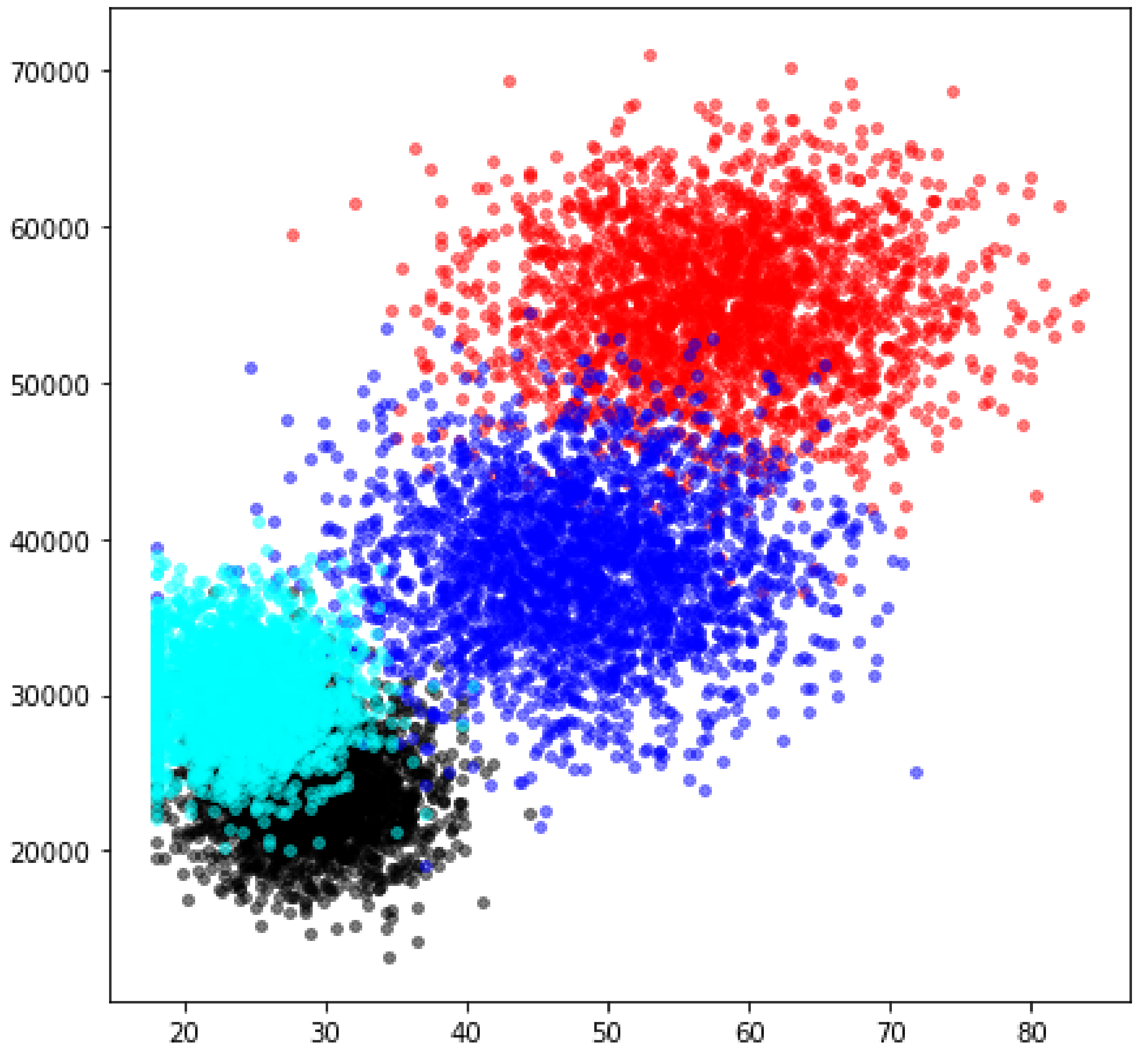


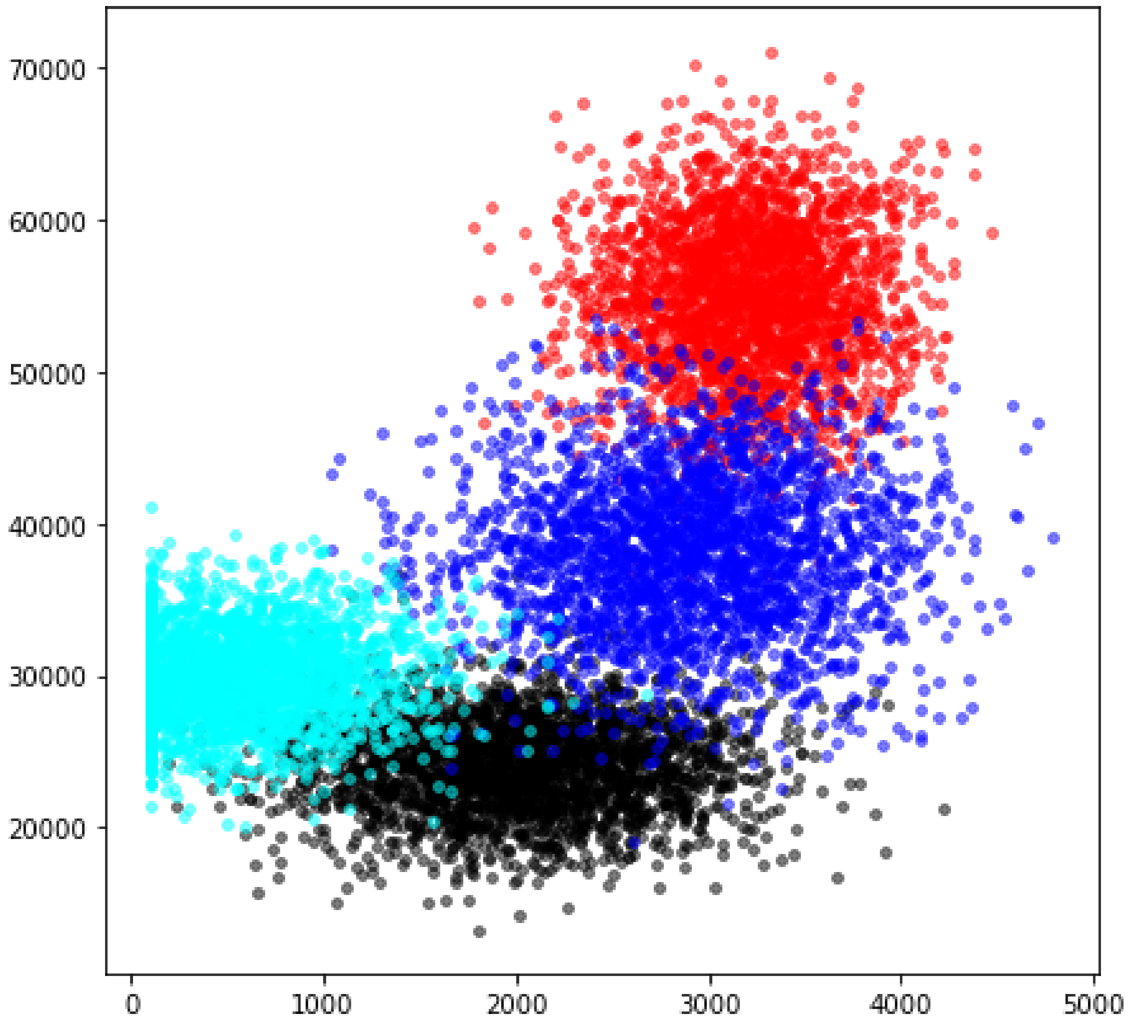
Figure 7: Scatter plot of 'Age' vs 'Salary' with each data point colour coded by cluster

Then plot Annual Spend against Salary

```
plt.figure(figsize=(7,7))
plt.scatter(customer_data[y_km ==0]['Annual Spend'],
                customer_data[y_km == 0]['Salary'],
                s=15, c='red', alpha=.5)
plt.scatter(customer_data[y_km ==1]['Annual Spend'],
                customer_data[y_km == 1]['Salary'],
                s=15, c='black', alpha=.5)
plt.scatter(customer_data[y_km ==2]['Annual Spend'],
                customer_data[y_km == 2]['Salary'],
                s=15, c='blue', alpha=.5)
plt.scatter(customer_data[y_km ==3]['Annual Spend'],
                customer_data[y_km == 3]['Salary'],
                s=15, c='cyan', alpha=.5)
plt.show
```

```
<function matplotlib.pyplot.show(close=None, block=None)>
```

Scatter plot of ‘Annual Spend’ vs ‘Salary’ with each data point colour coded by cluster



