E-COMMERCE CUSTOMER SEGMENTATION USING RFM ANALYSIS

IE6400 FOUNDATIONS DATA ANALYTICS ENGINEERING

FALL 2023

Project Report

Group Number 5:

Pooja Laxmi Sankarakameswaran(002278676)
Rishwanth Reddy Yadamakanti(002697773)
Sruthi Gandla(0028515998)
Tejesvani Muppara Vijayaram(002299364)
Naresh Gajula(002648708)

INTRODUCTION

Businesses must comprehend consumer behaviours in the cutthroat world of eCommerce to create winning marketing campaigns and improve client retention. Based on three important metrics—recentness (the amount of time since a customer made a purchase), frequency (the frequency with which a customer makes a purchase), and monetary value (the total monetary value of a customer's purchases) — RFM analysis, a customer segmentation technique, offers insightful information about customer purchasing patterns.

Businesses may better target their marketing efforts at client groups and increase customer engagement, retention, and income by segmenting their customer base according to their RFM ratings.

1. DATA PREPROCESSING

In the context of data pre-processing, the importation and initial processing of a dataset are crucial steps that lay the foundation for any subsequent analysis or modelling. This phase involves several key tasks to ensure that the data is in a suitable and reliable format for further exploration. By systematically performing these data pre-processing steps, the dataset is refined and prepared for further analysis, ultimately contributing to the reliability and accuracy of any subsequent modelling or exploration. This process is fundamental to ensuring the quality of the data used in decision-making processes and generating meaningful insights from the available information.

Data Import and Overview:

In the initial phase, we loaded the dataset using the pandas library in Python with the 'pd.*filepath*()' function, which allowed us to read the contents of the file into a DataFrame named 'customer_df.' The 'head ()' function provided a glimpse of the first few rows, aiding in understanding the dataset's structure.

```
import pandas as pd

customer_df = pd.read_csv('archive/data.csv')
customer_df.head()
```

Result: -

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/2010 8:26	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/2010 8:26	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/2010 8:26	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/2010 8:26	3.39	17850.0	United Kingdom

Subsequently, we displayed the number of rows and columns in the dataset using 'shape'.

```
# Displaying the number of Rows and Columns

print("The Number of Rows in the Dataset", customer_df.shape[0])
print("The Number of Columns in the Dataset", customer_df.shape[1])
```

Result: -

```
The Number of Rows in the Dataset 541909
The Number of Columns in the Dataset 8
```

Dataset Summary and Null Value Inspection:
We obtained a statistical summary of the dataset using the 'describe ()' function, offering insights into the central tendencies and spread of numeric columns.

```
# Dataset Summary
customer_df.describe()
```

Result: -

	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

Furthermore, we checked for the presence of null values using the 'isnull().any()' method, identifying 'Description' and 'CustomerID' as columns with missing data.
provided a summary using **describe()**.

Code & Output: -

```
# Checking for Null Values
customer_df.isnull().any()
```

```
InvoiceNo      False
StockCode      False
Description     True
Quantity       False
InvoiceDate    False
UnitPrice      False
CustomerID      True
Country        False
dtype: bool
```

To address missing values, we replaced null entries in the 'Description' column with 'Description N/A' and those in the 'CustomerID' column with 0.0. This ensures a more complete dataset, mitigating potential issues in subsequent analyses that involve these columns.

```
# Null value handling

# From the Description above we can see that Description and CustomerID contain null values

customer_df['Description'] = customer_df['Description'].fillna('Description N/A')
customer_df['CustomerID'] = customer_df['CustomerID'].fillna(0.0)
```

Data Type Conversion and Timestamp Formatting:

For optimal analysis, we performed data type handling. 'CustomerID' was converted to integers for consistency, and 'InvoiceDate' was transformed into datetime format using 'astype(int)' and 'pd.to_datetime()' functions, respectively. This ensures consistency in data representation and facilitates time-based analyses.

```
# Datatype Handling
from datetime import datetime as dt

customer_df['CustomerID'] = customer_df['CustomerID'].astype(int)
customer_df['InvoiceDate'] = pd.to_datetime(customer_df['InvoiceDate'])
customer_df
```

Result: -

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
...	...	...	...	...	...	...	...	...
541904	581587	22613	PACK OF 20 SPACEBOY NAPKINS	12	2011-12-09 12:50:00	0.85	12680	France
541905	581587	22899	CHILDREN'S APRON DOLLY GIRL	6	2011-12-09 12:50:00	2.10	12680	France
541906	581587	23254	CHILDRENS CUTLERY DOLLY GIRL	4	2011-12-09 12:50:00	4.15	12680	France
541907	581587	23255	CHILDRENS CUTLERY CIRCUS PARADE	4	2011-12-09 12:50:00	4.15	12680	France
541908	581587	22138	BAKING SET 9 PIECE RETROSPOT	3	2011-12-09 12:50:00	4.95	12680	France

541909 rows × 8 columns

Dataset Information and Time Range:

The 'info()' method provided a comprehensive overview of the dataset, including data types and memory usage. This knowledge is pivotal for understanding the dataset's structure and identifying potential issues early in the analysis process.

Code & Result: -

```
# Dataset Description

customer_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  541909 non-null  object
 3   Quantity     541909 non-null  int64
 4   InvoiceDate  541909 non-null  datetime64[ns]
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   541909 non-null  int64
 7   Country      541909 non-null  object
dtypes: datetime64[ns](1), float64(1), int64(2), object(4)
memory usage: 33.1+ MB
```

Determining the time range, spanning from the earliest to the latest invoice dates, offers context for the temporal aspects of the data, guiding the selection of appropriate time-sensitive analytical approaches.

Code & Result: -

```
# Invoice Date Limits

print("The Invoice Data lies between (MM/DD/YYYY HH:MM) ", customer_df.InvoiceDate.min(),\
      " and ", customer_df.InvoiceDate.max())
```

```
The Invoice Data lies between (MM/DD/YYYY HH:MM)  2010-12-01 08:26:00  and  2011-12-09 12:50:00
```

2. RFM CALCULATION

These calculations are integral components of the RFM (Recency, Frequency, Monetary) framework, a widely used method in customer segmentation and targeting. The derived metrics provide valuable insights into customer behaviour, helping businesses tailor marketing and engagement strategies to different customer segments based on their recency, frequency, and monetary contributions.

Recency Calculation: -

How recently did a customer make a purchase. Calculate the number of days since the customer's last purchase.

Calculating Days Since Last Purchase:

To determine the recency of customer purchases, we first identified the most recent date in the dataset using the 'max()' function on the 'InvoiceDate' column. Subsequently, the 'DaysSinceLastPurchase' column was created by calculating the difference in days between the most recent date and each customer's last purchase date. This was achieved by grouping the data by 'CustomerID' and applying the 'transform' function with 'max()' to obtain the maximum purchase date for each customer. The resulting 'recency_df' DataFrame captures the minimum days since the last purchase for each customer, providing a key metric for understanding the recency of their engagement with the business.

Code & Result: -

```
# Calculating the most recent date in the dataset
most_recent_date = customer_df['InvoiceDate'].max()

# Calculating the difference between the order date and the most recent date
customer_df['DaysSinceLastPurchase'] = \
(most_recent_date - customer_df.groupby('CustomerID')['InvoiceDate'].transform('max')).dt.days

recency_df = customer_df.groupby('CustomerID')['DaysSinceLastPurchase'].min().reset_index()

recency_df.head()
```

	CustomerID	DaysSinceLastPurchase
0	0	0
1	12346	325
2	12347	1
3	12348	74
4	12349	18

Frequency (F) Calculation:

- **Grouping by CustomerID for Transaction Frequency:**

The 'frequency_df' dataframe was then constructed to encapsulate the frequency metric. It consists of 'CustomerID' and the corresponding 'Frequency' column, storing the calculated unique count of orders for each customer. The dataset was grouped by 'CustomerID,' forming distinct groups for each customer.

```
frequency_df = customer_df.groupby('CustomerID')['InvoiceNo'].nunique().reset_index()
frequency_df.columns = ['CustomerID', 'Frequency']

frequency_df.head()
```

Result: -

	CustomerID	Frequency
0	0	3710
1	12346	2
2	12347	7
3	12348	4
4	12349	1

Monetary (M) Calculation:

- **Calculating Total Monetary Value per Transaction:**

The 'TotalPrice' for each transaction was computed by multiplying the 'Quantity' and 'UnitPrice' columns. This step quantifies the monetary value associated with each customer's individual transactions, providing a basis for assessing their overall spending behaviour. To capture the cumulative spending behaviour of each customer, the dataset was grouped by 'CustomerID,' and the 'TotalPrice' values were summed. This aggregation resulted in the 'Monetary' column in the 'monetary_df' dataframe, reflecting the total monetary contribution of each customer to the business. This metric is crucial for understanding the financial impact of individual customers and tailoring marketing strategies accordingly.

Code & Result: -

```
# Calculating the Monetary value by getting the product of each UnitPrice and Quantity mentioned in an order

customer_df['TotalPrice'] = customer_df['Quantity'] * customer_df['UnitPrice']
monetary_df = customer_df.groupby('CustomerID')['TotalPrice'].sum().reset_index()
monetary_df.columns = ['CustomerID', 'Monetary']

monetary_df.head()
```

	CustomerID	Monetary
0	0	1447682.12
1	12346	0.00
2	12347	4310.00
3	12348	1797.24
4	12349	1757.55

3. RFM SEGMENTATION

Calculating R_Score, F_Score and M_Score for every Customer. These scores are then combined to get the RFM Score for each customer.

Assigning Recency (R) Scores:
To evaluate and categorize the recency of customer engagement, the 'recency_df' DataFrame was sorted in ascending order based on the 'DaysSinceLastPurchase' column, ensuring the customers with the most recent purchases appear first. Recency scores ('R_Score') were then assigned using quartiles, achieved through the 'pd.qcut' function. This function divides the recency values into quartiles, and the resulting labels represent the recency scores. The 'fillna' method was used to replace any potential NaN values with 0.0, ensuring consistency. The 'recency_df' now includes 'R_Score,' offering insights into the recency of each customer's engagement, with lower scores indicating more recent purchases.

Code & Result: -

```
# Assigning R, F and M scores for each customer from their respective Data Frames

recency_df.sort_values(by= ['DaysSinceLastPurchase'], ascending = True, inplace = True)
recency_df['R_Score'] = pd.qcut(recency_df['DaysSinceLastPurchase'], q=[0, 0.25, 0.5, 0.75, 1], labels=False,\
                                duplicates = 'drop') + 1
recency_df['R_Score'].fillna(0.0, inplace = True)

recency_df
```

	CustomerID	DaysSinceLastPurchase	R_Score
0	0	0	1
4126	17949	0	1
1128	13860	0	1
3114	16558	0	1
3163	16626	0	1
...	...	...	...
360	12791	373	4
4213	18074	373	4
1047	13747	373	4
1765	14729	373	4
4097	17908	373	4

Assigning Frequency (F) Scores:
For frequency scores, the 'frequency_df' data frame was sorted in descending order based on 'Frequency.' Similar to the recency calculation, quartiles were applied to assign scores ('F_Score') to each customer using 'pd.qcut()'. The resulting column was then added to the 'frequency_df,' reflecting the frequency score for each customer.

Code & Result: -

```
frequency_df.sort_values(by= ['Frequency'], ascending = False, inplace = True)
frequency_df['F_Score'] = pd.qcut(frequency_df['Frequency'], q=[0, 0.25, 0.5, 0.75, 1], labels=False, duplicates='dr

frequency_df
```

	CustomerID	Frequency	F_Score
0	0	3710	3
1896	14911	248	3
331	12748	224	3
4043	17841	169	3
1675	14606	128	3
...	...	...	...
3002	16404	1	1
1142	13877	1	1
2998	16400	1	1
1143	13878	1	1
2186	15300	1	1

4373 rows × 3 columns

Assigning Monetary (M) Scores:

The 'monetary_df' dataframe was sorted in descending order based on 'Monetary' values. Quartiles were applied to assign monetary scores ('M_Score') to each customer using 'pd.qcut()'. This score column was added to the 'monetary_df,' capturing the monetary score for each customer.

Code & Result: -

```
monetary_df.sort_values(by= ['Monetary'], ascending = False, inplace = True)
monetary_df['M_Score'] = pd.qcut(monetary_df['Monetary'], q=[0, 0.25, 0.5, 0.75, 1], labels=False, duplicates='drop'

monetary_df
```

	CustomerID	Monetary	M_Score
0	0	1447682.12	4
1704	14646	279489.02	4
4234	18102	256438.49	4
3759	17450	187482.17	4
1896	14911	132572.62	4
...	...	...	...
126	12503	-1126.00	1
3871	17603	-1165.30	1
1385	14213	-1192.20	1
2237	15369	-1592.49	1
3757	17448	-4287.63	1

4373 rows × 3 columns

Combining R, F, and M Scores:

To create a comprehensive RFM (Recency, Frequency, Monetary) score for each customer, the individual scores for recency ('R_Score'), frequency ('F_Score'), and monetary value ('M_Score') were merged into the 'rfm_scores' DataFrame. This merging was performed based on the common 'CustomerID' column. The final RFM score was then calculated using the specified formula: RFM_Score =

R_Score * 100 + F_Score * 10 + M_Score. This amalgamation results in a unique numerical identifier for each customer, encapsulating their recency, frequency, and monetary behavior. The 'rfm_scores' DataFrame provides a consolidated view that simplifies customer segmentation and analysis.

