Customer Segmentation Using K-means Algorithm

February 12, 2025

1 Customer Segmentation using Unsupervised Machine Learning Algorithm in Python

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.

1.0.1 Import Libraries

[1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import plotly.graph_objects as go
```

```
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 yellowbrick.cluster import KElbowVisualizer, SilhouetteVisualizer
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

%matplotlib inline

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

[2]:
```
# Initializing Plotly for use in the notebook
from plotly.offline import init_notebook_mode
init_notebook_mode(connected=True)
```

[3]:
```
# Configuring Seaborn plot styles: Seting background color and using dark grid
sns.set(rc={'axes.facecolor': '#fcf0dc'}, style='darkgrid')
```

1.0.2 Loading the Dataset

[4]:
```
df = pd.read_csv(r"C:\Users\user\Desktop\E-books\Datasets1\CustomerSegmentation.
 ↪csv", encoding="ISO-8859-1")
```

Dataset Description

- **InvoiceNo** - Code representing each unique transaction. If this code starts with letter 'c', it indicates a cancellation.
- **StockCode** - Code uniquely assigned to each distinct product.
- **Description** - Description of each product.
- **Quantity** - The number of units of a product in a transaction.
- **InvoiceDate** - The date and time of the transaction.
- **UnitPrice** - The unit price of the product in sterling.
- **CustomerID** - Identifier uniquely assigned to each customer.
- **Country** - The country of the customer.

Dataset Overview First I will perform a preliminary analysis to understand the structure and types of data columns:

[5]:
```
df.head(10)
```

[5]:
```
  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
```

[6]:
```
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
```

Inferences: The dataset consists of **541,909 entries** and **8 columns**. Here is a brief overview of each column:

- **InvoiceNo:** This is an object data type column that contains the invoice number for each transaction. Each invoice number can represent multiple items purchased in a single transaction.

- **StockCode:** An object data type column representing the product code for each item.
- **Description:** This column, also an object data type, contains descriptions of the products. It has some missing values, with 540,455 non-null entries out of 541,909.
- **Quantity:** This is an integer column indicating the quantity of products purchased in each transaction.
- **InvoiceDate:** A datetime column that records the date and time of each transaction.
- **UnitPrice:** A float column representing the unit price of each product.
- **CustomerID:** A float column that contains the customer ID for each transaction. This column has a significant number of missing values, with only 406,829 non-null entries out of 541,909.
- **Country:** An object column recording the country where each transaction took place.

From a preliminary overview, it seems that there are missing values in the Description and CustomerID columns which need to be addressed. The InvoiceDate column is already in datetime format, which will facilitate further time series analysis. We also observe that a single customer can have multiple transactions as inferred from the repeated CustomerID in the initial rows.

The next steps would include deeper data cleaning and preprocessing to handle missing values, potentially erroneous data, and to create new features that can help in achieving the project goals.

Summary Statistics Generatimg summary statistics will help gain initial insights into the distribution of data

[7]:
```
#summary statistics for numerical  variables
df.describe().T
```

[7]:
```
               count          mean          std        min       25%       50%  \
Quantity    541909.0      9.552250   218.081158 -80995.00      1.00      3.00
UnitPrice   541909.0      4.611114    96.759853 -11062.06      1.25      2.08
CustomerID  406829.0  15287.690570  1713.600303  12346.00  13953.00  15152.00

                 75%      max
Quantity       10.00  80995.0
UnitPrice       4.13  38970.0
CustomerID  16791.00  18287.0
```

[8]:
```
#summary statistics for categorical  variables
df.describe(include = "object").T
```

[8]:
```
              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
```

Inferences: **Quantity:** - The average quantity of products in a transaction is approximately 9.55. - The quantity has a wide range, with a minimum value of -80995 and a maximum value of 80995. The negative values indicate returned or cancelled orders, which need to be handled appropriately. - The standard deviation is quite large, indicating a significant spread in the data. The presence of outliers is indicated by a large difference between the maximum and the 75th percentile values.

UnitPrice: - The average unit price of the products is approximately 4.61. - The unit price also shows a wide range, from -11062.06 to 38970, which suggests the presence of errors or noise in the data, as negative prices don't make sense. - Similar to the Quantity column, the presence of outliers is indicated by a large difference between the maximum and the 75th percentile values.

CustomerID: - There are 406829 non-null entries, indicating missing values in the dataset which need to be addressed. - The Customer IDs range from 12346 to 18287, helping in identifying unique customers.

InvoiceNo: - There are 25900 unique invoice numbers, indicating 25900 separate transactions. - The most frequent invoice number is 573585, appearing 1114 times, possibly representing a large transaction or an order with multiple items.

StockCode: - There are 4070 unique stock codes representing different products. - The most frequent stock code is 85123A, appearing 2313 times in the dataset.

Description: - There are 4223 unique product descriptions. - The most frequent product description is "WHITE HANGING HEART T-LIGHT HOLDER", appearing 2369 times. - There are some missing values in this column which need to be treated.

Country: - The transactions come from 38 different countries, with a dominant majority of the transactions (approximately 91.4%) originating from the United Kingdom.

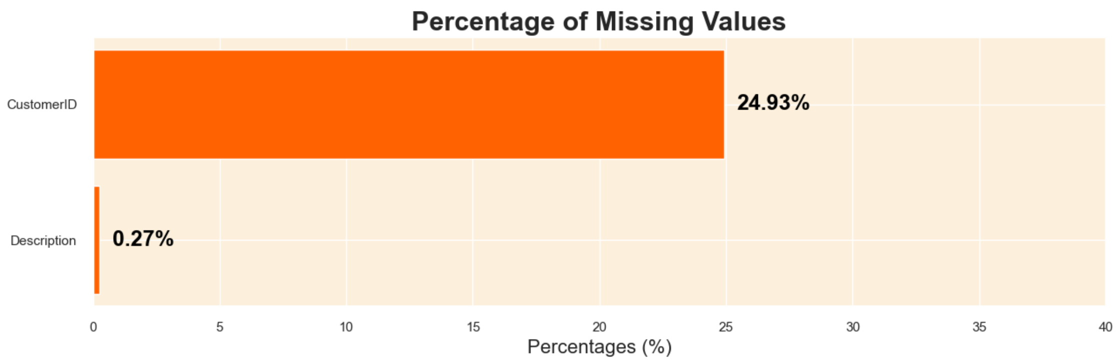
1.0.3 Data Cleaning & Transformation

This step encompasses a comprehensive cleaning and transformation process to refine the dataset. It includes addressing missing values, eliminating duplicate entries, correcting anomalies in product codes and descriptions, and other necessary adjustments to prepare the data for in-depth analysis and modeling.

