custsegtb

May 12, 2024

P8 Building Customer Segmentation Models using Python

Full Name: Tushar Madhukar Bhagat

Internship Registraion ID: INTERNSHIP_170003939765548ae545859

Problem:

In this project, we delve deep into the thriving sector of **online retail** by analyzing a **transactional dataset** from a UK-based retailer, available at the UCI Machine Learning Repository. This dataset documents all transactions between 2010 and 2011. Our primary objective is to amplify the efficiency of marketing strategies and boost sales through **customer segmentation**. We aim to transform the transactional data into a customer-centric dataset by creating new features that will facilitate the segmentation of customers into distinct groups using the **K-means clustering** algorithm. This segmentation will allow us to understand the distinct **profiles** and preferences of different customer groups. Building upon this, we intend to develop a **recommendation system** that will suggest top-selling products to customers within each segment who haven't purchased those items yet, ultimately enhancing marketing efficacy and fostering increased sales.

Objectives:

- **Data Cleaning & Transformation**: Clean the dataset by handling missing values, duplicates, and outliers, preparing it for effective clustering.
- **Feature Engineering**: Develop new features based on the transactional data to create a customer-centric dataset, setting the foundation for customer segmentation.
- **Data Preprocessing**: Undertake feature scaling and dimensionality reduction to streamline the data, enhancing the efficiency of the clustering process.
- **Customer Segmentation using K-Means Clustering**: Segment customers into distinct groups using K-means, facilitating targeted marketing and personalized strategies.
- **Cluster Analysis & Evaluation**: Analyze and profile each cluster to develop targeted marketing strategies and assess the quality of the clusters formed.
- **Recommendation System**: Implement a system to recommend best-selling products to customers within the same cluster who haven't purchased those products, aiming to boost sales and marketing effectiveness.

#

Step 1 | Setup and Initialization

[6]:
```
pip install yellowbrick
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: yellowbrick in
c:\users\admin\appdata\roaming\python\python39\site-packages (1.5)
Requirement already satisfied: matplotlib!=3.0.0,>=2.0.2 in
c:\programdata\anaconda3\lib\site-packages (from yellowbrick) (3.5.2)
Requirement already satisfied: scipy>=1.0.0 in
c:\programdata\anaconda3\lib\site-packages (from yellowbrick) (1.9.1)
Requirement already satisfied: cycler>=0.10.0 in
c:\programdata\anaconda3\lib\site-packages (from yellowbrick) (0.11.0)
Requirement already satisfied: numpy>=1.16.0 in
c:\programdata\anaconda3\lib\site-packages (from yellowbrick) (1.21.5)
Requirement already satisfied: scikit-learn>=1.0.0 in
c:\programdata\anaconda3\lib\site-packages (from yellowbrick) (1.0.2)
Requirement already satisfied: packaging>=20.0 in
c:\programdata\anaconda3\lib\site-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (21.3)
Requirement already satisfied: fonttools>=4.22.0 in
c:\programdata\anaconda3\lib\site-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (4.25.0)
Requirement already satisfied: pillow>=6.2.0 in
c:\programdata\anaconda3\lib\site-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (9.2.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
c:\programdata\anaconda3\lib\site-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (1.4.2)
Requirement already satisfied: python-dateutil>=2.7 in
c:\programdata\anaconda3\lib\site-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (2.8.2)
Requirement already satisfied: pyparsing>=2.2.1 in
c:\programdata\anaconda3\lib\site-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (3.0.9)
Requirement already satisfied: threadpoolctl>=2.0.0 in
c:\programdata\anaconda3\lib\site-packages (from scikit-
learn>=1.0.0->yellowbrick) (2.2.0)
Requirement already satisfied: joblib>=0.11 in
c:\programdata\anaconda3\lib\site-packages (from scikit-
learn>=1.0.0->yellowbrick) (1.1.0)
Requirement already satisfied: six>=1.5 in c:\programdata\anaconda3\lib\site-
packages (from python-dateutil>=2.7->matplotlib!=3.0.0,>=2.0.2->yellowbrick)
(1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

[7]:
```
# importing necessary library
```

```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import matplotlib.gridspec as gridspec
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import colors as mcolors
from scipy.stats import linregress
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score, calinski_harabasz_score,␣
↪davies_bouldin_score
from sklearn.cluster import KMeans
from tabulate import tabulate
from collections import Counter
from yellowbrick.cluster import KElbowVisualizer, SilhouetteVisualizer
%matplotlib inline
```

[8]:
```
#Configuring Seaborn plot styles setting background colorb and grid

sns.set(rc={'axes.facecolor': '#89CFF0'}, style='darkgrid')
```

[9]:
```
#loading the dataset

df = pd.read_csv(r'C:\Users\admin\Pictures\customer_data_internship.csv'␣
↪,encoding='ISO-8859-1')
```

#

Step 2 | Initial Data Analysis

[10]:
```
#Step 2 Intial Data Analysis
```

[11]:
```
df.head(10)
```

[11]:
```
  InvoiceNo StockCode                          Description  Quantity  \
0    536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1    536365     71053                  WHITE METAL LANTERN         6
2    536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3    536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4    536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6
5    536365     22752         SET 7 BABUSHKA NESTING BOXES         2
6    536365     21730    GLASS STAR FROSTED T-LIGHT HOLDER         6
```

```
7    536366     22633             HAND WARMER UNION JACK         6
8    536366     22632          HAND WARMER RED POLKA DOT         6
9    536367     84879       ASSORTED COLOUR BIRD ORNAMENT       32

      InvoiceDate  UnitPrice  CustomerID         Country
0  12/1/2010 8:26       2.55     17850.0  United Kingdom
1  12/1/2010 8:26       3.39     17850.0  United Kingdom
2  12/1/2010 8:26       2.75     17850.0  United Kingdom
3  12/1/2010 8:26       3.39     17850.0  United Kingdom
4  12/1/2010 8:26       3.39     17850.0  United Kingdom
5  12/1/2010 8:26       7.65     17850.0  United Kingdom
6  12/1/2010 8:26       4.25     17850.0  United Kingdom
7  12/1/2010 8:28       1.85     17850.0  United Kingdom
8  12/1/2010 8:28       1.85     17850.0  United Kingdom
9  12/1/2010 8:34       1.69     13047.0  United Kingdom
```

[12]:
```
df.info()
```

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

[13]:
```
# Summary of Dataset for Numerical Values

df.describe()
```

[13]:

	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

[14]:
```
# Summary of Dataset for Categorical Values

df.describe(include='object').T
```

[14]:

	count	unique	top	freq
InvoiceNo	541909	25900	573585	1114
StockCode	541909	4070	85123A	2313
Description	540455	4223	WHITE HANGING HEART T-LIGHT HOLDER	2369
InvoiceDate	541909	23260	10/31/2011 14:41	1114
Country	541909	38	United Kingdom	495478

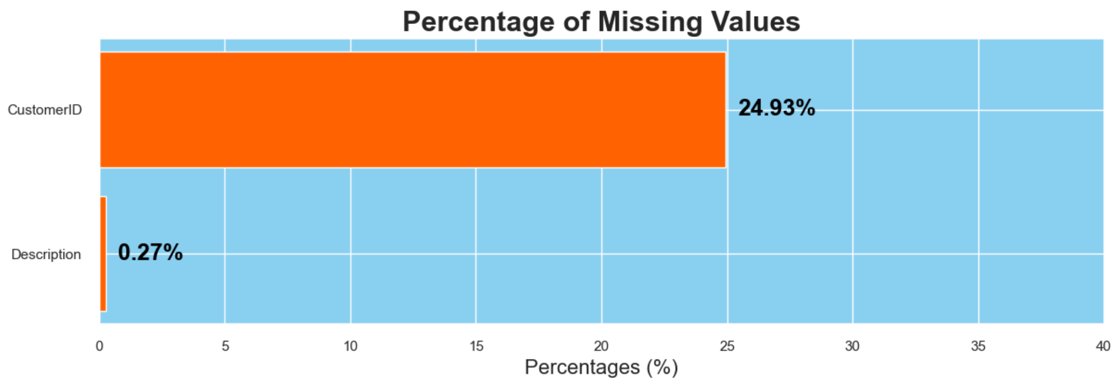
#

Step 3 | Data Cleaning & Transformation

Step 3.1 | Handling Missing Values

[15]:
```
# Calculating the percentage of missing values for each column
missing_data = df.isnull().sum()
missing_percentage = (missing_data[missing_data > 0] / df.shape[0]) * 100

# Preparing values

missing_percentage.sort_values(ascending=True, inplace=True)


# Plot the barh chart

fig, ax = plt.subplots(figsize=(14, 4))
ax.barh(missing_percentage.index, missing_percentage, color='#ff6200')


# Annotate the values and indexes

for i, (value, name) in enumerate(zip(missing_percentage, missing_percentage.
 ↪index)):
    ax.text(value+0.5, i, f"{value:.2f}%", ha='left', va='center',␣
 ↪fontweight='bold', fontsize=18, color='black')


# Set x-axis limit

ax.set_xlim([0, 40])


# Add title and xlabel

plt.title("Percentage of Missing Values", fontweight='bold', fontsize=22)
```

```
plt.xlabel('Percentages (%)', fontsize=16)
plt.show()
```



Handling Missing Values Strategy:

- **CustomerID (24.93% missing values)**
  - The CustomerID column contains nearly a 25% of missing data. This column is essential for clustering customers and creating a recommendation system. Imputing such a large percentage of missing values might introduce significant bias or noise into the analysis.
- **Description (0.27% missing values)**
  - The Description column has a minor percentage of missing values. However, it has been noticed that there are inconsistencies in the data where the same StockCode does not always have the same Description. This indicates data quality issues and potential errors in the product descriptions.
  - Given these inconsistencies, imputing the missing descriptions based on StockCode might not be reliable. Moreover, since the missing percentage is quite low, it would be prudent to remove the rows with missing Descriptions to avoid propagating errors and inconsistencies into the subsequent analyses.

By removing rows with missing values in the CustomerID and Description columns, we aim to construct a cleaner and more reliable dataset, which is essential for achieving accurate clustering and creating an effective recommendation system.

[16]:
```
# Extracting rows with missing values in 'CustomerID' or 'Description' columns

df[df['CustomerID'].isnull() | df['Description'].isnull()]
```

[16]:
```
        InvoiceNo StockCode                      Description  Quantity  \
622        536414     22139                              NaN        56
1443       536544     21773  DECORATIVE ROSE BATHROOM BOTTLE         1
1444       536544     21774  DECORATIVE CATS BATHROOM BOTTLE         2
1445       536544     21786               POLKADOT RAIN HAT          4
1446       536544     21787            RAIN PONCHO RETROSPOT         2
…             …       …                                …       …
541536     581498    85099B           JUMBO BAG RED RETROSPOT         5
```

```
541537    581498    85099C   JUMBO  BAG BAROQUE BLACK WHITE         4
541538    581498     85150    LADIES & GENTLEMEN METAL SIGN         1
541539    581498     85174                 S/4 CACTI CANDLES         1
541540    581498       DOT                    DOTCOM POSTAGE         1

             InvoiceDate  UnitPrice  CustomerID         Country
622      12/1/2010 11:52       0.00         NaN  United Kingdom
1443     12/1/2010 14:32       2.51         NaN  United Kingdom
1444     12/1/2010 14:32       2.51         NaN  United Kingdom
1445     12/1/2010 14:32       0.85         NaN  United Kingdom
1446     12/1/2010 14:32       1.66         NaN  United Kingdom
…                    …          …         …             …
541536   12/9/2011 10:26       4.13         NaN  United Kingdom
541537   12/9/2011 10:26       4.13         NaN  United Kingdom
541538   12/9/2011 10:26       4.96         NaN  United Kingdom
541539   12/9/2011 10:26      10.79         NaN  United Kingdom
541540   12/9/2011 10:26    1714.17         NaN  United Kingdom

[135080 rows x 8 columns]
```

[17]:
```
# Removing rows with missing values in 'CustomerID' and 'Description' columns

df = df.dropna(subset=['CustomerID','Description'])
```

[18]:
```
df.sample(10)
```

[18]:
```
       InvoiceNo StockCode                          Description  Quantity  \
321786    565201     22625                   RED KITCHEN SCALES         2
233245    557466     23307  SET OF 60 PANTRY DESIGN CAKE CASES         24
144017    548715     22272                  FELTCRAFT DOLL MARIA         3
81358     543123     20718             RED RETROSPOT SHOPPER BAG        10
315310    564725    84558A          3D DOG PICTURE PLAYING CARDS         1
349157    567482    85014B              RED RETROSPOT UMBRELLA         2
78185     542836    85099B               JUMBO BAG RED RETROSPOT         6
26176     538507     22791          T-LIGHT GLASS FLUTED ANTIQUE        15
261860    559893     22979               PANTRY WASHING UP BRUSH         4
330028    565865     23289             DOLLY GIRL CHILDRENS BOWL         8

             InvoiceDate  UnitPrice  CustomerID         Country
321786    9/1/2011 16:41       8.50     14907.0  United Kingdom
233245   6/20/2011 13:08       0.55     13815.0         Germany
144017    4/3/2011 15:22       2.95     17758.0  United Kingdom
81358     2/3/2011 14:08       1.25     18178.0  United Kingdom
315310   8/28/2011 12:28       2.95     18041.0  United Kingdom
349157   9/20/2011 13:42       5.95     16464.0  United Kingdom
78185     2/1/2011 11:31       1.95     16745.0  United Kingdom
26176   12/12/2010 13:26       1.25     15547.0  United Kingdom
```

```
261860   7/13/2011 12:06        1.45      14498.0  United Kingdom
330028    9/7/2011 15:07        1.25      12637.0          France
```

[19]:
```
# Verifying the removal of missing values

df.isnull().sum().sum()
```

[19]:
```
0
```

Step 3.2 | Handling Duplicates

[20]:
```
# Finding duplicate rows while keeping all instances

duplicate_rows = df[df.duplicated(keep=False)]


# Sorting the data by certain columns to see the duplicate rows next to each
↪other

duplicate_rows_sorted = duplicate_rows.sort_values(by=['InvoiceNo',
↪'StockCode', 'Description', 'CustomerID', 'Quantity'])


# Displaying the first 10 records


duplicate_rows_sorted.head(20)
```

[20]:
```
    InvoiceNo StockCode                          Description  Quantity  \
494    536409     21866          UNION JACK FLAG LUGGAGE TAG         1
517    536409     21866          UNION JACK FLAG LUGGAGE TAG         1
485    536409     22111         SCOTTIE DOG HOT WATER BOTTLE         1
539    536409     22111         SCOTTIE DOG HOT WATER BOTTLE         1
489    536409     22866        HAND WARMER SCOTTY DOG DESIGN         1
527    536409     22866        HAND WARMER SCOTTY DOG DESIGN         1
521    536409     22900      SET 2 TEA TOWELS I LOVE LONDON          1
537    536409     22900      SET 2 TEA TOWELS I LOVE LONDON          1
578    536412     21448            12 DAISY PEGS IN WOOD BOX         1
598    536412     21448            12 DAISY PEGS IN WOOD BOX         1
565    536412     21448            12 DAISY PEGS IN WOOD BOX         2
601    536412     21448            12 DAISY PEGS IN WOOD BOX         2
604    536412     21448            12 DAISY PEGS IN WOOD BOX         2
612    536412     21706  FOLDING UMBRELLA RED/WHITE POLKADOT         1
618    536412     21706  FOLDING UMBRELLA RED/WHITE POLKADOT         1
607    536412     21708      FOLDING UMBRELLA CREAM POLKADOT         1
616    536412     21708      FOLDING UMBRELLA CREAM POLKADOT         1
574    536412     22141        CHRISTMAS CRAFT TREE TOP ANGEL         1
```

```
594    536412     22141       CHRISTMAS CRAFT TREE TOP ANGEL         1
556    536412     22273                 FELTCRAFT DOLL MOLLY         1

          InvoiceDate  UnitPrice  CustomerID         Country
494  12/1/2010 11:45       1.25     17908.0  United Kingdom
517  12/1/2010 11:45       1.25     17908.0  United Kingdom
485  12/1/2010 11:45       4.95     17908.0  United Kingdom
539  12/1/2010 11:45       4.95     17908.0  United Kingdom
489  12/1/2010 11:45       2.10     17908.0  United Kingdom
527  12/1/2010 11:45       2.10     17908.0  United Kingdom
521  12/1/2010 11:45       2.95     17908.0  United Kingdom
537  12/1/2010 11:45       2.95     17908.0  United Kingdom
578  12/1/2010 11:49       1.65     17920.0  United Kingdom
598  12/1/2010 11:49       1.65     17920.0  United Kingdom
565  12/1/2010 11:49       1.65     17920.0  United Kingdom
601  12/1/2010 11:49       1.65     17920.0  United Kingdom
604  12/1/2010 11:49       1.65     17920.0  United Kingdom
612  12/1/2010 11:49       4.95     17920.0  United Kingdom
618  12/1/2010 11:49       4.95     17920.0  United Kingdom
607  12/1/2010 11:49       4.95     17920.0  United Kingdom
616  12/1/2010 11:49       4.95     17920.0  United Kingdom
574  12/1/2010 11:49       2.10     17920.0  United Kingdom
594  12/1/2010 11:49       2.10     17920.0  United Kingdom
556  12/1/2010 11:49       2.95     17920.0  United Kingdom
```

Handling Duplicates Strategy:

In the context of this project, the presence of completely identical rows, including identical transaction times, suggests that these might be data recording errors rather than genuine repeated transactions. Keeping these duplicate rows can introduce noise and potential inaccuracies in the clustering and recommendation system.

Therefore, I am going to remove these completely identical duplicate rows from the dataset. Removing these rows will help in achieving a cleaner dataset, which in turn would aid in building more accurate customer clusters based on their unique purchasing behaviors. Moreover, it would help in creating a more precise recommendation system by correctly identifying the products with the most purchases.

[21]:
```python
# Displaying the number of duplicate rows

print(f"The dataset contains {df.duplicated().sum()} duplicate rows that need␣
↪to be removed.")