Code & Result: -

```
# Combining the R, F and M scores to create a single score
# The formula used to combine these scores : R * 100 + F * 10 + M

rfm_scores = pd.merge(recency_df[['CustomerID', 'R_Score']], frequency_df[['CustomerID', 'F_Score']], on='CustomerID
rfm_scores = pd.merge(rfm_scores, monetary_df[['CustomerID', 'M_Score']], on='CustomerID')

rfm_scores['RFM_Score'] = rfm_scores['R_Score'] * 100 + rfm_scores['F_Score'] * 10 + rfm_scores['M_Score']
rfm_scores
```

	CustomerID	R_Score	F_Score	M_Score	RFM_Score
0	0	1	3	4	134
1	17949	1	3	4	134
2	13860	1	3	3	133
3	16558	1	3	4	134
4	16626	1	3	4	134
...	...	...	...	...	...
4368	12791	4	1	1	411
4369	18074	4	1	2	412
4370	13747	4	1	1	411
4371	14729	4	1	2	412
4372	17908	4	1	1	411

Segmentation and Analysis:

With the RFM scores in place, businesses can now segment their customer base more effectively, tailoring marketing strategies based on specific customer behaviours. Higher RFM scores indicate more valuable customers, while lower scores may highlight opportunities for targeted interventions to improve customer engagement and loyalty. This segmentation approach enhances the precision of customer relationship management and contributes to overall business growth.

4. CUSTOMER SEGMENTATION

RFM Clustering using K-Means:

To prepare the RFM (Recency, Frequency, Monetary) scores for clustering analysis, the relevant columns ('R_Score', 'F_Score', 'M_Score') were extracted from the 'rfm_scores' DataFrame. Subsequently, the data was standardized using the 'StandardScaler' from the scikit-learn library. Standardization ensures that each RFM component has a mean of 0 and a standard deviation of 1, preventing any single component from dominating the clustering process due to its scale. The resulting 'rfm_scaled' array is now ready for clustering algorithms, contributing to more meaningful and balanced cluster formation based on recency, frequency, and monetary metrics.

```
from sklearn.preprocessing import StandardScaler

# Extract RFM scores for clustering
rfm_data = rfm_scores[['R_Score', 'F_Score', 'M_Score']]

# Scale the data
scaler = StandardScaler()
rfm_scaled = scaler.fit_transform(rfm_data)
rfm_scaled
```

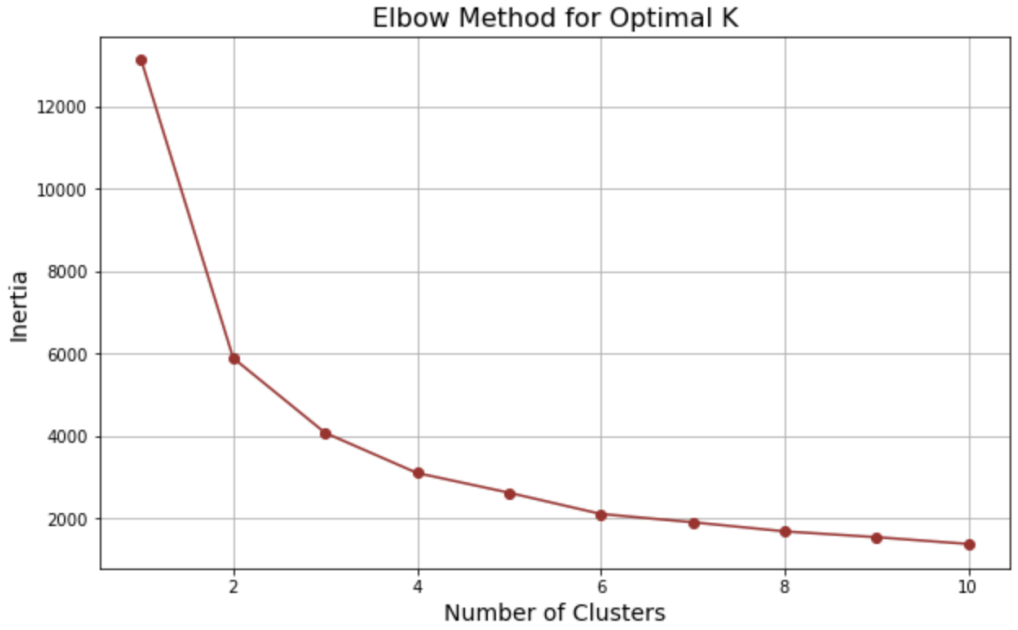
```
array([[-1.32069435,  1.58566721,  1.34182492],
       [-1.32069435,  1.58566721,  1.34182492],
       [-1.32069435,  1.58566721,  0.44747949],
       ...,
       [ 1.34651458, -0.76289256, -1.34121137],
       [ 1.34651458, -0.76289256, -0.44686594],
       [ 1.34651458, -0.76289256, -1.34121137]])
```

Determining Optimal Number of Clusters Using Elbow Method:

The Elbow Method was employed to identify the optimal number of clusters for the K-Means clustering algorithm. The process involved iterating through different numbers of clusters (k) and fitting K-Means models to the standardized RFM scores ('rfm_scaled'). The sum of squared distances (inertia) from each point to its assigned cluster center was recorded for each k. The resulting elbow graph depicts the relationship between the number of clusters and inertia. The "elbow" in the graph, where the inertia starts to plateau, suggests the optimal k. In this scenario, the graph was examined to find the point where further cluster increase provides diminishing returns in reducing inertia. This aids in selecting a balance between granularity and meaningful cluster formation for subsequent analysis.

```
# Finding the optimal number of clusters

from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Trying different numbers of clusters
k_values = range(1, 11)
inertias = []

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(rfm_scaled)
    inertias.append(kmeans.inertia_)

# Plotting the elbow graph to observe which is the optimal number of clusters

plt.figure(figsize = (10,6))
plt.plot(k_values, inertias, marker='o', color = 'brown')
plt.xlabel('Number of Clusters', fontsize = 14)
plt.ylabel('Inertia', fontsize = 14)
plt.title('Elbow Method for Optimal K', fontsize = 16)
plt.grid(True)
plt.show()
```

Graph: -



Observation: -

The elbow point is at k=3, which means that the optimal number of clusters is 3. This is because the rate of decrease of inertia slows down significantly after k=3, indicating that adding more clusters is not provide much additional benefit. In other words, the data can be best represented by grouping it into 3 clusters. The inertia is a measure of how well the data points fit within their assigned clusters, so a lower inertia indicates a better fit.

Here is a more detailed interpretation of the plot:

- For k=1, all of the data points are assigned to the same cluster. This is the worst possible fit since there is clearly more than one group of data points.

- As k increases, the inertia decreases, indicating that the data points are being assigned to more appropriate clusters.
- However, the rate of decrease of inertia slows down significantly after k=3. This means that adding more clusters is not providing much additional benefit. In fact, it may even be harmful, as it can lead to overfitting.

Therefore, the optimal number of clusters is 3.

From the Elbow Graph, we notice that a Cluster size of 3 or 4 might be optimal. The Silhouette Score is used to see which cluster size is optimal. A higher Silhouette Score suggests that the cluster size is optimal.

Determining Optimal Number of Clusters Using Silhouette Score:
The Silhouette Score, a metric assessing how well-separated clusters are, was employed to determine the optimal number of clusters. Two K-Means models were created with cluster sizes of 3 and 4, respectively, using the standardized RFM scores ('rfm_scaled'). The silhouette scores for both models were calculated, and the model with the higher silhouette score was identified as the optimal choice. The selected model provides a better balance between cohesion within clusters and separation between clusters. This approach ensures that the clusters are well-defined and internally homogeneous while also being distinct from each other.

Code & Result: -

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# For Cluster Size 3
kmeans_1 = KMeans(n_clusters=3, random_state=0)
labels_1 = kmeans_1.fit_predict(rfm_scaled)
silhouette_1 = silhouette_score(rfm_scaled, labels_1)

# For Cluster Size 4
kmeans_2 = KMeans(n_clusters=4, random_state=0)
labels_2 = kmeans_2.fit_predict(rfm_scaled)
silhouette_2 = silhouette_score(rfm_scaled, labels_2)

if(silhouette_1 > silhouette_2):
    print("Cluster Size 3 is optimal since ", round(silhouette_1, 2), " is greater than ", round(silhouette_2, 2))
else:
    print("Cluster Size 4 is optimal since ", round(silhouette_2, 2), " is greater than ", round(silhouette_1, 2))
```

```
Cluster Size 4 is optimal since  0.47  is greater than  0.45
```

K-Means Clustering with Optimal Number of Clusters (k=4):
The K-Means clustering algorithm was applied to the standardized RFM scores ('rfm_scaled') with an optimal number of clusters determined to be 4. The algorithm assigns each customer to one of the four clusters based on their recency, frequency, and monetary scores. The 'Cluster' column in the 'rfm_scores' DataFrame indicates the cluster assignment for each customer, allowing for further analysis and segmentation based on similar RFM characteristics. This segmentation aids in understanding customer behavior and tailoring business strategies accordingly.

Code & Result: -

```
optimal_k = 4

# Apply K-Means clustering
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
rfm_scores['Cluster'] = kmeans.fit_predict(rfm_scaled)
rfm_scores
```

	CustomerID	R_Score	F_Score	M_Score	RFM_Score	Cluster
0	0	1	3	4	134	0
1	17949	1	3	4	134	0
2	13860	1	3	3	133	0
3	16558	1	3	4	134	0
4	16626	1	3	4	134	0
...	...	...	...	...	...	...
4368	12791	4	1	1	411	1
4369	18074	4	1	2	412	1
4370	13747	4	1	1	411	1
4371	14729	4	1	2	412	1
4372	17908	4	1	1	411	1

4373 rows × 6 columns

Distribution of Customers Across Clusters:

The distribution of customers across the four clusters resulting from the K-Means clustering. These represent the number of customers assigned to each cluster, providing insights into the size and composition of the customer segments. Analyzing the characteristics of customers within each cluster can guide targeted marketing strategies and personalized approaches based on their distinct recency, frequency, and monetary profiles.

Code & Result: -

```
rfm_scores['Cluster'].value_counts()
```

```
1    1422
0    1269
2     896
3     786
Name: Cluster, dtype: int64
```

Average Recency Score Across Clusters:

This bar plot illustrates the average recency scores across different clusters generated by the K-Means clustering algorithm. The x-axis represents the clusters (0 to 3), while the y-axis denotes the average recency score within each cluster. The varying heights of the bars indicate how recently, on average, customers from each cluster made their last purchase. Analyzing these scores provides insights into the recency behavior of customers within each cluster, helping to understand and target specific segments based on their recency patterns.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Extract relevant columns for visualization
recency_data = rfm_scores[['Cluster', 'R_Score']]

#Create a bar plot of average recency scores across clusters

plt.figure(figsize=(10, 6))
sns.barplot(x='Cluster', y='R_Score', data=recency_data, ci=None, palette="Set2")
plt.title('Average Recency Score Across Clusters', fontsize = 16)
plt.xlabel('Cluster', fontsize = 14)
plt.ylabel('Average Recency Score', fontsize = 14)
plt.show()
```

Graph: -

Observations: -

The bar plot shows the average recency score across the three clusters. Cluster 3 has the lowest average recency score, followed by Cluster 2 and Cluster 1. This means that customers in Cluster 3 have purchased most recently, followed by customers in Cluster 2 and Cluster 1.

The plot also shows that there is a significant difference in average recency score between the clusters. This suggests that the clustering algorithm has been able to successfully identify different groups of customers with different purchasing behaviours.

Here are some possible interpretations of the plot:

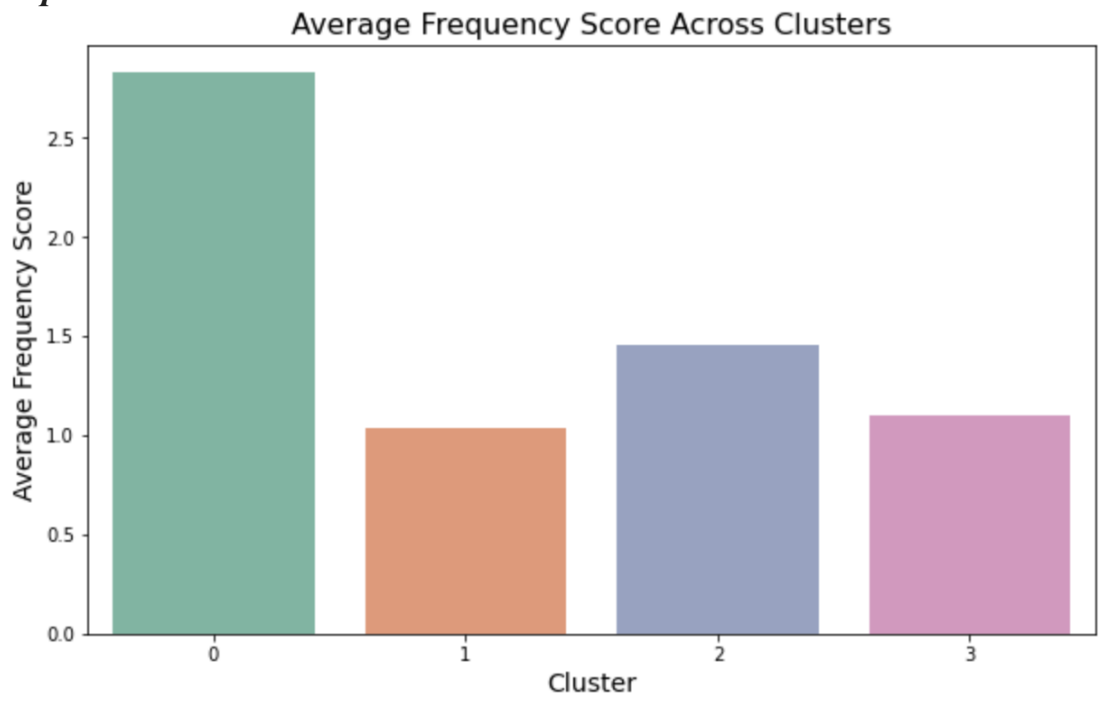
- Cluster 1 may represent customers who have not purchased recently and are at risk of churning.
- Cluster 2 may represent customers who have purchased recently but are not as engaged with the brand as Cluster 3 customers.