Handling Missing Values I will determine the percentage of missing values present in each column and select the most effective strategy to address them:

[9]:
```
# 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)

# Ploting the barh chart
fig, ax = plt.subplots(figsize=(15, 4))
ax.barh(missing_percentage.index, missing_percentage, color='#ff6200')

# Annotating 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')

# Seting x-axis limit
ax.set_xlim([0, 40])

# Adding 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 quarter 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.
- Moreover, since the clustering is based on customer behavior and preferences, it's crucial to have accurate data on customer identifiers. Therefore, removing the rows with missing CustomerIDs seems to be the most reasonable approach to maintain the integrity of the clusters and 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.

[10]:
```
# Extracting rows with missing values in 'CustomerID' or 'Description' columns
df[df["CustomerID"].isnull()|df["Description"].isnull()].head()
```

[10]:
```
      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

          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
```

[11]:
```
# Removing rows with missing values in 'CustomerID' and 'Description' columns
df = df.dropna(subset=['CustomerID', 'Description'])
```

[12]:
```
df.isnull().sum().sum()
```

[12]: 0

Handling Duplicates

[13]:
```
# Finding duplicate rows (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(10)
```

[13]:
```
    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
```

```
        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
```

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.

[14]:
```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.
```

[15]:
```python
df.shape[0]
```

[15]: 401604

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:

[16]:
```python
# Filtering 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')
```

```
# Analyzing the characteristics of these rows (considering the new column)
cancelled_transactions = df[df['Transaction_Status'] == 'Cancelled']
cancelled_transactions.describe().drop('CustomerID', axis=1)
```

[16]:

	Quantity	UnitPrice
count	8872.000000	8872.000000
mean	-30.774910	18.899512
std	1172.249902	445.190864
min	-80995.000000	0.010000
25%	-6.000000	1.450000
50%	-2.000000	2.950000
75%	-1.000000	4.950000
max	-1.000000	38970.000000

Inferences from the Cancelled Transactions Data: All quantities in the cancelled transactions are negative, indicating that these are indeed orders that were cancelled. The UnitPrice column has a considerable spread, showing that a variety of products, from low to high value, were part of the cancelled transactions.

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.

[17]:
```
# 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%
```

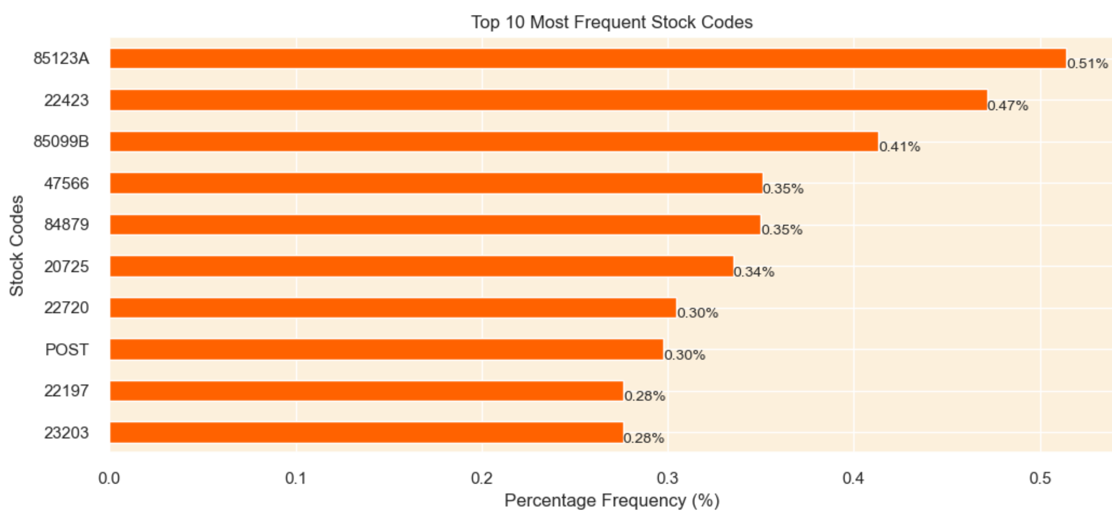
Correcting StockCode Anomalies Lets find the number of unique stock codes and to plot the top 10 most frequent stock codes along with their percentage frequency:

[18]:
```
# 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
```

[19]:

```python
# 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=(12, 5))
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 contains 3684 unique stock codes, indicating a substantial variety of products available in the online retail store. This diversity can potentially lead to the identification of distinct customer clusters, with preferences for different types of products. Popular Items: A closer look at the top 10 most frequent stock codes can offer insights into the popular products or categories that are frequently purchased by customers.
- **Stock Code Anomalies:** We observe that while most stock codes are composed of 5 or 6 characters, there are some anomalies like the code 'POST'. These anomalies might represent services or non-product transactions (perhaps postage fees) rather than actual products. To maintain the focus of the project, which is clustering based on product purchases and creating

a recommendation system, these anomalies should be further investigated and possibly treated appropriately to ensure data integrity.

To delve 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:

[20]:
```python
# 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
Name: count, dtype: int64
```

Inference: The output indicates the following:

- A majority of the unique stock codes (3676 out of 3684) contain exactly 5 numeric characters, which seems to be the standard format for representing product codes in this dataset.
- There are a few anomalies: 7 stock codes contain no numeric characters and 1 stock code contains only 1 numeric character. These are clearly deviating from the standard format and need further investigation to understand their nature and whether they represent valid product transactions. Now, let's identify the stock codes that contain 0 or 1 numeric characters to further understand these anomalies:

[21]:
```python
# 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("Anomalous stock codes:")
print("-"*22)
for code in anomalous_stock_codes:
    print(code)
print("-"*22)
```

```
Anomalous stock codes:
----------------------
POST
D
```

```
C2
M
BANK CHARGES
PADS
DOT
CRUK
----------------------
```

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

[22]:
```
# 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 the analysis, we find that a very small proportion of the records, **0.48%**, have anomalous stock codes, which deviate from the typical format observed in the majority of the data. Also, these anomalous codes are just a fraction among all unique stock codes **(only 8 out of 3684)**.

These codes seem to represent non-product transactions like **"BANK CHARGES"**, **"POST"** (possibly postage fees), etc. Since they do not represent actual products and are a very small proportion of the dataset, including them in the analysis might introduce noise and distort the clustering and recommendation system.

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:

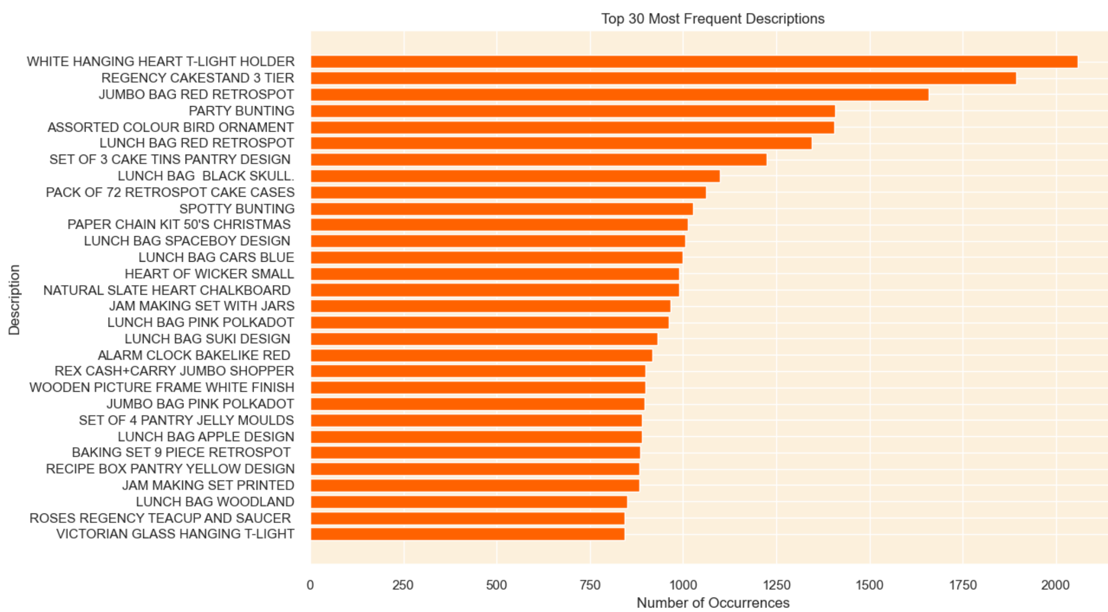
[23]:
```
# Removing rows with anomalous stock codes from the dataset
df = df[~df['StockCode'].isin(anomalous_stock_codes)]
```

[24]:
```
# Getting the number of rows in the dataframe
df.shape[0]
```

[24]:
```
399689
```

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

[25]:
```python
# Calculating the occurrence of each unique description and sort them
description_counts = df['Description'].value_counts()

# Geting the top 30 descriptions
top_30_descriptions = description_counts[:30]

# 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')

# Showing 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.
- 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.