# Removing duplicate rows

df.drop_duplicates(inplace=True)
```

```
The dataset contains 5225 duplicate rows that need to be removed.
```

[22]:
```
df.shape[0]
```

[22]: 401604

Step 3.3 | Treating Cancelled Transactions

To refine our understanding of customer behavior and preferences, we need to take into account the transactions that were cancelled. Initially, we will identify these transactions by filtering the rows where the `InvoiceNo` starts with "C". Subsequently, we will analyze these rows to understand their common characteristics or patterns:

[23]:
```
# Filter out the rows with InvoiceNo starting with "C" and create a new column␣
↪indicating the transaction status

df['Transaction_Status'] = np.where(df['InvoiceNo'].astype(str).str.
↪startswith('C'), 'Cancelled', 'Completed')
'''
C14245345  Cancelled
35365563   Completed
'''

# Analyze the characteristics of these rows by analysing the new column

cancelled_transactions = df[df['Transaction_Status'] == 'Cancelled']
cancelled_transactions.describe()
```

[23]:

	Quantity	UnitPrice	CustomerID
count	8872.000000	8872.000000	8872.000000
mean	-30.774910	18.899512	14990.152953
std	1172.249902	445.190864	1708.230387
min	-80995.000000	0.010000	12346.000000
25%	-6.000000	1.450000	13505.000000
50%	-2.000000	2.950000	14868.000000
75%	-1.000000	4.950000	16393.000000
max	-1.000000	38970.000000	18282.000000

Inferences from the Cancelled Transactions Data:

- All the quantaties from the cancelled transcation are negative , indicating that these are the orders which are cancelled by the Customer.
- The UnitPrice column shows the variety of items which are spread across different categories from low to medium to high value items in cancelled transcations.

Strategy for Handling Cancelled Transactions:

Considering the project's objective to cluster customers based on their purchasing behavior and preferences and to eventually create a recommendation system, it's imperative to understand the cancellation patterns of customers. Therefore, the strategy is to retain these cancelled transactions in the dataset, marking them distinctly to facilitate further analysis. This approach will:

- Enhance the clustering process by incorporating patterns and trends observed in cancellation data, which might represent certain customer behaviors or preferences.
- Allow the recommendation system to possibly prevent suggesting products that have a high likelihood of being cancelled, thereby improving the quality of recommendations.

[24]:
```
# Finding the percentage of cancelled transactions

cancelled_percentage = (cancelled_transactions.shape[0] / df.shape[0]) * 100


# Printing the percentage of cancelled transactions

print(f"The percentage of cancelled transactions in the dataset is:␣
↪{cancelled_percentage:.2f}%")
```

```
The percentage of cancelled transactions in the dataset is: 2.21%
```

Step 3.4 | Correcting StockCode Anomalies

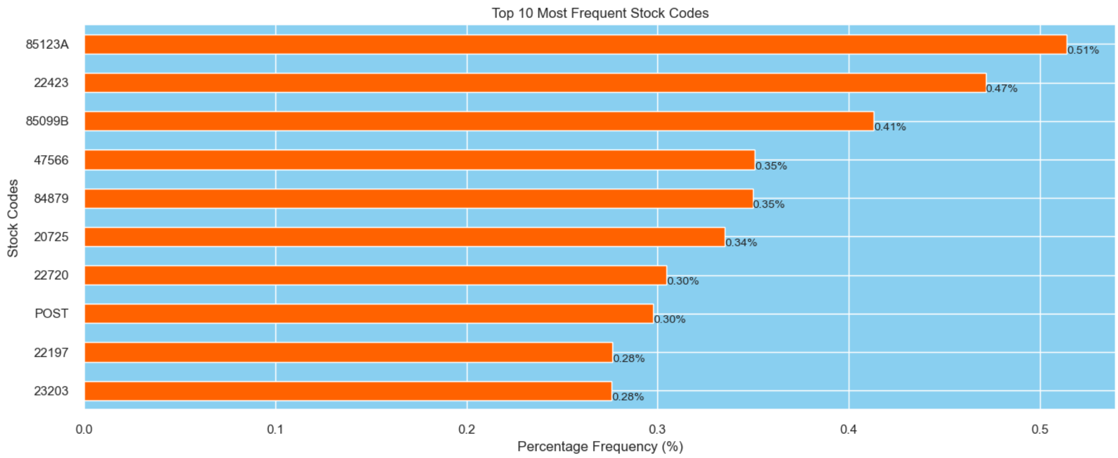
First of all, lets find the number of unique stock codes and to plot the top 10 most frequent stock codes along with their percentage frequency:

[25]:
```
# Finding the number of unique stock codes


unique_stock_codes = df['StockCode'].nunique()


# Printing the number of unique stock codes
print(f"The number of unique stock codes in the dataset is:␣
↪{unique_stock_codes}")
```

```
The number of unique stock codes in the dataset is: 3684
```

[26]:
```
# Finding the top 10 most frequent stock codes

top_10_stock_codes = df['StockCode'].value_counts(normalize=True).head(10) * 100


# Plotting the top 10 most frequent stock codes

plt.figure(figsize=(16, 6))
top_10_stock_codes.plot(kind='barh', color='#ff6200')


# Adding the percentage frequency on the bars

for index, value in enumerate(top_10_stock_codes):
```

```
    plt.text(value, index+0.25, f'{value:.2f}%', fontsize=10)

plt.title('Top 10 Most Frequent Stock Codes')
plt.xlabel('Percentage Frequency (%)')
plt.ylabel('Stock Codes')
plt.gca().invert_yaxis()
plt.show()
```



Inferences on Stock Codes:

- **Product Variety** : The dataset have 3684 unique stock codes. It indicates that the wide variety of available items, products on sale. This might be to attract wide variety of customers to the retail store according to there needs and preference of product.
- **Popular items** : The top 10 frequent stock codes represent the items which are people more likely and frequently tend to purchase from these categories from total offering.
- **Stock Code Anamolies** : While observing the frequent stock codes i noticed that there is stock code named **POST** . As mentioned in the description of dataset the stock codes are made of 5 or 6 characters of number and letters. These anomalies is may be displaying some service by it's description like distpatch or deliver or some kind of charges or so. To maintain the target outcome of project and to maintain the data useful for further operations we have to look at the anamolies and has to remove from the dataset.

To dive deeper into identifying these anomalies, let's explore the frequency of the number of numeric characters in the stock codes, which can provide insights into the nature of these unusual entries:

[27]:
```
# Finding the number of numeric characters in each unique stock code

unique_stock_codes = df['StockCode'].unique()
numeric_char_counts_in_unique_codes = pd.Series(unique_stock_codes).
 ↪apply(lambda x: sum(c.isdigit() for c in str(x))).value_counts()
```

```
# Printing the value counts for unique stock codes

print("Value counts of numeric character frequencies in unique stock codes:")
print("-"*70)
print(numeric_char_counts_in_unique_codes)
```

```
Value counts of numeric character frequencies in unique stock codes:
----------------------------------------------------------------------
5    3676
0       7
1       1
dtype: int64
```

Inference:

The output indicates the following:

- The majority of stock code i.e 3676 out of 3684 contain exactly 5 numeric character. which is the standard format of representing the stock codes in the dataset.
- There are few anomalies are present exactly 7 out of 3684 stock codes has a Zero numeric value which is impossible because stock code should have at least one numeric value in it to represent, and one stock code with one numeric value which is quite surprising.

  Now let check the stock codes with 0 and 1 numeric character to understand these anamolies

[28]:
```
# Finding and printing the stock codes with 0 and 1 numeric characters

anomalous_stock_codes = [code for code in unique_stock_codes if sum(c.isdigit()␣
↪for c in str(code)) in (0, 1)]


# Printing each stock code on a new line

print("Anamolus stock codes:")
print("-"*22)
for code in anomalous_stock_codes:
    print(code)
```

```
Anamolus stock codes:
----------------------
POST
D
C2
M
BANK CHARGES
PADS
DOT
CRUK
```

Let's calculate the percentage of records with these anomalous stock codes:

[29]:
```python
# Calculating the percentage of records with these stock codes

percentage_anomalous = (df['StockCode'].isin(anomalous_stock_codes).sum() /␣
 ↪len(df)) * 100


# Printing the percentage

print(f"The percentage of records with anomalous stock codes in the dataset is:␣
 ↪{percentage_anomalous:.2f}%")
```

```
The percentage of records with anomalous stock codes in the dataset is: 0.48%
```

Inference:

Based on our analysis there **0.48%** anamolies are present in the dataset. Which are unusal from their format.

Strategy:

Given the context of the project, where the aim is to cluster customers based on their product purchasing behaviors and develop a product recommendation system, it would be prudent to exclude these records with anomalous stock codes from the dataset. This way, the focus remains strictly on genuine product transactions, which would lead to a more accurate and meaningful analysis.

Thus, the strategy would be to filter out and remove rows with these anomalous stock codes from the dataset before proceeding with further analysis and model development:

[30]:
```python
# Removing rows with anomalous stock codes from the dataset

df = df[~df['StockCode'].isin(anomalous_stock_codes)]
```

[31]:
```python
# Getting the number of rows in the dataframe

df.shape[0]
```

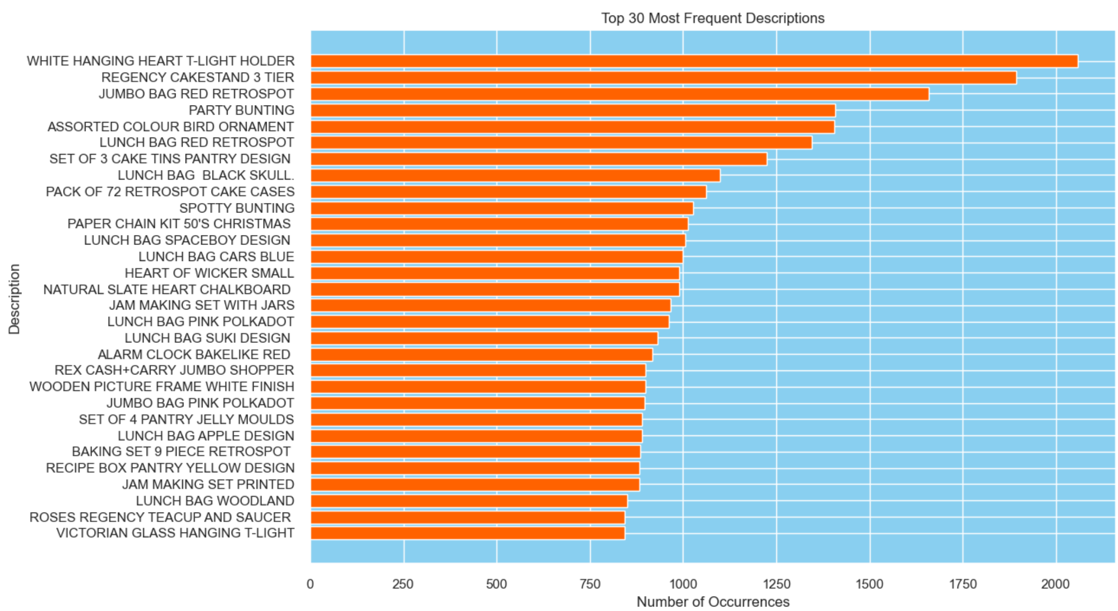
[31]:
```
399689
```

Step 3.5 | Cleaning Description Column

First, we will calculate the occurrence count of each unique description in the dataset. Then, we will plot the top 30 descriptions. This visualization will give a clear view of the highest occurring descriptions in the dataset:

[32]:
```python
# Calculate the occurrence of each unique description and sort them

description_counts = df['Description'].value_counts()

# Get the top 30 descriptions

top_30_descriptions = description_counts[:30]
```

```python
# Plotting

plt.figure(figsize=(12,8))
plt.barh(top_30_descriptions.index[::-1], top_30_descriptions.values[::-1],␣
 ↪color='#ff6200')

# Adding labels and title

plt.xlabel('Number of Occurrences')
plt.ylabel('Description')
plt.title('Top 30 Most Frequent Descriptions')

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



Inferences on Descriptions:

- The most frequent descriptions are generally household items, particularly those associated with kitchenware, lunch bags, and decorative items.
- The most frequent itmes holds a significant amount of orders (Occurrence) in the data
- Interestingly, all the descriptions are in uppercase, which might be a standardized format for entering product descriptions in the database. However, considering the inconsistencies and anomalies encountered in the dataset so far, it would be prudent to check if there are descriptions entered in lowercase or a mix of case styles.

[33]:
```python
# Find unique descriptions containing lowercase characters

lowercase_descriptions = df['Description'].unique()
print(lowercase_descriptions)
lowercase_descriptions = [desc for desc in lowercase_descriptions if any(char.
 ↪islower() for char in desc)]


# Print the unique descriptions containing lowercase characters

print("The unique descriptions containing lowercase characters are:")
print("-"*60)
for desc in lowercase_descriptions:
    print(desc)
```

```
['WHITE HANGING HEART T-LIGHT HOLDER' 'WHITE METAL LANTERN'
 'CREAM CUPID HEARTS COAT HANGER' … 'PINK CRYSTAL SKULL PHONE CHARM'
 'CREAM HANGING HEART T-LIGHT HOLDER' 'PAPER CRAFT , LITTLE BIRDIE']
The unique descriptions containing lowercase characters are:
------------------------------------------------------------
BAG 500g SWIRLY MARBLES
POLYESTER FILLER PAD 45x45cm
POLYESTER FILLER PAD 45x30cm
POLYESTER FILLER PAD 40x40cm
FRENCH BLUE METAL DOOR SIGN No
BAG 250g SWIRLY MARBLES
BAG 125g SWIRLY MARBLES
3 TRADITIONAl BISCUIT CUTTERS  SET
NUMBER TILE COTTAGE GARDEN No
FOLK ART GREETING CARD,pack/12
ESSENTIAL BALM 3.5g TIN IN ENVELOPE
POLYESTER FILLER PAD 65CMx65CM
NUMBER TILE VINTAGE FONT No
POLYESTER FILLER PAD 30CMx30CM
POLYESTER FILLER PAD 60x40cm
FLOWERS HANDBAG blue and orange
Next Day Carriage
THE KING GIFT BAG 25x24x12cm
High Resolution Image
```

Inference:

- Upon reviewing the descriptions that contain lowercase characters, it is evident that some entries are not product descriptions, such as "**Next Day Carriage**" and "**High Resolution Image**". These entries seem to be unrelated to the actual products and might represent other types of information or service details.

Strategy:

- **Step 1**: Remove the rows where the descriptions contain service-related information like "**Next Day Carriage**" and "**High Resolution Image**", as these do not represent actual products and would not contribute to the clustering and recommendation system we aim to build.
- **Step 2**: For the remaining descriptions with mixed case, standardize the text to uppercase to maintain uniformity across the dataset. This will also assist in reducing the chances of having duplicate entries with different case styles.

By implementing the above strategy, we can enhance the quality of our dataset, making it more suitable for the analysis and modeling phases of our project.

[34]:
```
service_related_descriptions = ["Next Day Carriage", "High Resolution Image"]

# Calculate the percentage of records with service-related descriptions

service_related_percentage = df[df['Description'].
 ↪isin(service_related_descriptions)].shape[0] / df.shape[0] * 100

# Print the percentage of records with service-related descriptions

print(f"The percentage of records with service-related descriptions in the␣
 ↪dataset is: {service_related_percentage:.2f}%")
```

```
The percentage of records with service-related descriptions in the dataset is:
0.02%
```

[35]:
```
# Remove rows with service-related information in the description

df = df[~df['Description'].isin(service_related_descriptions)]

# Standardize the text to uppercase to maintain uniformity across the dataset

df['Description'] = df['Description'].str.upper()

# Getting the number of rows in the dataframe

df.shape[0]
```

[35]:
```
399606
```

Step 3.6 | Treating Zero Unit Prices

In this step, first I am going to take a look at the statistical description of the `UnitPrice` column:

[36]:
```
df['UnitPrice'].describe()
```

[36]:
```
count    399606.000000
mean          2.904957
```

```
std             4.448796
min             0.000000
25%             1.250000
50%             1.950000
75%             3.750000
max           649.500000
Name: UnitPrice, dtype: float64
```

Inference:

The minimum unit price value is zero. This indicates that there are some transactions where the unit price are zero, there may be chance of data entry errror or free item. To understand the nature of dataset is important to take action on zero unit price transactions . A detailed analysis of the product descriptions associated with zero unit prices will conducted to determine do they have any specific patterns.

[37]:
```
df[df['UnitPrice']==0].describe()['Quantity']
```

[37]:
```
count       33.000000
mean       420.515152
std       2176.713608
min          1.000000
25%          2.000000
50%         11.000000
75%         36.000000
max      12540.000000
Name: Quantity, dtype: float64
```

Inferences on UnitPrice:

- The transactions with a unit price of zero are relatively few in number (33 transactions).
- These transactions have a large variability in the quantity of items involved, ranging from 1 to 12540, with a substantial standard deviation.
- Including these transactions in the clustering analysis might introduce noise and could potentially distort the customer behavior patterns identified by the clustering algorithm.

