EXPERIMENT 3

K MEANS CLUSTERING UNSUPERVISED LEARNING

TITLE: *THE K-MEANS CLUSTERING CLASSIFICATION USING PYTHON WITH SCIKIT-LEARN LIBRARY*

INDEX

1.1 OBJECTIVE(S):

- To gain the ability to segment large datasets into meaningful clusters. This allows for the identification of underlying patterns or structures within the data that may not be apparent through manual inspection.
- To identify anomalies or outliers within a dataset.
- To effectively apply this algorithm in real-world scenarios like segment customers based on their purchasing behavior, group customers with similar shopping habits to provide personalized product recommendation or identify unusual network traffic patterns that may indicate a security breach or cyber-attack.

1.2 EQUIPMENT/SETUP:

- A Computer (PC) with Internet connection
- Operating system
- Any browser to use Google Colab (*https://colab.research.google.com*)

1.3 BACKGROUND:

The K Means Clustering algorithm works on unlabeled data and is a type of **unsupervised machine learning algorithm** used for **clustering/segmentation** and to some extend **outlier detection**. The algorithm's name "K-means" comes from the fact that it aims to partition a dataset into K distinct clusters. The "means" part refers to the calculation of the cluster centroids, which represent the mean of all data points within a cluster.

The algorithm aims to minimize the variance within clusters, effectively trying to find the configuration of clusters that best represents the underlying structure of the data.

Since its inception, K-means clustering has been applied to a wide range of fields and applications, including image segmentation, natural language processing, marketing, biology, and more. It's considered a fundamental technique in unsupervised learning and is often used as a building block for more complex algorithms and analyses.

K-means clustering is a ***model-based approach***. It involves an iterative process of assigning data points to clusters and updating cluster centroids until convergence is reached. The algorithm builds a model that consists of the cluster centroids and assigns data points to clusters based on their similarity to these centroids.

K-means clustering is a ***parametric learning algorithm***. It assumes that the data can be partitioned into K clusters, where K is a predefined parameter. It makes specific assumptions about the data's distribution and structure.

1.4 K-MEANS CLUSTERING STRATEGY:

The K-means clustering algorithm is a strategy for partitioning a dataset into K distinct clusters based on certain characteristics of the data points. Figure 1.1 depicts the conceptual block diagram of how K-Means Clustering works. Here is a step-by-step strategy for applying K-means clustering:

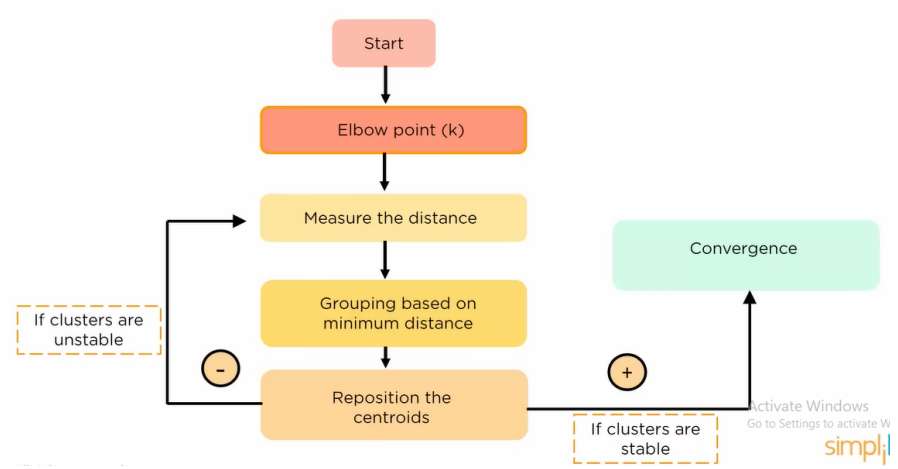
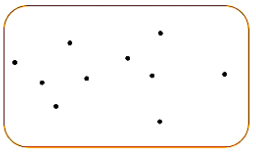


Figure 1.1
Conceptual block diagram of K-Means Clustering

Step 1: Determine the Number of Clusters (K):

- Let's say, you have a dataset for a **Grocery shop**



- Now, the important question is, "***how would you choose the optimum number of clusters?***"

Figure 1.2
How to choose optimum number of clusters?

- Decide how many clusters (K) you want to divide your data into. This can be based on domain knowledge, prior experience, or using techniques like the **Elbow Method** or silhouette analysis to find an optimal K.

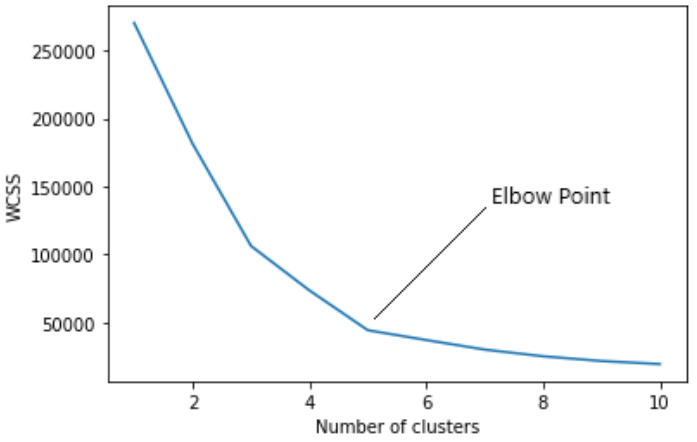


Figure 1.3
Elbow Method to choose optimal number of clusters

Step 2: Initialize Cluster Centroids:

- Randomly select K data points from your dataset as initial cluster centroids as shown in Figure 1.4. These will serve as the starting points for the clusters.