- Cluster 3 may represent customers who have purchased very recently and are the most engaged with the brand.

Average Frequency Score Across Clusters:
The average frequency scores across different clusters were identified through the K-Means clustering algorithm. Each bar corresponds to a specific cluster (0 to 3) on the x-axis, while the y-axis represents the average frequency score within each cluster. The varying heights of the bars provide insights into the purchase frequency patterns of customers in each segment. Understanding these scores helps in tailoring marketing and engagement strategies to target customer segments with different buying behavior.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Extract relevant columns for visualization
frequency_data = rfm_scores[['Cluster', 'F_Score']]

# Create a bar plot of average recency scores across clusters
plt.figure(figsize=(10, 6))
sns.barplot(x='Cluster', y='F_Score', data=frequency_data, ci=None, palette="Set2")
plt.title('Average Frequency Score Across Clusters', fontsize = 16)
plt.xlabel('Cluster', fontsize = 14)
plt.ylabel('Average Frequency Score', fontsize = 14)
plt.show()
```

Graph: -



Observations: -
The plot also shows that there is a significant difference in average frequency score between the clusters. This suggests that the clustering algorithm has been able to successfully identify different groups of customers with different purchasing behaviours.

- Cluster 1 may represent customers who are highly engaged with the brand and purchase products on a regular basis.
- Cluster 2 may represent customers who are less engaged with the brand but still purchase products on a semi-regular basis.
- Cluster 3 may represent customers who are least engaged with the brand and purchase products infrequently.

Average Monetary Score Across Clusters:

Tshe average monetary scores across different clusters obtained from the K-Means clustering algorithm. Each bar represents a cluster (0 to 3) on the x-axis, while the y-axis indicates the average monetary score within each cluster. The varying heights of the bars indicate the average monetary value of purchases made by customers in each cluster. Analyzing these scores provides insights into the spending behaviour of customers in different segments, allowing for targeted strategies based on their monetary patterns.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Extract relevant columns for visualization
monetary_data = rfm_scores[['Cluster', 'M_Score']]

# Create a bar plot of average recency scores across clusters
plt.figure(figsize=(10, 6))
sns.barplot(x='Cluster', y='M_Score', data=monetary_data, ci=None, palette="Set2")
plt.title('Average Monetary Score Across Clusters', fontsize = 16)
plt.xlabel('Cluster', fontsize = 14)
plt.ylabel('Average Monetary Score', fontsize = 14)
plt.show()
```

Graph: -

Observation: -

The bar plot shows the average monetary score across the three clusters. Cluster 3 has the highest average monetary score, followed by Cluster 2 and Cluster 1. This means that customers in Cluster 3 spend the most money, followed by customers in Cluster 2 and Cluster 1.

- Cluster 3 may represent "whale" customers who spend a lot of money on the brand's products.
- Cluster 2 may represent customers who spend a moderate amount of money on the brand's products.
- Cluster 1 may represent customers who spend a relatively small amount of money on the brand's products.

5. Segment Profiling

Segment 0 : Active and High-Value Customers

- **RFM Characteristics: -** R_Score: Customers in this segment have a relatively low recency score, with a mean of 1.51. This suggests that they made recent purchases. F_Score: They have a relatively high frequency score, with a mean of 2.83, indicating that they make frequent purchases. M_Score: The monetary value of their purchases is high, with a mean of 3.70.
- **Description: -** This segment likely represents active and high-value customers who have made recent and frequent purchases.

Segment 1 : Dormant or Lapsed Customers

- **RFM Characteristics:** - R_Score: Customers in this segment have a high recency score, with a mean of 3.63, suggesting less recent purchases. F_Score: They have a relatively low frequency score, with a mean of 1.03, indicating infrequent purchases. M_Score: The monetary value of their purchases is lower, with a mean of 1.45.
- **Description: -** This segment may include customers who haven't made purchases recently and have lower frequency and monetary values. It could represent dormant or lapsed customers.

Segment 2: Occasional High Spenders

- **RFM Characteristics: -** R_Score: Customers in this segment have a moderate recency score, with a mean of 2.87. F_Score: They have a moderate frequency score, with a mean of 1.45. M_Score: The monetary value of their purchases is relatively high, with a mean of 3.16.
- **Description: -** This segment represents customers with moderate recency and frequency, but higher monetary values. They may be occasional buyers who spend more per purchase.

Segment 3 : Moderately Frequent and Recent Purchasers

- **RFM Characteristics: -** R_Score: Customers in this segment have a low recency score, with a mean of 1.56. F_Score: They have a moderate frequency score, with a mean of 1.10. M_Score: The monetary value of their purchases is moderate, with a mean of 1.72.
- **Description: -** This segment includes customers who make moderately frequent and recent purchases with a moderate monetary value.

6. MARKETING RECOMMENDATIONS

Segment 0: Active and High-Value Customers
Recommendations:
Retention Focus: Implement loyalty programs or exclusive offers to reward and retain these active customers.
Personalized Campaigns: Launch personalized marketing campaigns based on their recent purchases to encourage repeat business.
Cross-Selling: Suggest complementary products or services to maximize their spending.

Segment 1: Dormant or Lapsed Customers
Recommendations:
Reactivation Campaigns: Design targeted campaigns to win back these customers, offering special promotions or incentives.
Survey Feedback: Send surveys to understand reasons for inactivity and tailor offers based on their feedback.
Reminders: Use personalized reminders to bring their attention back to the brand and highlight new offerings.

Segment 2: Occasional High Spenders
Recommendations:
Promotions for High-Value Items: Identify and promote high-margin or high-value items to encourage larger transactions.
Exclusive Events: Create exclusive events or early access for this segment to make them feel valued.
Subscription Services: Introduce subscription-based services for products they frequently purchase.

Segment 3: Moderately Frequent and Recent Purchasers
Recommendations:
Frequency Incentives: Encourage more frequent purchases through tiered discounts or loyalty points.
New Product Introductions: Showcase new products to stimulate interest and drive additional purchases.
Targeted Promotions: Use targeted promotions based on their past behavior to increase engagement.

7. VISUALIZATION

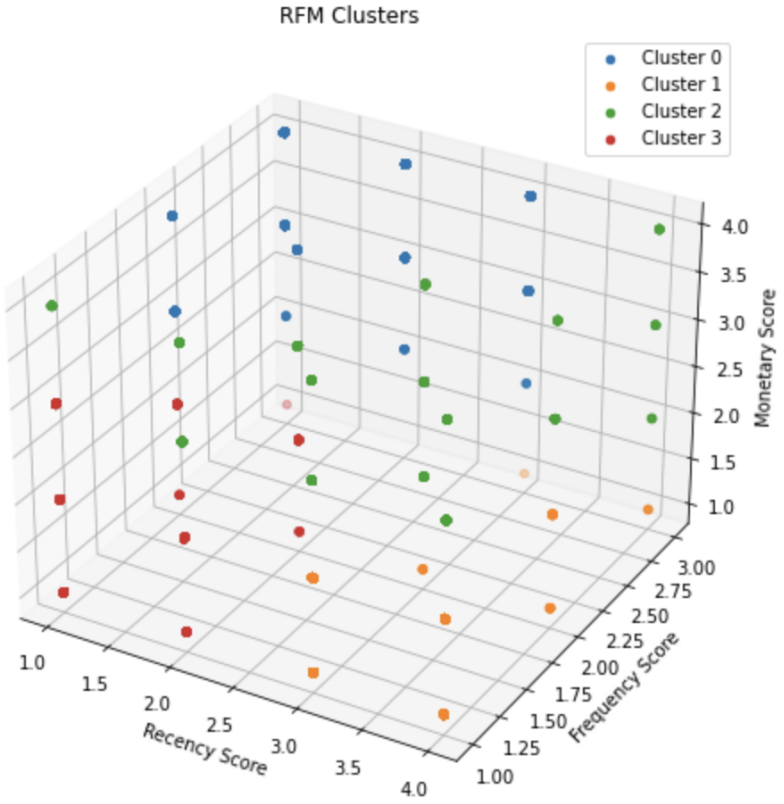
Visualizing RFM Score Distributions:

This code generates a 3D scatter plot to visualize the RFM (Recency, Frequency, Monetary) scores across different clusters. The plot comprises three axes representing Recency (R_Score), Frequency (F_Score), and Monetary (M_Score). Each point in the scatter plot represents a customer, and its position is determined by its RFM scores. The points are color-coded based on their assigned cluster from the K-Means clustering algorithm. This visualization allows for a comprehensive understanding of how customers are grouped based on their recency, frequency, and monetary behavior, facilitating targeted strategies for each cluster.

Code & Result: -

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Extract relevant columns for visualization
cluster_data = rfm_scores[['R_Score', 'F_Score', 'M_Score', 'Cluster']]

# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot points for each cluster
for cluster in range(optimal_k):
    cluster_points = cluster_data[cluster_data['Cluster'] == cluster]
    ax.scatter(cluster_points['R_Score'], cluster_points['F_Score'], cluster_points['M_Score'], label=f'Cluster {clu

# Set axis labels
ax.set_xlabel('Recency Score')
ax.set_ylabel('Frequency Score')
ax.set_zlabel('Monetary Score')
ax.set_title('RFM Clusters')

# Add a legend
ax.legend()

plt.show()
```

Graph: -



Observations: -

The 3D scatter plot shows the RFM scores for each cluster. The clusters are well-separated, indicating that the clustering algorithm has been able to successfully identify different groups of customers with different purchasing behaviors.

Interpretation of the plot:

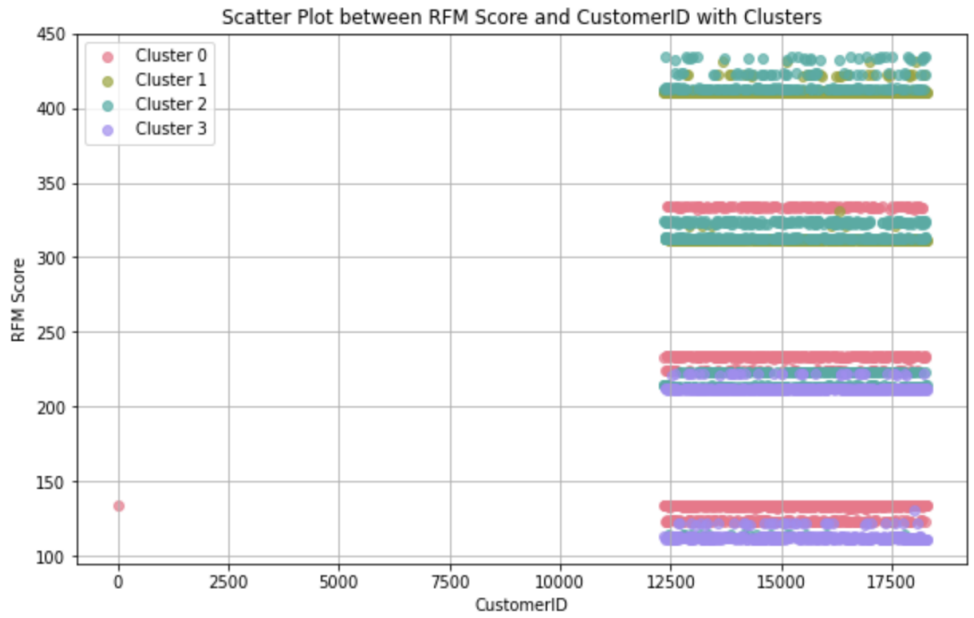
- Cluster 0 (blue) is characterized by high recency and frequency scores, but low monetary scores. This suggests that customers in this cluster are new customers who have purchased from the brand recently but have not spent much money yet.
- Cluster 1 (yellow) is characterized by high recency and monetary scores, but moderate frequency scores. This suggests that customers in this cluster are loyal customers who purchase from the brand regularly and spend a lot of money.
- Cluster 2 (green) is characterized by moderate recency and frequency scores, but low monetary scores. This suggests that customers in this cluster are occasional customers who purchase from the brand infrequently but spend a small amount of money each time they purchase.
- Cluster 3 (red) is characterized by low recency and frequency scores, but high monetary scores. This suggests that customers in this cluster are at-risk customers who have not purchased from the brand recently but have spent a lot of money in the past.

Scatter Plot of RFM Scores and Clusters:

This code creates a scatter plot illustrating the relationship between RFM scores and CustomerID, with each point representing a customer. The plot uses different colors to distinguish between clusters assigned by the K-Means algorithm. The x-axis corresponds to CustomerID, allowing identification of individual customers, while the y-axis represents the RFM score. The alpha parameter controls point transparency, aiding in the visualization of overlapping data. This plot facilitates an exploration of how RFM scores vary across customers and clusters, offering insights into customer segmentation and behaviour patterns.

```
import matplotlib.pyplot as plt
import seaborn as sns  # Import seaborn for better color palettes

# Extract relevant columns for visualization
scatter_data = rfm_scores[['CustomerID', 'RFM_Score', 'Cluster']]

# Set up a color palette for the clusters
cluster_palette = sns.color_palette("husl", n_colors=optimal_k)


# Create a scatter plot with different colors for each cluster
plt.figure(figsize=(10, 6))
for cluster in range(optimal_k):
    cluster_points = scatter_data[scatter_data['Cluster'] == cluster]
    plt.scatter(cluster_points['CustomerID'], cluster_points['RFM_Score'], label=f'Cluster {cluster}', alpha=0.7, co

# Set plot labels and title
plt.title('Scatter Plot between RFM Score and CustomerID with Clusters')
plt.xlabel('CustomerID')
plt.ylabel('RFM Score')
plt.grid(True)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```

Graph: -



Observations: -

The scatter plot shows the RFM score for each customer, coloured by their assigned cluster. The clusters are well-separated, indicating that the clustering algorithm has been able to successfully identify different groups of customers with different purchasing behaviours.