Strategy:

Given the small number of these transactions and their potential to introduce noise in the data analysis, the strategy should be to remove these transactions from the dataset. This would help in maintaining a cleaner and more consistent dataset, which is essential for building an accurate and reliable clustering model and recommendation system.

[38]:
```
# Removing records with a unit price of zero to avoid potential data entry␣
↪errors

df = df[df['UnitPrice'] > 0]
```

Step 3.7 | Outlier Treatment

In K-means clustering, the algorithm is sensitive to both the scale of data and the presence of outliers, as they can significantly influence the position of centroids, potentially leading to incorrect cluster assignments. However, considering the context of this project where the final goal is to understand customer behavior and preferences through K-means clustering, it would be more prudent to address the issue of outliers **after the feature engineering phase** where we create a customer-centric dataset. At this stage, the data is transactional, and removing outliers might eliminate valuable information that could play a crucial role in segmenting customers later on. Therefore, we will postpone the outlier treatment and proceed to the next stage for now.

[39]:
```
# Resetting the index of the cleaned dataset

df.reset_index(drop=True, inplace=True)
```

1 Getting the number of rows in the dataframe

df.shape[0]

#

Step 4 | Feature Engineering

Tabel of Contents

In order to create a comprehensive customer-centric dataset for clustering and recommendation, the following features can be engineered from the available data:

Step 4.1 | RFM Features

RFM is a method used for analyzing customer value and segmenting the customer base. It is an acronym that stands for:

- **Recency (R):** This metric indicates how recently a customer has made a purchase. A lower recency value means the customer has purchased more recently, indicating higher engagement with the brand.
- **Frequency (F):** This metric signifies how often a customer makes a purchase within a certain period. A higher frequency value indicates a customer who interacts with the business more often, suggesting higher loyalty or satisfaction.
- **Monetary (M):** This metric represents the total amount of money a customer has spent over a certain period. Customers who have a higher monetary value have contributed more to the business, indicating their potential high lifetime value.

Together, these metrics help in understanding a customer's buying behavior and preferences, which is pivotal in personalizing marketing strategies and creating a recommendation system.

Step 4.1.1 | Recency (R)

In this step, we focus on understanding how recently a customer has made a purchase. This is a crucial aspect of customer segmentation as it helps in identifying the engagement level of customers. Here, I am going to define the following feature:

- **Days Since Last Purchase**: This feature represents the number of days that have passed since the customer's last purchase. A lower value indicates that the customer has purchased recently, implying a higher engagement level with the business, whereas a higher value may indicate a lapse or decreased engagement. By understanding the recency of purchases, businesses can tailor their marketing strategies to re-engage customers who have not made purchases in a while, potentially increasing customer retention and fostering loyalty.

[40]:
```
# Convert InvoiceDate to datetime type
#extract date

df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'])

# Convert InvoiceDate to datetime and extract only the date

df['InvoiceDay'] = df['InvoiceDate'].dt.date

# Find the most recent purchase date for each customer

customer_data = df.groupby('CustomerID')['InvoiceDay'].max().reset_index()

# Find the most recent date in the entire dataset

most_recent_date = df['InvoiceDay'].max()

# Convert InvoiceDay to datetime type before subtraction

customer_data['InvoiceDay'] = pd.to_datetime(customer_data['InvoiceDay'])
most_recent_date = pd.to_datetime(most_recent_date)

# Calculate the number of days since the last purchase for each customer

customer_data['Days_Since_Last_Purchase'] = (most_recent_date -
 ↪customer_data['InvoiceDay']).dt.days

# Remove the InvoiceDay column

customer_data.drop(columns=['InvoiceDay'], inplace=True)
print(customer_data)
```

```
      CustomerID  Days_Since_Last_Purchase
0        12346.0                       325
1        12347.0                         2
2        12348.0                        75
3        12349.0                        18
4        12350.0                       310
…          …                     …
4357     18280.0                       277
4358     18281.0                       180
4359     18282.0                         7
4360     18283.0                         3
```

```
4361     18287.0                          42

[4362 rows x 2 columns]
```

Now, **customer_data** dataframe contains the `Days_Since_Last_Purchase` feature:

[41]:
```
customer_data.head()
```

[41]:
```
   CustomerID  Days_Since_Last_Purchase
0     12346.0                       325
1     12347.0                         2
2     12348.0                        75
3     12349.0                        18
4     12350.0                       310
```

Note:

- We've named the customer-centric dataframe as **customer_data**, which will eventually contain all the customer-based features we plan to create.

Step 4.1.2 | Frequency (F)

In this step, I am going to create two features that quantify the frequency of a customer's engagement with the retailer:

- **Total Transactions**: This feature represents the total number of transactions made by a customer. It helps in understanding the engagement level of a customer with the retailer.
- **Total Products Purchased**: This feature indicates the total number of products (sum of quantities) purchased by a customer across all transactions. It gives an insight into the customer's buying behavior in terms of the volume of products purchased.

These features will be crucial in segmenting customers based on their buying frequency, which is a key aspect in determining customer segments for targeted marketing and personalized recommendations.

[42]:
```
# Calculate the total number of transactions made by each customer

total_transactions = df.groupby('CustomerID')['InvoiceNo'].nunique().
 ↪reset_index()
total_transactions.rename(columns={'InvoiceNo': 'Total_Transactions'},␣
 ↪inplace=True)

# Calculate the total number of products purchased by each customer

total_products_purchased = df.groupby('CustomerID')['Quantity'].sum().
 ↪reset_index()
total_products_purchased.rename(columns={'Quantity':␣
 ↪'Total_Products_Purchased'}, inplace=True)

# Merge the new features into the customer_data dataframe
```

```
customer_data = pd.merge(customer_data, total_transactions, on='CustomerID')
customer_data = pd.merge(customer_data, total_products_purchased,␣
 ↪on='CustomerID')

# Display the first few rows of the customer_data dataframe

customer_data.sample(10)
```

[42]:

```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
938      13599.0                         1                  33
1789     14768.0                        17                   3
1133     13873.0                       119                   4
1780     14757.0                        75                   1
2597     15857.0                        18                   1
1491     14367.0                         8                  19
2717     16031.0                        92                   2
1541     14438.0                       306                   1
201      12596.0                        51                   2
3770     17481.0                         3                   5

      Total_Products_Purchased
938                       3169
1789                        34
1133                       138
1780                       160
2597                       308
1491                      4398
2717                       421
1541                        82
201                        468
3770                      1060
```

Step 4.1.3 | Monetary (M)

In this step, I am going to create two features that represent the monetary aspect of customer's transactions:

- **Total Spend**: This feature represents the total amount of money spent by each customer. It is calculated as the sum of the product of `UnitPrice` and `Quantity` for all transactions made by a customer. This feature is crucial as it helps in identifying the total revenue generated by each customer, which is a direct indicator of a customer's value to the business.
- **Average Transaction Value**: This feature is calculated as the **Total Spend** divided by the **Total Transactions** for each customer. It indicates the average value of a transaction carried out by a customer. This metric is useful in understanding the spending behavior of customers per transaction, which can assist in tailoring marketing strategies and offers to different customer segments based on their average spending patterns.

[43]:
```
# Calculate the total spend by each customer

df['Total_Spend'] = df['UnitPrice'] * df['Quantity']
total_spend = df.groupby('CustomerID')['Total_Spend'].sum().reset_index()

# Calculate the average transaction value for each customer

average_transaction_value = total_spend.merge(total_transactions,␣
↪on='CustomerID')
average_transaction_value['Average_Transaction_Value'] =␣
↪average_transaction_value['Total_Spend'] /␣
↪average_transaction_value['Total_Transactions']

# Merge the new features into the customer_data dataframe

customer_data = pd.merge(customer_data, total_spend, on='CustomerID')
customer_data = pd.merge(customer_data,␣
↪average_transaction_value[['CustomerID', 'Average_Transaction_Value']],␣
↪on='CustomerID')

# Display the first few rows of the customer_data dataframe

customer_data.sample(5)
```

[43]:
```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
1072     13791.0                        43                   3
2316     15485.0                        30                   3
2443     15654.0                         9                   2
2105     15204.0                       357                   1
2347     15526.0                        33                   2

      Total_Products_Purchased  Total_Spend  Average_Transaction_Value
1072                       828      1047.68                 349.226667
2316                      1281      2564.50                 854.833333
2443                       474       907.53                 453.765000
2105                       258       316.58                 316.580000
2347                        24       148.44                  74.220000
```

Step 4.2 | Product Diversity

In this step, we are going to understand the diversity in the product purchase behavior of customers. Understanding product diversity can help in crafting personalized marketing strategies and product recommendations. Here, I am going to define the following feature:

- **Unique Products Purchased**: This feature represents the number of distinct products bought by a customer. A higher value indicates that the customer has a diverse taste or preference, buying a wide range of products, while a lower value might indicate a focused or specific preference. Understanding the diversity in product purchases can help in segmenting customers based on their buying diversity, which can be a critical input in personalizing product recommendations.

[44]:
```python
# Calculate the number of unique products purchased by each customer
unique_products_purchased = df.groupby('CustomerID')['StockCode'].nunique().
 ↪reset_index()
unique_products_purchased.rename(columns={'StockCode':
 ↪'Unique_Products_Purchased'}, inplace=True)

# Merge the new feature into the customer_data dataframe
customer_data = pd.merge(customer_data, unique_products_purchased,␣
 ↪on='CustomerID')

# Display the first few rows of the customer_data dataframe
customer_data.head()
```

[44]:

```
   CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
0     12346.0                       325                   2
1     12347.0                         2                   7
2     12348.0                        75                   4
3     12349.0                        18                   1
4     12350.0                       310                   1

   Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
0                         0         0.00                   0.000000
1                      2458      4310.00                 615.714286
2                      2332      1437.24                 359.310000
3                       630      1457.55                1457.550000
4                       196       294.40                 294.400000

   Unique_Products_Purchased
0                          1
1                        103
2                         21
3                         72
4                         16
```

[45]:
```python
customer_data.sample(25)
```

[45]:

```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
3850     17589.0                        60                   4
3775     17491.0                         1                  11
1037     13741.0                        71                   4
3435     17004.0                        46                   2
2165     15276.0                        66                   1
712      13285.0                        23                   4
234      12633.0                        58                   5
```

3381	16929.0	3	7
2699	16011.0	8	16
4088	17912.0	310	6
92	12457.0	58	11
1036	13740.0	240	1
3719	17410.0	16	5
1743	14704.0	19	7
2603	15864.0	22	8
471	12950.0	2	3
2746	16073.0	291	2
460	12937.0	15	4
1791	14770.0	232	2
306	12720.0	2	28
4342	18259.0	24	3
2204	15332.0	366	4
2346	15525.0	2	4
3526	17134.0	106	2
2652	15942.0	133	1

	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	\
3850	1630	2402.92	600.730000	
3775	2012	3541.92	321.992727	
1037	217	666.33	166.582500	
3435	771	1312.14	656.070000	
2165	61	128.63	128.630000	
712	2051	2709.12	677.280000	
234	1136	1906.63	381.326000	
3381	929	1295.39	185.055714	
2699	2035	3352.96	209.560000	
4088	182	294.66	49.110000	
92	686	1597.78	145.252727	
1036	425	350.75	350.750000	
3719	616	1214.72	242.944000	
1743	1035	1461.62	208.802857	
2603	1044	1769.78	221.222500	
471	1380	1843.00	614.333333	
2746	37	94.35	47.175000	
460	1007	1504.27	376.067500	
1791	490	876.42	438.210000	
306	4618	5065.28	180.902857	
4342	714	2338.60	779.533333	
2204	652	1661.06	415.265000	
2346	383	716.97	179.242500	
3526	98	413.20	206.600000	
2652	232	337.44	337.440000	

Unique_Products_Purchased

```
3850            171
3775             72
1037             34
3435             38
2165             49
712             157
234              72
3381             51
2699             99
4088             35
92               52
1036             23
3719             57
1743            286
2603             16
471              13
2746              1
460              81
1791             49
306             210
4342             27
2204             29
2346            132
3526             21
2652             14
```

Step 4.3 | Behavioral Features

In this step, we aim to understand and capture the shopping patterns and behaviors of customers. These features will give us insights into the customers' preferences regarding when they like to shop, which can be crucial information for personalizing their shopping experience. Here are the features I am planning to introduce:

- **Average Days Between Purchases**: This feature represents the average number of days a customer waits before making another purchase. Understanding this can help in predicting when the customer is likely to make their next purchase, which can be a crucial metric for targeted marketing and personalized promotions.
- **Favorite Shopping Day**: This denotes the day of the week when the customer shops the most. This information can help in identifying the preferred shopping days of different customer segments, which can be used to optimize marketing strategies and promotions for different days of the week.
- **Favorite Shopping Hour**: This refers to the hour of the day when the customer shops the most. Identifying the favorite shopping hour can aid in optimizing the timing of marketing campaigns and promotions to align with the times when different customer segments are most active.

By including these behavioral features in our dataset, we can create a more rounded view of our customers, which will potentially enhance the effectiveness of the clustering algorithm, leading to more meaningful customer segments.

[46]:
```
# Extract day of week and hour from InvoiceDate

df['Day_Of_Week'] = df['InvoiceDate'].dt.dayofweek
df['Hour'] = df['InvoiceDate'].dt.hour

# Calculate the average number of days between consecutive purchases

days_between_purchases = df.groupby('CustomerID')['InvoiceDay'].apply(lambda x:␣
↪(x.diff().dropna()).apply(lambda y: y.days))
average_days_between_purchases = days_between_purchases.groupby('CustomerID').
↪mean().reset_index()
average_days_between_purchases.rename(columns={'InvoiceDay':␣
↪'Average_Days_Between_Purchases'}, inplace=True)

# Find the favorite shopping day of the week

favorite_shopping_day = df.groupby(['CustomerID', 'Day_Of_Week']).size().
↪reset_index(name='Count')
favorite_shopping_day = favorite_shopping_day.loc[favorite_shopping_day.
↪groupby('CustomerID')['Count'].idxmax()][['CustomerID', 'Day_Of_Week']]

# Find the favorite shopping hour of the day

favorite_shopping_hour = df.groupby(['CustomerID', 'Hour']).size().
↪reset_index(name='Count')
favorite_shopping_hour = favorite_shopping_hour.loc[favorite_shopping_hour.
↪groupby('CustomerID')['Count'].idxmax()][['CustomerID', 'Hour']]

# Merge the new features into the customer_data dataframe

customer_data = pd.merge(customer_data, average_days_between_purchases,␣
↪on='CustomerID')
customer_data = pd.merge(customer_data, favorite_shopping_day, on='CustomerID')
customer_data = pd.merge(customer_data, favorite_shopping_hour, on='CustomerID')

# Display the first few rows of the customer_data dataframe

customer_data.head(10)
```

[46]:
```
   CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
0     12346.0                       325                   2
1     12347.0                         2                   7
2     12348.0                        75                   4
3     12349.0                        18                   1
4     12350.0                       310                   1
```

```
5     12352.0                          36                    8
6     12353.0                         204                    1
7     12354.0                         232                    1
8     12355.0                         214                    1
9     12356.0                          22                    3

   Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
0                         0         0.00                   0.000000
1                      2458      4310.00                 615.714286
2                      2332      1437.24                 359.310000
3                       630      1457.55                1457.550000
4                       196       294.40                 294.400000
5                       463      1265.41                 158.176250
6                        20        89.00                  89.000000
7                       530      1079.40                1079.400000
8                       240       459.40                 459.400000
9                      1573      2487.43                 829.143333

   Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
0                          1                        0.000000            1
1                        103                        2.016575            1
2                         21                       10.884615            3
3                         72                        0.000000            0
4                         16                        0.000000            2
5                         57                        3.132530            1
6                          4                        0.000000            3
7                         58                        0.000000            3
8                         13                        0.000000            0
9                         52                        5.315789            1

   Hour
0    10
1    14
2    19
3     9
4    16
5    14
6    17
7    13
8    13
9     9
```

Step 4.4 | Geographic Features

In this step, we will introduce a geographic feature that reflects the geographical location of customers. Understanding the geographic distribution of customers is pivotal for several reasons:

- **Country**: This feature identifies the country where each customer is located. Including the country data can help us understand region-specific buying patterns and preferences. Different regions might have varying preferences and purchasing behaviors which can be critical in personalizing marketing strategies and inventory planning. Furthermore, it can be instrumental in logistics and supply chain optimization, particularly for an online retailer where shipping and delivery play a significant role.

[47]:
```
df['Country'].value_counts(normalize=True).head()
```

[47]:
```
United Kingdom    0.890971
Germany           0.022722
France            0.020402
EIRE              0.018440
Spain             0.006162
Name: Country, dtype: float64
```

Inference:

In above observation is seen that majority portion **(89%)** of the Transactions are from the **United Kingdom** . We can consider to make an Binary feature indicating that the transaction is from UK or not . This approach can help us in Clustering process without losing any important information, especially when we are using sensitive algorithm like K-means which is sensitive to the dimensionality of feature space.

Methodology:

- First, we will group the data by `CustomerID` and `Country` and calculate the number of transactions per country for each customer.
- Next, we will identify the main country for each customer (the country from which they have the maximum transactions).
- Then, we will create a binary column indicating whether the customer is from the UK or not.
- Finally, we will merge this information with the `customer_data` dataframe to include the new feature in our analysis.