[26]:
```python
# Finding unique descriptions containing lowercase characters
lowercase_descriptions = df['Description'].unique()
lowercase_descriptions = [desc for desc in lowercase_descriptions if any(char.
↪islower() for char in desc)]

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

```
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.

[27]:
```
service_related_descriptions = ["Next Day Carriage", "High Resolution Image"]

# Calculating the percentage of records with service-related descriptions
service_related_percentage = df[df['Description'].
 ↪isin(service_related_descriptions)].shape[0] / df.shape[0] * 100

# Printing 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}%")

# Removing rows with service-related information in the description
df = df[~df['Description'].isin(service_related_descriptions)]

# Standardizing the text to uppercase to maintain uniformity across the dataset
df['Description'] = df['Description'].str.upper()
```

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

[28]:
```
df.shape[0]
```

[28]:
```
399606
```

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

[29]:
```
df['UnitPrice'].describe()
```

[29]:
```
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 suggests that there are some transactions where the unit price is zero, potentially indicating a free item or a data entry error. To understand their nature, it is essential to investigate these zero unit price transactions further. A detailed analysis of the product descriptions associated with zero unit prices will be conducted to determine if they adhere to a specific pattern:

[30]:
```
df[df["UnitPrice"]==0].describe()[["Quantity"]]
```

[30]:
```
        Quantity
count      33.000000
mean      420.515152
std      2176.713608
min         1.000000
25%         2.000000
50%        11.000000
75%        36.000000
max     12540.000000
```

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.

[31]:
```python
df = df[df['UnitPrice'] > 0]
```

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.

[32]:
```python
# Resetting the index of the cleaned dataset
df.reset_index(drop=True, inplace=True)
```

[33]:
```python
df.shape[0]
```

[33]: 399573

1.0.4 Feature Engineering

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

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.

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 Purchas:** 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.

[34]:
```python
# Converting InvoiceDate to datetime type
df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'])

# Converting InvoiceDate to datetime and extracting only the date
df['InvoiceDay'] = df['InvoiceDate'].dt.date

# Finding the most recent purchase date for each customer
customer_data = df.groupby('CustomerID')['InvoiceDay'].max().reset_index()

# Finding the most recent date in the entire dataset
most_recent_date = df['InvoiceDay'].max()

# Converting InvoiceDay to datetime type before subtraction
customer_data['InvoiceDay'] = pd.to_datetime(customer_data['InvoiceDay'])
most_recent_date = pd.to_datetime(most_recent_date)

# Calculating 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

# Removing the InvoiceDay column
customer_data.drop(columns=['InvoiceDay'], inplace=True)
```

Now, customer_data dataframe contains the Days_Since_Last_Purchase feature:

[35]:
```
customer_data.head()
```

[35]:

	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

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

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.

[36]:
```
# Calculating 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)

# Calculating 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)

# Merging 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')

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

[36]:

	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
0                         0
1                      2458
2                      2332
3                       630
4                       196
```

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.

[37]:
```python
# Calculating the total spend by each customer
df['Total_Spend'] = df['UnitPrice'] * df['Quantity']
total_spend = df.groupby('CustomerID')['Total_Spend'].sum().reset_index()

# Calculating 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']

# Merging 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')

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

[37]:
```
   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
```

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.

[38]:
```python
# Calculating 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)

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

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

[38]:
```
   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
```

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.

[39]:
```python
# Extracting day of week and hour from InvoiceDate
df['Day_Of_Week'] = df['InvoiceDate'].dt.dayofweek
df['Hour'] = df['InvoiceDate'].dt.hour

# Calculating 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)

# Finding 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']]
```

```python
# Finding 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']]

# Merging 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')

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

[39]:

```
   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  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

   Hour
0    10
1    14
2    19
3     9
4    16
```

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.

[40]:
```
df['Country'].value_counts(normalize=True).head()
```

[40]:
```
Country
United Kingdom    0.890971
Germany           0.022722
France            0.020402
EIRE              0.018440
Spain             0.006162
Name: proportion, dtype: float64
```

Inference: Given that a substantial portion **(89%)** of transactions are originating from the **United Kingdom**, we might consider creating a binary feature indicating whether the transaction is from the UK or not. This approach can potentially streamline the clustering process without losing critical geographical information, especially when considering the application of algorithms like K-means which are sensitive to the dimensionality of the feature space.

Methodology:

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

[41]:
```
# Grouping 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')

# Getting 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')

# Creating 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)

# Merging this data with our customer_data dataframe
```

```python
customer_data = pd.merge(customer_data, customer_main_country[['CustomerID',␣
↪'Is_UK']], on='CustomerID', how='left')

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

[41]:

```
   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  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

   Hour  Is_UK
0    10      1
1    14      0
2    19      0
3     9      0
4    16      0
```

[42]:

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

[42]:

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

Cancellation Insights: In this step, I am 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.

[43]:
```python
# Calculating the total number of transactions made by each customer
total_transactions = df.groupby('CustomerID')['InvoiceNo'].nunique().
 ↪reset_index()

# Calculating 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)

# Merging the Cancellation Frequency data into the customer_data dataframe
customer_data = pd.merge(customer_data, cancellation_frequency,␣
 ↪on='CustomerID', how='left')

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

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

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

[43]:
```
   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  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

   Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate
0    10      1                     1.0                0.5
1    14      0                     0.0                0.0
2    19      0                     0.0                0.0
3     9      0                     0.0                0.0
4    16      0                     0.0                0.0
```

Seasonality & Trends: In this step, I 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.