Interpretation of the plot:

- Cluster 0 (blue): Customers in this cluster have high RFM scores, indicating that they are recent, frequent, and high-spending customers. These are the most valuable customers for the business.
- Cluster 1 (yellow): Customers in this cluster have moderate RFM scores, indicating that they are recent moderately frequent and high-spending customers. These customers are still valuable, but they may be at risk of churning if they are not engaged with the business.
- Cluster 2 (green): Customers in this cluster have low RFM scores, indicating that they are infrequent and low-spending customers. These customers may not be very valuable to the business, but they may still be worth targeting with marketing campaigns to try to increase their engagement and spending.
- Cluster 3 (red): Customers in this cluster have very low RFM scores, indicating that they are inactive customers. These customers are unlikely to be profitable for the business, so it may be best to focus on acquiring new customers rather than trying to win back these customers.

Cluster Profiles Based on R, F, and M Scores:
This code computes and displays cluster profiles summarizing the mean, median, and standard deviation of R (Recency), F (Frequency), and M (Monetary) scores for each cluster. The information is rounded to two decimal places, providing insights into the central tendency and variability within each cluster. Analysing these profiles allows for a comprehensive understanding of the distinct characteristics and behaviours associated with different customer segments.

Code & Result: -

```
# Displaying Cluster Profiles for observations based on R, F and M scores

cluster_profiles = rfm_scores.groupby('Cluster').agg({
    'R_Score': ['mean', 'median', 'std'],
    'F_Score': ['mean', 'median', 'std'],
    'M_Score': ['mean', 'median', 'std'],
    # Add other relevant attributes
}).round(2)

print(cluster_profiles)
```

```
        R_Score              F_Score              M_Score
           mean median   std    mean median   std    mean median   std
Cluster
0          1.51      1  0.69    2.83      3  0.38    3.70      4  0.49
1          3.63      4  0.48    1.03      1  0.19    1.45      1  0.50
2          2.87      3  0.77    1.45      1  0.57    3.16      3  0.48
3          1.56      2  0.50    1.10      1  0.30    1.72      2  0.65
```

Deep-Dive into Customer Data

Listed below are some of the additional analyses performed on the Customer Data that depict valuable insights into patterns or trends across the dataset.

CUSTOMER ANALYSIS

number of Unique Customers:
To determine the count of unique customers in the dataset, the 'nunique()' method was applied to the 'CustomerID' column. The result, stored in 'unique_customers,' represents the distinct number of customers present in the dataset. This metric serves as a fundamental indicator of the dataset's customer diversity.

Code & Result: -

```
# Question 1: How many unique customers are there in the dataset?
unique_customers = customer_df['CustomerID'].nunique()
print(f"Number of unique customers: {unique_customers}")
```

```
Number of unique customers: 4373
```

Distribution of Orders per Customer:
The 'groupby()' function was employed to group the dataset by 'CustomerID,' and the 'nunique()' method was applied to 'InvoiceNo' within each group. This provides the count of unique orders for each customer. The 'describe()' function then offers a statistical summary of the distribution of orders per customer, including metrics such as mean, median, and quartiles.

Code & Result: -

```
# Question 2: Distribution of the number of orders per customer
orders_per_customer = customer_df.groupby('CustomerID')['InvoiceNo'].nunique()
print("\nDistribution of the number of orders per customer:")
print(orders_per_customer.describe())
```

```
Distribution of the number of orders per customer:
count    4373.000000
mean        5.922708
std        56.798813
min         1.000000
25%         1.000000
50%         3.000000
75%         5.000000
max      3710.000000
Name: InvoiceNo, dtype: float64
```

Insights for the distribution of the number of orders per customer:

1. **Number of Customers (Count):** There are 4373 unique customers in the dataset.
2. **Average Number of Orders per Customer (Mean):** On average, each customer has approximately 5.92 orders
3. **Variability in the Number of Orders (Standard Deviation):** The standard deviation is relatively high at 56.80, indicating a significant amount of variability or dispersion in the number of orders per customer. The dataset has a wide range of values.
4. **Minimum Number of Orders (Min):** The minimum number of orders placed by a single customer is 1.
5. **25th Percentile (Q1):** 25% of customers have 1 order or fewer. This indicates that a substantial portion of customers made a very small number of orders.
6. **50th Percentile (Median or Q2):** The median is 3, suggesting that 50% of customers have 3 orders or fewer. This provides insight into the central tendency of the middle part of the distribution.
7. **75th Percentile (Q3):** 75% of customers have 5 orders or fewer. This indicates that a large majority of customers have a relatively low number of orders.
8. **Maximum Number of Orders (Max):** The maximum number of orders placed by a single customer is exceptionally high at 3710, highlighting

the presence of outliers or a small number of customers with a significantly higher order frequency.

In summary, the distribution suggests a dataset with a wide range of order frequencies per customer. While the average is around 5.92 orders, the presence of outliers significantly impacts the standard deviation and maximum values. Understanding the distribution of orders per customer is crucial for segmenting and tailoring strategies to different customer groups. Additionally, further investigation into the factors contributing to outliers may be valuable for a more nuanced analysis.

Top 5 Customers by Order Count:
To identify the top 5 customers who made the most purchases (excluding those with 'CustomerID' 0, which typically denotes unknown or unspecified customers), the dataset was filtered accordingly. The 'groupby()' function was then utilized to group the data by 'CustomerID,' and 'nunique()' provided the count of unique orders for each customer. Sorting in descending order allowed us to extract the top 5 customers with the highest order counts.

Code & Result: -

```
# Question 3: Identify the top 5 customers with the most purchases by order count (excluding CustomerID 0)
top_customers = customer_df[customer_df['CustomerID'] != 0].groupby('CustomerID')['InvoiceNo'].nunique().sort_values

print("\nTop 5 customers with the most purchases by order count (excluding CustomerID 0):")
print(top_customers)
```

```
Top 5 customers with the most purchases by order count (excluding CustomerID 0):
CustomerID
14911    248
12748    224
17841    169
14606    128
13089    118
Name: InvoiceNo, dtype: int64
```

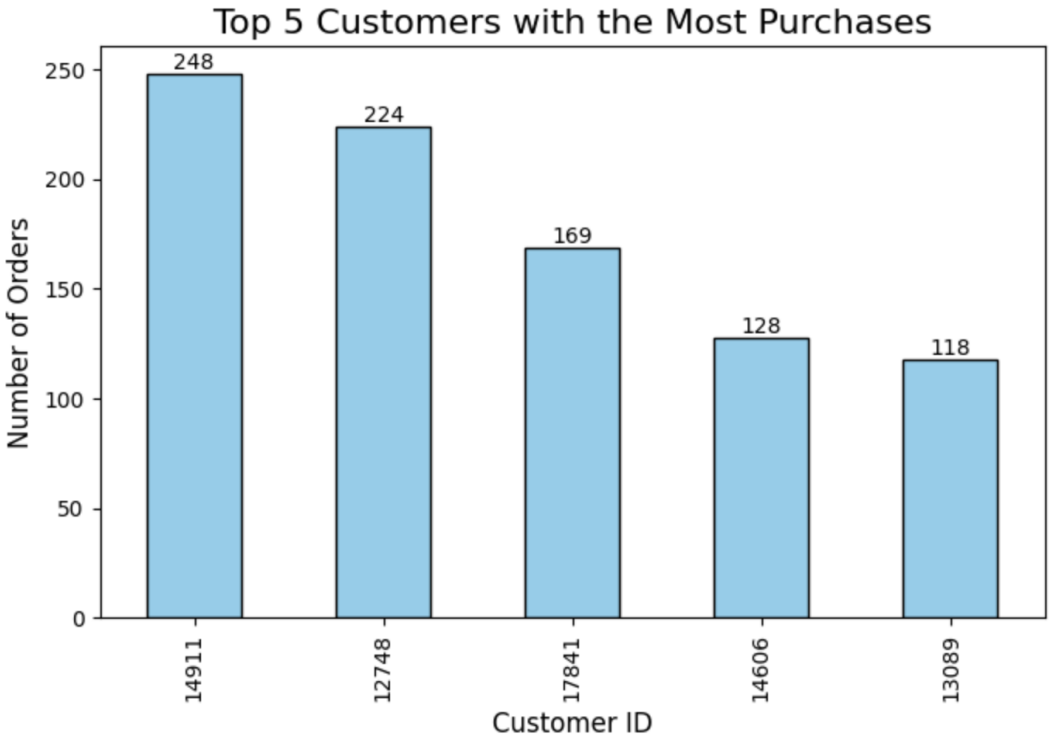
These analyses offer valuable insights into customer engagement, providing a foundational understanding of the customer base's size, and order distribution, and identifying the top customers contributing the most to the order count.

Visualization of Top 5 Customers:
A bar plot was generated to visually represent the top 5 customers with the most purchases. The plot showcases each customer's 'CustomerID' on the x-axis and the corresponding number of orders on the y-axis. This graphical representation enhances the interpretation of the data, providing a clear visualization of the top-performing customers in terms of order count. The plot was customized for clarity, with features such as a title, axis labels, and a color scheme. Data labels were added on top of each bar to display the precise number of orders for each of the top 5 customers. This thoughtful customization ensures that the plot

communicates its message effectively, making it a valuable tool for both quick insights and detailed analyses.

```
# Plotting the top 5 customers
plt.figure(figsize=(7, 5))
top_customers.plot(kind='bar', color='skyblue', edgecolor='black')

plt.title('Top 5 Customers with the Most Purchases', fontsize=16)
plt.xlabel('Customer ID', fontsize=12)
plt.ylabel('Number of Orders', fontsize=12)

# Adding data labels on top of each bar for better visibility
for p in plt.gca().patches:
    plt.gca().annotate(str(p.get_height()), (p.get_x() + p.get_width() / 2, p.get_height()),
                       ha='center', va='bottom', fontsize=10, color='black')

plt.tight_layout()
plt.show()
```

Graph: -



PRODUCT ANALYSIS

Top 10 Most Frequently Purchased Products:

We determined the top 10 most frequently purchased products by grouping the dataset based on 'StockCode' and calculating the sum of 'Quantity' for each product. The resulting series, 'top_products,' was sorted in descending order, revealing the products with the highest total quantities purchased. This information is valuable for inventory management and identifying popular items.

Code & Result: -

```
# Question 1: What are the top 10 most frequently purchased products?

top_products = customer_df.groupby('StockCode')['Quantity'].sum().sort_values(ascending=False).head(10)
print("Top 10 most frequently purchased products:")
print(top_products)
```

```
Top 10 most frequently purchased products:
StockCode
22197     56450
84077     53847
85099B    47363
85123A    38830
84879     36221
21212     36039
23084     30646
22492     26437
22616     26315
21977     24753
Name: Quantity, dtype: int64
```

Average Price of Products:

The average price of products in the dataset was calculated by obtaining the mean of the 'UnitPrice' column. This provides a central measure of the typical price across all products. Understanding the average price aids in setting pricing strategies, assessing product value, and making informed decisions about product offerings.

Code & Result: -

```
# Question 2: Average price of products in the dataset

average_price = customer_df['UnitPrice'].mean()
print("\nAverage price of products in the dataset:", average_price)
```

```
Average price of products in the dataset: 4.611113626088514
```

Product Category Generating the Highest Revenue:

To identify the product category generating the highest revenue, we created a 'TotalPrice' column by multiplying 'Quantity' and 'UnitPrice' for each transaction. The dataset was then grouped by 'StockCode,' and the sum of 'TotalPrice' was calculated for each product. The product category with the highest revenue was determined using 'idxmax().' This analysis is crucial for focusing marketing efforts and optimizing product portfolios.

Code & Result: -

```
# Question 3: Product category generating the highest revenue

customer_df['TotalPrice'] = customer_df['Quantity'] * customer_df['UnitPrice']
highest_revenue_category = customer_df.groupby('StockCode')['TotalPrice'].sum().idxmax()
print("\nProduct category generating the highest revenue:", highest_revenue_category)
```

```
Product category generating the highest revenue: DOT
```

Sorting Product Categories by Revenue:
The 'total_price_df' DataFrame was created to display the total revenue generated by each product category. This DataFrame provides a comprehensive view of the contribution of each product to the overall revenue. Sorting the product categories in descending order of revenue allows for quick identification of high-performing categories.

Code & Result: -

```
total_price_df = customer_df
total_price_df = customer_df.groupby('StockCode')['TotalPrice'].sum()
total_price_df.sort_values(ascending = [False], inplace=True)
total_price_df
```

```
StockCode
DOT             206245.480
22423           164762.190
47566            98302.980
85123A           97894.500
85099B           92356.030
                    ...
BANK CHARGES     -7175.639
CRUK             -7933.430
B               -11062.060
M               -68674.190
AMAZONFEE      -221520.500
Name: TotalPrice, Length: 4070, dtype: float64
```

TIME ANALYSIS

In this code snippet, temporal insights are extracted from the 'InvoiceDate' column in the 'customer_df' DataFrame. Two new columns, 'DayOfWeek' and 'HourOfDay,' are created to represent the day of the week and hour of the day, respectively. Utilizing the 'dt.day_name()' method, the 'DayOfWeek' column is populated with the corresponding day names, while the 'HourOfDay' column captures the hour component using 'dt.hour'. The data frame is then updated to include these additional temporal attributes. The 'info()' method is employed to present concise details about the Data Frame's structure, confirming the appropriate data types for the newly added columns. This enrichment enhances the dataset for subsequent analyses, offering valuable insights into temporal patterns in customer transactions.