[48]:
```
# Group by CustomerID and Country to get the number of transactions per country␣
↪for each customer
customer_country = df.groupby(['CustomerID', 'Country']).size().
↪reset_index(name='Number_of_Transactions')

# Get the country with the maximum number of transactions for each customer (in␣
↪case a customer has transactions from multiple countries)
customer_main_country = customer_country.sort_values('Number_of_Transactions',␣
↪ascending=False).drop_duplicates('CustomerID')

# Create a binary column indicating whether the customer is from the UK or not
customer_main_country['Is_UK'] = customer_main_country['Country'].apply(lambda␣
↪x: 1 if x == 'United Kingdom' else 0)
```

```
# Merge this data with our customer_data dataframe
customer_data = pd.merge(customer_data, customer_main_country[['CustomerID',␣
↪'Is_UK']], on='CustomerID', how='left')

# Display the first few rows of the customer_data dataframe
customer_data.sample(15)
```

[48]:

```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
431      12901.0                         8                  34
4064     17975.0                        15                  14
1107     13865.0                        58                   4
2825     16233.0                        71                   5
1441     14329.0                         8                  14
2582     15894.0                       253                   2
2675     16029.0                        38                  66
880      13534.0                         2                  43
2367     15602.0                         8                  14
1794     14813.0                       369                   2
2019     15124.0                        22                   2
2026     15133.0                       127                   3
2996     16471.0                       274                   1
2914     16362.0                        32                   9
2866     16297.0                       249                   3

      Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
431                      21299     16316.14                 479.886471
4064                      1547      4384.76                 313.197143
1107                       204       501.56                 125.390000
2825                       165       422.13                  84.426000
1441                      3067      4889.14                 349.224286
2582                       163       257.60                 128.800000
2675                     33687     60369.93                 914.695909
880                       2876      5613.08                 130.536744
2367                       999      1102.37                  78.740714
1794                        66       152.88                  76.440000
2019                       233       184.19                  92.095000
2026                       680       982.42                 327.473333
2996                       141       223.95                 223.950000
2914                       314       627.29                  69.698889
2866                       181       278.55                  92.850000

      Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
431                          30                        2.147541            2
4064                        188                        1.202055            1
1107                         26                        7.965517            2
2825                         22                       12.291667            3
1441                        218                        1.202247            4
```

```
2582            34    3.314286    6
2675            43    1.293436    1
880            119    1.043988    2
2367            30    7.200000    4
1794            25    0.000000    6
2019            14   12.071429    2
2026            27    6.214286    3
2996            13    0.000000    3
2914            58    0.694118    4
2866            20    0.904762    2

      Hour  Is_UK
431     11      1
4064    12      1
1107     8      1
2825    17      1
1441    13      1
2582    14      1
2675    11      1
880     10      1
2367    14      1
1794    11      1
2019     9      1
2026    10      1
2996     9      1
2914    14      1
2866     8      1
```

[49]:

```python
# Display feature distribution
customer_data['Is_UK'].value_counts()
```

[49]:

```
1    3866
0     416
Name: Is_UK, dtype: int64
```

Step 4.5 | Cancellation Insights

In this step, We are going to delve deeper into the cancellation patterns of customers to gain insights that can enhance our customer segmentation model. The features I am planning to introduce are:

- **Cancellation Frequency**: This metric represents the total number of transactions a customer has canceled. Understanding the frequency of cancellations can help us identify customers who are more likely to cancel transactions. This could be an indicator of dissatisfaction or other issues, and understanding this can help us tailor strategies to reduce cancellations and enhance customer satisfaction.
- **Cancellation Rate**: This represents the proportion of transactions that a customer has canceled out of all their transactions. This metric gives a normalized view of cancellation behavior. A high cancellation rate might be indicative of an unsatisfied customer segment.

By identifying these segments, we can develop targeted strategies to improve their shopping experience and potentially reduce the cancellation rate.

By incorporating these cancellation insights into our dataset, we can build a more comprehensive view of customer behavior, which could potentially aid in creating more effective and nuanced customer segmentation.

[50]:
```python
# Calculate the total number of transactions made by each customer
total_transactions = df.groupby('CustomerID')['InvoiceNo'].nunique().
    ↪reset_index()

# Calculate the number of cancelled transactions for each customer
cancelled_transactions = df[df['Transaction_Status'] == 'Cancelled']
cancellation_frequency = cancelled_transactions.
    ↪groupby('CustomerID')['InvoiceNo'].nunique().reset_index()
cancellation_frequency.rename(columns={'InvoiceNo': 'Cancellation_Frequency'},␣
    ↪inplace=True)

#no. of cancellations per customer
# Merge the Cancellation Frequency data into the customer_data dataframe
customer_data = pd.merge(customer_data, cancellation_frequency,␣
    ↪on='CustomerID', how='left')

# Replace NaN values with 0 (for customers who have not cancelled any␣
    ↪transaction)
customer_data['Cancellation_Frequency'].fillna(0, inplace=True)

# Calculate the Cancellation Rate
customer_data['Cancellation_Rate'] = customer_data['Cancellation_Frequency'] /␣
    ↪total_transactions['InvoiceNo']

# Display the first few rows of the customer_data dataframe
customer_data.sample(10)
```

[50]:
```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
2773     16169.0                         8                   4
3201     16759.0                         7                   4
1767     14775.0                        58                   3
3303     16900.0                        11                   4
2549     15844.0                        46                   1
3671     17430.0                        32                   2
3111     16638.0                        22                  11
3817     17633.0                        31                   6
2343     15569.0                       103                   5
633      13189.0                        18                   2

      Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
```

```
2773                          1325       1822.97                 455.742500
3201                           608        772.84                 193.210000
1767                          1174       1011.90                 337.300000
3303                           724        869.20                 217.300000
2549                           134        130.74                 130.740000
3671                           190        265.76                 132.880000
3111                          1254       1676.47                 152.406364
3817                           630       1242.34                 207.056667
2343                           666       1375.71                 275.142000
633                            842        260.68                 130.340000

      Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
2773                         71                        0.797619            3
3201                         40                        0.547619            2
1767                         57                        5.322034            0
3303                         77                        0.666667            4
2549                         11                        0.000000            0
3671                         14                       25.615385            0
3111                         92                        3.680851            1
3817                         60                        4.619718            3
2343                         27                        7.633333            6
633                          29                        3.464286            1

      Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate
2773    15      1                     0.0           0.000000
3201     9      1                     1.0           0.500000
1767    12      1                     0.0           0.000000
3303    13      1                     1.0           1.000000
2549    15      1                     0.0           0.000000
3671     9      1                     0.0           0.000000
3111     9      1                     1.0           0.166667
3817    15      1                     2.0           2.000000
2343    10      1                     0.0           0.000000
633     13      1                     0.0           0.000000
```

Step 4.6 | Seasonality & Trends

In this step, we will delve into the seasonality and trends in customers' purchasing behaviors, which can offer invaluable insights for tailoring marketing strategies and enhancing customer satisfaction. Here are the features I am looking to introduce:

- **Monthly_Spending_Mean**: This is the average amount a customer spends monthly. It helps us gauge the general spending habit of each customer. A higher mean indicates a customer who spends more, potentially showing interest in premium products, whereas a lower mean might indicate a more budget-conscious customer.
- **Monthly_Spending_Std**: This feature indicates the variability in a customer's monthly spending. A higher value signals that the customer's spending fluctuates significantly month-to-month, perhaps indicating sporadic large purchases. In contrast, a lower value suggests

more stable, consistent spending habits. Understanding this variability can help in crafting personalized promotions or discounts during periods they are expected to spend more.

- **Spending_Trend**: This reflects the trend in a customer's spending over time, calculated as the slope of the linear trend line fitted to their spending data. A positive value indicates an increasing trend in spending, possibly pointing to growing loyalty or satisfaction. Conversely, a negative trend might signal decreasing interest or satisfaction, highlighting a need for re-engagement strategies. A near-zero value signifies stable spending habits. Recognizing these trends can help in developing strategies to either maintain or alter customer spending patterns, enhancing the effectiveness of marketing campaigns.

By incorporating these detailed insights into our customer segmentation model, we can create more precise and actionable customer groups, facilitating the development of highly targeted marketing strategies and promotions.

[51]:
```
# Extract month and year from InvoiceDate
df['Year'] = df['InvoiceDate'].dt.year
df['Month'] = df['InvoiceDate'].dt.month

# Calculate monthly spending for each customer
monthly_spending = df.groupby(['CustomerID', 'Year', 'Month'])['Total_Spend'].
↪sum().reset_index()
#print(monthly_spending)
#Calculate Seasonal Buying Patterns: We are using monthly frequency as a proxy␣
↪for seasonal buying patterns
seasonal_buying_patterns = monthly_spending.
↪groupby('CustomerID')['Total_Spend'].agg(['mean', 'std']).reset_index()
seasonal_buying_patterns.rename(columns={'mean': 'Monthly_Spending_Mean', 'std':
↪ 'Monthly_Spending_Std'}, inplace=True)

# Replace NaN values in Monthly_Spending_Std with 0, implying no variability␣
↪for customers with single transaction month
seasonal_buying_patterns['Monthly_Spending_Std'].fillna(0, inplace=True)

# Calculate Trends in Spending
# We are using the slope of the linear trend line fitted to the customer's␣
↪spending over time as an indicator of spending trends
def calculate_trend(spend_data):
    # If there are more than one data points, we calculate the trend using␣
↪linear regression
    if len(spend_data) > 1:
        x = np.arange(len(spend_data))
        slope, _, _, _, _ = linregress(x, spend_data)
        return slope
    # If there is only one data point, no trend can be calculated, hence we␣
↪return 0
    else:
        return 0
```

```
# Apply the calculate_trend function to find the spending trend for each␣
 ↪customer
spending_trends = monthly_spending.groupby('CustomerID')['Total_Spend'].
 ↪apply(calculate_trend).reset_index()
spending_trends.rename(columns={'Total_Spend': 'Spending_Trend'}, inplace=True)
#The calculated slope represents the rate of change in spending over time.
# #Positive slopes indicate increasing spending trends, while negative slopes␣
 ↪suggest decreasing trends.

# Merge the new features into the customer_data dataframe
customer_data = pd.merge(customer_data, seasonal_buying_patterns,␣
 ↪on='CustomerID')
customer_data = pd.merge(customer_data, spending_trends, on='CustomerID')

# Display the first few rows of the customer_data dataframe
customer_data.sample(10)
```

[51]:

```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
3176     16726.0                        26                   5
4172     18136.0                        63                   4
3026     16515.0                        52                   5
2044     15156.0                         1                   3
670      13243.0                       203                   1
3371     16996.0                        30                   9
3222     16784.0                        19                   1
147      12530.0                        59                   5
143      12524.0                         9                   8
513      13016.0                        73                   3

      Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
3176                       945      1358.74                 271.748000
4172                       373       761.83                 190.457500
3026                       826      1624.14                 324.828000
2044                       597       961.49                 320.496667
670                        232       585.12                 585.120000
3371                       534      1427.46                 158.606667
3222                        24       107.60                 107.600000
147                        891      1461.63                 292.326000
143                       3669      3945.72                 493.215000
513                        527       789.89                 263.296667

      Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
3176                        121                        1.892655            6
4172                         34                        6.736842            0
3026                         89                        2.354545            1
2044                         51                        0.592593            0
```

```
670                           56                       0.000000           4
3371                          14                       8.200000           2
3222                           3                       0.000000           6
147                           50                       4.396825           1
143                          112                       2.400000           3
513                           39                       1.086957           0

      Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate  \
3176    13      1                     0.0                0.0
4172    12      1                     1.0                1.0
3026    11      1                     0.0                0.0
2044     9      1                     0.0                0.0
670     13      1                     0.0                0.0
3371    14      1                     3.0                1.5
3222    12      1                     0.0                0.0
147     10      0                     1.0                0.2
143     13      0                     0.0                0.0
513     15      1                     0.0                0.0

      Monthly_Spending_Mean  Monthly_Spending_Std  Spending_Trend
3176             339.685000            214.649606      109.124000
4172             253.943333             61.023317      -52.755000
3026             324.828000             47.057137        7.710000
2044             480.745000            230.439029      325.890000
670              585.120000              0.000000        0.000000
3371             285.492000            111.548694       67.083000
3222             107.600000              0.000000        0.000000
147              365.407500             33.531044       17.409000
143              563.674286            404.113005       28.038929
513              394.945000             42.857742      -60.610000
```

[52]:
```python
# Changing the data type of 'CustomerID' to string as it is a unique identifier and not used in mathematical operations
customer_data['CustomerID'] = customer_data['CustomerID'].astype(str)

# Convert data types of columns to optimal types
customer_data = customer_data.convert_dtypes()
```

[53]:
```python
customer_data.head(10)
```

[53]:
```
  CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
0    12346.0                       325                   2
1    12347.0                         2                   7
2    12348.0                        75                   4
3    12349.0                        18                   1
4    12350.0                       310                   1
5    12352.0                        36                   8
```

```
6   12353.0                        204                   1
7   12354.0                        232                   1
8   12355.0                        214                   1
9   12356.0                         22                   3

   Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
0                         0          0.0                        0.0
1                      2458       4310.0                 615.714286
2                      2332      1437.24                     359.31
3                       630      1457.55                    1457.55
4                       196        294.4                      294.4
5                       463      1265.41                  158.17625
6                        20         89.0                       89.0
7                       530       1079.4                     1079.4
8                       240        459.4                      459.4
9                      1573      2487.43                 829.143333

   Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
0                          1                             0.0            1
1                        103                        2.016575            1
2                         21                       10.884615            3
3                         72                             0.0            0
4                         16                             0.0            2
5                         57                         3.13253            1
6                          4                             0.0            3
7                         58                             0.0            3
8                         13                             0.0            0
9                         52                        5.315789            1

   Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate  \
0    10      1                       1                0.5
1    14      0                       0                0.0
2    19      0                       0                0.0
3     9      0                       0                0.0
4    16      0                       0                0.0
5    14      0                       1              0.125
6    17      0                       0                0.0
7    13      0                       0                0.0
8    13      0                       0                0.0
9     9      0                       0                0.0

   Monthly_Spending_Mean  Monthly_Spending_Std  Spending_Trend
0                    0.0                   0.0             0.0
1             615.714286            341.070789        4.486071
2                 359.31            203.875689        -100.884
3                1457.55                   0.0             0.0
4                  294.4                   0.0             0.0
```

```
5           316.3525      134.700629       9.351
6               89.0             0.0         0.0
7             1079.4             0.0         0.0
8              459.4             0.0         0.0
9         829.143333      991.462585    -944.635
```

[54]:
```python
customer_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4282 entries, 0 to 4281
Data columns (total 16 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   CustomerID                      4282 non-null   string
 1   Days_Since_Last_Purchase        4282 non-null   Int64
 2   Total_Transactions              4282 non-null   Int64
 3   Total_Products_Purchased        4282 non-null   Int64
 4   Total_Spend                     4282 non-null   Float64
 5   Average_Transaction_Value       4282 non-null   Float64
 6   Unique_Products_Purchased       4282 non-null   Int64
 7   Average_Days_Between_Purchases  4282 non-null   Float64
 8   Day_Of_Week                     4282 non-null   Int64
 9   Hour                            4282 non-null   Int64
 10  Is_UK                           4282 non-null   Int64
 11  Cancellation_Frequency          4282 non-null   Int64
 12  Cancellation_Rate               4282 non-null   Float64
 13  Monthly_Spending_Mean           4282 non-null   Float64
 14  Monthly_Spending_Std            4282 non-null   Float64
 15  Spending_Trend                  4282 non-null   Float64
dtypes: Float64(7), Int64(8), string(1)
memory usage: 631.4 KB
```

Let's review the descriptions of the columns in our newly created `customer_data` dataset:

Customer Dataset Description: Fill Missing Values

Variable	Description
CustomerID	Unique Identity number to identify each unique customer
Days_Since_Last_Purchase	The number of days from the last purchase date.
Total_Transactions	The total number of transactions done by a customer.
Total_Products_Purchased	The total quantity of items which is purchased by a customer.
Total_Spend	The total amount of money the customer has spent across all transactions.

Variable	Description
Average_Transaction_Value	The average value of the customer's transactions, calculated as total spend divided by the number of transactions.
Unique_Products_Purchased	The number of different products the customer has purchased.
Average_Days_Between_Purchases	The average number of days between consecutive purchases made by the customer.
Day_Of_Week	The day of the week when the customer prefers to shop, represented numerically (0 for Monday, 6 for Sunday).
Hour	The hour of the day when the customer prefers to shop, represented in a 24-hour format.
Is_UK	A Binary identifier to check customer from uk(1) or not(0).
Cancellation_Frequency	How often a customer cancelled his order
Cancellation_Rate	The proportion of transactions that the customer has cancelled, calculated as cancellation frequency divided by total transactions.
Monthly_Spending_Mean	The average monthly spending of the customer.
Monthly_Spending_Std	their spending pattern.
Spending_Trend	A numerical representation of the trend in the customer's spending over time. A positive value indicates an increasing trend, a negative value indicates a decreasing trend, and a value close to zero indicates a stable trend.

We've done a great job so far! We have created a dataset that focuses on our customers, using a variety of new features that give us a deeper understanding of their buying patterns and preferences.

Now that our dataset is ready, we can move on to the next steps of our project. This includes looking at our data more closely to find any patterns or trends, making sure our data is in the best shape by checking for and handling any outliers, and preparing our data for the clustering process. All of these steps will help us build a strong foundation for creating our customer segments and, eventually, a personalized recommendation system.

Let's dive in!

#

Step 5 | Outlier Detection and Treatment

Tabel of Contents

In this section, I will identify and handle outliers in our dataset. Outliers are data points that are significantly different from the majority of other points in the dataset. These points can potentially skew the results of our analysis, especially in k-means clustering where they can significantly influence the position of the cluster centroids. Therefore, it is essential to identify and treat these outliers appropriately to achieve more accurate and meaningful clustering results.