[44]:
```python
# Extracting month and year from InvoiceDate
df['Year'] = df['InvoiceDate'].dt.year
df['Month'] = df['InvoiceDate'].dt.month
```

```
# Calculating monthly spending for each customer
monthly_spending = df.groupby(['CustomerID', 'Year', 'Month'])['Total_Spend'].
    ↪sum().reset_index()

# Calculating 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)

# Replacing 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)

# Calculating 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

# Applying 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)

# Merging 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')

# Displaying 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  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

   Hour  Is_UK  Cancellation_Frequency  Cancellation_Rate  \
0    10      1                     1.0                0.5
1    14      0                     0.0                0.0
2    19      0                     0.0                0.0
3     9      0                     0.0                0.0
4    16      0                     0.0                0.0

   Monthly_Spending_Mean  Monthly_Spending_Std  Spending_Trend
0               0.000000              0.000000        0.000000
1             615.714286            341.070789        4.486071
2             359.310000            203.875689     -100.884000
3            1457.550000              0.000000        0.000000
4             294.400000              0.000000        0.000000
```

[45]:

```
# 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)

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

[46]:

```
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          0.0                        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
```

[47]:

```
customer_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 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   Int32
 9   Hour                           4282 non-null   Int32
 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), Int32(2), Int64(6), string(1)
memory usage: 564.7 KB
```

Customer Dataset Description:

- **CustomerID:** Identifier uniquely assigned to each customer, used to distinguish individual customers.
- **Days_Since_Last_Purchase:** The number of days that have passed since the customer's last purchase.
- **Total_Transactions:** The total number of transactions made by the customer.
- **Total_Products_Purchased:** The total quantity of products purchased by the customer across all transactions.
- **Total_Spend:** The total amount of money the customer has spent across all transactions.
- **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 variable indicating whether the customer is based in the UK (1) or not (0).
- **Cancellation_Frequency:** The total number of transactions that the customer has cancelled.
- **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:** The standard deviation of the customer's monthly spending, indicating the variability in 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.

1.0.5 Outlier Detection and Treatment

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.

[48]:
```
# 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']]

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

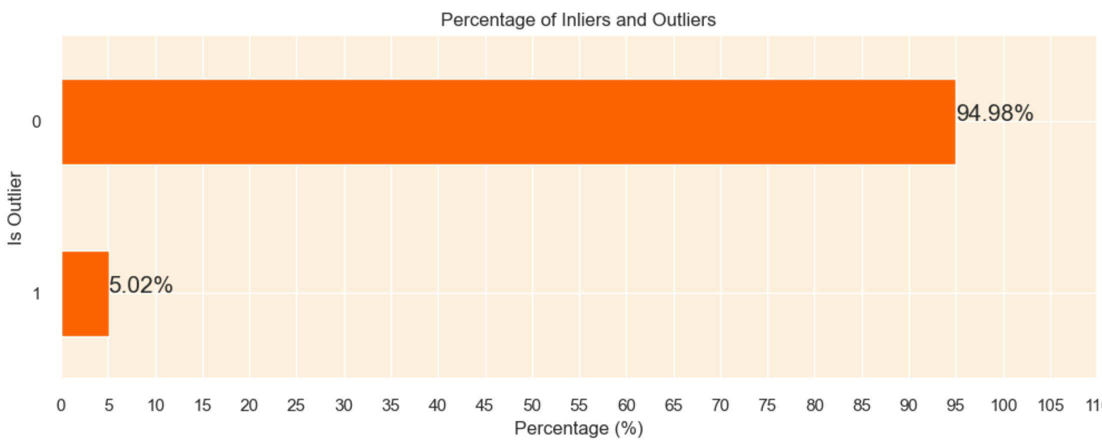
[48]:
```
   CustomerID  Days_Since_Last_Purchase  Total_Transactions  \
0     12346.0                       325                   2
1     12347.0                         2                   7
2     12348.0                        75                   4
3     12349.0                        18                   1
4     12350.0                       310                   1

   Total_Products_Purchased  Total_Spend  Average_Transaction_Value  \
0                         0          0.0                        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

   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

   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

   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

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

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:

[49]:

```python
# Calculating 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=(12, 4))
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: From the above plot, we can observe that about **5%** of the customers have been identified as outliers in our dataset. This percentage seems to be a reasonable proportion, not too high to lose a significant amount of data, and not too low to retain potentially noisy data points. It suggests that our isolation forest algorithm has worked well in identifying a moderate percentage of outliers, which will be critical in refining our customer segmentation.

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.

[50]:
```
# Separating the outliers for analysis
outliers_data = customer_data[customer_data['Is_Outlier'] == 1]

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

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

# Resetting the index of the cleaned data
customer_data_cleaned.reset_index(drop=True, inplace=True)
```

[51]:
```
customer_data_cleaned.shape[0]
```

[51]: 4067

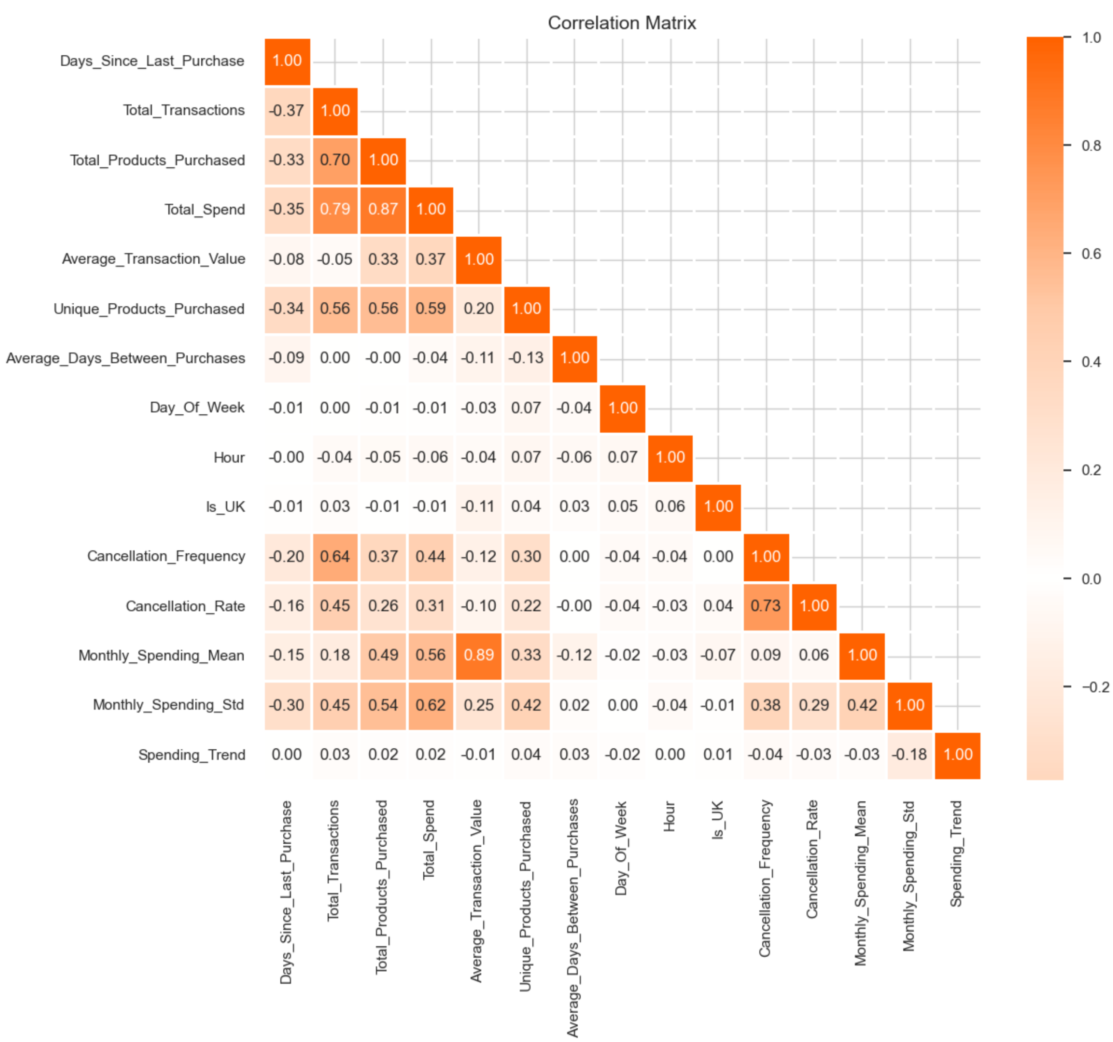
1.0.6 Correlation Analysis

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.