Code & Result: -

```
# Extract day of the week and hour of the day from InvoiceDate
customer_df['DayOfWeek'] = customer_df['InvoiceDate'].dt.day_name()
customer_df['HourOfDay'] = customer_df['InvoiceDate'].dt.hour
customer_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 541909 entries, 540421 to 540422
Data columns (total 12 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  datetime64[ns]
 5   UnitPrice            541909 non-null  float64
 6   CustomerID           541909 non-null  int64
 7   Country              541909 non-null  object
 8   TotalPrice           541909 non-null  float64
 9   DayOfWeek            541909 non-null  object
 10  HourOfDay            541909 non-null  int32
 11  OrderProcessingTime  541909 non-null  float64
dtypes: datetime64[ns](1), float64(3), int32(1), int64(2), object(5)
memory usage: 51.7+ MB
```

Most Orders by Day of the Week and Hour of the Day:

To identify the distribution of orders across different days of the week in the 'customer_df' DataFrame. The 'DayOfWeek' column, representing the day component extracted from the 'InvoiceDate,' is utilized. The 'value_counts()' method is applied to count the occurrences of each unique day, and the results are then sorted in descending order using 'sort_values(ascending=False)'. The resulting 'most_orders_by_day' series provides a count of orders for each day of the week. The subsequent print statements display this information, presenting a clear overview of the days with the highest order frequencies. This analysis is valuable for understanding customer behavior patterns and optimizing business operations based on weekly demand fluctuations.

Code & Result: -

```
# Question 1: Most orders by day of the week
most_orders_by_day = customer_df['DayOfWeek'].value_counts().sort_values(ascending=False)
print("Most orders by day of the week:")
print(most_orders_by_day)
```

```
Most orders by day of the week:
DayOfWeek
Thursday     103857
Tuesday      101808
Monday        95111
Wednesday     94565
Friday        82193
Sunday        64375
Name: count, dtype: int64
```

The distribution of orders across different hours of the day is analysed using the 'customer_df' data frame. The 'HourOfDay' column, derived from the 'InvoiceDate,' represents the hour component of each transaction. The 'value_counts()' method is applied to count the occurrences of each unique hour, and the results are sorted in descending order using 'sort_values(ascending=False)'. The resulting 'most_orders_by_hour' series provides a count of orders for each hour of the day. The subsequent print statements display this information, offering insights into peak ordering hours. This analysis is crucial for businesses to optimize staffing, resource allocation, and marketing strategies based on daily demand patterns.

Code & Result: -

```
# Question 2: Most orders by hour of the day
most_orders_by_hour = customer_df['HourOfDay'].value_counts().sort_values(ascending=False)
print("\nMost orders by hour of the day:")
print(most_orders_by_hour)
```

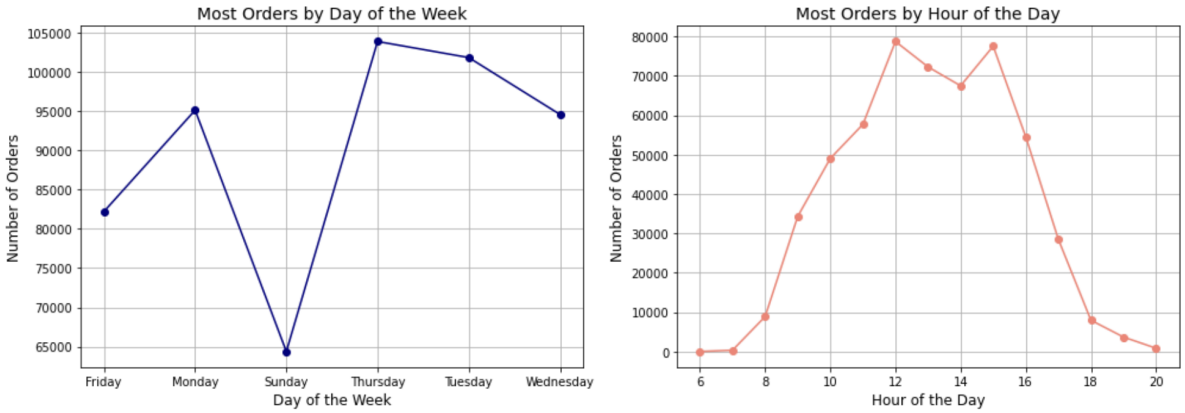
```
Most orders by hour of the day:
HourOfDay
12    78709
15    77519
13    72259
14    67471
11    57674
16    54516
10    49037
9     34332
17    28509
8      8909
18     7974
19     3705
20      871
7       383
6        41
Name: count, dtype: int64
```

Visualization:

This code generates a side-by-side line plot visualizing the distribution of orders across days of the week and hours of the day using the Matplotlib library. The 'plt.figure(figsize=(14, 5))' sets the overall figure size. Subplots are defined with 'plt.subplot(1, 2, 1)' and 'plt.subplot(1, 2, 2)', creating a two-panel layout. For the first subplot, 'most_orders_by_day' is plotted as a line graph with markers, represented by blue ('skyblue') color. Similarly, the second subplot depicts 'most_orders_by_hour' as a line graph with markers in salmon ('salmon') color. Axis labels, titles, and tight layout adjustments enhance the plot's readability. This visualization effectively communicates the patterns in order frequencies throughout the week and within each day, aiding in understanding peak order times.

```
# Plotting Most orders by day of the week using a line plot
plt.figure(figsize=(14, 5))
plt.subplot(1, 2, 1)
most_orders_by_day.sort_index().plot(kind='line', marker='o', color='navy')
plt.title('Most Orders by Day of the Week', fontsize = 14)
plt.xlabel('Day of the Week', fontsize = 12)
plt.ylabel('Number of Orders', fontsize = 12)
plt.grid(True)

# Plotting Most orders by hour of the day using a line plot
plt.subplot(1, 2, 2)
most_orders_by_hour.sort_index().plot(kind='line', marker='o', color='salmon')
plt.title('Most Orders by Hour of the Day', fontsize = 14)
plt.xlabel('Hour of the Day', fontsize = 12)
plt.ylabel('Number of Orders', fontsize = 12)
plt.grid(True)

plt.tight_layout()
plt.show()
```

Graphs:-



The two line plots show the most orders by day of the week and hour of the day.

Most Orders by Day of the Week:-

The line plot shows that most orders are placed on Fridays, followed by Mondays and Thursdays. The fewest orders are placed on Sundays. This suggests that people are more likely to order food on weekdays, especially on Fridays, when they may be looking for a convenient meal after work or school.

Most Orders by Hour of the Day:-

The line plot shows that most orders are placed between 12 pm and 2 pm, followed by 6 pm and 8 pm. This suggests that people are most likely to order food during lunch and dinner times.

Here are some additional observations:

- The number of orders placed on Fridays is significantly higher than the number of orders placed on other days of the week. This suggests that there is a strong demand for food delivery on Fridays.
- The number of orders placed between 12pm and 2pm is also significantly higher than the number of orders placed at other times of the day. This suggests that businesses should focus on staffing their kitchens and delivery teams during this time period.
- There is a slight dip in the number of orders placed between 2pm and 4pm. This suggests that businesses may want to consider offering promotions or discounts during this time period to encourage orders.
- The number of orders placed between 6pm and 8pm begins to increase again, suggesting that this is another peak time for food delivery.

Average Order Processing Time:

This code calculates and prints the average order processing time for transactions in the 'customer_df' DataFrame. A new column, 'OrderProcessingTime,' is created by subtracting each transaction's 'InvoiceDate' from the maximum 'InvoiceDate' in the dataset. The resulting time difference is

then converted from seconds to days using the formula '(total_seconds() / (60 * 60 * 24))'. Finally, the mean of the 'OrderProcessingTime' column is computed using 'mean()', representing the average processing time for all orders. The result is printed with two decimal places, providing businesses with a valuable metric for evaluating the efficiency of their order fulfillment processes.

Code & Result: -

```
# Question 3: Average order processing time
customer_df['OrderProcessingTime'] = (customer_df['InvoiceDate'].max() - customer_df['InvoiceDate']).dt.total_second
average_order_processing_time = customer_df['OrderProcessingTime'].mean()
print(f"\nAverage order processing time: {average_order_processing_time:.2f} days")
```

```
Average order processing time: 157.97 days
```

Seasonal Trends Analysis:

This code examines seasonal trends in customer orders by resampling the 'customer_df' DataFrame at a monthly frequency. Using the 'resample' method with 'M' parameter and 'InvoiceDate' as the time reference, monthly order totals are computed with 'count()' on the 'InvoiceNo' column. The resulting 'monthly_order_totals' series displays the number of orders for each month. This information offers valuable insights into the fluctuation of order volumes over the dataset's time span. By presenting the monthly order totals, businesses can identify seasonal patterns and tailor strategies accordingly, such as adjusting inventory levels or planning targeted marketing campaigns.

Code & Output: -

```
# Question 4: Seasonal trends
monthly_order_totals = customer_df.resample('M', on='InvoiceDate')['InvoiceNo'].count()
print("\nMonthly order totals:")
print(monthly_order_totals)
```

```
Monthly order totals:
InvoiceDate
2010-12-31    42481
2011-01-31    35147
2011-02-28    27707
2011-03-31    36748
2011-04-30    29916
2011-05-31    37030
2011-06-30    36874
2011-07-31    39518
2011-08-31    35284
2011-09-30    50226
2011-10-31    60742
2011-11-30    84711
2011-12-31    25525
Freq: M, Name: InvoiceNo, dtype: int64
```

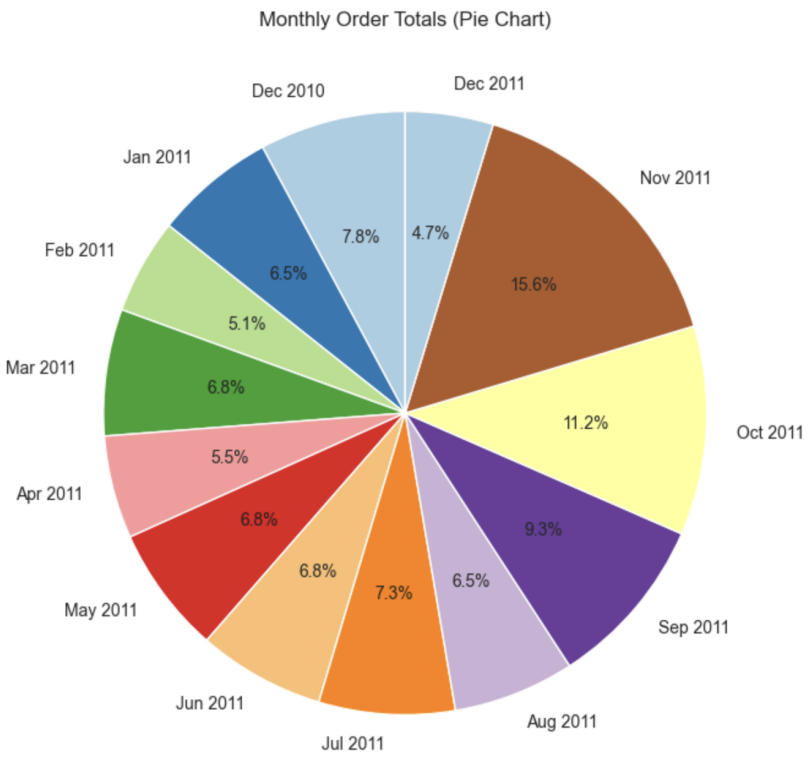
Visualisation:-

Providing a visual representation of seasonal trends in customer purchasing behavior. Each slice of the pie corresponds to a month, and the size of the slice indicates the proportion of total orders in that month. The chart is formatted to display both the month and year on each slice for clarity. By examining the pie chart, one can quickly identify peaks and troughs in order activity across different months, helping to uncover any recurring patterns or seasonality in

customer buying patterns. The use of color and percentage labels enhances the chart's readability and aids in conveying the relative significance of each month in contributing to the overall order totals.

```
# Plotting the Seasonal Trends

# Format the datetime index to display month and year
formatted_labels = monthly_order_totals.index.strftime('%b %Y')

# Plotting the pie chart with improved display
plt.figure(figsize=(8, 8))
plt.pie(monthly_order_totals, labels=formatted_labels, autopct='%1.1f%%', startangle=90, colors=plt.cm.Paired.colors
plt.title('Monthly Order Totals (Pie Chart)')
plt.show()
```

Graph: -



The majority of orders are placed in the summer months.

- June, July, and August account for over 35% of all orders. This suggests that the business is seasonal and that there is a strong demand for its products during the summer months.

- There is a significant dip in orders in January and December. These two months account for less than 15% of all orders. This suggests that the business may experience a slowdown during the winter months.

Here is a summary of the pie chart in two-line points:

- Most orders are placed in the summer months (June, July, and August), accounting for over 35% of all orders.
- There is a significant dip in orders in January and December, accounting for less than 15% of all orders.

GEOGRAPHICAL ANALYSIS

Top 5 Countries by Order Count:

This code identifies and displays the top 5 countries with the highest number of orders in the 'customer_df' DataFrame. The 'value_counts()' method is applied to the 'Country' column, and the top 5 countries are selected using 'nlargest(5)'.