Given the multi-dimensional nature of the data, it would be prudent to use algorithms that can detect outliers in multi-dimensional spaces. I am going to use the **Isolation Forest** algorithm for this task. This algorithm works well for multi-dimensional data and is computationally efficient. It isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Let's proceed with this approach:

[55]:
```
# Initializing the IsolationForest model with a contamination parameter of 0.05
model = IsolationForest(contamination=0.05, random_state=0)

# Fitting the model on our dataset (converting DataFrame to NumPy to avoid␣
↪warning)
customer_data['Outlier_Scores'] = model.fit_predict(customer_data.iloc[:, 1:].
↪to_numpy())

# Creating a new column to identify outliers (1 for inliers and -1 for outliers)
customer_data['Is_Outlier'] = [1 if x == -1 else 0 for x in␣
↪customer_data['Outlier_Scores']]

# Display the first few rows of the customer_data dataframe
customer_data.head(10)
```

[55]:
```
  CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
0    12346.0                       325                   2
1    12347.0                         2                   7
2    12348.0                        75                   4
3    12349.0                        18                   1
4    12350.0                       310                   1
5    12352.0                        36                   8
6    12353.0                       204                   1
7    12354.0                       232                   1
8    12355.0                       214                   1
9    12356.0                        22                   3

   Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
0                         0          0.0                        0.0
1                      2458       4310.0                 615.714286
2                      2332      1437.24                     359.31
3                       630      1457.55                    1457.55
4                       196        294.4                      294.4
5                       463      1265.41                  158.17625
6                        20         89.0                       89.0
7                       530       1079.4                     1079.4
```

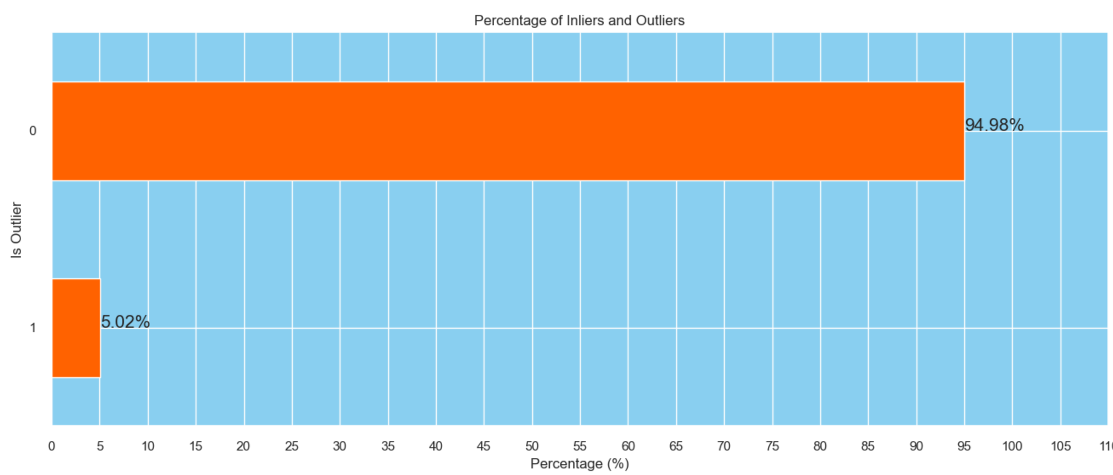
```
8                    240       459.4                         459.4
9                   1573     2487.43                    829.143333

   Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
0                          1                             0.0            1
1                        103                        2.016575            1
2                         21                       10.884615            3
3                         72                             0.0            0
4                         16                             0.0            2
5                         57                         3.13253            1
6                          4                             0.0            3
7                         58                             0.0            3
8                         13                             0.0            0
9                         52                        5.315789            1

   Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate  \
0    10      1                       1                0.5
1    14      0                       0                0.0
2    19      0                       0                0.0
3     9      0                       0                0.0
4    16      0                       0                0.0
5    14      0                       1              0.125
6    17      0                       0                0.0
7    13      0                       0                0.0
8    13      0                       0                0.0
9     9      0                       0                0.0

   Monthly_Spending_Mean  Monthly_Spending_Std  Spending_Trend  \
0                    0.0                   0.0             0.0
1             615.714286            341.070789        4.486071
2                 359.31            203.875689        -100.884
3                1457.55                   0.0             0.0
4                  294.4                   0.0             0.0
5               316.3525            134.700629           9.351
6                   89.0                   0.0             0.0
7                 1079.4                   0.0             0.0
8                  459.4                   0.0             0.0
9             829.143333            991.462585        -944.635

   Outlier_Scores  Is_Outlier
0               1           0
1               1           0
2               1           0
3               1           0
4               1           0
5               1           0
6               1           0
```

```
7                1          0
8                1          0
9               -1          1
```

After applying the Isolation Forest algorithm, we have identified the outliers and marked them in a new column named `Is_Outlier`. We have also calculated the outlier scores which represent the anomaly score of each record.

Now let's visualize the distribution of these scores and the number of inliers and outliers detected by the model:

[56]:
```python
# Calculate the percentage of inliers and outliers
outlier_percentage = customer_data['Is_Outlier'].value_counts(normalize=True) *␣
 ↪100

# Plotting the percentage of inliers and outliers
plt.figure(figsize=(16,6))
outlier_percentage.plot(kind='barh', color='#ff6200')

# Adding the percentage labels on the bars
for index, value in enumerate(outlier_percentage):
    plt.text(value, index, f'{value:.2f}%', fontsize=15)

plt.title('Percentage of Inliers and Outliers')
plt.xticks(ticks=np.arange(0, 115, 5))
plt.xlabel('Percentage (%)')
plt.ylabel('Is Outlier')
plt.gca().invert_yaxis()
plt.show()
```



Inference:

By Observing the above barplot we have noticed that there are about 5% of the customer are idntified as outliers. This number is not too high or not to low it is resonable amount of outliers are present in the dataset. This Observation suggest that our isolation forest algorithm has worked well to identifying the resonable percentage of outliers in dataset . It will play acrucial role in refining our Customer Segamentation

Strategy:

Considering the nature of the project (customer segmentation using clustering), it is crucial to handle these outliers to prevent them from affecting the clusters' quality significantly. Therefore, I will separate these outliers for further analysis and remove them from our main dataset to prepare it for the clustering analysis.

Let's proceed with the following steps:

- Separate the identified outliers for further analysis and save them as a separate file (optional).
- Remove the outliers from the main dataset to prevent them from influencing the clustering process.
- Drop the `Outlier_Scores` and `Is_Outlier` columns as they were auxiliary columns used for the outlier detection process.

Let's implement these steps:

[57]:
```
# Separate the outliers for analysis
outliers_data = customer_data[customer_data['Is_Outlier'] == 1]

# Remove the outliers from the main dataset
customer_data_cleaned = customer_data[customer_data['Is_Outlier'] == 0]

# Drop the 'Outlier_Scores' and 'Is_Outlier' columns
customer_data_cleaned = customer_data_cleaned.drop(columns=['Outlier_Scores',␣
↪'Is_Outlier'])

# Reset the index of the cleaned data
customer_data_cleaned.reset_index(drop=True, inplace=True)
print(customer_data_cleaned.tail(5))
```

```
      CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
4062     18280.0                       277                   1
4063     18281.0                       180                   1
4064     18282.0                         7                   3
4065     18283.0                         3                  16
4066     18287.0                        42                   3

      Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
4062                        45        180.6                      180.6
4063                        54        80.82                      80.82
4064                        98        176.6                  58.866667
4065                      1355      2039.58                  127.47375
4066                      1586      1837.28                 612.426667
```

```
      Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
4062                         10                             0.0            0
4063                          7                             0.0            6
4064                         12                        9.916667            4
4065                        262                        0.465181            3
4066                         59                        2.304348            2

      Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate  \
4062     9      1                       0                0.0
4063    10      1                       0                0.0
4064    13      1                       1           0.142857
4065    14      1                       0                0.0
4066    10      1                       0                0.0

      Monthly_Spending_Mean  Monthly_Spending_Std  Spending_Trend
4062                  180.6                   0.0             0.0
4063                  80.82                   0.0             0.0
4064                   88.3             14.792674          -20.92
4065                203.958            165.798738       22.319273
4066                 918.64            216.883792          306.72
```

We have successfully separated the outliers for further analysis and cleaned our main dataset by removing these outliers. This cleaned dataset is now ready for the next steps in our customer segmentation project, which includes scaling the features and applying clustering algorithms to identify distinct customer segments.

[58]:
```
customer_data_cleaned.shape[0]
```

[58]:
```
4067
```

#

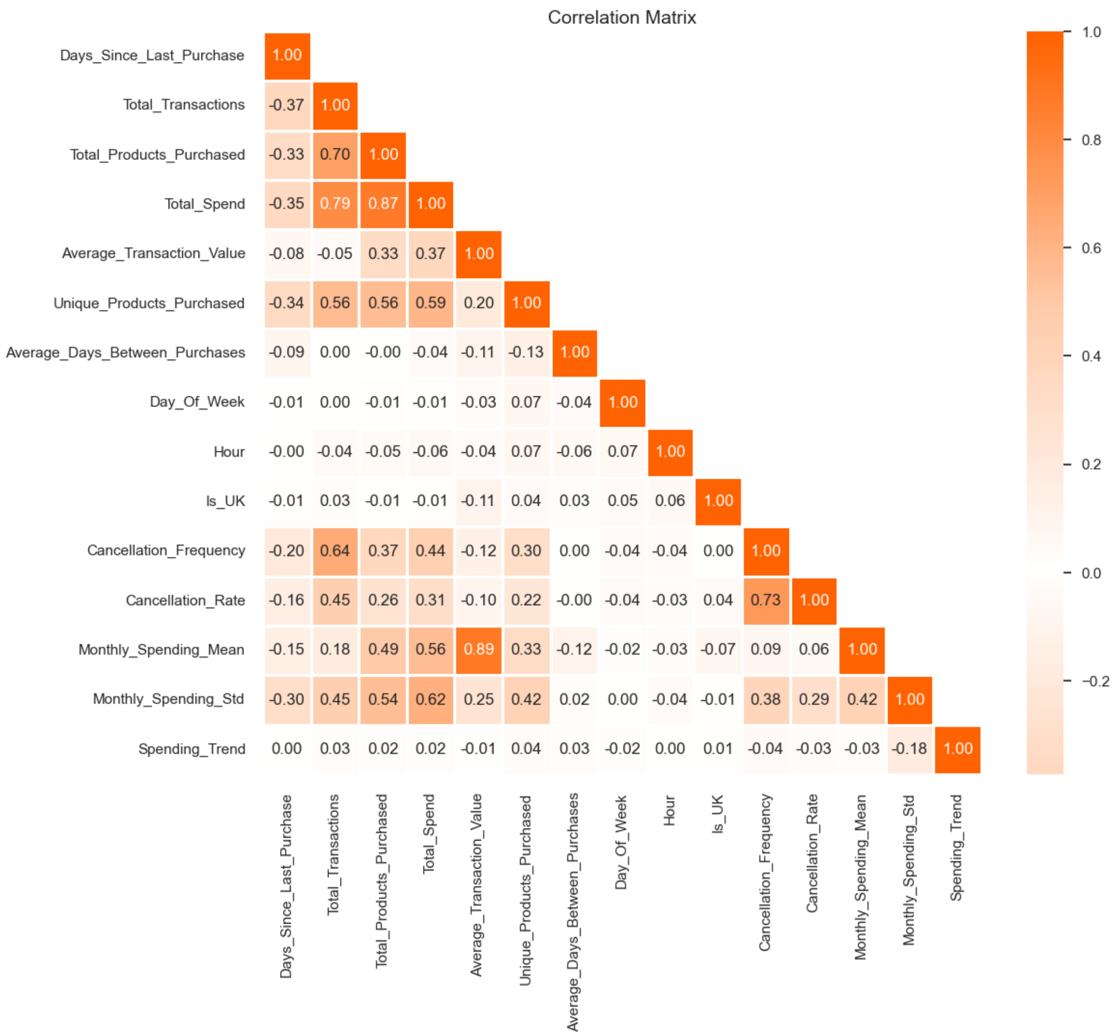
Step 6 | Correlation Analysis

Before we proceed to KMeans clustering, it's essential to check the correlation between features in our dataset. The presence of **multicollinearity**, where **features are highly correlated**, can potentially affect the clustering process by not allowing the model to learn the actual underlying patterns in the data, as the features do not provide unique information. This could lead to clusters that are not well-separated and meaningful.

If we identify multicollinearity, we can utilize dimensionality reduction techniques like PCA. These techniques help in neutralizing the effect of multicollinearity by transforming the correlated features into a new set of uncorrelated variables, preserving most of the original data's variance. This step not only enhances the quality of clusters formed but also makes the clustering process more computationally efficient.

[59]:
```
# Reset background style
sns.set_style('whitegrid')
```

```
# Calculate the correlation matrix excluding the 'CustomerID' column
corr = customer_data_cleaned.drop(columns=['CustomerID']).corr()

# Define a custom colormap
colors = ['#ff6200', '#ffcaa8', 'white', '#ffcaa8', '#ff6200']
my_cmap = LinearSegmentedColormap.from_list('custom_map', colors, N=256)

# Create a mask to only show the lower triangle of the matrix (since it's␣
↪mirrored around its
# top-left to bottom-right diagonal)
mask = np.zeros_like(corr)
mask[np.triu_indices_from(mask, k=1)] = True

# Plot the heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(corr, mask=mask, cmap=my_cmap, annot=True, center=0, fmt='.2f',␣
↪linewidths=2)
plt.title('Correlation Matrix', fontsize=14)
plt.show()
```



Inference:

Looking at the heatmap, we can see that there are some pairs of variables that have high correlations, for instance:

- `Monthly_Spending_Mean` and `Average_Transaction_Value`
- `Total_Spend` and `Total_Products_Purchased`
- `Total_Transactions` and `Total_Spend`
- `Cancellation_Rate` and `Cancellation_Frequency`
- `Total_Transactions` and `Total_Products_Purchased`

These high correlations indicate that these variables move closely together, implying a degree of multicollinearity.

Before moving to the next steps, considering the impact of multicollinearity on KMeans clustering, it might be beneficial to treat this multicollinearity possibly through dimensionality reduction techniques such as PCA to create a set of uncorrelated variables. This will help in achieving more stable clusters during the KMeans clustering process.

#

Step 7 | Feature Scaling

Before we move forward with the clustering and dimensionality reduction, it's imperative to scale our features. This step holds significant importance, especially in the context of distance-based algorithms like K-means and dimensionality reduction methods like PCA. Here's why:

- **For K-means Clustering**: K-means relies heavily on the concept of '**distance**' between data points to form clusters. When features are not on a similar scale, features with larger values can disproportionately influence the clustering outcome, potentially leading to incorrect groupings.
- **For PCA**: PCA aims to find the directions where the data varies the most. When features are not scaled, those with larger values might dominate these components, not accurately reflecting the underlying patterns in the data.

Methodology:

Therefore, to ensure a balanced influence on the model and to reveal the true patterns in the data, I am going to standardize our data, meaning transforming the features to have a mean of 0 and a standard deviation of 1. However, not all features require scaling. Here are the exceptions and the reasons why they are excluded:

- **CustomerID**: This feature is just an identifier for the customers and does not contain any meaningful information for clustering.
- **Is_UK**: This is a binary feature indicating whether the customer is from the UK or not. Since it already takes a value of 0 or 1, scaling it won't make any significant difference.
- **Day_Of_Week**: This feature represents the most frequent day of the week that the customer made transactions. Since it's a categorical feature represented by integers (1 to 7), scaling it would not be necessary.

I will proceed to scale the other features in the dataset to prepare it for PCA and K-means clustering.

[60]:
```
# Initialize the StandardScaler
scaler = StandardScaler()

# List of columns that don't need to be scaled
columns_to_exclude = ['CustomerID', 'Is_UK', 'Day_Of_Week']

# List of columns that need to be scaled
columns_to_scale = customer_data_cleaned.columns.difference(columns_to_exclude)

# Copy the cleaned dataset
customer_data_scaled = customer_data_cleaned.copy()

# Applying the scaler to the necessary columns in the dataset
```

```
customer_data_scaled[columns_to_scale] = scaler.
 ↪fit_transform(customer_data_scaled[columns_to_scale])

# Display the first few rows of the scaled data
customer_data_scaled.head()
```

[60]:
```
   CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
0     12346.0                  2.345802           -0.477589
1     12347.0                 -0.905575            0.707930
2     12348.0                 -0.170744           -0.003381
3     12349.0                 -0.744516           -0.714692
4     12350.0                  2.194809           -0.714692

   Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
0                 -0.754491    -0.813464                  -1.317106
1                  2.005048     2.366920                   1.528132
2                  1.863591     0.247087                   0.343279
3                 -0.047205     0.262074                   5.418285
4                 -0.534446    -0.596223                   0.043327

   Unique_Products_Purchased  Average_Days_Between_Purchases  Day_Of_Week  \
0                  -0.908471                       -0.310564            1
1                   0.815119                       -0.128438            1
2                  -0.570512                        0.672476            3
3                   0.291283                       -0.310564            0
4                  -0.655002                       -0.310564            2

       Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate  \
0 -1.086929      1                0.420541           0.417623
1  0.647126      0               -0.545753          -0.432111
2  2.814696      0               -0.545753          -0.432111
3 -1.520443      0               -0.545753          -0.432111
4  1.514154      0               -0.545753          -0.432111

   Monthly_Spending_Mean  Monthly_Spending_Std  Spending_Trend
0              -1.329018             -0.713318        0.090868
1               0.989511              1.259961        0.116774
2               0.023997              0.466213       -0.491708
3               4.159521             -0.713318        0.090868
4              -0.220428             -0.713318        0.090868
```

#

Step 8 | Dimensionality Reduction

Why We Need Dimensionality Reduction?