Figure 1.4
Initializing Cluster Centroids

Step 3: Assign Data Points to Nearest Cluster:

- Calculate the distance (typically **Euclidean distance** shown in Figure 1.5) between each data point and all cluster centroids.

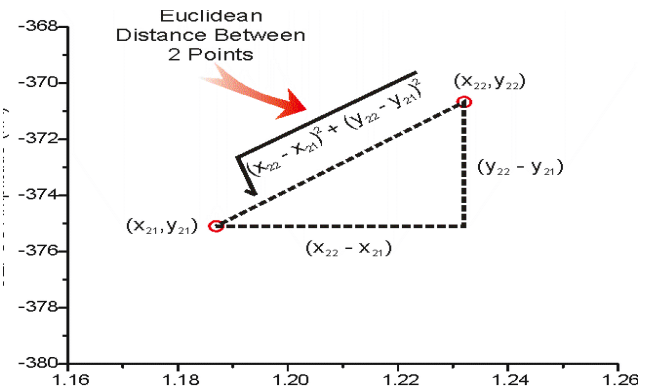


Figure 1.5

- Assign each data point to the cluster whose centroid is the nearest as shown in Figure 1.6. This is based on the "nearest neighbor" principle.

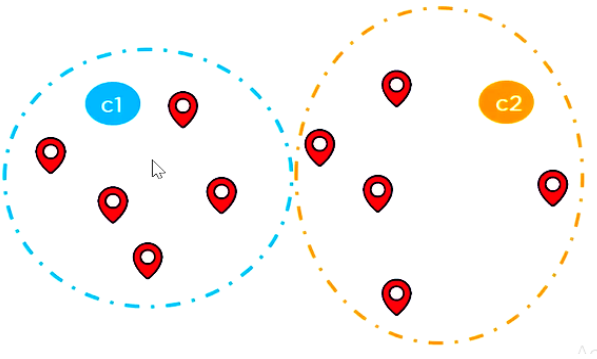


Figure 1.6
Assigning each data point to K clusters

Step 4: Update Cluster Centroids:

- Recalculate the centroid (mean) of each cluster using the data points that are currently assigned to it (using Figure 1.7 equation). Visualization is given in Figure 1.8.

$$c_i = \frac{1}{m_i} \sum_{x \in C_i} x$$

Figure 1.7
Finding mean point or new centroid of a cluster

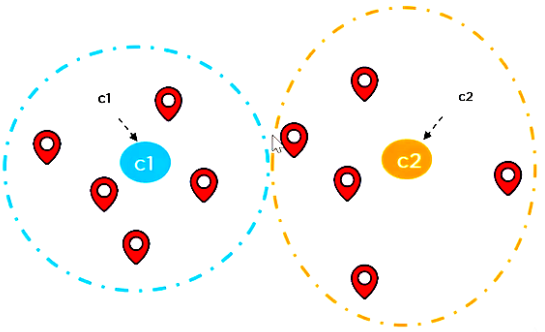


Figure 1.8
Reassigning each data point to new centroids

Step 5: Repeat Steps 3 and 4:

- Iteratively repeat the assignment and centroid update steps until one of the stopping criteria is met, such as a maximum number of iterations or when the centroids no longer change significantly between iterations. That's called convergence.

Step 6: Finalize Clustering:

- The algorithm terminates when one of the stopping criteria is met. At this point, you have your clusters, and each data point belongs to one of them as shown in Figure 1.9.

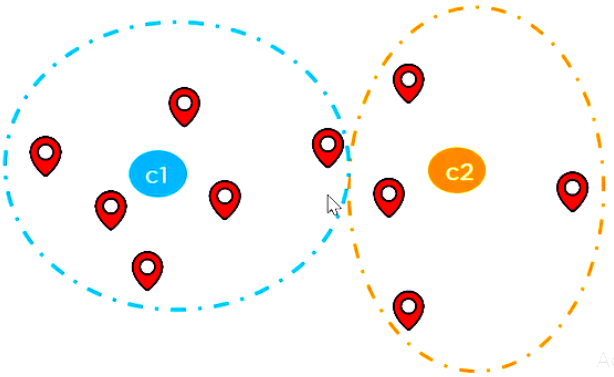


Figure 1.9
Clusters reaching to a convergence level by not changing the centroids

Step 7: Evaluate and Interpret Results:

- Assess the quality of your clustering results using metrics like Sum Squared Error (SSE) or silhouette score. Visualize the clusters if possible.

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} dist^2(c_i, x)$$

Figure 1.10
Assessing quality using SSE

- Interpret the clusters to understand the patterns and relationships within your data.

Step 8: Repeat or Refine (Optional):

- You can experiment with different initializations or vary the number of clusters to see how it affects the results. Sometimes running K-means multiple times with different starting centroids can help improve the robustness of the clustering.

Remember that K-means can be sensitive to the initial placement of centroids and may converge to local optima. Therefore, it's often a good idea to run the algorithm

multiple times with different initializations and select the best result based on a chosen evaluation metric.

1.5 PROBLEM DESCRIPTION:

In this project, we are presented with a transnational dataset representing all the transactions of a UK-based online retail company from December 1, 2010, to December 9, 2011. The company specializes in selling unique all-occasion gifts, and a significant portion of its customer base consists of wholesalers.

The goal of this project is to segment the company's customer base using the **FM model (Frequency, Monetary)** through the application of K-means clustering. The FM model is a widely used customer segmentation technique that classifies customers based on their purchasing behavior.