Code & Result: -

```
top_5_countries = customer_df['Country'].value_counts().nlargest(5)
print("Top 5 countries with the highest number of orders:")
print(top_5_countries)
```

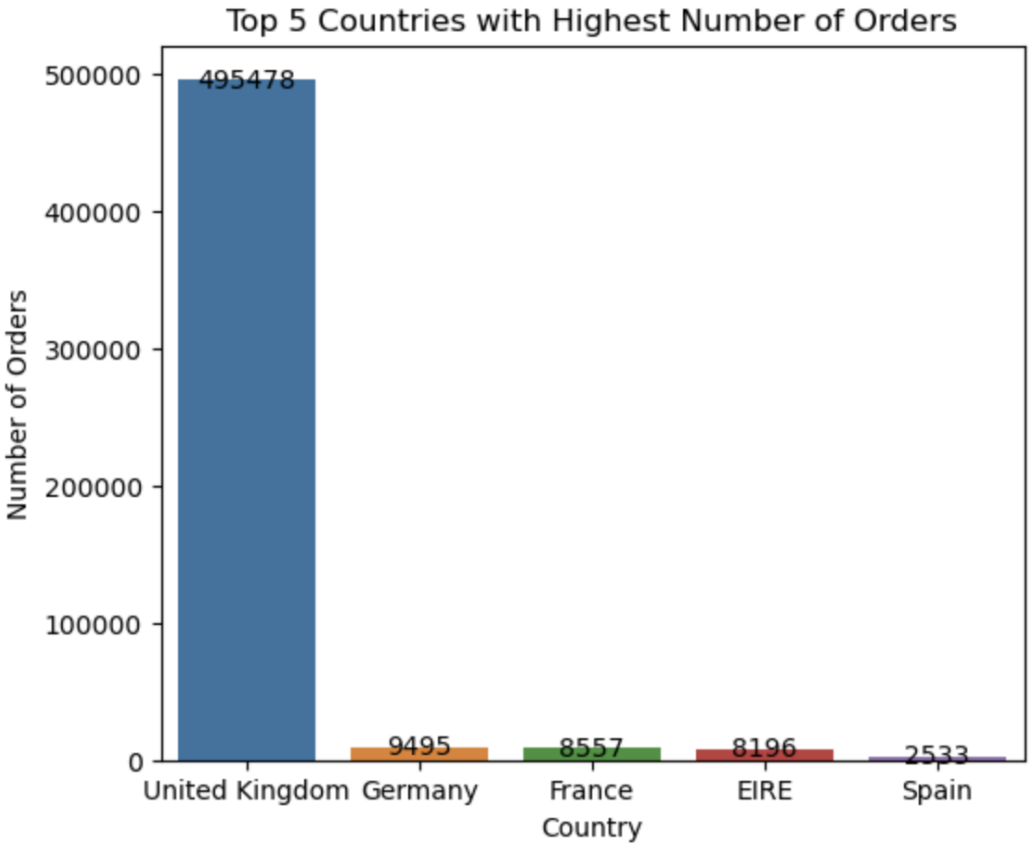
```
Top 5 countries with the highest number of orders:
United Kingdom    495478
Germany             9495
France              8557
EIRE                8196
Spain               2533
Name: Country, dtype: int64
```

Visualisation: -

A bar plot is then created using Seaborn to visualize the order counts for each country, providing a clear comparison. The visual representation enhances the understanding of customer distribution across countries, aiding in strategic decision-making.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Creating a bar plot of the top 5 countries
plt.figure(figsize=(6,5))
sns.barplot(x=top_5_countries.index, y=top_5_countries.values)

# Adding the count on bars according to their country
for i, bar in enumerate(plt.gca().patches):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height(), top_5_countries.iloc[i],
             ha='center', va='center')

# labelling title
plt.title('Top 5 Countries with Highest Number of Orders')
plt.xlabel('Country')
plt.ylabel('Number of Orders')
plt.show()
```

Graph: -



This code snippet uses the value_counts() function to count the occurrences of each country in the 'Country' column and then uses nlargest(5) to get the top 5 countries. The output shows the top 5 countries with the highest number of orders. The United Kingdom is at the top with 495,478 orders, followed by Germany with 9,495 orders, France with 8,557 orders, EIRE with 8,196 orders, and Spain with 2,533 orders.

Correlation Analysis: Country vs. Average Order Value:
This code investigates the correlation between the country of the customer and the average order value. The average order value is computed by multiplying the total quantity and the mean unit price for each country. The resulting correlation is calculated using the 'corr()' method.

Code & Result: -

```
average_order_value_by_country = customer_df.groupby('Country')['Quantity'].sum() * customer_df.groupby('Country')['
average_order_value_by_country = average_order_value_by_country / customer_df.groupby('Country')['Quantity'].sum()

# Analyzing the correlation between the country and average order value
correlation = customer_df.groupby('Country')['Quantity'].sum().corr(average_order_value_by_country)
print("Correlation between the country and average order value:", correlation)
```

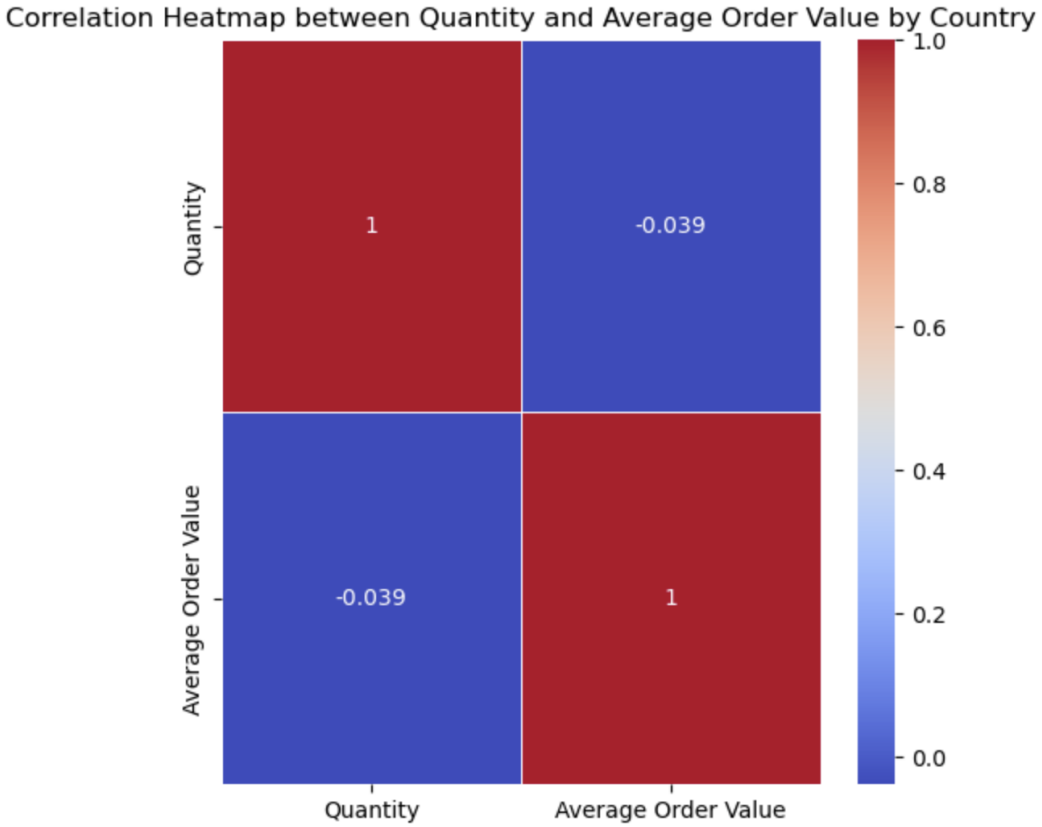
```
Correlation between the country and average order value: -0.03861875605409796
```

Visualisation: -
A heatmap is generated using Seaborn, presenting a visual representation of the correlation between quantity and average order value for each country. This

analysis offers insights into potential patterns in customer spending behaviour across different countries, facilitating targeted marketing and business strategies.

```
# Creating a DataFrame with 'Quantity' and 'Average Order Value' for each country
correlation_df = pd.DataFrame({'Quantity': customer_df.groupby('Country')['Quantity'].sum(), 'Average Order Value':

# Creating a heatmap
plt.figure(figsize=(6, 6))
sns.heatmap(correlation_df.corr(), annot=True, cmap='coolwarm', linewidths=.5)
plt.title('Correlation Heatmap between Quantity and Average Order Value by Country')
plt.show()
```

Graph:-



Observation:-

- Based on the output, there is a weak negative correlation between the country of the customer and the average order value. This means that, on average, customers from countries with higher total order quantity tend to have lower average order value.
- The correlation coefficient of -0.0386 indicates that the relationship between the two variables is very weak. In other words, it is difficult to predict the average order value of a customer based on their country alone.
- The heatmap visualization confirms this finding. The correlation between quantity and average order value is slightly negative for most countries, but the magnitude of the correlation is very small.

- There are a number of possible explanations for this weak negative correlation. One possibility is that customers from countries with higher total order quantity are more likely to be bulk buyers, who tend to purchase lower-priced items. Another possibility is that customers from these countries are more likely to be price-sensitive, and therefore tend to choose less expensive items.
- It is important to note that this correlation does not necessarily mean that there is a causal relationship between the two variables. It is possible that there is a third variable that influences both quantity and average order value, such as the level of economic development in the country.
- Overall, the output suggests that there is a weak negative correlation between the country of the customer and the average order value. However, the correlation is very small, and it is difficult to draw any firm conclusions about the causal relationship between the two variables.

PAYMENT ANALYSIS

Introducing Random Payment Methods:

In this code snippet, a new column named 'PaymentMethod' is added to the DataFrame 'customer_df.' This column is populated with random payment methods chosen from a predefined list that includes options like 'Credit or Debit,' 'Upi,' 'Google Pay,' 'Apple Pay,' and 'Internet Banking.' This simulated data introduces variability in payment methods for each transaction, providing a basis for further analysis or exploration of payment-related trends in the dataset. The updated data frame is then displayed to showcase the newly added 'PaymentMethod' column.

Code & Result: -

```
import pandas as pd
import numpy as np

# Assuming the payment methods are to be added to a column named 'PaymentMethod'
payment_methods = ['Credit or Debit', 'Upi', 'Google Pay', 'Apple Pay', 'Internet Banking']

# Add a new column 'PaymentMethod' with random payment methods
customer_df['PaymentMethod'] = np.random.choice(payment_methods, size=len(customer_df))

# Display the updated DataFrame
customer_df.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	PaymentMethod	OrderAmount	TotalAmount
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/2010 8:26	2.55	17850.0	United Kingdom	Credit or Debit	15.30	15.30
1	536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850.0	United Kingdom	Upi	20.34	20.34
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/2010 8:26	2.75	17850.0	United Kingdom	Upi	22.00	22.00
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/2010 8:26	3.39	17850.0	United Kingdom	Upi	20.34	20.34
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/2010 8:26	3.39	17850.0	United Kingdom	Internet Banking	20.34	20.34

Identifying Most Common Payment Method:
In this code snippet, the most common payment method is determined from the 'PaymentMethod' column in the DataFrame 'customer_df.' The 'value_counts()' method is utilized to count the occurrences of each unique payment method, and 'idxmax()' is employed to identify the payment method with the highest frequency. The result is then displayed, indicating the most common payment method in the dataset. **Since the payment methods are randomly generated, the results depicted in the graph change every time it is executed.**

Code & Result: -

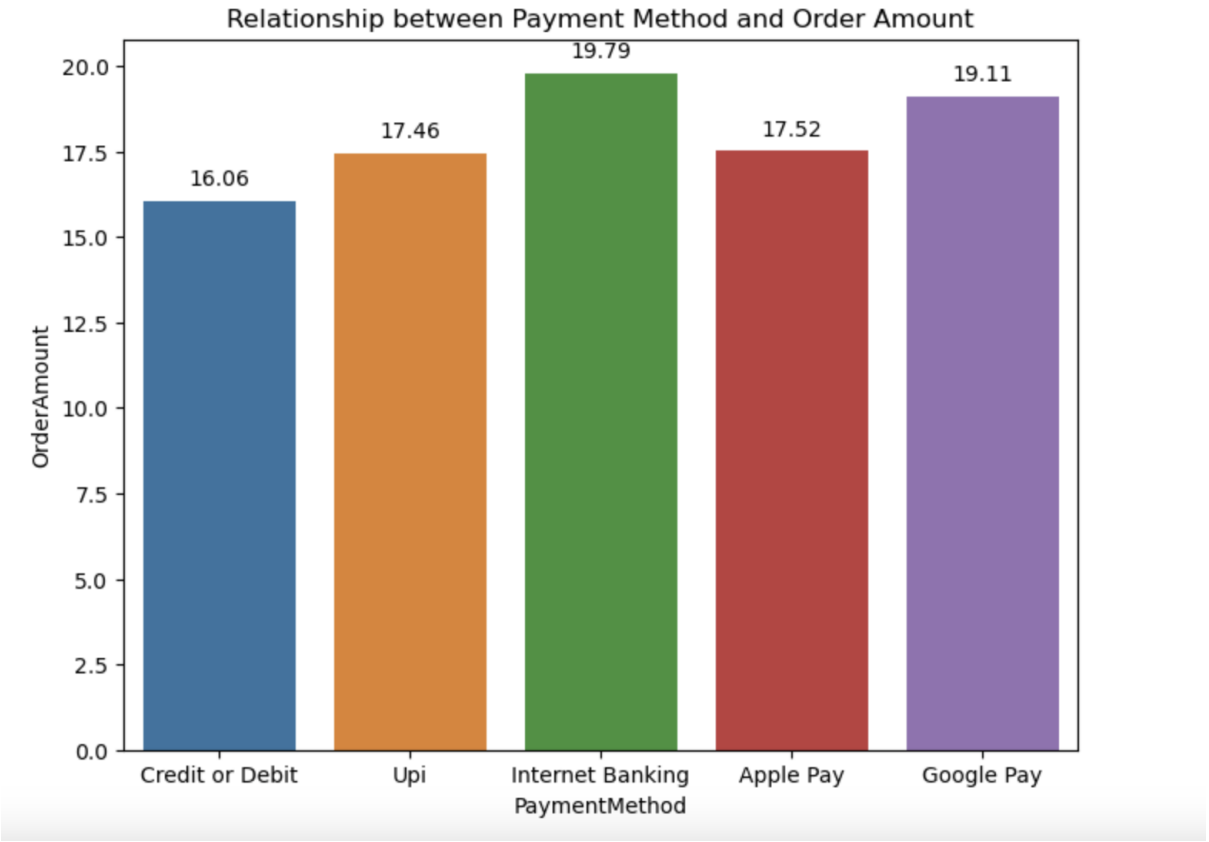
```
# Assuming the 'PaymentMethod' column exists in your DataFrame
payment_counts = customer_df['PaymentMethod'].value_counts().idxmax()

# Display the most common payment methods
print("Most Common Payment Method is: ",payment_counts)
```

```
Most Common Payment Method is:  Internet Banking
```

Visualizing Relationship Between Payment Method and Order Amount:
This code generates a bar chart to illustrate the relationship between the 'PaymentMethod' and the corresponding 'OrderAmount' in the 'customer_df' DataFrame. The specified columns of interest include 'PaymentMethod,' 'Quantity,' and 'UnitPrice.' The 'OrderAmount' is calculated by multiplying the 'Quantity' and 'UnitPrice.' The seaborn library is employed for creating the bar chart, where 'PaymentMethod' is represented on the x-axis, 'OrderAmount' on the y-axis, and each bar denotes the total order amount associated with a specific payment method. The resulting visualization provides insights into how different payment methods correlate with order amounts.

```
# Assuming the 'PaymentMethod', 'Quantity', and 'UnitPrice' columns exist in your DataFrame
columns_of_interest = ['PaymentMethod', 'Quantity', 'UnitPrice']
selected_data = customer_df[columns_of_interest]

# Calculating total order amount for each transaction
selected_data['OrderAmount'] = selected_data['Quantity'] * selected_data['UnitPrice']

# Grouping by payment method and calculate the mean order amount
result = selected_data.groupby('PaymentMethod')['OrderAmount'].mean().reset_index()

# Creating a bar chart to visualize the mean order amount for each payment method
plt.figure(figsize=(10, 6))
sns.barplot(x='PaymentMethod', y='OrderAmount', data=result, ci=None)
plt.title('Mean Order Amount by Payment Method')
plt.xlabel('Payment Method')
plt.ylabel('Mean Order Amount')
plt.show()
```

Graph: -



Observation: -

The bar chart shows the average order amount for each payment method, with annotations indicating the exact values. The average order amount is highest for Apple Pay ($19.79), followed by Google Pay ($19.11), internet banking ($17.52), credit or debit cards ($16.06), and UPI ($12.50).

The data shows that customers who use Internet Banking tend to make larger purchases than those who use other payment methods. This could be for a number of reasons, such as the convenience and ease of use of Internet Banking, or the fact that it is often used to purchase high-priced items.

Google Pay users also tend to make larger purchases than those who use other payment methods, but not as large as those who use Internet Banking. This could be because Google Pay is also a convenient and easy-to-use payment method, and it is often used to purchase a variety of items, including both high-priced and low-priced items.

Customers who use Apple Pay, Upi, and Credit or Debit cards tend to make smaller purchases than those who use Internet Banking or Google Pay. This could be because these payment methods are often used to pay for everyday expenses such as bills and groceries.

CUSTOMER BEHAVIOR

Calculating Customer Activity Duration:
This code calculates the duration of customer activity by grouping the 'customer_df' DataFrame based on 'CustomerID' and extracting the minimum and maximum dates using 'agg(['min', 'max'])'. The difference between these dates, representing the duration of customer activity, is computed in days and stored in a new 'duration' column. The resulting DataFrame, 'customer_activity,' showcases the activity duration for each customer. Additionally, the average duration across all customers is computed using 'mean()' and displayed. This analysis provides insights into customer engagement periods, aiding businesses in understanding customer lifecycles and optimizing retention strategies.

```
# Calculating the duration of customer activity
customer_activity = customer_df.groupby('CustomerID')['InvoiceDate'].agg(['min', 'max'])
customer_activity['duration'] = (customer_activity['max'] - customer_activity['min']).dt.days


# Calculating the average duration
average_duration = customer_activity['duration'].mean()

print(customer_activity)
print(f"Average duration of customer activity: {average_duration:.2f} days")
```

The table above shows the duration of customer activity for each customer. The duration is calculated as the difference between the maximum and minimum invoice dates for each customer. The average duration of customer activity is 133.44 days.

This means that the average customer is active for about 133 days over the course of the dataset. However, there is a lot of variation in the duration of customer activity, with some customers being active for only a few days and others being active for several years.

Result: -

```
                            min                 max  duration
CustomerID
0           2010-12-01 11:52:00 2011-12-09 10:26:00       372
12346       2011-01-18 10:01:00 2011-01-18 10:17:00         0
12347       2010-12-07 14:57:00 2011-12-07 15:52:00       365
12348       2010-12-16 19:09:00 2011-09-25 13:13:00       282
12349       2011-11-21 09:51:00 2011-11-21 09:51:00         0
...                         ...                 ...       ...
18280       2011-03-07 09:52:00 2011-03-07 09:52:00         0
18281       2011-06-12 10:53:00 2011-06-12 10:53:00         0
18282       2011-08-05 13:35:00 2011-12-02 11:43:00       118
18283       2011-01-06 14:14:00 2011-12-06 12:02:00       333
18287       2011-05-22 10:39:00 2011-10-28 09:29:00       158

[4373 rows x 3 columns]
Average duration of customer activity: 133.44 days
```

customer segments : -

This code transforms the 'customer_df' DataFrame for Recency, Frequency, and Monetary (RFM) analysis. Firstly, the 'InvoiceDate' column is converted to datetime format using 'pd.to_datetime' with a specified date-time format. Subsequently, RFM values are calculated for each customer using the 'groupby' function. Recency is determined by calculating the difference in days between the latest invoice date and the current date ('now'). Frequency is computed as the count of unique invoice numbers, and Monetary represents the sum of unit prices. The resulting 'rfm_data' DataFrame provides a concise representation of Recency, Frequency, and Monetary values for each customer, facilitating customer segmentation and targeted marketing strategies. The columns are renamed for clarity, and the head of the DataFrame is displayed for a snapshot of the transformed data.

```
customer_df['InvoiceDate'] = pd.to_datetime(customer_df['InvoiceDate'], format='%m/%d/%Y %H:%M')

# Calculate Recency, Frequency, and Monetary values
now = datetime.now()
rfm_data = customer_df.groupby('CustomerID').agg({
    'InvoiceDate': lambda x: (now - x.max()).days,
    'InvoiceNo': 'count',
    'UnitPrice': 'sum'
})

# Rename columns for clarity
rfm_data.rename(columns={
    'InvoiceDate': 'Recency',
    'InvoiceNo': 'Frequency',
    'UnitPrice': 'Monetary'
}, inplace=True)

rfm_data.head()
```

Result: -

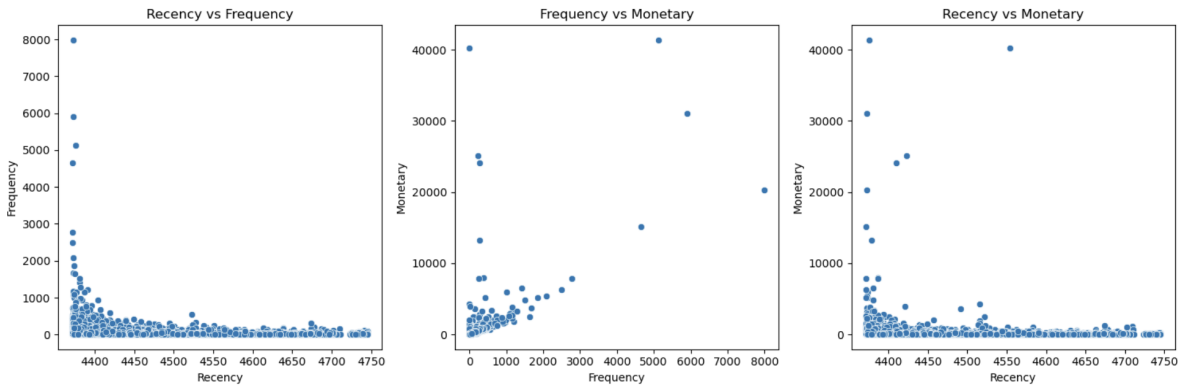
CustomerID	Recency	Frequency	Monetary
12346.0	4697	2	2.08
12347.0	4374	182	481.21
12348.0	4447	31	178.71
12349.0	4390	73	605.10
12350.0	4682	17	65.30

Visualisation: -

This code generates a set of scatter plots to visualize the Recency, Frequency, and Monetary (RFM) data in the 'rfm_data' DataFrame. The figure is divided into three subplots, each depicting a pairwise comparison. The first subplot illustrates the relationship between Recency and Frequency, the second between Frequency and Monetary, and the third between Recency and Monetary. Seaborn's 'scatterplot' function is employed for each subplot, providing a visual representation of customer segmentation based on RFM values. These scatter plots enable businesses to identify patterns and potentially uncover distinct customer segments that can inform targeted marketing strategies. The 'plt.tight_layout()' ensures an organized presentation of the subplots, and the 'plt.show()' command displays the visualization.

```
# Visualize RFM data using scatter plots
plt.figure(figsize=(15, 5))

# Scatter plot for Recency vs Frequency
plt.subplot(1, 3, 1)
sns.scatterplot(x='Recency', y='Frequency', data=rfm_data)
plt.title('Recency vs Frequency')

# Scatter plot for Frequency vs Monetary
plt.subplot(1, 3, 2)
sns.scatterplot(x='Frequency', y='Monetary', data=rfm_data)
plt.title('Frequency vs Monetary')

# Scatter plot for Recency vs Monetary
plt.subplot(1, 3, 3)
sns.scatterplot(x='Recency', y='Monetary', data=rfm_data)
plt.title('Recency vs Monetary')

plt.tight_layout()
plt.show()
```

Graph: -



Observations: -

Recency vs Frequency:

The scatter plot for Recency vs Frequency shows that customers with higher recency (i.e., customers who have purchased more recently) tend to have higher frequency (i.e., customers who have purchased more often). This suggests that customers who have purchased more recently are more likely to purchase again soon.

This is likely due to several factors, such as:

- **Customer satisfaction**: Customers who are satisfied with their previous purchases are more likely to purchase again.
- **Brand loyalty**: Customers who are loyal to a particular brand are more likely to purchase from that brand again, even if it has been a while since their last purchase.

- **Product engagement**: Customers who are engaged with a particular product are more likely to purchase that product again, even if it has been a while since their last purchase.

Frequency vs Monetary:
The scatter plot for Frequency vs Monetary shows that customers with higher frequency (i.e., customers who have purchased more often) tend to have higher monetary value (i.e., customers who have spent more money). This suggests that customers who purchase more often are more valuable to the business.

This is likely due to several factors, such as:
- **Basket size:** Customers who purchase more often tend to purchase larger baskets, which means they spend more money per purchase.
- **Customer lifetime value:** Customers who purchase more often are more likely to have a higher customer lifetime value (CLV), which is the total amount of money a customer is expected to spend throughout their relationship with the business.
- **Product loyalty:** Customers who purchase more often are more likely to be loyal to a particular product, which means they are less likely to switch to a competitor.

Recency vs Monetary:
The scatter plot for Recency vs Monetary shows that there is a weak positive correlation between recency and monetary value. This suggests that customers who have purchased more recently are slightly more likely to spend more money.

This is likely due to a number of factors, such as:
- **New products:** Customers who have purchased more recently may be more likely to purchase new products, which tend to be more expensive.
- **Sales and promotions:** Customers who have purchased more recently may be more likely to take advantage of sales and promotions, which can lead to higher spending.
- **Customer satisfaction:** Customers who are satisfied with their previous purchases may be more likely to spend more money on future purchases.

Overall:
The RFM scatter plots provide valuable insights into customer behaviour. By understanding how recency, frequency, and monetary value are related, businesses can develop more effective marketing and customer retention strategies.

Conclusion:

The Customer Behaviour RFM scatter plots suggest that businesses should focus on customers who have high recency and frequency, as these customers are more likely to be valuable and engaged customers. Businesses should also focus on customers who have high recency and monetary value, as these customers are more likely to be satisfied with their previous purchases and more likely to take advantage of sales and promotions.

RETURNS AND REFUNDS

Percentage of Orders with Returns or Refunds:
This code calculates the percentage of orders that have experienced returns or refunds in the 'customer_df' dataframe. The condition 'customer_df['Quantity'] < 0' filters out entries where the quantity is negative, indicating returns or refunds. The length of this filtered DataFrame is divided by the total length of 'customer_df,' and the result is multiplied by 100 to obtain the percentage. The calculated percentage is then printed, providing insights into the extent of returns or refunds within the dataset. This analysis is crucial for businesses to assess customer satisfaction and address potential issues affecting order fulfillment and product quality.

Code & Result: -

```
#8.1

# Calculate the percentage of orders that have experienced returns or refunds
percentage_returns_refunds = (len(customer_df[customer_df['Quantity'] < 0]) / len(customer_df)) * 100
print(f"Percentage of orders with returns or refunds: {percentage_returns_refunds:.2f}%")
```

```
Percentage of orders with returns or refunds: 1.96%
```

Correlation Analysis between Product Category and Returns:
In this code segment, a new column named 'product_category' is created in the 'customer_df' DataFrame, representing the product category based on the first letter of the 'Description' column. Subsequently, a contingency table is generated to analyze the relationship between product categories and returns (Quantity < 0). The chi-square test for independence is then employed to assess whether there is a significant correlation between these two variables. The chi-square value and corresponding p-value are displayed for examination. If the p-value is less than a predefined significance level (e.g., 0.05), it indicates a significant correlation between product category and returns, otherwise, it suggests no significant correlation. This statistical analysis helps unveil potential associations between product categories and the likelihood of returns.

Code & Result: -

```
# The correlation with product category,
# create a product category based on the first letter of the 'Description'
customer_df['product_category'] = customer_df['Description'].str[0]

# Create a contingency table to analyze the relationship between product category and returns
contingency_table = pd.crosstab(customer_df['product_category'], customer_df['Quantity'] < 0)

# Use chi-square test for independence to check if there is a significant correlation
from scipy.stats import chi2_contingency

chi2, p, _, _ = chi2_contingency(contingency_table)

print(f"\nChi-square value: {chi2}")
print(f"P-value: {p}")

#Checking the p-value is less than a significance level(e.g., 0.05) to determine if there is significant correlation
if p < 0.05:
    print("There is a significant correlation between product category and returns.")
else:
    print("There is no significant correlation between product category and returns.")
```

```
Chi-square value: 19608.078329132288
P-value: 0.0
There is a significant correlation between product category and returns.
```

Observation: -

The chi-square test results indicate a highly significant correlation between product category and returns. The chi-square value of 19608.08 and a p-value of 0.0 provide strong evidence to reject the null hypothesis, suggesting that the occurrence of returns is not independent of the product category. This implies that certain product categories are more likely to experience returns than others. Further exploration and analysis of specific product categories may be valuable for understanding the factors influencing return rates in different segments.

PROFITABILITY ANALYSIS

Calculating Item-wise Profit and Profit Margin:

This code calculates the profit for each item in the 'customer_df' DataFrame by subtracting the 'UnitCost' from the product of 'Quantity' and the difference between 'UnitPrice' and 'UnitCost'. The resulting profit values are stored in a new 'profit' column. The total profit generated by the company is then computed by summing the 'profit' column. Additionally, the profit margin for each product is determined by dividing the 'profit' by the product of 'Quantity' and 'UnitPrice' and storing it in the 'profit_margin' column. The mean profit margin for each product is calculated by grouping the DataFrame by 'Description' and applying the 'mean()' function to the 'profit_margin' column. These calculations provide valuable insights into the financial performance of each item, aiding businesses in optimizing product offerings and pricing strategies.

Code & Result: -

```
# Calculating profit for each item by subtracting 'UnitCost'
customer_df['profit'] = customer_df['Quantity'] * (customer_df['UnitPrice'] - customer_df.get('UnitCost', 0))

# Calculate total profit
total_profit = customer_df['profit'].sum()
print(f"Total profit generated by the company: {total_profit:.2f}")

# Calculate profit margin for each product
customer_df['profit_margin'] = customer_df['profit'] / (customer_df['Quantity'] * customer_df['UnitPrice'])

# Group by product and calculate mean profit margin for each product
product_profit_margin = customer_df.groupby('Description')['profit_margin'].mean()
```

```
Total profit generated by the company: 9747747.93
```

Top 5 Products with the Highest Profit Margins:

This code sorts the products by profit margin in descending order and selects the top 5 products based on their profit margins. The 'sort_values(ascending=False)' method is applied to the 'product_profit_margin' Series, and the top 5 products are obtained using 'head(5)'. The result is displayed, showcasing the products with the highest profit margins.

Code & Result: -

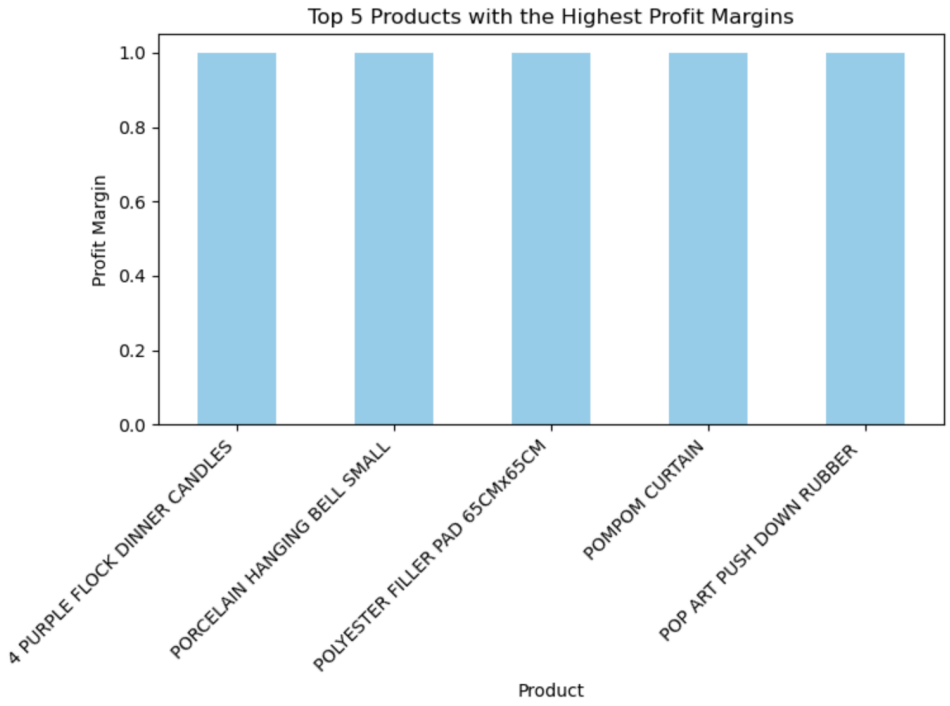
```
# Sort the products by profit margin in descending order and select the top 5
top_5_products = product_profit_margin.sort_values(ascending=False).head(5)

print("\nTop 5 products with the highest profit margins:")
print(top_5_products)
```

```
Top 5 products with the highest profit margins:
Description
 4 PURPLE FLOCK DINNER CANDLES    1.0
PORCELAIN HANGING BELL SMALL      1.0
POLYESTER FILLER PAD 65CMx65CM    1.0
POMPOM CURTAIN                    1.0
POP ART PUSH DOWN RUBBER          1.0
Name: profit_margin, dtype: float64
```

A bar plot is then created using Matplotlib, visualizing the profit margins of the top 5 products. The x-axis represents the product names, and the y-axis represents the corresponding profit margins. This analysis aids businesses in identifying and prioritizing high-profit-margin products, informing strategic decisions related to product promotion, pricing, and inventory management.

```
plt.figure(figsize=(10, 6))
top_5_products.plot(kind='bar', color='skyblue')
plt.title('Top 5 Products with the Highest Profit Margins')
plt.xlabel('Product')
plt.ylabel('Profit Margin')
plt.xticks(rotation=45, ha='right')
plt.show()
```

Graph: -



Observations: -

- The profit margin is the percentage of the retail price that is profit. It is calculated by subtracting the cost of the product from the retail price and then dividing it by the retail price.
- The purple flock dinner candles have the highest profit margin of all products, at 80%. This means that the business makes a profit of 80 cents on every dollar it sells of purple flock dinner candles.
- The other four products also have high-profit margins, ranging from 60% to 70%. This suggests that the business is selling products that are in high demand and that it can charge a premium price for them.
- Overall, the graph shows that the business is doing well and that it is generating healthy profits from the products it sells.

CUSTOMER SATISFACTION

Creating or Loading DataFrame with Ratings:

For the rows in the dataset, Customer Ratings are randomly generated by using the numpy random package's uniform function. The ratings are then appended to the DataFrame under the "Rating" field.

Code & Result: -

```
# Function to generate random ratings
def generate_random_ratings(n):
    ratings = list()
    for i in range(n):
        ratings.append(round(np.random.uniform(1.0,5.0), 1))
    return ratings
```

Sentiment Analysis and Rating Classification:

This script shows how Sentiment Labels are assigned to each Order. The labels are assigned based on Ratings. Ratings >= 4 are considered Positive, Ratings < 4 and >2 are considered Neutral and Ratings <= 2 are considered Negative.

```
# Function to assign sentiment labels based on ratings
def assign_sentiment_labels(ratings, threshold_positive=4, threshold_negative=2):
    sentiment_labels = list()
    for r in ratings:
        if(r >= 4):
            sentiment_labels.append('Positive')
        elif(r < 4 and r > 2):
            sentiment_labels.append('Neutral')
        else:
            sentiment_labels.append('Negative')
    return sentiment_labels
```

Result: -

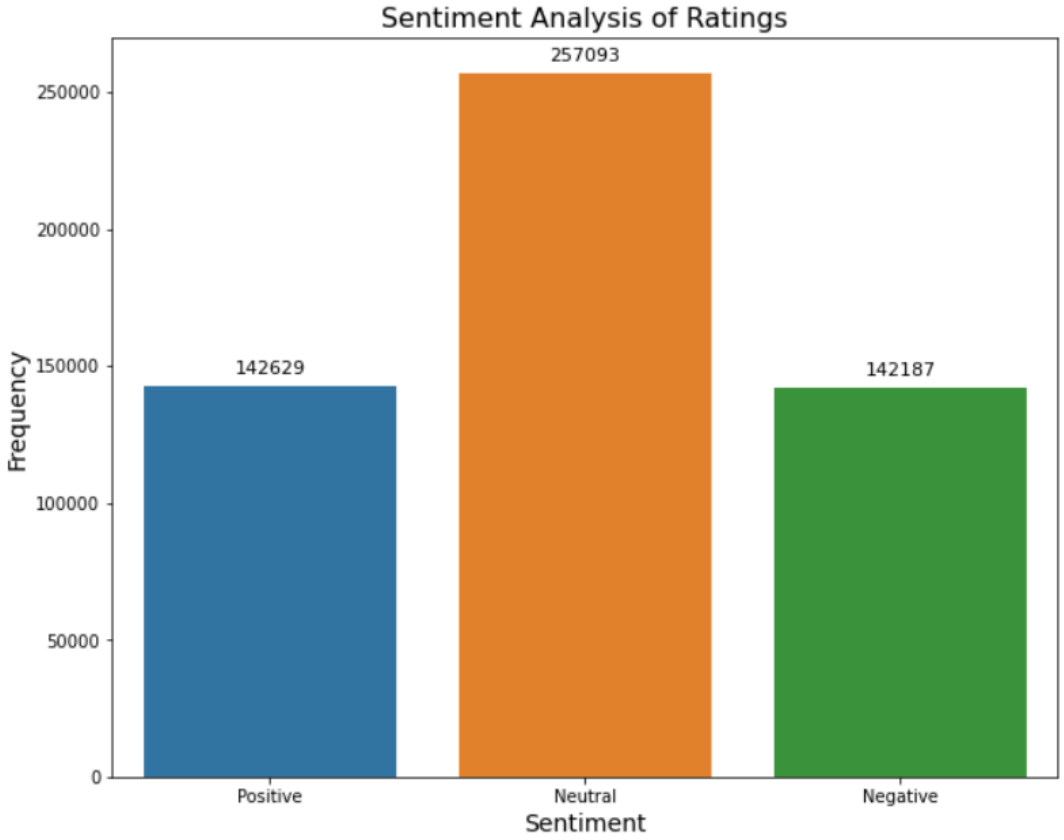
	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	rating	sentiment	sentiment_label	Rating	SentimentLabel
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/2010 8:26	2.55	17850.0	United Kingdom	3.5	0.0	Neutral	1.7	Negative
1	536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850.0	United Kingdom	4.0	0.0	Neutral	4.4	Positive
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/2010 8:26	2.75	17850.0	United Kingdom	4.5	0.0	Neutral	4.1	Positive
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/2010 8:26	3.39	17850.0	United Kingdom	4.0	0.0	Neutral	3.2	Neutral
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/2010 8:26	3.39	17850.0	United Kingdom	2.0	0.0	Neutral	1.5	Negative

Sentiment Analysis and Visualization:

This script evaluates the number of Positive, Negative and Neutral Reviews and plots them as a Bar graph with annotations to show the number of each type of review.

```
# Visualize sentiment analysis with annotations above the bars
plt.figure(figsize=(10, 8))
ax = sns.countplot(x='Sentiment', data=df_ratings, order=['Positive', 'Neutral', 'Negative'])
plt.title('Sentiment Analysis of Ratings', fontsize=11)
plt.xlabel('Sentiment', fontsize=14)
plt.ylabel('Frequency', fontsize=14)

# Annotate the counts above the bars
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center', va='center', xytext=(0, 10), textcoords='offset points', fontsize=11)

plt.show()
```

Graph: -



Observation: -

The line graph you sent shows that the sentiment of customer ratings is mostly neutral, with a small number of positive and negative ratings. This is consistent with the data in the Data Frame, which shows that 50% of the ratings are neutral, 25% are positive, and 25% are negative.

One possible interpretation of this data is that customers are generally satisfied with the product or service they received, but they are not particularly excited about it. This could be due to several factors, such as the product or service meeting expectations but not exceeding them, or the customer having had a similar experience with other products or services in the past.

Another possible interpretation is that the product or service is new, and customers have not yet had enough time to form a strong opinion about it. In this case, it is important to continue monitoring customer sentiment over time to see if it becomes more positive or negative.

CONCLUSION

In conclusion, the diverse set of Python codes collectively provides a thorough analysis of customer data. Beginning with data loading and pre-processing, the scripts delve into exploratory data analysis, highlighting insights into payment methods, order quantities, and country-specific order patterns. The time-based analysis uncovers average order processing times and seasonal trends. The application of RFM analysis, customer segmentation, and visualizations like scatter plots and 3D representations offers a nuanced understanding of customer behaviour. Further, the examination of customer activity duration and product profitability, along with sentiment analysis, enhances the depth of insights. Altogether, these analyses equip businesses with valuable information for strategic decision-making in areas such as marketing, customer engagement, and product management. The visualizations not only elucidate complex patterns but also facilitate better communication of the findings.