- **Multicollinearity Detected**: In the previous steps, we identified that our dataset contains multicollinear features. Dimensionality reduction can help us remove redundant information and alleviate the multicollinearity issue.

- **Better Clustering with K-means**: Since K-means is a distance-based algorithm, having a large number of features can sometimes dilute the meaningful underlying patterns in the data. By reducing the dimensionality, we can help K-means to find more compact and well-separated clusters.
- **Noise Reduction**: By focusing only on the most important features, we can potentially remove noise in the data, leading to more accurate and stable clusters.
- **Enhanced Visualization**: In the context of customer segmentation, being able to visualize customer groups in two or three dimensions can provide intuitive insights. Dimensionality reduction techniques can facilitate this by reducing the data to a few principal components which can be plotted easily.
- **Improved Computational Efficiency**: Reducing the number of features can speed up the computation time during the modeling process, making our clustering algorithm more efficient.

Let's proceed to select an appropriate dimensionality reduction method to our data.

Which Dimensionality Reduction Method?

In this step, we are considering the application of dimensionality reduction techniques to simplify our data while retaining the essential information. Among various methods such as KernelPCA, ICA, ISOMAP, TSNE, and UMAP, I am starting with **PCA (Principal Component Analysis)**. Here's why:

PCA is an excellent starting point because it works well in capturing linear relationships in the data, which is particularly relevant given the multicollinearity we identified in our dataset. It allows us to reduce the number of features in our dataset while still retaining a significant amount of the information, thus making our clustering analysis potentially more accurate and interpretable. Moreover, it is computationally efficient, which means it won't significantly increase the processing time.

However, it's essential to note that we are keeping our options open. After applying PCA, if we find that the first few components do not capture a significant amount of variance, indicating a loss of vital information, we might consider exploring other non-linear methods. These methods can potentially provide a more nuanced approach to dimensionality reduction, capturing complex patterns that PCA might miss, albeit at the cost of increased computational time and complexity.

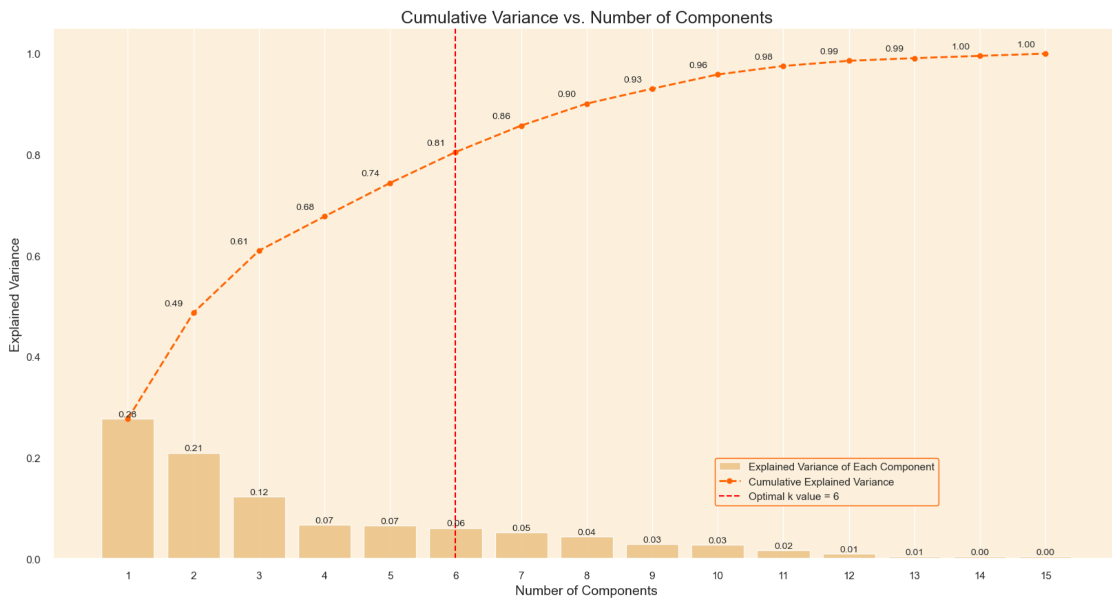
Methodology

We will apply PCA on all the available components and plot the cumulative variance explained by them. This process will allow me to visualize how much variance each additional principal component can explain, thereby helping me to pinpoint the optimal number of components to retain for the analysis:

[61]:
```
# Setting CustomerID as the index column
customer_data_scaled.set_index('CustomerID', inplace=True)

# Apply PCA
pca = PCA().fit(customer_data_scaled)
```

```python
# Calculate the Cumulative Sum of the Explained Variance
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_explained_variance = np.cumsum(explained_variance_ratio)

# Set the optimal k value (based on our analysis, we can choose 6)
optimal_k = 6

# Set seaborn plot style
sns.set(rc={'axes.facecolor': '#fcf0dc'}, style='darkgrid')

# Plot the cumulative explained variance against the number of components
plt.figure(figsize=(20, 10))

# Bar chart for the explained variance of each component
barplot = sns.barplot(x=list(range(1, len(cumulative_explained_variance) + 1)),
                      y=explained_variance_ratio,
                      color='#fcc36d',
                      alpha=0.8)

# Line plot for the cumulative explained variance
lineplot, = plt.plot(range(0, len(cumulative_explained_variance)),␣
 ↪cumulative_explained_variance,
                     marker='o', linestyle='--', color='#ff6200', linewidth=2)

# Plot optimal k value line
optimal_k_line = plt.axvline(optimal_k - 1, color='red', linestyle='--',␣
 ↪label=f'Optimal k value = {optimal_k}')

# Set labels and title
plt.xlabel('Number of Components', fontsize=14)
plt.ylabel('Explained Variance', fontsize=14)
plt.title('Cumulative Variance vs. Number of Components', fontsize=18)

# Customize ticks and legend
plt.xticks(range(0, len(cumulative_explained_variance)))
plt.legend(handles=[barplot.patches[0], lineplot, optimal_k_line],
           labels=['Explained Variance of Each Component', 'Cumulative␣
 ↪Explained Variance', f'Optimal k value = {optimal_k}'],
           loc=(0.62, 0.1),
           frameon=True,
           framealpha=1.0,
           edgecolor='#ff6200')

# Display the variance values for both graphs on the plots
x_offset = -0.3
y_offset = 0.01
```

```
for i, (ev_ratio, cum_ev_ratio) in enumerate(zip(explained_variance_ratio,␣
↪cumulative_explained_variance)):
    plt.text(i, ev_ratio, f"{ev_ratio:.2f}", ha="center", va="bottom",␣
↪fontsize=10)
    if i > 0:
        plt.text(i + x_offset, cum_ev_ratio + y_offset, f"{cum_ev_ratio:.2f}",␣
↪ha="center", va="bottom", fontsize=10)

plt.grid(axis='both')
plt.show()
```



Conclusion

The plot and the cumulative explained variance values indicate how much of the total variance in the dataset is captured by each principal component, as well as the cumulative variance explained by the first n components.

Here, we can observe that:

- The first component explains approximately 28% of the variance.
- The first two components together explain about 49% of the variance.
- The first three components explain approximately 61% of the variance, and so on.

To choose the optimal number of components, we generally look for a point where adding another component doesn't significantly increase the cumulative explained variance, often referred to as the "**elbow point**" in the curve.

From the plot, we can see that the increase in cumulative variance starts to slow down after the **6th component** (which **captures about 81% of the total variance**).

Considering the context of customer segmentation, we want to retain a sufficient amount of information to identify distinct customer groups effectively. Therefore, retaining **the first 6 components** might be a balanced choice, as they together explain a substantial portion of the total variance while reducing the dimensionality of the dataset.

[62]:
```
# Creating a PCA object with 6 components
pca = PCA(n_components=6)

# Fitting and transforming the original data to the new PCA dataframe
customer_data_pca = pca.fit_transform(customer_data_scaled)

# Creating a new dataframe from the PCA dataframe, with columns labeled PC1,␣
↪PC2, etc.
customer_data_pca = pd.DataFrame(customer_data_pca, columns=['PC'+str(i+1) for␣
↪i in range(pca.n_components_)])

# Adding the CustomerID index back to the new PCA dataframe
customer_data_pca.index = customer_data_scaled.index
```

[63]:
```
# Displaying the resulting dataframe based on the PCs
customer_data_pca.head(10)
```

[63]:

CustomerID	PC1	PC2	PC3	PC4	PC5	PC6
12346.0	-2.186469	-1.705370	-1.576745	1.008187	-0.411803	-1.658012
12347.0	3.290264	-1.387375	1.923310	-0.930990	-0.010591	0.873150
12348.0	0.584684	0.585019	0.664727	-0.655411	-0.470280	2.306657
12349.0	1.791116	-2.695652	5.850040	0.853418	0.677111	-1.520098
12350.0	-1.997139	-0.542639	0.578781	0.183682	-1.484838	0.062672
12352.0	0.428268	-1.482771	-0.758378	-0.593492	-0.376960	0.652029
12353.0	-2.391640	0.494249	-0.471649	-0.303112	-1.392030	0.827928
12354.0	0.259452	0.371751	4.204593	0.839086	-0.583905	-1.122417
12355.0	-1.341366	-2.608400	1.298483	0.321563	-0.651027	-0.416435
12358.0	-0.493193	-1.684943	1.040100	-0.525585	1.262327	-0.597757

Now, let's extract the coefficients corresponding to each principal component to better understand the transformation performed by PCA:

[64]:
```
# Define a function to highlight the top 3 absolute values in each column of a␣
↪dataframe
def highlight_top3(column):
    top3 = column.abs().nlargest(3).index
    return ['background-color:  #ffeacc' if i in top3 else '' for i in column.
↪index]

# Create the PCA component DataFrame and apply the highlighting function
```

```
pc_df = pd.DataFrame(pca.components_.T, columns=['PC{}'.format(i+1) for i in␣
↪range(pca.n_components_)],
                     index=customer_data_scaled.columns)

pc_df.style.apply(highlight_top3, axis=0)
```

[64]: <pandas.io.formats.style.Styler at 0x23391a53220>

#

Step 9 | K-Means Clustering

K-Means:

- **K-Means** is an unsupervised machine learning algorithm that clusters data into a specified number of groups (K) by minimizing the **within-cluster sum-of-squares (WCSS)**, also known as **inertia**. The algorithm iteratively assigns each data point to the nearest centroid, then updates the centroids by calculating the mean of all assigned points. The process repeats until convergence or a stopping criterion is reached.

Drawbacks of K-Means:

Here are the main drawbacks of the K-means clustering algorithm and their corresponding solutions:

- 1 **Inertia is influenced by the number of dimensions**: The value of inertia tends to increase in high-dimensional spaces due to the curse of dimensionality, which can distort the Euclidean distances between data points.

  Solution: Performing dimensionality reduction, such as **PCA**, before applying K-means to alleviate this issue and speed up computations.

- 2 **Dependence on Initial Centroid Placement**: The K-means algorithm might find a local minimum instead of a global minimum, based on where the centroids are initially placed.

  Solution: To enhance the likelihood of locating the global minimum, we can employ the **k-means++ initialization** method.

- 3 **Requires specifying the number of clusters**: K-means requires specifying the number of clusters (K) beforehand, which may not be known in advance.

  Solution: Using methods such as the **elbow method** and **silhouette analysis** to estimate the optimal number of clusters.

- 4 **Sensitivity to unevenly sized or sparse clusters**: K-means might struggle with clusters of different sizes or densities.

  Solution: Increasing the number of random initializations (n_init) or consider using algorithms that handle unevenly sized clusters better, like GMM or DBSCAN.

- 5 **Assumes convex and isotropic clusters**: K-means assumes that clusters are spherical and have similar variances, which is not always the case. It may struggle with elongated or irregularly shaped clusters.

  Solution: Considering using clustering algorithms that do not make these assumptions, such as DBSCAN or Gaussian Mixture Model (GMM).

Taking into account the aforementioned considerations, I initially applied PCA to the dataset. For the KMeans algorithm, I will set the `init` parameter to `k-means++` and `n_init` to `10`. To determine the optimal number of clusters, I will employ the elbow method and silhouette analysis. Additionally, it might be beneficial to explore the use of alternative clustering algorithms such as GMM and DBSCAN in future analyses to potentially enhance the segmentation results.

Step 9.1 | Determining the Optimal Number of Clusters

To ascertain the optimal number of clusters (k) for segmenting customers, I will explore two renowned methods:

- **Elbow Method**
- **Silhouette Method**

It's common to utilize both methods in practice to corroborate the results.

Step 9.1.1 | Elbow Method

What is the Elbow Method?

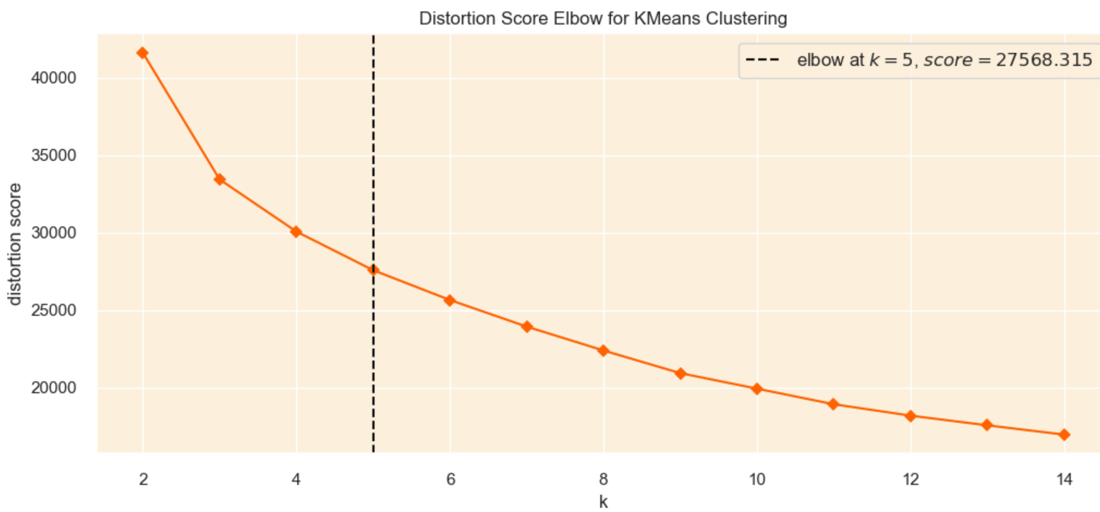
The Elbow Method is a technique for identifying the ideal number of clusters in a dataset. It involves iterating through the data, generating clusters for various values of k. The k-means algorithm calculates the sum of squared distances between each data point and its assigned cluster centroid, known as the **inertia** or **WCSS** score. By plotting the inertia score against the k value, we create a graph that typically exhibits an elbow shape, hence the name "**Elbow Method**". The **elbow point** represents the k-value where the reduction in inertia achieved by increasing k becomes negligible, indicating the optimal stopping point for the number of clusters.

Utilizing the YellowBrick Library

In this section, I will employ the **YellowBrick** library to facilitate the implementation of the **Elbow method**. YellowBrick, an extension of the Scikit-Learn API, is renowned for its ability to rapidly generate insightful visualizations in the field of machine learning.

[65]:
```
# Set plot style, and background color
sns.set(style='darkgrid', rc={'axes.facecolor': '#fcf0dc'})

# Set the color palette for the plot
sns.set_palette(['#ff6200'])

# Instantiate the clustering model with the specified parameters
km = KMeans(init='k-means++', n_init=10, max_iter=100, random_state=0)

# Create a figure and axis with the desired size
```

```
fig, ax = plt.subplots(figsize=(12, 5))

# Instantiate the KElbowVisualizer with the model and range of k values, and␣
↪disable the timing plot
visualizer = KElbowVisualizer(km, k=(2, 15), timings=False, ax=ax)

# Fit the data to the visualizer
visualizer.fit(customer_data_pca)

# Finalize and render the figure
visualizer.show();
```



Optimal k Value: Elbow Method Insights

The optimal value of k for the KMeans clustering algorithm can be found at the **elbow point**. Using the YellowBrick library for the Elbow method, we observe that the suggested optimal k value is **5**. However, **we don't have a very distinct elbow point in this case**, which is common in real-world data. From the plot, we can see that the inertia continues to decrease significantly up to k=5, indicating that **the optimum value of k could be between 3 and 7**. To choose the best k within this range, we can employ the **silhouette analysis**, another cluster quality evaluation method. Additionally, incorporating business insights can help determine a practical k value.

Step 9.1.2 | Silhouette Method

What is the Silhouette Method?

The **Silhouette Method** is an approach to find the optimal number of clusters in a dataset by evaluating the consistency within clusters and their separation from other clusters. It computes the **silhouette coefficient for each data point**, which measures how similar a point is to its own cluster compared to other clusters.

What is the Silhouette Coefficient?

To determine the silhouette coefficient for a given point i, follow these steps:

- **Calculate a(i)**: Compute the average distance between point i and all other points within its cluster.
- **Calculate b(i)**: Compute the average distance between point i and all points in the nearest cluster to its own.
- **Compute the silhouette coefficient**, s(i), for point i using the following formula:

$$s(i) = \frac{b(i) - a(i)}{\max(b(i), a(i))}$$

Note: The silhouette coefficient quantifies the similarity of a point to its own cluster (cohesion) relative to its separation from other clusters. This value ranges from -1 to 1, with higher values signifying that the point is well aligned with its cluster and has a low similarity to neighboring clusters.

What is the Silhouette Score?

The **silhouette score** is the **average silhouette coefficient** calculated for all data points in a dataset. It provides an overall assessment of the clustering quality, taking into account both cohesion within clusters and separation between clusters. A higher silhouette score indicates a better clustering configuration.

What are the Advantages of Silhouette Method over the Elbow Method?

- The **Silhouette Method** evaluates cluster quality by considering **both** the **cohesion within clusters** and their **separation** from other clusters. This provides a more comprehensive measure of clustering performance compared to the **Elbow Method**, which only considers the **inertia** (sum of squared distances within clusters).
- The **Silhouette Method** produces a silhouette score that directly quantifies the quality of clustering, making it easier to compare different values of k. In contrast, the **Elbow Method** relies on the subjective interpretation of the elbow point, which can be less reliable in cases where the plot does not show a clear elbow.
- The **Silhouette Method** generates a visual representation of silhouette coefficients for each data point, allowing for easier identification of fluctuations and outliers within clusters. This helps in determining the optimal number of clusters with higher confidence, as opposed to the **Elbow Method**, which relies on visual inspection of the inertia plot.

Methodology

In the following analysis:

- I will initially choose a range of 2-6 for the number of clusters (k) based on the Elbow method from the previous section. Next, I will plot **Silhouette scores** for each k value to determine the one with the highest score.

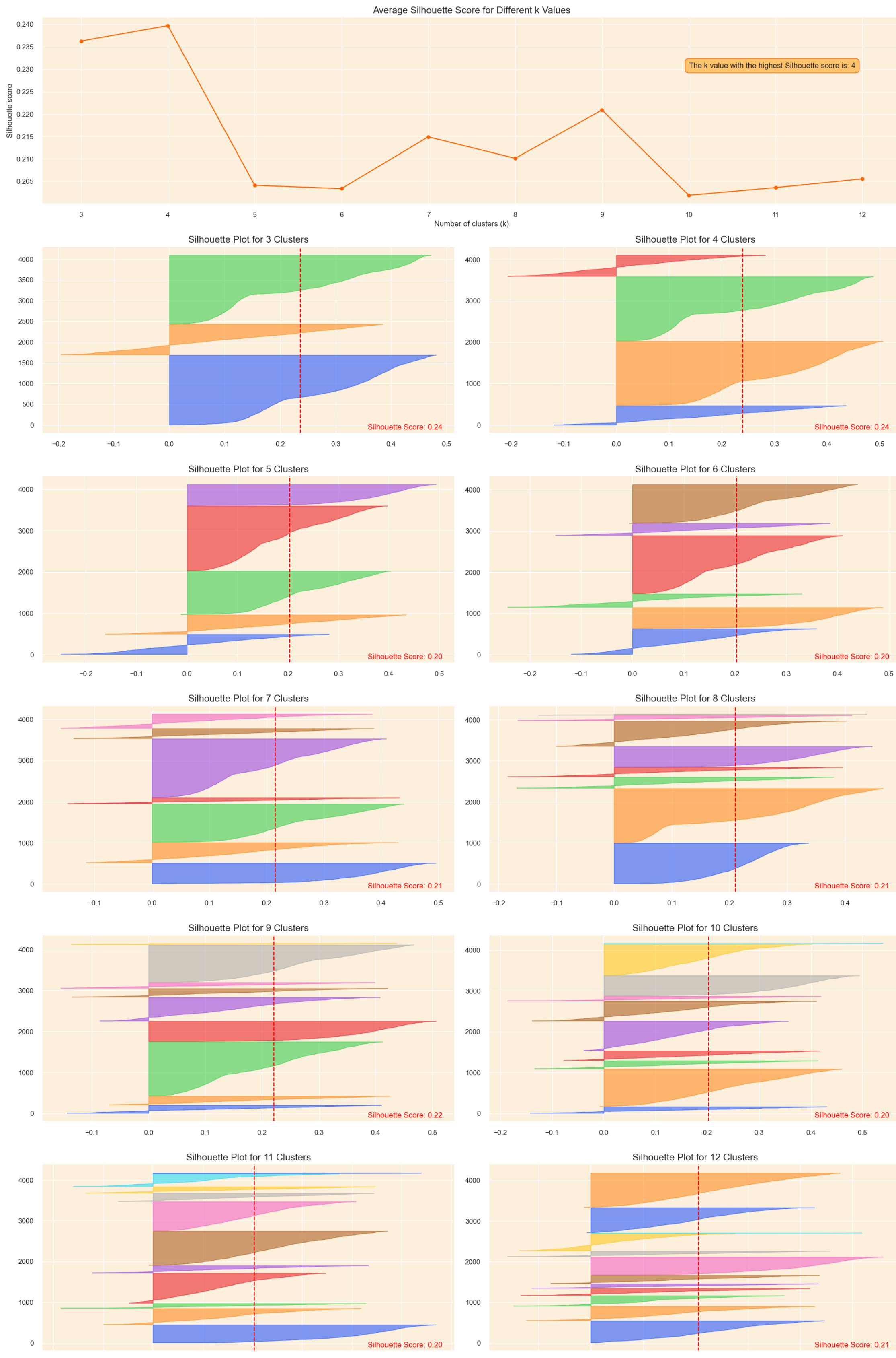
- Subsequently, to fine-tune the selection of the most appropriate k, I will generate **Silhouette plots** that visually display the **silhouette coefficients for each data point within various clusters**.

The **YellowBrick** library will be utilized once again to create these plots and facilitate a comparative analysis.

[66]:
```
def silhouette_analysis(df, start_k, stop_k, figsize=(15, 16)):
    """
    Perform Silhouette analysis for a range of k values and visualize the␣
↪results.
    """

    # Set the size of the figure
    plt.figure(figsize=figsize)

    # Create a grid with (stop_k - start_k + 1) rows and 2 columns
    grid = gridspec.GridSpec(stop_k - start_k + 1, 2)

    # Assign the first plot to the first row and both columns
    first_plot = plt.subplot(grid[0, :])

    # First plot: Silhouette scores for different k values
    sns.set_palette(['darkorange'])

    silhouette_scores = []

    # Iterate through the range of k values
    for k in range(start_k, stop_k + 1):
        km = KMeans(n_clusters=k, init='k-means++', n_init=10, max_iter=100,␣
↪random_state=0)
        km.fit(df)
        labels = km.predict(df)
        score = silhouette_score(df, labels)
        silhouette_scores.append(score)

    best_k = start_k + silhouette_scores.index(max(silhouette_scores))

    plt.plot(range(start_k, stop_k + 1), silhouette_scores, marker='o')
    plt.xticks(range(start_k, stop_k + 1))
    plt.xlabel('Number of clusters (k)')
    plt.ylabel('Silhouette score')
    plt.title('Average Silhouette Score for Different k Values', fontsize=15)

    # Add the optimal k value text to the plot
    optimal_k_text = f'The k value with the highest Silhouette score is:␣
↪{best_k}'
    plt.text(10, 0.23, optimal_k_text, fontsize=12, verticalalignment='bottom',
```

```
            horizontalalignment='left', bbox=dict(facecolor='#fcc36d',␣
↪edgecolor='#ff6200', boxstyle='round, pad=0.5'))


    # Second plot (subplot): Silhouette plots for each k value
    colors = sns.color_palette("bright")

    for i in range(start_k, stop_k + 1):
        km = KMeans(n_clusters=i, init='k-means++', n_init=10, max_iter=100,␣
↪random_state=0)
        row_idx, col_idx = divmod(i - start_k, 2)

        # Assign the plots to the second, third, and fourth rows
        ax = plt.subplot(grid[row_idx + 1, col_idx])

        visualizer = SilhouetteVisualizer(km, colors=colors, ax=ax)
        visualizer.fit(df)

        # Add the Silhouette score text to the plot
        score = silhouette_score(df, km.labels_)
        ax.text(0.97, 0.02, f'Silhouette Score: {score:.2f}', fontsize=12, \
                ha='right', transform=ax.transAxes, color='red')

        ax.set_title(f'Silhouette Plot for {i} Clusters', fontsize=15)

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

[67]:
```
silhouette_analysis(customer_data_pca, 3, 12, figsize=(20, 50))
```


Average Silhouette Score for Different k Values
Silhouette score
Number of clusters (k)
The k value with the highest Silhouette score is: 4
Silhouette Plot for 3 Clusters
Silhouette Score: 0.24
Silhouette Plot for 4 Clusters
Silhouette Score: 0.24
Silhouette Plot for 5 Clusters
Silhouette Score: 0.20
Silhouette Plot for 6 Clusters
Silhouette Score: 0.20
Silhouette Plot for 7 Clusters
Silhouette Score: 0.21
Silhouette Plot for 8 Clusters
Silhouette Score: 0.21
Silhouette Plot for 9 Clusters
Silhouette Score: 0.22
Silhouette Plot for 10 Clusters
Silhouette Score: 0.20
Silhouette Plot for 11 Clusters
Silhouette Score: 0.20
Silhouette Plot for 12 Clusters
Silhouette Score: 0.21

Guidelines to Interpret Silhouette Plots and Determine the Optimal K:

To interpret silhouette plots and identify the optimal number of clusters ((k)), consider the following criteria:

- 1 **Analyze the Silhouette Plots**:
  - **Silhouette Score Width**:
    - **Wide Widths (closer to +1)**: Indicate that the data points in the cluster are well separated from points in other clusters, suggesting well-defined clusters.
    - **Narrow Widths (closer to -1)**: Show that data points in the cluster are not distinctly separated from other clusters, indicating poorly defined clusters.
  - **Average Silhouette Score**:
    - **High Average Width**: A cluster with a high average silhouette score indicates well-separated clusters.
    - **Low Average Width**: A cluster with a low average silhouette score indicates poor separation between clusters.

- 2 **Uniformity in Cluster Size**:

  2.1 **Cluster Thickness**:

  - **Uniform Thickness**: Indicates that clusters have a roughly equal number of data points, suggesting a balanced clustering structure.
  - **Variable Thickness**: Signifies an imbalance in the data point distribution across clusters, with some clusters having many data points and others too few.

- 3 **Peaks in Average Silhouette Score**:
  - **Clear Peaks**: A clear peak in the **average** silhouette score plot for a specific (k) value indicates this (k) might be optimal.

- 4 **Minimize Fluctuations in Silhouette Plot Widths**:
  - **Uniform Widths**: Seek silhouette plots with similar widths across clusters, suggesting a more balanced and optimal clustering.
  - **Variable Widths**: Avoid wide fluctuations in silhouette plot widths, indicating that clusters are not well-defined and may vary in compactness.

- 5 **Optimal Cluster Selection**:
  - **Maximize the Overall Average Silhouette Score**: Choose the (k) value that gives the highest average silhouette score across all clusters, indicating well-defined clusters.
  - **Avoid Below-Average Silhouette Scores**: Ensure most clusters have above-average silhouette scores to prevent suboptimal clustering structures.

- 6 **Visual Inspection of Silhouette Plots**:
  - **Consistent Cluster Formation**: Visually inspect the silhouette plots for each (k) value to evaluate the consistency and structure of the formed clusters.
  - **Cluster Compactness**: Look for more compact clusters, with data points having silhouette scores closer to +1, indicating better clustering.

Optimal k Value: Silhouette Method Insights

Based on above guidelines and after carefully considering the silhouette plots, it's clear that choosing **(k = 3)** is the better option. This choice gives us clusters that are more evenly matched and well-defined, making our clustering solution stronger and more reliable.

Step 9.2 | Clustering Model - K-means

In this step, I am going to apply the K-means clustering algorithm to segment customers into different clusters based on their purchasing behaviors and other characteristics, using the optimal number of clusters determined in the previous step.

It's important to note that the K-means algorithm might assign different labels to the clusters in each run. To address this, we have taken an additional step to swap the labels based on the frequency of samples in each cluster, ensuring a consistent label assignment across different runs.

[68]:
```
# Apply KMeans clustering using the optimal k
kmeans = KMeans(n_clusters=4, init='k-means++', n_init=10, max_iter=100,␣
↪random_state=0)
kmeans.fit(customer_data_pca)

# Get the frequency of each cluster
cluster_frequencies = Counter(kmeans.labels_)

# Create a mapping from old labels to new labels based on frequency
label_mapping = {label: new_label for new_label, (label, _) in
                 enumerate(cluster_frequencies.most_common())}

# Reverse the mapping to assign labels as per your criteria
label_mapping = {v: k for k, v in {3: 0 ,2: 1, 1: 2, 0: 3}.items()}

# Apply the mapping to get the new labels
new_labels = np.array([label_mapping[label] for label in kmeans.labels_])

# Append the new cluster labels back to the original dataset
customer_data_cleaned['cluster'] = new_labels

# Append the new cluster labels to the PCA version of the dataset
customer_data_pca['cluster'] = new_labels
```

[69]:
```
# Display the first few rows of the original dataframe
customer_data_cleaned.head(10)
```

[69]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions
0	12346.0	325	2
1	12347.0	2	7
2	12348.0	75	4
3	12349.0	18	1
4	12350.0	310	1
5	12352.0	36	8
6	12353.0	204	1
7	12354.0	232	1
8	12355.0	214	1
9	12358.0	1	2

	Total_Products_Purchased	Total_Spend	Average_Transaction_Value
0	0	0.0	0.0
1	2458	4310.0	615.714286
2	2332	1437.24	359.31
3	630	1457.55	1457.55
4	196	294.4	294.4
5	463	1265.41	158.17625
6	20	89.0	89.0
7	530	1079.4	1079.4
8	240	459.4	459.4
9	242	928.06	464.03

	Unique_Products_Purchased	Average_Days_Between_Purchases	Day_Of_Week
0	1	0.0	1
1	103	2.016575	1
2	21	10.884615	3
3	72	0.0	0
4	16	0.0	2
5	57	3.13253	1
6	4	0.0	3
7	58	0.0	3
8	13	0.0	0
9	12	9.3125	1

	Hour	Is_UK	Cancellation_Frequency	Cancellation_Rate
0	10	1	1	0.5
1	14	0	0	0.0
2	19	0	0	0.0
3	9	0	0	0.0
4	16	0	0	0.0
5	14	0	1	0.125
6	17	0	0	0.0
7	13	0	0	0.0
8	13	0	0	0.0
9	10	0	0	0.0

	Monthly_Spending_Mean	Monthly_Spending_Std	Spending_Trend	cluster
0	0.0	0.0	0.0	2
1	615.714286	341.070789	4.486071	0
2	359.31	203.875689	-100.884	0
3	1457.55	0.0	0.0	0
4	294.4	0.0	0.0	2
5	316.3525	134.700629	9.351	2
6	89.0	0.0	0.0	1
7	1079.4	0.0	0.0	0
8	459.4	0.0	0.0	2
9	464.03	83.679016	118.34	2

#

Step 10 | Clustering Evaluation

After determining the optimal number of clusters (which is 3 in our case) using elbow and silhouette analyses, I move onto the evaluation step to assess the quality of the clusters formed. This step is essential to validate the effectiveness of the clustering and to ensure that the clusters are **coherent** and **well-separated**. The evaluation metrics and a visualization technique I plan to use are outlined below:

- 1 **3D Visualization of Top PCs**
- 2 **Cluster Distribution Visualization**
- 3 **Evaluation Metrics**
  - Silhouette Score
  - Calinski Harabasz Score
  - Davies Bouldin Score

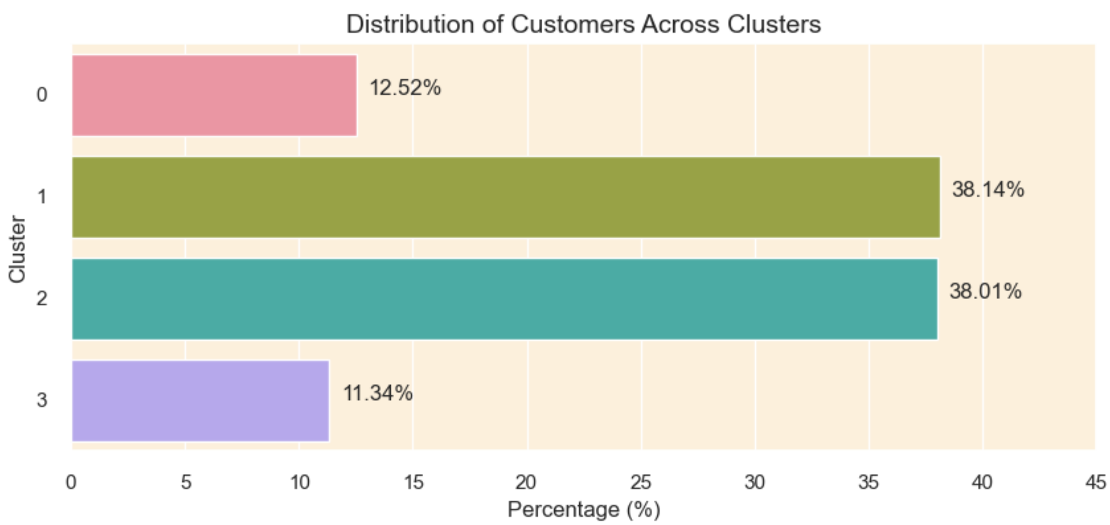
Note: We are using the PCA version of the dataset for evaluation because this is the space where the clusters were actually formed, capturing the most significant patterns in the data. Evaluating in this space ensures a more accurate representation of the cluster quality, helping us understand the true cohesion and separation achieved during clustering. This approach also aids in creating a clearer 3D visualization using the top principal components, illustrating the actual separation between clusters.

| Cluster Distribution Visualization

I am going to utilize a bar plot to visualize the percentage of customers in each cluster, which helps in understanding if the clusters are balanced and significant:

[70]:
```
# Calculate the percentage of customers in each cluster
cluster_percentage = (customer_data_pca['cluster'].value_counts(normalize=True)␣
↪* 100).reset_index()
cluster_percentage.columns = ['Cluster', 'Percentage']
cluster_percentage.sort_values(by='Cluster', inplace=True)
```

```python
# Create a horizontal bar plot
plt.figure(figsize=(10, 4))
sns.barplot(x='Percentage', y='Cluster', data=cluster_percentage, orient='h')

# Adding percentages on the bars
for index, value in enumerate(cluster_percentage['Percentage']):
    plt.text(value+0.5, index, f'{value:.2f}%')

plt.title('Distribution of Customers Across Clusters', fontsize=14)
plt.xticks(ticks=np.arange(0, 50, 5))
plt.xlabel('Percentage (%)')

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



Inference

The distribution of customers across the clusters, as depicted by the bar plot, suggests a fairly balanced distribution with clusters 0 and 1 holding around 41% of customers each and cluster 2 accommodating approximately 18% of the customers.

This balanced distribution indicates that our clustering process has been largely successful in identifying meaningful patterns within the data, rather than merely grouping noise or outliers. It implies that each cluster represents a substantial and distinct segment of the customer base, thereby offering valuable insights for future business strategies.

Moreover, the fact that no cluster contains a very small percentage of customers, assures us that each cluster is significant and not just representing outliers or noise in the data. This setup allows for a more nuanced understanding and analysis of different customer segments, facilitating effective and informed decision-making.

| Evaluation Metrics

To further scrutinize the quality of our clustering, I will employ the following metrics:

- **Silhouette Score**: A measure to evaluate the separation distance between the clusters. Higher values indicate better cluster separation. It ranges from -1 to 1.
- **Calinski Harabasz Score**: This score is used to evaluate the dispersion between and within clusters. A higher score indicates better defined clusters.
- **Davies Bouldin Score**: It assesses the average similarity between each cluster and its most similar cluster. Lower values indicate better cluster separation.

[71]:
```
# Compute number of customers
num_observations = len(customer_data_pca)

# Separate the features and the cluster labels
X = customer_data_pca.drop('cluster', axis=1)
clusters = customer_data_pca['cluster']

# Compute the metrics
sil_score = silhouette_score(X, clusters)
calinski_score = calinski_harabasz_score(X, clusters)
davies_score = davies_bouldin_score(X, clusters)

# Create a table to display the metrics and the number of observations
table_data = [
    ["Number of Observations", num_observations],
    ["Silhouette Score", sil_score],
    ["Calinski Harabasz Score", calinski_score],
    ["Davies Bouldin Score", davies_score]
]

# Print the table
print(tabulate(table_data, headers=["Metric", "Value"], tablefmt='pretty'))
```

```
+-------------------------+---------------------+
|         Metric          |        Value        |
+-------------------------+---------------------+
| Number of Observations  |        4067         |
|    Silhouette Score     | 0.23970087614430569 |
| Calinski Harabasz Score | 1083.1662511726674  |
|  Davies Bouldin Score   | 1.4731764647334638  |
+-------------------------+---------------------+
```

Clustering Quality Inference

- The **Silhouette Score** of approximately 0.236, although not close to 1, still indicates a fair amount of separation between the clusters. It suggests that the clusters are somewhat distinct, but there might be slight overlaps between them. Generally, a score closer to 1 would be ideal, indicating more distinct and well-separated clusters.

- The **Calinski Harabasz Score** is 1257.17, which is considerably high, indicating that the clusters are well-defined. A higher score in this metric generally signals better cluster definitions, thus implying that our clustering has managed to find substantial structure in the data.
- The **Davies Bouldin Score** of 1.37 is a reasonable score, indicating a moderate level of similarity between each cluster and its most similar one. A lower score is generally better as it indicates less similarity between clusters, and thus, our score here suggests a decent separation between the clusters.

In conclusion, the metrics suggest that the clustering is of good quality, with clusters being well-defined and fairly separated. However, there might still be room for further optimization to enhance cluster separation and definition, potentially by trying other clustering and dimensionality reduction algorithms.

#

Step 11 | Cluster Analysis and Profiling

Tabel of Contents

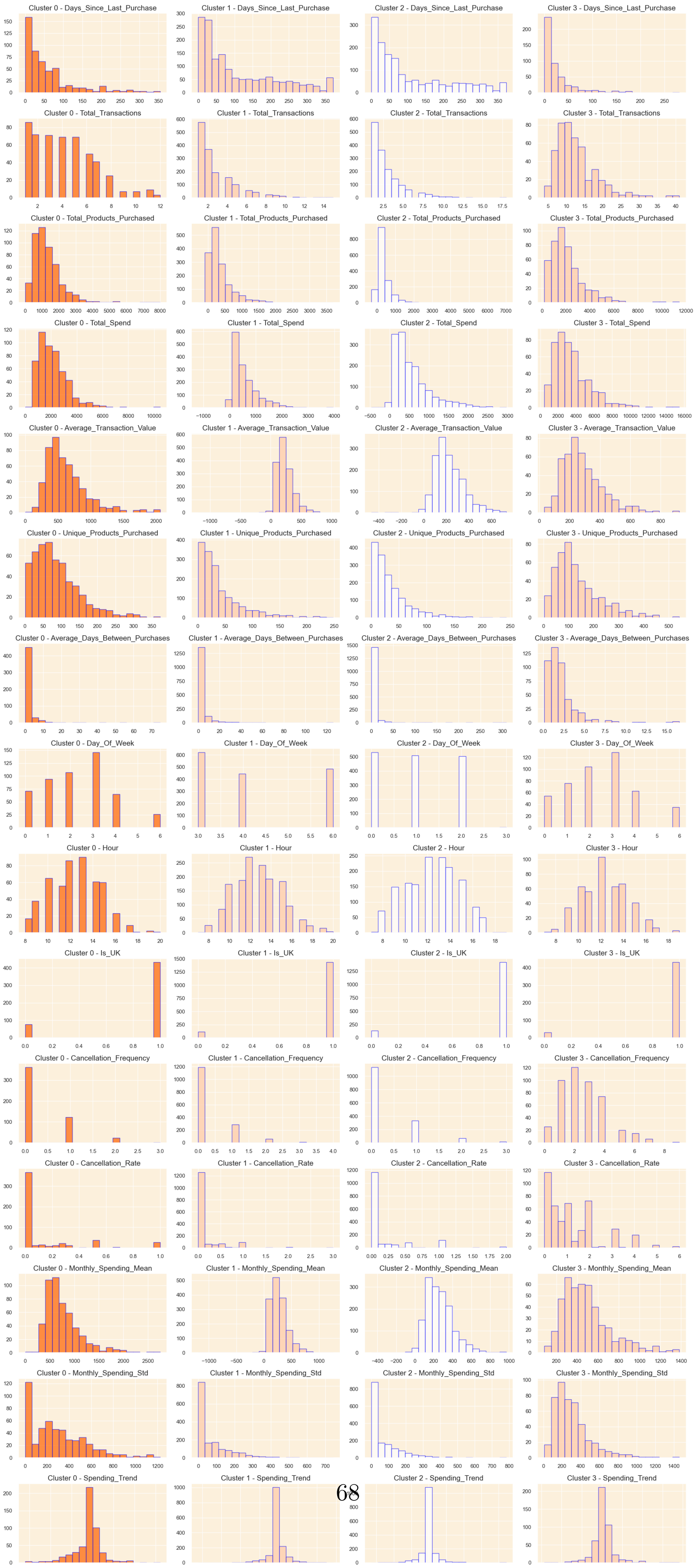
In this section, We are going to analyze the characteristics of each cluster to understand the distinct behaviors and preferences of different customer segments and also profile each cluster to identify the key traits that define the customers in each cluster.

Step 11.1 | Histogram Chart Approach

We can plot histograms for each feature segmented by the cluster labels. These histograms will allow us to visually inspect the distribution of feature values within each cluster.

[72]:
```
# Plot histograms for each feature segmented by the clusters
features = customer_data_cleaned.columns[1:-1]
clusters = customer_data_cleaned['cluster'].unique()
clusters.sort()

# Setting up the subplots
n_rows = len(features)
n_cols = len(clusters)
fig, axes = plt.subplots(n_rows, n_cols, figsize=(20, 3*n_rows))

# Plotting histograms
for i, feature in enumerate(features):
    for j, cluster in enumerate(clusters):
        data = customer_data_cleaned[customer_data_cleaned['cluster'] ==␣
↪cluster][feature]
        axes[i, j].hist(data, bins=20, color=colors[j], edgecolor='Blue',␣
↪alpha=0.7)
        axes[i, j].set_title(f'Cluster {cluster} - {feature}', fontsize=15)
        axes[i, j].set_xlabel('')
        axes[i, j].set_ylabel('')
```

```python
# Adjusting layout to prevent overlapping
plt.tight_layout()
plt.show()
```


Cluster 0 - Days_Since_Last_Purchase
Cluster 1 - Days_Since_Last_Purchase
Cluster 2 - Days_Since_Last_Purchase
Cluster 3 - Days_Since_Last_Purchase
Cluster 0 - Total_Transactions
Cluster 1 - Total_Transactions
Cluster 2 - Total_Transactions
Cluster 3 - Total_Transactions
Cluster 0 - Total_Products_Purchased
Cluster 1 - Total_Products_Purchased
Cluster 2 - Total_Products_Purchased
Cluster 3 - Total_Products_Purchased
Cluster 0 - Total_Spend
Cluster 1 - Total_Spend
Cluster 2 - Total_Spend
Cluster 3 - Total_Spend
Cluster 0 - Average_Transaction_Value
Cluster 1 - Average_Transaction_Value
Cluster 2 - Average_Transaction_Value
Cluster 3 - Average_Transaction_Value
Cluster 0 - Unique_Products_Purchased
Cluster 1 - Unique_Products_Purchased
Cluster 2 - Unique_Products_Purchased
Cluster 3 - Unique_Products_Purchased
Cluster 0 - Average_Days_Between_Purchases
Cluster 1 - Average_Days_Between_Purchases
Cluster 2 - Average_Days_Between_Purchases
Cluster 3 - Average_Days_Between_Purchases
Cluster 0 - Day_Of_Week
Cluster 1 - Day_Of_Week
Cluster 2 - Day_Of_Week
Cluster 3 - Day_Of_Week
Cluster 0 - Hour
Cluster 1 - Hour
Cluster 2 - Hour
Cluster 3 - Hour
Cluster 0 - Is_UK
Cluster 1 - Is_UK
Cluster 2 - Is_UK
Cluster 3 - Is_UK
Cluster 0 - Cancellation_Frequency
Cluster 1 - Cancellation_Frequency
Cluster 2 - Cancellation_Frequency
Cluster 3 - Cancellation_Frequency
Cluster 0 - Cancellation_Rate
Cluster 1 - Cancellation_Rate
Cluster 2 - Cancellation_Rate
Cluster 3 - Cancellation_Rate
Cluster 0 - Monthly_Spending_Mean
Cluster 1 - Monthly_Spending_Mean
Cluster 2 - Monthly_Spending_Mean
Cluster 3 - Monthly_Spending_Mean
Cluster 0 - Monthly_Spending_Std
Cluster 1 - Monthly_Spending_Std
Cluster 2 - Monthly_Spending_Std
Cluster 3 - Monthly_Spending_Std
Cluster 0 - Spending_Trend
Cluster 1 - Spending_Trend
Cluster 2 - Spending_Trend
Cluster 3 - Spending_Trend

#

Step 12 | Recommendation System

In the final phase of this project, I am set to develop a recommendation system to enhance the online shopping experience. This system will suggest products to customers based on the purchasing patterns prevalent in their respective clusters. Earlier in the project, during the customer data preparation stage, I isolated a small fraction (5%) of the customers identified as outliers and reserved them in a separate dataset called `outliers_data`.

Now, focusing on the core 95% of the customer group, I analyze the cleansed customer data to pinpoint the top-selling products within each cluster. Leveraging this information, the system will craft personalized recommendations, suggesting **the top three products** popular within their cluster that they have not yet purchased. This not only facilitates targeted marketing strategies but also enriches the personal shopping experience, potentially boosting sales. For the outlier group, a basic approach could be to recommend random products, as a starting point to engage them.

[92]:
```python
# Step 1: Extract the CustomerIDs of the outliers and remove their transactions␣
↪from the main dataframe
outlier_customer_ids = outliers_data['CustomerID'].astype('float').unique()
df_filtered = df[~df['CustomerID'].isin(outlier_customer_ids)]

# Step 2: Ensure consistent data type for CustomerID across both dataframes␣
↪before merging
customer_data_cleaned['CustomerID'] = customer_data_cleaned['CustomerID'].
↪astype('float')

# Step 3: Merge the transaction data with the customer data to get the cluster␣
↪information for each transaction
merged_data = df_filtered.merge(customer_data_cleaned[['CustomerID',␣
↪'cluster']], on='CustomerID', how='inner')

# Step 4: Identify the top 10 best-selling products in each cluster based on␣
↪the total quantity sold
best_selling_products = merged_data.groupby(['cluster', 'StockCode',␣
↪'Description'])['Quantity'].sum().reset_index()
best_selling_products = best_selling_products.sort_values(by=['cluster',␣
↪'Quantity'], ascending=[True, False])
top_products_per_cluster = best_selling_products.groupby('cluster').head(10)

# Step 5: Create a record of products purchased by each customer in each cluster
customer_purchases = merged_data.groupby(['CustomerID', 'cluster',␣
↪'StockCode'])['Quantity'].sum().reset_index()

# Step 6: Generate recommendations for each customer in each cluster
```

```
recommendations = []
for cluster in top_products_per_cluster['cluster'].unique():
    top_products = top_products_per_cluster[top_products_per_cluster['cluster']␣
 ↪== cluster]
    customers_in_cluster =␣
 ↪customer_data_cleaned[customer_data_cleaned['cluster'] ==␣
 ↪cluster]['CustomerID']

    for customer in customers_in_cluster:
        # Identify products already purchased by the customer
        customer_purchased_products =␣
 ↪customer_purchases[(customer_purchases['CustomerID'] == customer) &
                                                                   ␣
 ↪(customer_purchases['cluster'] == cluster)]['StockCode'].tolist()

        # Find top 3 products in the best-selling list that the customer hasn't␣
 ↪purchased yet
        top_products_not_purchased = top_products[~top_products['StockCode'].
 ↪isin(customer_purchased_products)]
        top_3_products_not_purchased = top_products_not_purchased.head(3)

        # Append the recommendations to the list
        recommendations.append([customer, cluster] +␣
 ↪top_3_products_not_purchased[['StockCode', 'Description']].values.flatten().
 ↪tolist())

# Step 7: Create a dataframe from the recommendations list and merge it with␣
 ↪the original customer data
recommendations_df = pd.DataFrame(recommendations, columns=['CustomerID',␣
 ↪'cluster', 'Rec1_StockCode', 'Rec1_Description', \
                                                 'Rec2_StockCode',␣
 ↪'Rec2_Description', 'Rec3_StockCode', 'Rec3_Description'])
customer_data_with_recommendations = customer_data_cleaned.
 ↪merge(recommendations_df, on=['CustomerID', 'cluster'], how='right')
```

[93]:

```
# Display 10 random rows from the customer_data_with_recommendations dataframe
customer_data_with_recommendations.set_index('CustomerID').iloc[:, -6:].
 ↪sample(10, random_state=0)
```

[93]:

```
           Rec1_StockCode                     Rec1_Description Rec2_StockCode  \
CustomerID
16091.0             18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          17003
16080.0             18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          17003
16628.0             84077    WORLD WAR 2 GLIDERS ASSTD DESIGNS          84879
16601.0             84077    WORLD WAR 2 GLIDERS ASSTD DESIGNS          84879
14362.0             84077    WORLD WAR 2 GLIDERS ASSTD DESIGNS          84879
```

```
14520.0            18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          17003
14525.0            22616           PACK OF 12 LONDON TISSUES           84077
17877.0            18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          17003
17722.0            22616           PACK OF 12 LONDON TISSUES           84077
13192.0            18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          17003

                            Rec2_Description Rec3_StockCode  \
CustomerID
16091.0                    BROCADE RING PURSE          84879
16080.0                    BROCADE RING PURSE          84879
16628.0          ASSORTED COLOUR BIRD ORNAMENT        85123A
16601.0          ASSORTED COLOUR BIRD ORNAMENT        85123A
14362.0          ASSORTED COLOUR BIRD ORNAMENT        85123A
14520.0                    BROCADE RING PURSE          84879
14525.0      WORLD WAR 2 GLIDERS ASSTD DESIGNS         84879
17877.0                    BROCADE RING PURSE          84879
17722.0      WORLD WAR 2 GLIDERS ASSTD DESIGNS         84879
13192.0                    BROCADE RING PURSE         85123A

                             Rec3_Description
CustomerID
16091.0           ASSORTED COLOUR BIRD ORNAMENT
16080.0           ASSORTED COLOUR BIRD ORNAMENT
16628.0      WHITE HANGING HEART T-LIGHT HOLDER
16601.0      WHITE HANGING HEART T-LIGHT HOLDER
14362.0      WHITE HANGING HEART T-LIGHT HOLDER
14520.0           ASSORTED COLOUR BIRD ORNAMENT
14525.0           ASSORTED COLOUR BIRD ORNAMENT
17877.0           ASSORTED COLOUR BIRD ORNAMENT
17722.0           ASSORTED COLOUR BIRD ORNAMENT
13192.0      WHITE HANGING HEART T-LIGHT HOLDER
```