[52]:
```
# Resetting background style
sns.set_style('whitegrid')

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

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

```python
# Creating 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

# Ploting 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.

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.

[53]:
```
# Initializing the StandardScaler
scaler = StandardScaler()

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

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

```python
# Copying 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])

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

[53]:

```
   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
```

1.0.7 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.

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.

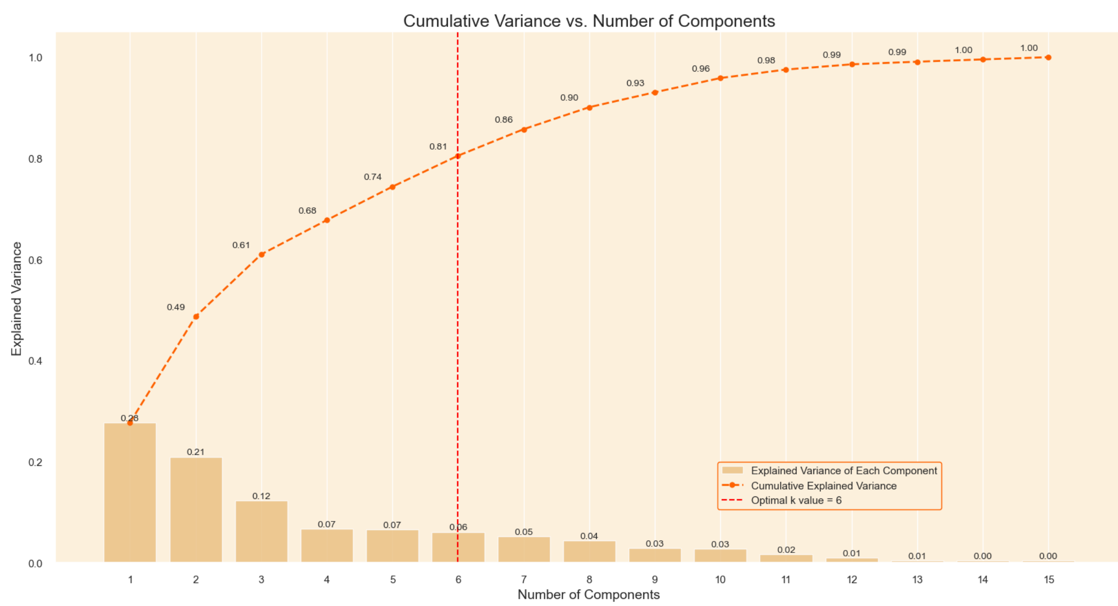
Methodology

I 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:

[54]:
```
# Setting CustomerID as the index column
customer_data_scaled.set_index('CustomerID', inplace=True)

# Applying PCA
pca = PCA().fit(customer_data_scaled)

# Calculating the Cumulative Sum of the Explained Variance
explained_variance_ratio = pca.explained_variance_ratio_
```

```python
cumulative_explained_variance = np.cumsum(explained_variance_ratio)

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

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

# Plotting 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)

# Ploting 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)

# Customizing 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')

# Displaying 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.

[55]:
```
# 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
```

[56]:
```
customer_data_pca.head()
```

[56]:

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

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

[57]:
```
# Defining 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]

# Creating 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)
```

[57]:
```
<pandas.io.formats.style.Styler at 0x23bbdc0bda0>
```

1.0.8 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:

- **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.
- **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.
- **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.
- **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.
- **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.

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.

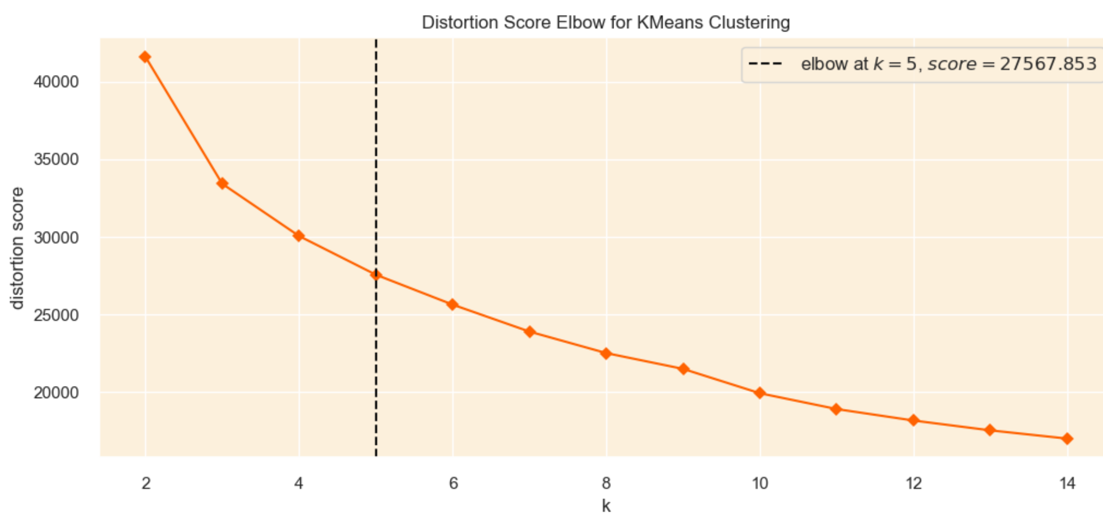
Elbow Method

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.

[58]:
```
# Setting plot style, and background color
sns.set(style='darkgrid', rc={'axes.facecolor': '#fcf0dc'})

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

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

# Creating a figure and axis with the desired size
fig, ax = plt.subplots(figsize=(12, 5))

# Instantiating 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)

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

# Finalizing and rendering 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.

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.

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:

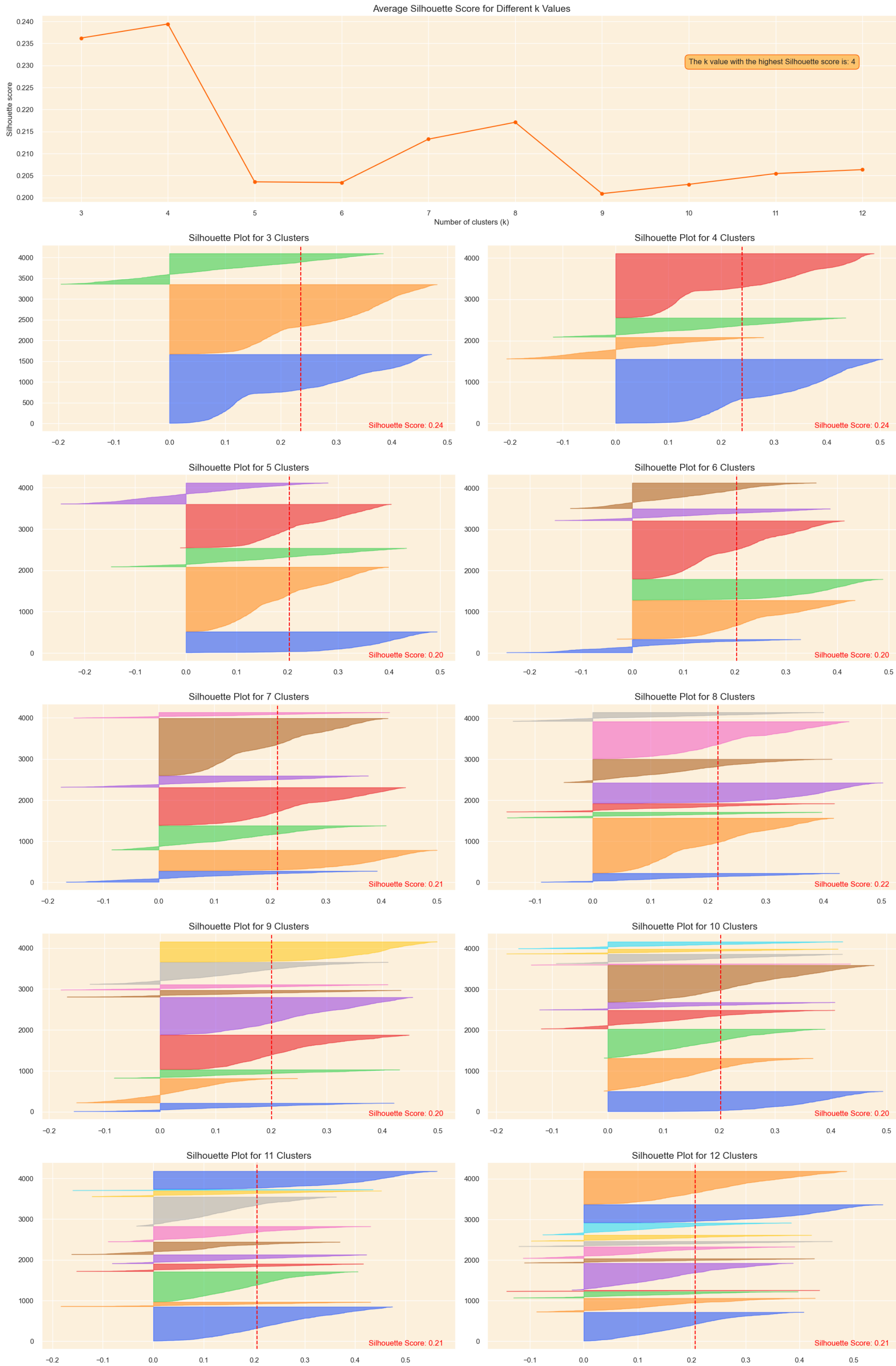
- I will initially choose a range of 2-6 for the number of clusters (k) based on the Elbow method from the previous section. Next, I will plot Silhouette scores for each k value to determine the one with the highest score.
- Subsequently, to fine-tune the selection of the most appropriate k, I will generate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

[59]:
```python
def silhouette_analysis(df, start_k, stop_k, figsize=(15, 16)):
    """
    Perform Silhouette analysis for a range of k values and visualize the␣
    ↪results.
    """

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

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

    # Assigning 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 = []

    # Iterating 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)

    # Addng 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)

    # Assigning 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)

    # Adding 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()
```

[60]:

```
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: 0.22
Silhouette Plot for 9 Clusters
Silhouette Score: 0.20
Silhouette Plot for 10 Clusters
Silhouette Score: 0.20
Silhouette Plot for 11 Clusters
Silhouette Score: 0.21
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:

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.

1.0.9 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.

[61]:
```
# Applying KMeans clustering using the optimal k
kmeans = KMeans(n_clusters=3, init='k-means++', n_init=10, max_iter=100,␣
↪random_state=0)
kmeans.fit(customer_data_pca)

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

# Creating 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())}

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

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

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

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

[62]:
```
customer_data_cleaned.head(10)
```

[62]:
```
  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        1
3                1457.55                   0.0             0.0        2
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	1
8	459.4	0.0	0.0	2
9	464.03	83.679016	118.34	2

1.0.10 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:

- 3D Visualization of Top PCs
- Cluster Distribution Visualization
- 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.

3D Visualization of Top Principal Components In this part, I am going to choose the top 3 PCs (which capture the most variance in the data) and use them to create a 3D visualization. This will allow us to visually inspect the quality of separation and cohesion of clusters to some extent:

[63]:
```
# Setting up the color scheme for the clusters (RGB order)
colors = ['#e8000b', '#1ac938', '#023eff']
```

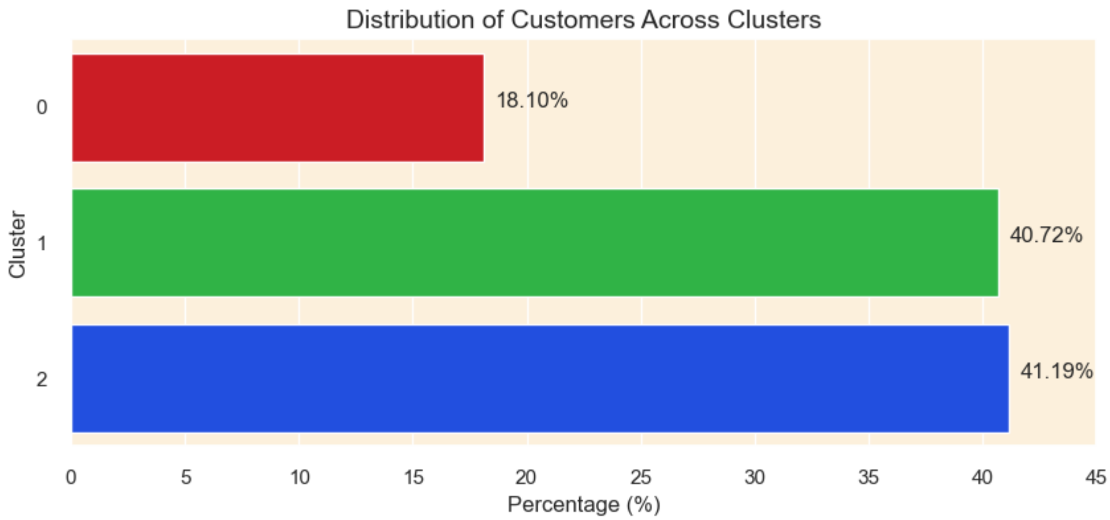
[64]:
```
# Creating separate data frames for each cluster
cluster_0 = customer_data_pca[customer_data_pca['cluster'] == 0]
cluster_1 = customer_data_pca[customer_data_pca['cluster'] == 1]
cluster_2 = customer_data_pca[customer_data_pca['cluster'] == 2]

# Creating a 3D scatter plot
fig = go.Figure()

# Adding data points for each cluster separately and specify the color
fig.add_trace(go.Scatter3d(x=cluster_0['PC1'], y=cluster_0['PC2'],␣
 ↪z=cluster_0['PC3'],
                           mode='markers', marker=dict(color=colors[0], size=5,␣
 ↪opacity=0.4), name='Cluster 0'))
fig.add_trace(go.Scatter3d(x=cluster_1['PC1'], y=cluster_1['PC2'],␣
 ↪z=cluster_1['PC3'],
```

```
                           mode='markers', marker=dict(color=colors[1], size=5,␣
↪opacity=0.4), name='Cluster 1'))
fig.add_trace(go.Scatter3d(x=cluster_2['PC1'], y=cluster_2['PC2'],␣
↪z=cluster_2['PC3'],
                           mode='markers', marker=dict(color=colors[2], size=5,␣
↪opacity=0.4), name='Cluster 2'))

# Setting the title and layout details
fig.update_layout(
    title=dict(text='3D Visualization of Customer Clusters in PCA Space', x=0.
↪5),
    scene=dict(
        xaxis=dict(backgroundcolor="#fcf0dc", gridcolor='white', title='PC1'),
        yaxis=dict(backgroundcolor="#fcf0dc", gridcolor='white', title='PC2'),
        zaxis=dict(backgroundcolor="#fcf0dc", gridcolor='white', title='PC3'),
    ),
    width=900,
    height=800
)

# Showing the plot
fig.show()
```

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:

[65]:
```
# Calculating 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)

# Creating a horizontal bar plot
plt.figure(figsize=(10, 4))
sns.barplot(x='Percentage', y='Cluster', data=cluster_percentage, orient='h',␣
↪palette=colors)

# 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 (%)')
```

```python
# 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.