We can even take a ‘3D’ view of the data

```
%matplotlib inline

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')
```

```python
ax.view_init(20, 20)
ax.set_xlabel('Annual Spend')
ax.set_ylabel('Salary')
ax.set_zlabel('Age')
ax.scatter(customer_data[y_km ==0]['Annual Spend'],
                customer_data[y_km == 0]['Salary'],
                customer_data[y_km == 0]['Age'],
                s=15, c='red', alpha=.3)
ax.scatter(customer_data[y_km ==1]['Annual Spend'],
                customer_data[y_km == 1]['Salary'],
                customer_data[y_km == 1]['Age'],
                s=15, c='black', alpha=.3)
ax.scatter(customer_data[y_km ==2]['Annual Spend'],
                customer_data[y_km == 2]['Salary'],
                customer_data[y_km == 2]['Age'],s=15,
                c='blue', alpha=.3)
ax.scatter(customer_data[y_km ==3]['Annual Spend'],
                customer_data[y_km == 3]['Salary'],
                customer_data[y_km == 3]['Age'],s=15,
                c='cyan', alpha=.3)

plt.show()
```

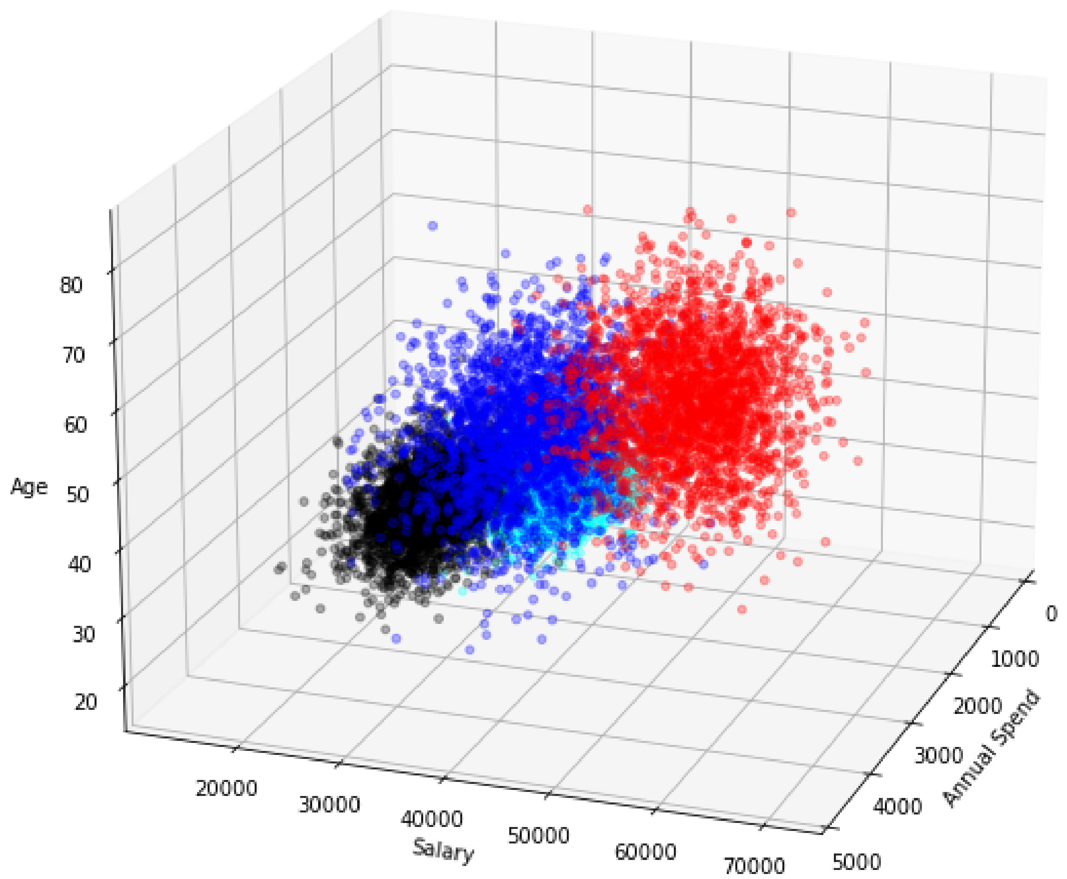


Figure 8: Scatter plot of ‘Annual Spend’ vs ‘Salary’ with each data point colour coded by cluster

In the above code I have included a line:

```
%matplotlib
```

This should cause a 3D graph to display in a pop-out window.

If you prefer, you can also display the chart as a static imagine ‘in line’ using the following command:

```
%matplotlib inline
```

6.11 Another method for Visualising Clusters

Another attractive method for displaying discovered clusters is provided by using sns pairplot ..

```python
# The 'X' on the following lines is simply to make
# this code easier to follow

X = standardized_customer_data_df[['Age', 'Salary', 'Annual Spend']].copy()
X["y_km"] = y_km.astype(str)
sns.pairplot(X, hue='y_km')

plt.show()
```

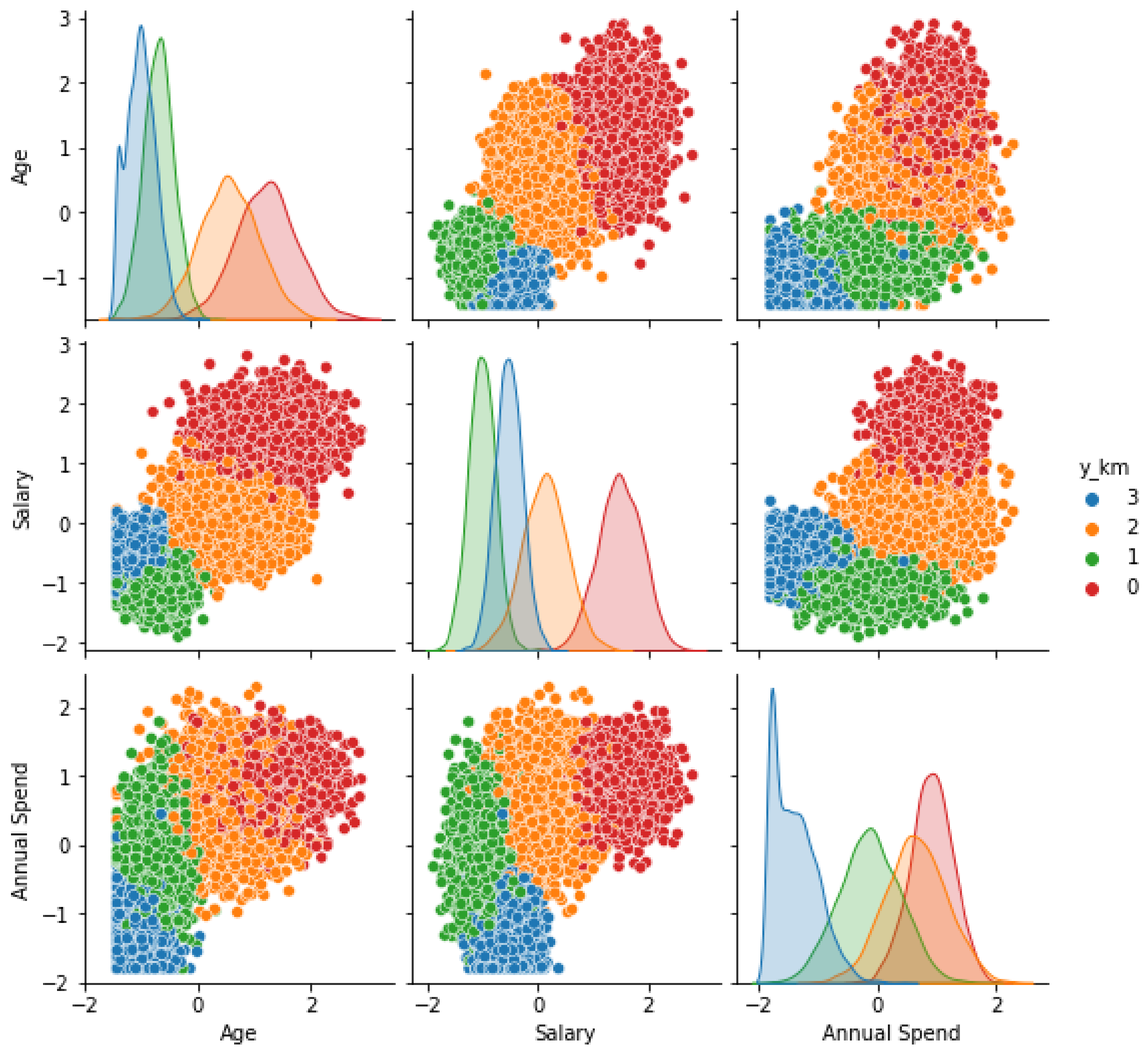


Figure 9: sns Pairplot of the dataset with each data point colour coded by cluster

6.12 Bonus Material - Experimenting with Other Clustering Algorythms

sklearn provides a large set of different clustering methods. If you have time available, then you could use your time usefully to explore some of those other methods. The main sklearn URL for clustering is here:

https://scikit-learn.org/stable/modules/clustering.html

And below are some of the examples I selected to experiment with.

6.12.1 DBSCAN

DBSCAN is documented here:

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

DBSCAN has a number of difference to kmeans: The algorithm ..

- Determines the optimum number of clusters - you are not required to pass this as a parameter
- Identifies 'noise points' that do not fit the model clustering.

First, create an object called 'db' which is an instance of the sklearn 'DBSCAN' class. For now, do not pass any parameters to this instantiation.

```python
db = DBSCAN()
```

Then, as is consistent througout sklearn, cluster the data using the '.fit' method, passing 'standardized_customer_data_df' as a parameter.

```python
db.fit(standardized_customer_data_df)
```

```
DBSCAN()
```

To make the following code slightly easier to read, I copied the result labels from the 'db.labels_' attribute into a variable called 'labels'. 'labels' is an array of dtype int64

```python
labels = db.labels_
print(f"labels = {labels}")
```

```
labels = [ 0  1  2 ...  0  2 -1]
```

It is useful to know how many different clusters have been identified. That can be obtained by reviewing the number of unique label values. An easy way to do this in Python is to create a set (using the 'set()' function). A set created from the list of labels will include unique values of each label value.

```python
label_set = set(labels)
print(f"label_set = {label_set}")
```

```
label_set = {0, 1, 2, 3, -1}
```

An interesting feature of DBSCAN is that it identifies 'noise points' - those that don't fit into the cluster model. These are labelled '-1'. We can count the number of points that have been identified as noise by first converting the labels into a list ..

```python
label_list= list(labels)
print(f"label_list[0:10] = {label_list[0:10]}")
```

```
label_list[0:10] = [0, 1, 2, 0, 0, 2, 1, 0, 1, 2]
```

Then counting the number of items in that list that are labelled '-1'. This can be achieved using the '.count(-1)' function.

```python
noise_point_count = label_list.count(-1)
print(f"noise_point_count = {noise_point_count}")
```

```
noise_point_count = 6
```

After that, we can create charts of the clusters as previously.

```python
X = standardized_customer_data_df[['Age', 'Salary', 'Annual Spend']].copy()
X["labels"] = labels.astype(str)
sns.pairplot(X, hue='labels')

plt.show()
```

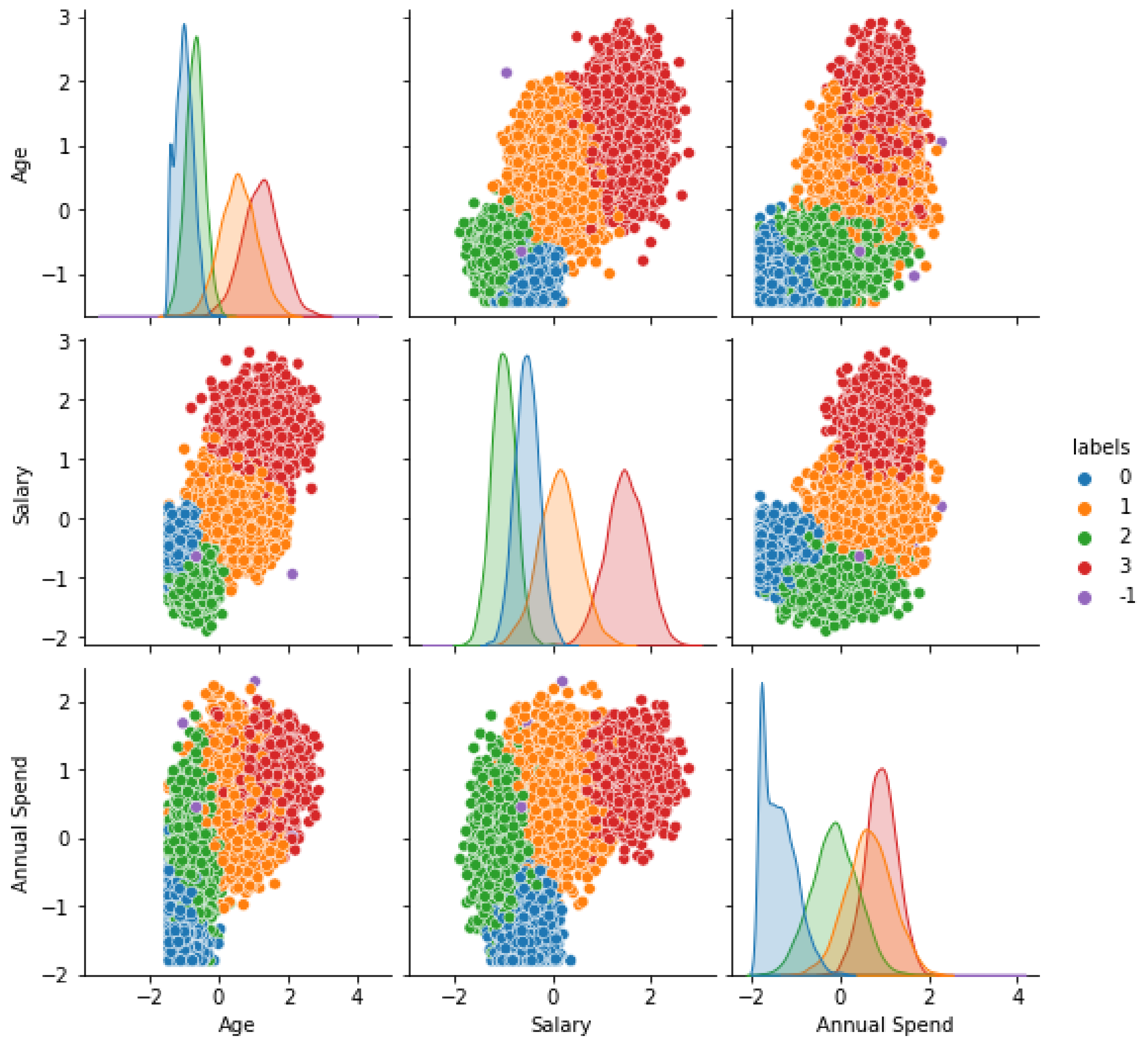


Figure 10: sns Pairplot for DBSCAN clustering

Note that the documentation points out that there is an important parameter for DBSCAN called ‘eps’. This represents the maximum distance between two points for them to be considered as part of the same neighborhood. So an instantiation of DBSCAN would normally look something like:

```
db = DBSCAN(eps=0.3, min_samples=10)
```

6.12.2 Optics

This is the best article I have found on the OPTICS algo: https://saturncloud.io/blog/python-implementation-of-optics-clustering-algorithm/

This article provides some more information: https://www.geeksforgeeks.org/ml-optics-clustering-explanation/

There are two another articles here .. although unfortunately they are behind the Medium paywal:

https://towardsdatascience.com/understanding-optics-and-implementation-with-python-143572abdfb6

https://pub.towardsai.net/fully-explained-optics-clustering-with-python-example-4553108fa04b

```python
opt = OPTICS(min_samples=50)
opt.fit(standardized_customer_data_df)
```

```
OPTICS(min_samples=50)
```

```python
labels = opt.labels_
print(f"labels = {labels}")

label_set = set(labels)
print(f"label_set = {label_set}")

label_list= list(labels)
print(f"label_list[0:10] = {label_list[0:10]}")

noise_point_count = label_list.count(-1)
print(f"noise_point_count = {noise_point_count}")

```

```
labels = [0 2 1 ... 0 1 0]
label_set = {0, 1, 2, 3}
label_list[0:10] = [0, 2, 1, 0, 0, 1, 2, 0, 2, 1]
noise_point_count = 0
```

```python
X = standardized_customer_data_df[['Age', 'Salary', 'Annual Spend']].copy()
X["labels"] = labels.astype(str)
sns.pairplot(X, hue='labels')

plt.show()
```