The expected outcome of this project is a clear segmentation of the customer base into distinct clusters, each representing a different type of customer. This will provide the company with actionable insights for more effective marketing, product offerings, and customer relationship management.

Figure 1.11
Online Retail

1.6 PROCEDURE:

- Get the '**OnlineRetail.csv**' and upload in your Google Drive
- Open Google Colab (*https://colab.research.google.com/*)
- Create/rename your project as '*OnlineRetail_KMeans.ipynb*'
- Follow the steps of next section (read and understand) and execute all the codes

1.7 LAB PRACTICE/EXPERIMENT:

1.7.1 Getting the dataset

Download the dataset '**OnlineRetail.csv**' and upload in your Google Drive. Please open the CSV file to see and understand the data.

1.7.2 Understanding the dataset

The '***OnlineRetail.csv***' contains eight features. The features are,

- **InvoiceNo:** It contains the purchase id of a customer.
- **Description:** It contains the description of the product purchased by the customer.
- **Quantity:** It contains the quantities of each product (item) per transaction.
- **InvoiceDate:** It contains the day and time when each transaction was generated.
- **UnitPrice:** It contains the unit price of a product.
- **CustomerID:** It contains a 5-digit integral number uniquely assigned to each customer.
- **Country:** It contains the name of the country where each customer resides.

The CSV file contains 541908 records in total. Table 1.1 shows the data of the CSV file with some missing values.

Table 1.1

*The dataset '**OnlineRetail.csv**'*

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	1/12/2010 8:26	2.55	17850	United Kingdom
536365	71053	WHITE METAL LANTERN	6	1/12/2010 8:26	3.39	17850	United Kingdom
536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	1/12/2010 8:26	2.75	17850	United Kingdom
536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	1/12/2010 8:26	3.39	17850	United Kingdom
536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	1/12/2010 8:26	3.39	17850	United Kingdom
536365	22752	SET 7 BABUSHKA NESTING BOXES	2	1/12/2010 8:26	7.65	17850	United Kingdom
536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	1/12/2010 8:26	4.25	17850	United Kingdom
536366	22633	HAND WARMER UNION JACK	6	1/12/2010 8:28	1.85	17850	United Kingdom
536366	22632	HAND WARMER RED POLKA DOT	6	1/12/2010 8:28	1.85	17850	United Kingdom
536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	1/12/2010 8:34	1.69	13047	United Kingdom
536367	22745	POPPY'S PLAYHOUSE BEDROOM	6	1/12/2010 8:34	2.1	13047	United Kingdom
536367	22748	POPPY'S PLAYHOUSE KITCHEN	6	1/12/2010 8:34	2.1	13047	United Kingdom
536367	22749	FELTCRAFT PRINCESS CHARLOTTE DOLL	8	1/12/2010 8:34	3.75	13047	United Kingdom
536367	22310	IVORY KNITTED MUG COSY	6	1/12/2010 8:34	1.65	13047	United Kingdom
536367	84969	BOX OF 6 ASSORTED COLOUR TEASPOONS	6	1/12/2010 8:34	4.25	13047	United Kingdom
536367	22623	BOX OF VINTAGE JIGSAW BLOCKS	3	1/12/2010 8:34	4.95	13047	United Kingdom
536367	22622	BOX OF VINTAGE ALPHABET BLOCKS	2	1/12/2010 8:34	9.95	13047	United Kingdom
536367	21754	HOME BUILDING BLOCK WORD	3	1/12/2010 8:34	5.95	13047	United Kingdom
536367	21755	LOVE BUILDING BLOCK WORD	3	1/12/2010 8:34	5.95	13047	United Kingdom
536367	21777	RECIPE BOX WITH METAL HEART	4	1/12/2010 8:34	7.95	13047	United Kingdom
536367	48187	DOORMAT NEW ENGLAND	4	1/12/2010 8:34	7.95	13047	United Kingdom
536368	22960	JAM MAKING SET WITH JARS	6	1/12/2010 8:34	4.25	13047	United Kingdom
536368	22913	RED COAT RACK PARIS FASHION	3	1/12/2010 8:34	4.95	13047	United Kingdom
536368	22912	YELLOW COAT RACK PARIS FASHION	3	1/12/2010 8:34	4.95	13047	United Kingdom
536368	22914	BLUE COAT RACK PARIS FASHION	3	1/12/2010 8:34	4.95	13047	United Kingdom
536369	21756	BATH BUILDING BLOCK WORD	3	1/12/2010 8:35	5.95	13047	United Kingdom
536370	22728	ALARM CLOCK BAKELIKE PINK	24	1/12/2010 8:45	3.75	12583	France
536370	22727	ALARM CLOCK BAKELIKE RED	24	1/12/2010 8:45	3.75	12583	France
536370	22726	ALARM CLOCK BAKELIKE GREEN	12	1/12/2010 8:45	3.75	12583	France
536370	21724	PANDA AND BUNNIES STICKER SHEET	12	1/12/2010 8:45	0.85	12583	France
536370	21883	STARS GIFT TAPE	24	1/12/2010 8:45	0.65	12583	France
536370	10002	INFLATABLE POLITICAL GLOBE	48	1/12/2010 8:45	0.85	12583	France
536370	21791	VINTAGE HEADS AND TAILS CARD GAME	24	1/12/2010 8:45	1.25	12583	France
536370	21035	SET/2 RED RETROSPOT TEA TOWELS	18	1/12/2010 8:45	2.95	12583	France
536370	22326	ROUND SNACK BOXES SET OF4 WOODLAND	24	1/12/2010 8:45	2.95	12583	France
536370	22629	SPACEBOY LUNCH BOX	24	1/12/2010 8:45	1.95	12583	France

1.7.3 Import Libraries in Google Colab

Write the codes shown in Figure 1.3 to import preliminary libraries and execute. More libraries are required to conduct the experiment which will be imported latter.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt

# import required libraries for clustering
import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

%matplotlib inline
```

Figure 1.3

Importing some preliminary libraries into colab

1.7.4 Fetching data from google drive

Write the following code shown in Figure 1.4 to fetch the dataset from Google drive and execute. Figure 1.5 shows some data from the dataset.