[66]:
```python
# Computing number of customers
num_observations = len(customer_data_pca)
```

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

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

# Creating 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]
]

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

```
+-------------------------+--------------------+
|         Metric          |       Value        |
+-------------------------+--------------------+
| Number of Observations  |        4067        |
|    Silhouette Score     | 0.2362284801709886 |
| Calinski Harabasz Score | 1257.1747766540623 |
|  Davies Bouldin Score   | 1.368269537607467  |
+-------------------------+--------------------+
```

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.

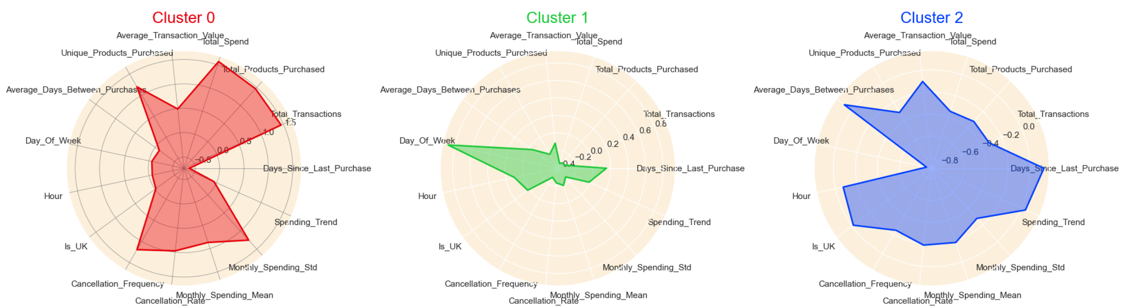
1.0.11 Cluster Analysis and Profiling

In this section, I am 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.

Radar Chart Approach First of all, I am going to create radar charts to visualize the centroid values of each cluster across different features. This can give a quick visual comparison of the profiles of different clusters.To construct the radar charts, it's essential to first compute the centroid for each cluster. This centroid represents the mean value for all features within a specific cluster. Subsequently, I will display these centroids on the radar charts, facilitating a clear visualization of the central tendencies of each feature across the various clusters:

[67]:
```python
# Setting 'CustomerID' column as index and assigning it to a new dataframe
df_customer = customer_data_cleaned.set_index('CustomerID')

# Standardizing the data (excluding the cluster column)
scaler = StandardScaler()
df_customer_standardized = scaler.fit_transform(df_customer.
 ↪drop(columns=['cluster'], axis=1))

# Creating a new dataframe with standardized values and add the cluster column␣
 ↪back
df_customer_standardized = pd.DataFrame(df_customer_standardized,␣
 ↪columns=df_customer.columns[:-1], index=df_customer.index)
df_customer_standardized['cluster'] = df_customer['cluster']

# Calculating the centroids of each cluster
cluster_centroids = df_customer_standardized.groupby('cluster').mean()

# Function to create a radar chart
def create_radar_chart(ax, angles, data, color, cluster):
    # Plot the data and fill the area
    ax.fill(angles, data, color=color, alpha=0.4)
    ax.plot(angles, data, color=color, linewidth=2, linestyle='solid')

    # Add a title
    ax.set_title(f'Cluster {cluster}', size=20, color=color, y=1.1)

# Set data
labels=np.array(cluster_centroids.columns)
num_vars = len(labels)

# Computing angle of each axis
angles = np.linspace(0, 2 * np.pi, num_vars, endpoint=False).tolist()

# The plot is circular, so we need to "complete the loop" and append the start␣
 ↪to the end
```

```
labels = np.concatenate((labels, [labels[0]]))
angles += angles[:1]

# Initializing the figure
fig, ax = plt.subplots(figsize=(20, 10), subplot_kw=dict(polar=True), nrows=1, ncols=3)

# Creating radar chart for each cluster
for i, color in enumerate(colors):
    data = cluster_centroids.loc[i].tolist()
    data += data[:1]  # Complete the loop
    create_radar_chart(ax[i], angles, data, color, i)

# Adding input data
ax[0].set_xticks(angles[:-1])
ax[0].set_xticklabels(labels[:-1])

ax[1].set_xticks(angles[:-1])
ax[1].set_xticklabels(labels[:-1])

ax[2].set_xticks(angles[:-1])
ax[2].set_xticklabels(labels[:-1])

# Adding a grid
ax[0].grid(color='grey', linewidth=0.5)

# Displaying the plot
plt.tight_layout()
plt.show()
```



Customer Profiles Derived from Radar Chart Analysis

Cluster 0 (Red Chart): Profile: Sporadic Shoppers with a Preference for Weekend Shopping

- Customers in this cluster tend to spend less, with a lower number of transactions and products purchased.
- They have a slight tendency to shop during the weekends, as indicated by the very high Day_of_Week value.
- Their spending trend is relatively stable but on the lower side, and they have a low monthly spending variation (low Monthly_Spending_Std).
- These customers have not engaged in many cancellations, showing a low cancellation frequency and rate.
- The average transaction value is on the lower side, indicating that when they do shop, they tend to spend less per transaction.

Cluster 1 (Green Chart): Profile: Infrequent Big Spenders with a High Spending Trend

- Customers in this cluster show a moderate level of spending, but their transactions are not very frequent, as indicated by the high Days_Since_Last_Purchase and Average_Days_Between_Purchases.
- They have a very high spending trend, indicating that their spending has been increasing over time.
- These customers prefer shopping late in the day, as indicated by the high Hour value, and they mainly reside in the UK.
- They have a tendency to cancel a moderate number of transactions, with a medium cancellation frequency and rate.
- Their average transaction value is relatively high, meaning that when they shop, they tend to make substantial purchases.