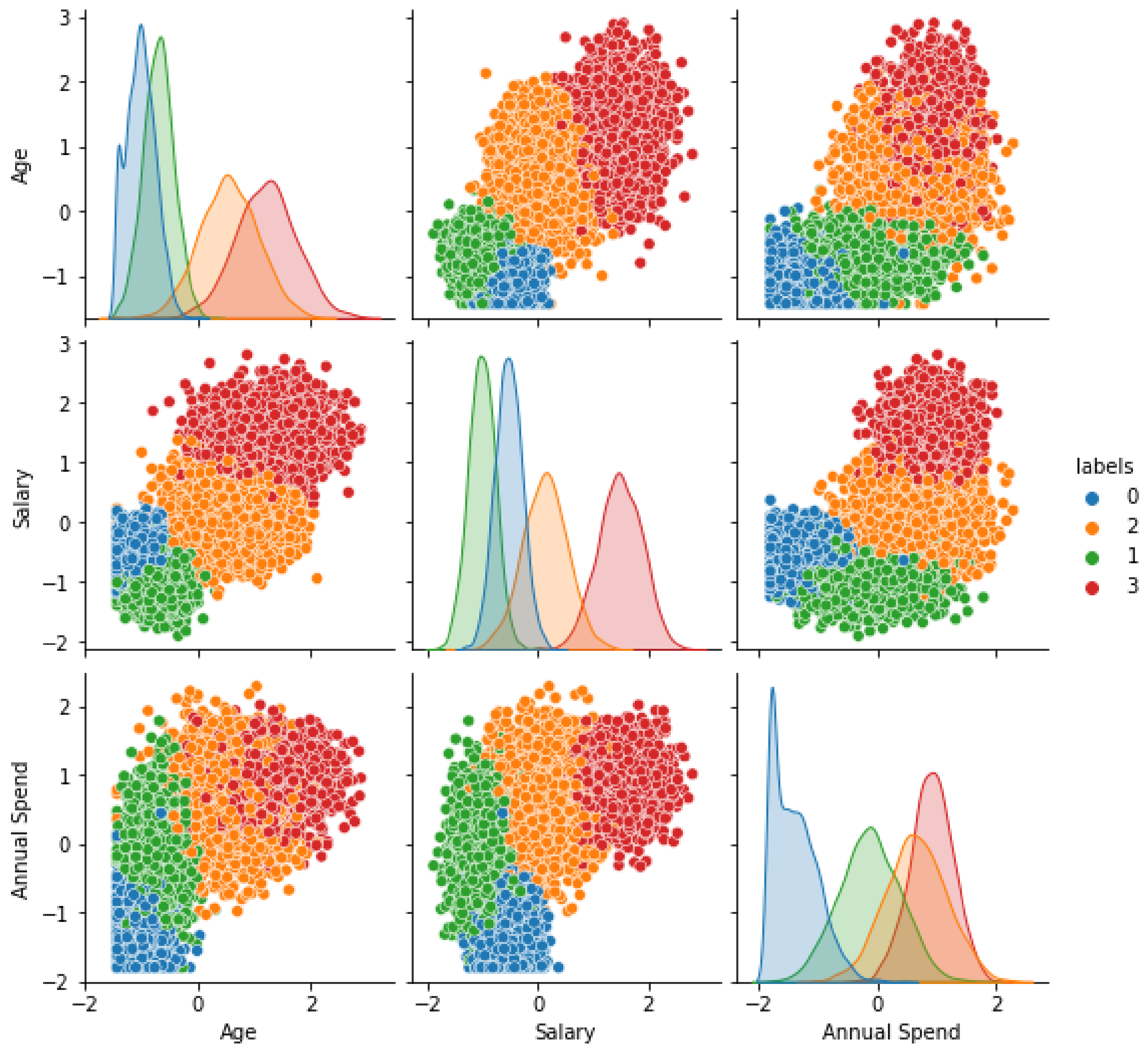


Figure 11: sns Pairplot for OPTICS clustering

To quote from https://www.geeksforgeeks.org/ml-optics-clustering-explanation/

"OPTICS (Ordering Points To Identify the Clustering Structure) is a density-based clustering algorithm, similar to DBSCAN (Density-Based Spatial Clustering of Applications with Noise), but it can extract clusters of varying densities and shapes. It is useful for identifying clusters of different densities in large, high-dimensional datasets.

The main idea behind OPTICS is to extract the clustering structure of a dataset by identifying the density-connected points. The algorithm builds a density-based representation of the data by creating an ordered list of points called the reachability plot. Each point in the list is associated with a reachability distance, which is a measure of how easy it is to reach that point from other points in the dataset. Points with similar reachability distances are likely to be in the same cluster."