```
#Fetching the dataset for training...
retail_df = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/K-Means Clustering ML/OnlineRetail.csv",
                        encoding="ISO-8859-1")
retail_df
```

Figure 1.4

Fetching dataset for training and test

Here, encoding="ISO-8859-1": This parameter specifies the character encoding used in the CSV file. "ISO-8859-1" (also known as "latin1") is a widely used encoding for CSV files.

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	01-12-2010 08:26	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	01-12-2010 08:26	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	01-12-2010 08:26	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	01-12-2010 08:26	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01-12-2010 08:26	3.39	17850.0	United Kingdom
...	...	...	...	...	...	...	...	...
541904	581587	22613	PACK OF 20 SPACEBOY NAPKINS	12	09-12-2011 12:50	0.85	12680.0	France
541905	581587	22899	CHILDREN'S APRON DOLLY GIRL	6	09-12-2011 12:50	2.10	12680.0	France
541906	581587	23254	CHILDRENS CUTLERY DOLLY GIRL	4	09-12-2011 12:50	4.15	12680.0	France
541907	581587	23255	CHILDRENS CUTLERY CIRCUS PARADE	4	09-12-2011 12:50	4.15	12680.0	France
541908	581587	22138	BAKING SET 9 PIECE RETROSPOT	3	09-12-2011 12:50	4.95	12680.0	France

Figure 1.5

Some data from the Data Frame (df)

1.7.5 Understanding the data

Write the following code shown in Figure 1.6 to understand the data. There are 541908 rows excluding the column name and 8 feature columns. Among the 8 columns, InvoiceNo, StockCode, Description, InvoiceDate and Country contains object/string data, and others contain integer/float data.

```
retail_df.shape
```

```
(541909, 8)
```

```
retail_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   InvoiceNo    541909 non-null  object
 1   StockCode    541909 non-null  object
 2   Description  540455 non-null  object
 3   Quantity     541909 non-null  int64
 4   InvoiceDate  541909 non-null  object
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      541909 non-null  object
dtypes: float64(2), int64(1), object(5)
memory usage: 33.1+ MB
```

Figure 1.6
Understanding the dataset

Write the code shown in Figure 1.7 to get a vivid description of the integer/float data type columns.

```
retail_df.describe()
```

	Quantity	UnitPrice	CustomerID
count	541909.000000	541909.000000	406829.000000
mean	9.552250	4.611114	15287.690570
std	218.081158	96.759853	1713.600303
min	-80995.000000	-11062.060000	12346.000000
25%	1.000000	1.250000	13953.000000
50%	3.000000	2.080000	15152.000000
75%	10.000000	4.130000	16791.000000
max	80995.000000	38970.000000	18287.000000

Figure 1.7
Understanding the dataset

1.7.7 Data Pre-processing

1.7.7.1 Checking the NULL values

Write the code shown in Figure 1.8 to check which feature has NULL values and how many.

```
retail_df.isnull().sum()
```

```
InvoiceNo          0
StockCode          0
Description     1454
Quantity           0
InvoiceDate        0
UnitPrice          0
CustomerID    135080
Country            0
dtype: int64
```

Figure 1.8
Code to check NULL values in a data-frame with corresponding outputs

1.7.7.2 Dropping the NULL value records

Write the code shown in Figure 1.9 to drop the NULL value records. The Null values can be replaced by the average values, or mean values, or any values base on the existing values of the corresponding feature columns. In this experiment, the Null value records are discarded.

```
retail_df = retail_df.dropna()
retail_df.isnull().sum()
```

```
InvoiceNo      0
StockCode      0
Description    0
Quantity       0
InvoiceDate    0
UnitPrice      0
CustomerID     0
Country        0
Amount         0
Difference     0
dtype: int64
```

Figure 1.9
Dropping the Null value records

After dropping the Null records, the total rows become 406828 (excluding column names), as presented in Figure 1.10. At the same time, the $retail\_df$ (Data Frame) is assigned to $retail\_df$.

```
retail_df.shape
```

```
(406829, 8)
```

Figure 1.10
Dataframe size after deleting Null values

1.7.7.3 Calculating Frequency and Monetary (Amount Spent) column

We are going to analysis the Customers based on below 2 factors:

- F (Frequency): Number of transactions of a customer.

- M (Monetary): Total amount of transactions (revenue contributed)

```
retail_df['CustomerID'] = retail_df['CustomerID'].astype(str)
```

Figure 1.11
Changing datatype of CustomerID Column

Write code given in Figure 1.11. Here, CustomerID column was numeric datatype, so it's been converted to string, because

- Avoiding Unintended Calculations:
  If **'CustomerID'** were initially stored as an integer or floating-point number, performing arithmetic operations on it might lead to unintended calculations. For example, you wouldn't want to perform mathematical operations on customer IDs.
- Maintaining Leading Zeros:
  If **'CustomerID'** contains leading zeros (e.g., '001234'), converting it to a string ensures that those zeros are preserved. In numerical form, leading zeros are typically dropped.

Now, we will calculate the monetary part from the dataset.

```
retail_df['Amount'] = retail_df['Quantity']*retail_df['UnitPrice']
fm_m = retail_df.groupby('CustomerID')['Amount'].sum()
fm_m = fm_m.reset_index()
fm_m.columns = ['CustomerID', 'Spend_Amount']
fm_m.head()
```

Figure 1.12
Calculation of monetary column from the dataset

Here's a step-by-step explanation of the code in Figure 1.12:

1. **retail_df['Amount'] = retail_df['Quantity']*retail_df['UnitPrice']:** This line creates a new column called **'Amount'** in the **retail_df** DataFrame. The values in this column are calculated by multiplying the **'Quantity'** and **'UnitPrice'** columns.
2. **fm_m = retail_df.groupby('CustomerID')['Amount'].sum():** This groups the DataFrame **retail_df** by the **'CustomerID'** column, calculates the sum of the **'Amount'** column for each group and assigns the result in **rfm_m**. This effectively gives you the total spending amount for each customer.
3. **fm_m = fm_m.reset_index():** This line resets the index of the DataFrame fm_m. When you perform group operations in pandas, the grouping columns become the index. This line ensures that **'CustomerID'** is a regular column in the DataFrame.
4. **fm_m.columns = ['CustomerID', 'Spend_Amount']:** This renames the columns of the DataFrame **fm_m**. The first column, which contains customer IDs, is renamed to **'CustomerID'**, and the second column, which contains the total spending amount, is renamed to **'Spend_Amount'**.

5. **fm_m.head():** This displays the first few rows of the DataFrame **fm_m** to give you a preview of the data.

In summary, this code calculates the total spending amount for each customer and stores it in a DataFrame called **fm_m** as shown in Figure 1.13. The resulting DataFrame has two columns: **'CustomerID'** and **'Spend_Amount'**.

	CustomerID	Spend_Amount
0	12346.0	0.00
1	12347.0	4310.00
2	12348.0	1797.24
3	12349.0	1757.55
4	12350.0	334.40

Figure 1.13
Output of the fm_m the dataset

Now, we will calculate the Frequency part from the dataset as shown in Figure 1.14

```
fm_f = retail_df.groupby('CustomerID')['InvoiceNo'].count()
fm_f = fm_f.reset_index()
fm_f.columns = ['CustomerID', 'Frequency']
fm_f.head()
```

	CustomerID	Frequency
0	12346.0	2
1	12347.0	182
2	12348.0	31
3	12349.0	73
4	12350.0	17

Figure 1.14
Calculation of Frequency and Output of the fm_f the dataset

Now, we will merge the fm_m and fm_f dataset as shown in Figure 1.15.

```
fm = pd.merge(fm_m, fm_f, on='CustomerID', how='inner')
fm.head()
```

	CustomerID	Spend_Amount	Frequency
0	12346.0	0.00	2
1	12347.0	4310.00	182
2	12348.0	1797.24	31
3	12349.0	1757.55	73
4	12350.0	334.40	17

Figure 1.15
Merging fm_m and fm_f dataset

fm = pd.merge(fm_m, fm_f, on='CustomerID', how='inner'): This line of code is performing an inner join operation using the **pd.merge()** function. **fm_m** and **fm_f** are two DataFrames that you are merging. ‘

on='CustomerID' specifies that the merge should be based on the **'CustomerID'** column in both DataFrames.

how='inner' specifies that only the rows with matching **'CustomerID'** values in both DataFrames should be included in the merged DataFrame **fm**.

1.7.7.4 Outlier Analysis of Amount and Frequency

Write the code given in Figure 1.16 to plot the dataset using boxplot for visualizing outliers. In figure 1.17, it shows that there are 4371 rows in **fm** dataframe.

```
attributes = ['Spend_Amount','Frequency']
plt.rcParams['figure.figsize'] = [10,8]
sns.boxplot(data = fm[attributes], orient="X", palette="Set2" ,whis=1.5,saturation=1, width=0.7)
plt.title("Outliers Variable Distribution", fontsize = 14, fontweight = 'bold')
plt.ylabel("Range", fontweight = 'bold')
plt.xlabel("Attributes", fontweight = 'bold')
```

Figure 1.16
Analyzing Amount and Frequency for outlier using boxplot

```
fm.shape
```

```
(4372, 3)
```

Figure 1.17
Shape of the dataset before removing outliers

```
# Removing (statistical) outliers for Spend_Amount
Q1 = fm.Spend_Amount.quantile(0.25)
Q3 = fm.Spend_Amount.quantile(0.75)
IQR = Q3 - Q1
fm = fm[(fm.Spend_Amount >= Q1 - 1.5*IQR) & (fm.Spend_Amount <= Q3 + 1.5*IQR)]


# Removing (statistical) outliers for Frequency
Q1 = fm.Frequency.quantile(0.25)
Q3 = fm.Frequency.quantile(0.75)
IQR = Q3 - Q1
fm = fm[(fm.Frequency >= Q1 - 1.5*IQR) & (fm.Frequency <= Q3 + 1.5*IQR)]
```

Figure 1.18
Removing outliers from fm dataset

Write the code in Figure 1.18. Step by step code explanation is given below:

1. **Q1** and **Q3**: These lines calculate the first quartile (**Q1**) and third quartile (**Q3**) for the variable **Spend_Amount**. These values are used to define the interquartile range (**IQR**).

2. **IQR = Q3 - Q1**: This line calculates the interquartile range.

3. **fm = fm[(fm.Spend_Amount >= Q1 - 1.5*IQR) & (fm.Spend_Amount <= Q3 + 1.5*IQR)]**: This line filters the DataFrame **fm** to keep only the rows where the **Spend_Amount** falls within a range defined by the lower bound (Q1 - 1.5*IQR) and the upper bound (Q3 + 1.5*IQR). This effectively removes potential outliers from the dataset.

After removing the outliers, we plot the dataset using boxplot to visualize it again. See Figure 1.18 and 1.19.

```
attributes = ['Spend_Amount','Frequency']
plt.rcParams['figure.figsize'] = [7,7]
sns.boxplot(data = fm[attributes], orient="X", palette="Set2" ,whis=1.5,saturation=1, width=0.7)
plt.title("Outliers Variable Distribution", fontsize = 14, fontweight = 'bold')
plt.ylabel("Range", fontweight = 'bold')
plt.xlabel("Attributes", fontweight = 'bold')
```

Figure 1.18
Plotting the fm dataset using boxplot after removing outliers

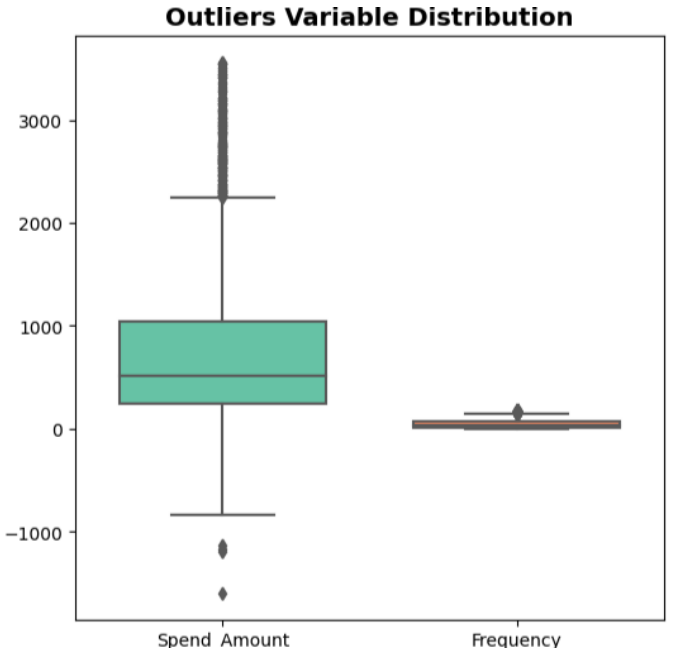


Figure 1.19
Boxplot view of the fm dataset

As you can see in Figure 1.20, after removing outliers, the rows of fm dataset has been reduced from 4371 to 3698 (excluding the row features)

```
fm.shape
```

```
(3697, 3)
```

Figure 1.20
Shape of the fm dataset after removing outliers

1.7.7.5 Feature Scaling for K-Means Clustering

Distance-based algorithms suffer greatly from data that is not on the same scale. The scale of the points may distort the real distance between values. To perform Feature Scaling, ***Scikit-Learn's StandardScaler class*** will be applied. As you can see in Figure 1.21, the standard deviation in Spend_Amount is 740 and in Frequency is 41. So, that's why we have to scale the data to improve our accuracy.

```
fm.describe()
```

	Spend_Amount	Frequency
count	3697.000000	3697.000000
mean	771.363098	46.658101
std	740.646223	41.705211
min	-1592.490000	1.000000
25%	248.100000	15.000000
50%	508.460000	32.000000
75%	1046.560000	68.000000
max	3563.850000	178.000000

Figure 1.21
Checking the data statistics before applying Scaling

Write the code in Figure 1.22(a). First of all, we copied the **fm** dataset's **Spend_Amount** and **Frequency** feature in **fm_df**. Then, we applied scaling on **fm_df** and put the resulting data in **fm_df_scaled**.

```
fm_df = fm[['Spend_Amount', 'Frequency']]

# Instantiate
scaler = StandardScaler()

# fit_transform
fm_df_scaled = scaler.fit_transform(fm_df)
fm_df_scaled.shape
```

```
(3697, 2)
```

Figure 1.22(a)
Scaling of datasets for applying k-means clustering

As you can see in Figure 1.22(b), the data is in array form after scaling. So, need to convert it to dataframe.

```
fm_df_scaled
```

```
array([[-1.04161396, -1.07094871],
       [ 1.38529792, -0.3754979 ],
       [ 1.33170234,  0.63170672],
       ...,
       [-0.93247827, -0.9510434 ],
       [-0.80314128, -0.80715703],
       [ 1.43936613,  0.55976353]])
```

Figure 1.22(b)

Dataset in array form after scaling

Write the code in Figure 1.23(a) to convert it to a dataframe.

```
fm_df_scaled = pd.DataFrame(fm_df_scaled)
fm_df_scaled.columns = ['Amount', 'Frequency']
fm_df_scaled.head()
```

	Amount	Frequency
0	-1.041614	-1.070949
1	1.385298	-0.375498
2	1.331702	0.631707
3	-0.590055	-0.711233
4	1.045238	1.159290

Figure 1.23(a)

Organizing the scaled data into DataFrame again

In figure 1.23(b), it's seen that, the standard deviation has been reduced for both the feature columns.

```
fm_df_scaled.describe()
```

	Amount	Frequency
count	3.697000e+03	3.697000e+03
mean	3.843888e-17	-1.345361e-17
std	1.000135e+00	1.000135e+00
min	-3.192041e+00	-1.094930e+00
25%	-7.065909e-01	-7.591949e-01
50%	-3.550125e-01	-3.515168e-01
75%	3.716135e-01	5.118014e-01
max	3.770848e+00	3.149718e+00

Figure 1.23(b)

Checking the statistical properties after scaling

1.7.8 Developing K-Means Clustering Model for our Dataset

1.7.8.1 Finding the best value of K for our dataset

In k-means, it is essential to provide the numbers of the cluster in the algorithm. When we do not know the number of the cluster, we have to use methods such as **Elbow method or Silhouette methods** for finding the optimal number of clusters in the dataset. Here, I will use both Elbow and Silhouette method to find the optimal number of clusters. In figure 1.24, we are trying to find the best value of **k.** Here is a step by step explanation.

1. **track_inertia = {}** and **silhouette = {}**: These lines initialize empty dictionaries **track_inertia** and **silhouette** to store the Inertia and Silhouette Score values for different values of **k**.

2. **kmeans = KMeans(n_clusters=k, init = 'k-means++', random_state=0)**: This line initializes a KMeans model with **k** clusters. **init = 'k-means++'** indicates that the initialization method for centroids is "smart" initialization, which can lead to faster convergence. **Random_state = 0**, set this to any int (generally 0 or 42) to get the same output when you run it multiple times. When there is a randomization process involved, you should use random_state to create reproducible output. If you don't use this parameter, you may get different output.

```
track_inertia = {}
silhouette = {}
for k in range(2,11):
  kmeans = KMeans(n_clusters=k, init = 'k-means++', random_state=0)
  kmeans.fit(fm_df_scaled)
  track_inertia[k] = kmeans.inertia_
  silhouette[k]=silhouette_score(fm_df_scaled,kmeans.labels_)
```

Figure 1.24

Script to calculate and plot WCSS for all possible K values

3. **kmeans.fit(fm_df_scaled)**: This fits the KMeans model to the scaled data **fm_df_scaled**.

4. **track_inertia[k] = kmeans.inertia_**: This stores the Inertia (sum of squared distances of samples to their closest cluster center) for the current **k** value in the **track_inertia** dictionary. Inertia gives within-cluster sum of squares. This is a total of the within-cluster sum of squares for all clusters.

5. **silhouette[k] = silhouette_score(fm_df_scaled, kmeans.labels_)**: This computes the Silhouette Score (a measure of how similar an object is to its own cluster compared to other clusters) for the current **k** value and stores it in the **silhouette** dictionary.

After running this loop, you will have dictionaries **track_inertia** and **silhouette** containing Inertia and Silhouette Score values for different numbers of clusters, which can help you decide the optimal number of clusters for your data.

1.7.8.2 Plotting K vs WCSS and K vs Silhouette score for finding best k

In Figure 1.25, we plot K vs WCSS(Intertia) using pointplot. The Elbow Method involves plotting **k** against the corresponding Inertia values and looking for the "elbow" point, where the Inertia starts to decrease at a slower rate. We can take k = 3 from the Figure 1.25.

Keep in mind that the choice of **k** is subjective and may require domain knowledge or further analysis to determine the most suitable number of clusters for your specific dataset.

```
sns.pointplot(x = list(track_inertia.keys()), y = list(track_inertia.values()))
```

```
<Axes: >
```

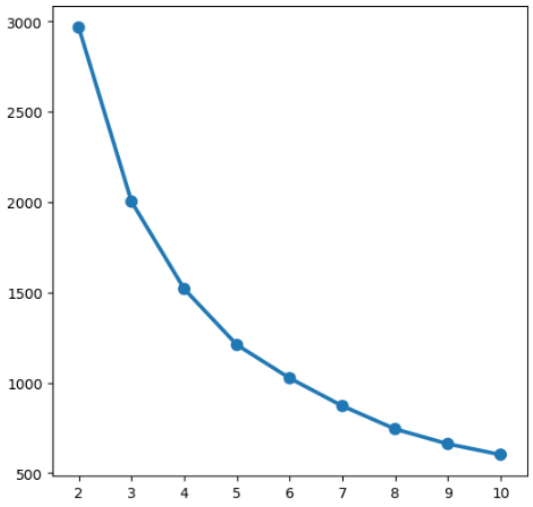


Figure 1.25

Plotting the K vs WCSS score to find the elbow point

```
sns.pointplot(x = list(silhouette.keys()), y = list(silhouette.values()))
```

```
<Axes: >
```

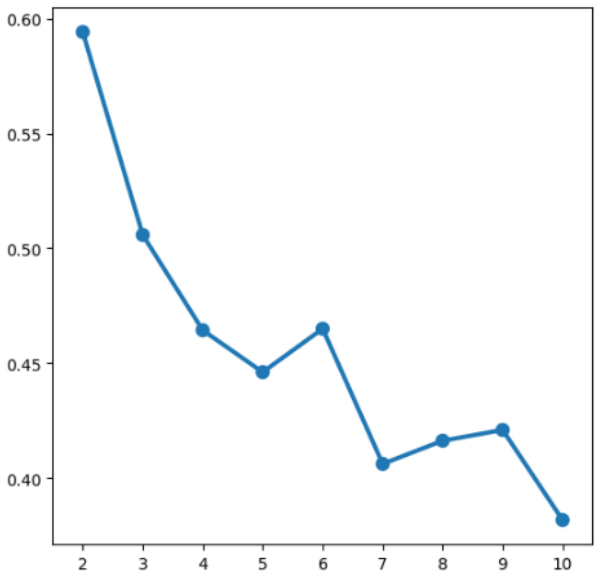


Figure 1.26

Plotting the K vs Silhouette score to find the best value of K

The Silhouette Score can be used to further validate the quality of the clusters. In Figure 1.26, we plot k vs silhouette scores to further validate the best value for k. Silhouette score ranges from -1 to 1.

1. **Positive Values (Closer to +1):**

    - A score closer to +1 suggests that the sample is far away from the neighboring clusters. This indicates that the clustering assignment is appropriate.

2. **Negative Values (Closer to -1):**

    - A score closer to -1 suggests that those data points have been assigned to the wrong cluster.

We will take k = 3 as well by observing silhouette score.

1.7.8.3 Fitting the model with the best value of k

In Figure 1.27, we train our fm_df_scaled dataset with the KMeans algorithm using k = 3 and for this, WCSS = 2002 and Silhouette score is 0.505866.

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, init = 'k-means++', max_iter = 300, n_init = 10, random_state=0 )

kmeans.fit(fm_df_scaled)

print("WCSS for k: %d and silhouette score: %f" %(kmeans.inertia_,silhouette_score(fm_df_scaled,kmeans.labels_)))
```

```
WCSS for k: 2002 and silhouette score: 0.505866
```

Figure 1.27

Training the fm_df_scaled dataframe for K = 5

The cluster_centers positions are shown in Figure 1.28.

```
kmeans.cluster_centers_
```

```
array([[ 0.32807792,  0.61837614],
       [ 2.0845487 ,  1.7117625 ],
       [-0.57393175, -0.63311871]])
```

Figure 1.28

3 Cluster Centers location

We are assigning the cluster ID or labels with each row by adding the Cluster_Id column in fm dataset as shown in Figure 1.29.

```
fm['Cluster_Id'] = kmeans.labels_
fm.head()
```

	CustomerID	Spend_Amount	Frequency	Cluster_Id
0	12346.0	0.00	2	2
2	12348.0	1797.24	31	0
3	12349.0	1757.55	73	0
4	12350.0	334.40	17	2
5	12352.0	1545.41	95	0

Figure 1.29

Adding the labels/cluster value column in fm dataset

1.7.8.4 Visualizing the clusters of our dataset

We will be plotting our dataset using the code given in Figure 1.30. We have used scatterplot for this. The output is given in Figure 1.31.

```
plt.figure(figsize=(7,7))
plt.scatter(fm_df_scaled["Amount"],fm_df_scaled["Frequency"], c=kmeans.labels_, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1],c='blue',s=200, marker = 'x')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.title('Scatter Plot of Amount vs Frequency')
plt.show()
```

Figure 1.30

Plotting Spend_Amount vs Frequency to visualize the clusters

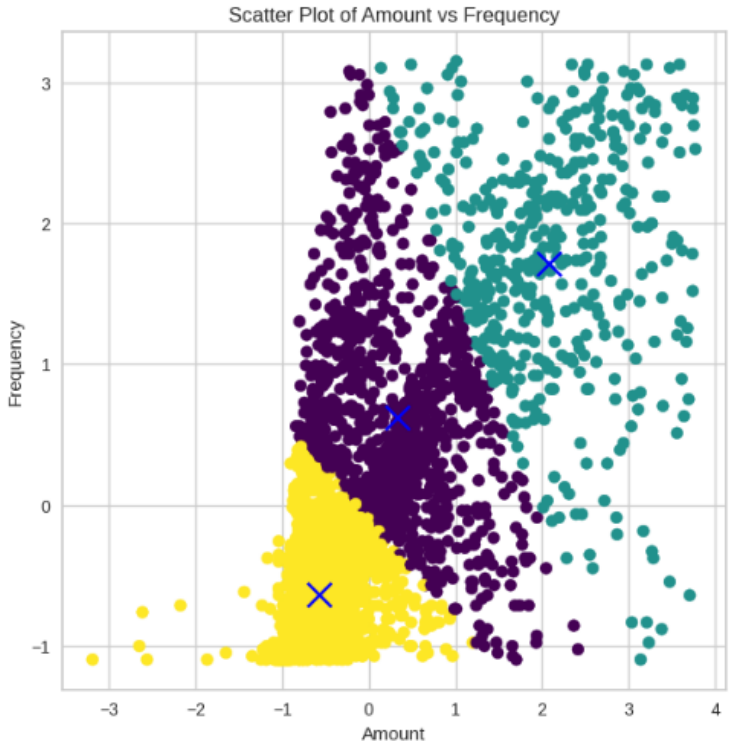


Figure 1.31

Plotting Spend_Amount vs Frequency to visualize the clusters using scatterplot

```
sns.relplot(x='Spend_Amount', y='Frequency', data=fm, hue=fm['Cluster_Id'], height=5);
```

Figure 1.32

Plotting Spend_Amount vs Frequency to visualize the clusters using relplot

You can use Figure 1.32 as well for visualization.

1.7.8.5 Predicting new data point cluster level

Use Figure 1.33 code to predict a new data point cluster level. Here you can see, a new_data_point is a customer whose spend_amount is 390.5 and his frequency (number of transactions) is 23. According to our model, he is in Cluster 2.

```
new_data_point = [390.5, 23]
new_data_point_scaled = scaler.transform([new_data_point])
predicted_cluster = kmeans.predict(new_data_point_scaled)
print(f"The new data point belongs to cluster {predicted_cluster[0]}")
```

```
The new data point belongs to cluster 2
```

Figure 1.33

Predicting new data point cluster level using our built model