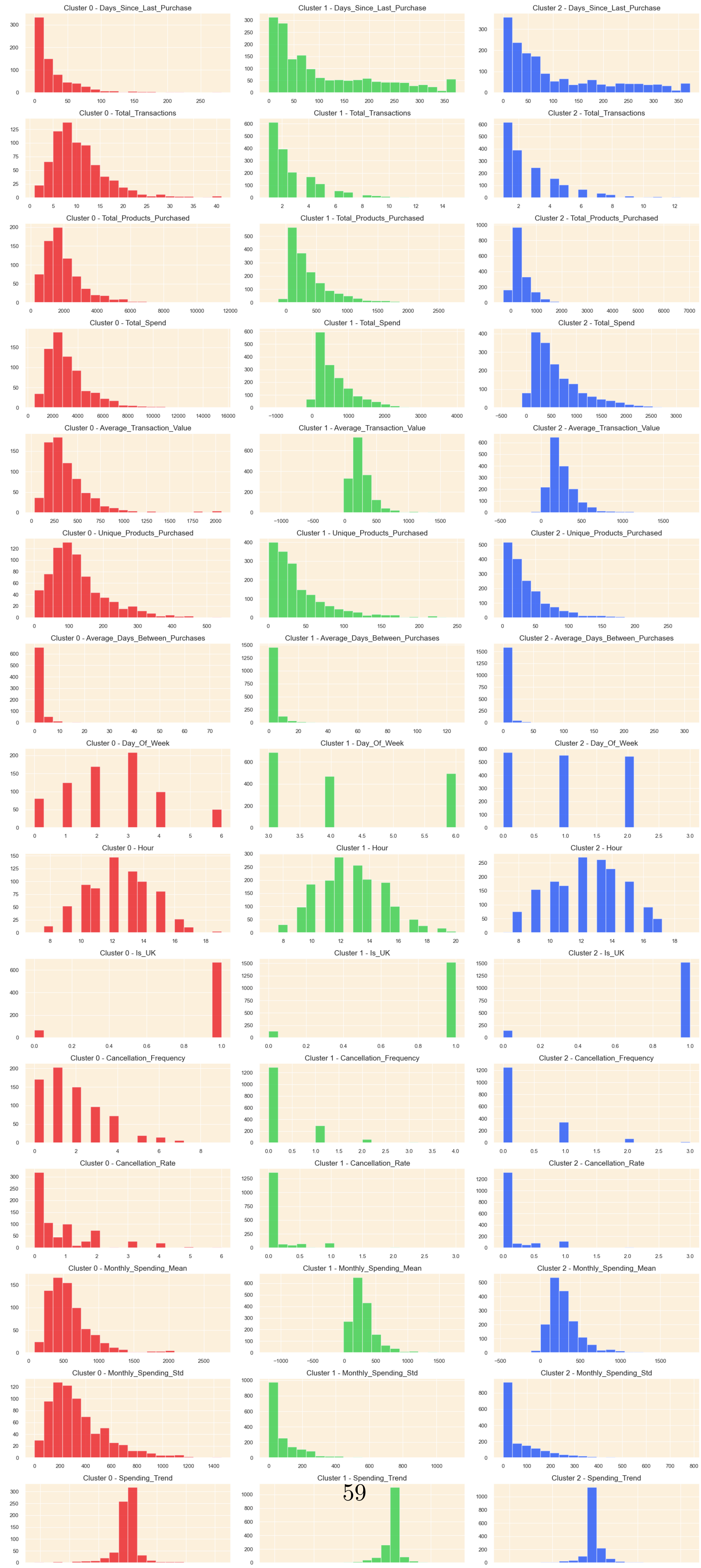
Cluster 2 (Blue Chart): Profile: Frequent High-Spenders with a High Rate of Cancellations

- Customers in this cluster are high spenders with a very high total spend, and they purchase a wide variety of unique products.
- They engage in frequent transactions, but also have a high cancellation frequency and rate.
- These customers have a very low average time between purchases, and they tend to shop early in the day (low Hour value).
- Their monthly spending shows high variability, indicating that their spending patterns might be less predictable compared to other clusters.
- Despite their high spending, they show a low spending trend, suggesting that their high spending levels might be decreasing over time.

Histogram Chart Approach To validate the profiles identified from the radar charts, 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, thereby confirming or refining the profiles we have created based on the radar charts.

[68]:
```python
# Plotting histograms for each feature segmented by the clusters
features = customer_data_cleaned.columns[1:-1]
clusters = customer_data_cleaned['cluster'].unique()
clusters.sort()
```

```python
# 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='w', alpha=0.7)
        axes[i, j].set_title(f'Cluster {cluster} - {feature}', fontsize=15)
        axes[i, j].set_xlabel('')
        axes[i, j].set_ylabel('')

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

The detailed insights from the histograms provide a more nuanced understanding of each cluster, helping in refining the profiles to represent the customer behaviors more accurately. Based on the detailed analysis from both the radar charts and the histograms, here are the refined profiles and titles for each cluster:

Cluster 0 - Casual Weekend Shoppers:

- Customers in this cluster usually shop less frequently and spend less money compared to the other clusters.
- They generally have a smaller number of transactions and buy fewer products.
- These customers have a preference for shopping during the weekends, possibly engaging in casual or window shopping.
- Their spending habits are quite stable over time, showing little fluctuation in their monthly spending.
- They rarely cancel their transactions, indicating a more decisive shopping behavior.
- When they do shop, their spending per transaction tends to be lower compared to other clusters.

Cluster 1 - Occasional Big Spenders:

- Customers in this cluster don't shop frequently but tend to spend a considerable amount when they do, buying a variety of products.
- Their spending has been on the rise, indicating a growing interest or investment in their purchases.
- They prefer to shop later in the day, possibly after work hours, and are mainly based in the UK.
- They have a moderate tendency to cancel transactions, which might be due to their higher spending; they perhaps reconsider their purchases more often.
- Their purchases are generally substantial, indicating a preference for quality or premium products.

Cluster 2 - Eager Early-Bird Shoppers:

- Customers in this cluster are characterized by their high spending habits. They tend to buy a wide array of unique products and engage in numerous transactions.
- Despite their high expenditure, they have a tendency to cancel a significant portion of their transactions, possibly indicating impulsive buying behaviors.
- They usually shop during the early hours of the day, perhaps finding time before their daily commitments or taking advantage of early bird deals.
- Their spending patterns are quite variable, with high fluctuations in their monthly spending, indicating a less predictable shopping pattern.
- Interestingly, their spending trend is showing a slight decrease, which might signal a future change in their shopping habits.

1.0.12 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.

[69]:
```
# Extracting 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)]

# Ensures consistent data type for CustomerID across both dataframes before
↪merging
customer_data_cleaned['CustomerID'] = customer_data_cleaned['CustomerID'].
↪astype('float')

# Merging 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')

# Identifying 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)

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

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

        # Finding 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)

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

# Step 7: Creating 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')
```

[70]:
```
customer_data_with_recommendations.set_index('CustomerID').iloc[:, -6:].
↪sample(5, random_state=0)
```

[70]:
```
           Rec1_StockCode                    Rec1_Description Rec2_StockCode  \
CustomerID
13243.0             84077    WORLD WAR 2 GLIDERS ASSTD DESIGNS          84879
13232.0             84077    WORLD WAR 2 GLIDERS ASSTD DESIGNS          84879
14997.0             18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          84879
14948.0             18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          84879
12596.0             18007  ESSENTIAL BALM 3.5G TIN IN ENVELOPE          84879

                        Rec2_Description Rec3_StockCode  \
CustomerID
13243.0      ASSORTED COLOUR BIRD ORNAMENT          15036
13232.0      ASSORTED COLOUR BIRD ORNAMENT          15036
```

```
14997.0     ASSORTED COLOUR BIRD ORNAMENT          17003
14948.0     ASSORTED COLOUR BIRD ORNAMENT          17003
12596.0     ASSORTED COLOUR BIRD ORNAMENT          17003

                     Rec3_Description
CustomerID
13243.0     ASSORTED COLOURS SILK FAN
13232.0     ASSORTED COLOURS SILK FAN
14997.0           BROCADE RING PURSE
14948.0           BROCADE RING PURSE
12596.0           BROCADE RING PURSE
```

[ ]: