📦 Online Retail Dataset Overview

🔍 About the Dataset

Context

In the world of e-commerce, datasets are often kept confidential, making it difficult to find high-quality, real-world data for analysis. However, the UCI Machine Learning Repository has provided a valuable exception. This dataset contains actual transaction records from a UK-based online retail company, spanning from December 1, 2010, to December 9, 2011. The dataset is publicly available under the title **"Online Retail."**

Dataset Content

This dataset captures every transaction made within the specified period by a non-store online retailer in the UK. The company specializes in selling unique, all-occasion gifts, and many of its customers are wholesalers. The dataset includes:

- **InvoiceNo:** A unique identifier for each transaction.
- **StockCode:** A unique identifier for each product.
- **Description:** Product description.
- **Quantity:** The number of products purchased per transaction.
- **InvoiceDate:** The date of the transaction.
- **UnitPrice:** Price per product.
- **CustomerID:** A unique identifier for each customer.
- **Country:** The country where the customer resides.

Acknowledgements

This dataset was generously provided by Dr. Daqing Chen, Director of the Public Analytics Group at the School of Engineering, London South Bank University. Dr. Chen's contributions have made it possible for researchers and data enthusiasts to explore real-world e-commerce data. For further inquiries, Dr. Chen can be contacted at chend '@' lsbu.ac.uk.

Inspiration for Analysis

The "Online Retail" dataset offers a rich playground for a variety of data analysis techniques, including but not limited to:

- **Time Series Analysis:** Examine trends over time, such as peak shopping periods, seasonal effects, and sales cycles.
- **Clustering:** Identify distinct customer segments, such as frequent buyers, high spenders, or seasonal shoppers.
- **Classification:** Predict customer behavior, such as likelihood of repeat purchases or propensity to buy certain products.
- **Market Basket Analysis:** Discover relationships between products often purchased together.

This dataset provides an excellent opportunity to explore the dynamics of online retail and to gain insights into consumer behavior in the e-commerce space.

1. Import libraries

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
```

In [ ]:
```
plt.style.use('ggplot')
```

2. Loading data

In [ ]:
```
df = pd.read_csv('data.csv', encoding = 'latin-1')
df.head()
```

Out[ ]:

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

3. Initial Analysis

3.1. Initial Clean-up

In [ ]:
```
df.head()
```

Out[ ]:

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

In [ ]:
```
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
```

In [ ]:
```
df['InvoiceDate'] = df['InvoiceDate'].astype('datetime64[ns]')
```

In [ ]:
```
df.isna().sum()
```

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

In [ ]:
```
df.dropna(subset=['CustomerID'], inplace=True)
```

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

Out[ ]:
```
0
```

In [ ]:
```
df['CustomerID'] = df['CustomerID'].astype(int).astype('object')
df.dtypes
```

Out[ ]:
```
InvoiceNo              object
StockCode              object
Description            object
Quantity                int64
InvoiceDate    datetime64[ns]
UnitPrice             float64
CustomerID             object
Country                object
dtype: object
```

In [ ]:
```
df.duplicated().sum()
```

Out[ ]:
```
5225
```

In [ ]:
```
df.drop_duplicates(inplace=True)
```

3.2. Summary Stats

In [ ]:
```
# df.describe(percentiles= np.arange(0, 1, 0.15)).T
df.describe().T
```

Out[ ]:

	count	mean	min	25%	50%	75%	max	std
Quantity	401604.0	12.183273	-80995.0	2.0	5.0	12.0	80995.0	250.283037
InvoiceDate	401604	2011-07-10 12:08:23.848567552	2010-12-01 08:26:00	2011-04-06 15:02:00	2011-07-29 15:40:00	2011-10-20 11:58:30	2011-12-09 12:50:00	NaN
UnitPrice	401604.0	3.474064	0.0	1.25	1.95	3.75	38970.0	69.764035

In [ ]:
```
df.describe(include='object').T
```

Out[ ]:

	count	unique	top	freq
InvoiceNo	401604	22190	576339	542
StockCode	401604	3684	85123A	2065
Description	401604	3896	WHITE HANGING HEART T-LIGH...	2058
CustomerID	401604	4372	17841	7812
Country	401604	37	United Kingdom	356728

`Quantity:`

- std of Quantity is so high, then it should contain outliers
- contains negative values, it might be a order cancelation or return

3.3. Some columns' problems

3.3.1. Product Variety

In [ ]:
```
df.groupby(by = 'StockCode', as_index=False)['Quantity'].sum()
```

Out[ ]:

	StockCode	Quantity
0	10002	823
1	10080	291
2	10120	192
3	10123C	5
4	10124A	16
...	...	...
3679	D	-1194
3680	DOT	16
3681	M	2944
3682	PADS	4
3683	POST	3002

3684 rows × 2 columns

It seems like `StockCode` contains 5-6 characters, but there are some anamolies.

In [ ]:
```
df['StockCodeLen'] = df['StockCode'].apply(lambda x: len(x))
df['StockCodeDigitLen'] = df['StockCode'].apply(lambda x: sum(c.isdigit() for c in str(x)))
df.head()
```

Out[ ]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	StockCodeLen	StockCodeDigitLen
0	536365	85123A	WHITE HANGING HEART T-LIGH...	6	2010-12-01 08:26:00	2.55	17850	United Kingdom	6	5
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	5	5
2	536365	84406B	CREAM CUPID HEARTS COAT HA...	8	2010-12-01 08:26:00	2.75	17850	United Kingdom	6	5
3	536365	84029G	KNITTED UNION FLAG HOT WAT...	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	6	5
4	536365	84029E	RED WOOLLY HOTTIE WHITE HE...	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	6	5

In [ ]:
```
print("Value counts of numeric character frequencies in unique stock codes:")
print("-"*70)
print(df['StockCodeDigitLen'].value_counts())
```

```
Value counts of numeric character frequencies in unique stock codes:
----------------------------------------------------------------------
StockCodeDigitLen
5    399689
0      1781
1       134
Name: count, dtype: int64
```

In [ ]:
```
df[df['StockCodeDigitLen'] != 5].groupby(by = 'StockCode', as_index=False)['Quantity'].sum().sort_values(by= 'Quantity', ascen
```

Out[ ]:

	StockCode	Quantity
7	POST	3002
5	M	2944
1	C2	133
4	DOT	16
0	BANK CHARGES	12
6	PADS	4
2	CRUK	-16
3	D	-1194

In [ ]:
```
df[df['StockCodeDigitLen'] != 5][['StockCode','Description']].drop_duplicates()
```

Out[ ]:

	StockCode	Description
45	POST	POSTAGE
141	D	Discount
1423	C2	CARRIAGE
2239	M	Manual
4406	BANK CHARGES	Bank Charges
157195	PADS	PADS TO MATCH ALL CUSHIONS
317507	DOT	DOTCOM POSTAGE
317508	CRUK	CRUK Commission

`StockCode` contains != 5 numerical characters are service-related products & add-on purchases and might be removed from the dataframe.

3.3.2. Unit Price

In [ ]:
```
df[df['UnitPrice'] == 0].head()
```

Out[ ]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	StockCodeLen	StockCodeDigitLen
9302	537197	22841	ROUND CAKE TIN VINTAGE GREEN	1	2010-12-05 14:02:00	0.0	12647	Germany	5	5
33576	539263	22580	ADVENT CALENDAR GINGHAM SACK	4	2010-12-16 14:36:00	0.0	16560	United Kingdom	5	5
40089	539722	22423	REGENCY CAKESTAND 3 TIER	10	2010-12-21 13:45:00	0.0	14911	EIRE	5	5
47068	540372	22090	PAPER BUNTING RETROSPOT	24	2011-01-06 16:41:00	0.0	13081	United Kingdom	5	5
47070	540372	22553	PLASTERS IN TIN SKULLS	24	2011-01-06 16:41:00	0.0	13081	United Kingdom	5	5

`UnitPrice` contains 0, it should be removed

3.3.3. InvoiceNo

In [ ]:
```
df[df['Quantity'] < 0 ]['InvoiceNo'].unique()
```

Out[ ]:
```
array(['C536379', 'C536383', 'C536391', ..., 'C581499', 'C581568',
       'C581569'], dtype=object)
```

`InvoiceNo` starts with "C" is the cancelled invoice.

In [ ]:
```
df['Invoice_Status'] = 'normal'
df.loc[df['InvoiceNo'].str.startswith('C'), 'Invoice_Status'] = 'cancel'
```

3.3.3. Resolve the problems

In [ ]:
```
df = df[df['StockCodeDigitLen'] == 5]
# df = df[df['UnitPrice'] > 0]
```

3.4. Feature Engineering

3.4.1. RFM features

Recency (R)

In [ ]:
```
df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'])
df['InvoiceDay'] = df['InvoiceDate'].dt.date
customer_data = df.groupby('CustomerID')['InvoiceDay'].max().reset_index()
most_recent_date = df['InvoiceDay'].max()
customer_data['InvoiceDay'] = pd.to_datetime(customer_data['InvoiceDay'])
most_recent_date = pd.to_datetime(most_recent_date)
customer_data['Days_Since_Last_Purchase'] = (most_recent_date - customer_data['InvoiceDay']).dt.days
customer_data.drop(columns=['InvoiceDay'], inplace=True)

customer_data.head()
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase
0	12346	325
1	12347	2
2	12348	75
3	12349	18
4	12350	310

Frequency (F)

In [ ]:

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

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

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

customer_data.head()
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased
0	12346	325	2	0
1	12347	2	7	2458
2	12348	75	4	2332
3	12349	18	1	630
4	12350	310	1	196

Monetary (M)

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

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

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

customer_data.head()
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value
0	12346	325	2	0	0.00	0.000000
1	12347	2	7	2458	4310.00	615.714286
2	12348	75	4	2332	1437.24	359.310000
3	12349	18	1	630	1457.55	1457.550000
4	12350	310	1	196	294.40	294.400000

3.4.2. Behaviour features

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

In [ ]:
```
# Calculate the average number of days between consecutive purchases
days_between_purchases = df.groupby('CustomerID')['InvoiceDay'].apply(lambda x: (x.diff().dropna()).apply(lambda y: y.days))
average_days_between_purchases = days_between_purchases.groupby('CustomerID').mean().reset_index()
average_days_between_purchases.rename(columns={'InvoiceDay': 'Average_Days_Between_Purchases'}, inplace=True)
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\1043944771.py:2: FutureWarning:

The behavior of array concatenation with empty entries is deprecated. In a future version, this will no longer exclude empty it
ems when determining the result dtype. To retain the old behavior, exclude the empty entries before the concat operation.
```

In [ ]:
```
# Find the favorite shopping day of the week
favorite_shopping_day = df.groupby(['CustomerID', 'Day_Of_Week']).size().reset_index(name='Count')
favorite_shopping_day = favorite_shopping_day.loc[favorite_shopping_day.groupby('CustomerID')['Count'].idxmax()][['CustomerID'
```

In [ ]:
```
# Find the favorite shopping hour of the day
favorite_shopping_hour = df.groupby(['CustomerID', 'Hour']).size().reset_index(name='Count')
favorite_shopping_hour = favorite_shopping_hour.loc[favorite_shopping_hour.groupby('CustomerID')['Count'].idxmax()][['Customer
```

In [ ]:
```
# Merge the new features into the customer_data dataframe
customer_data = pd.merge(customer_data, average_days_between_purchases, on='CustomerID')
customer_data = pd.merge(customer_data, favorite_shopping_day, on='CustomerID')
customer_data = pd.merge(customer_data, favorite_shopping_hour, on='CustomerID')
```

In [ ]:
```
# Display the first few rows of the customer_data dataframe
customer_data.head()
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Average_D
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	

3.4.3. Product diversity

In [ ]:
```
# Calculate the number of unique products purchased by each customer
unique_products_purchased = df.groupby('CustomerID')['StockCode'].nunique().reset_index()
```

```
unique_products_purchased.rename(columns={'StockCode': 'Unique_Products_Purchased'}, inplace=True)
```

In [ ]:
```
customer_data = pd.merge(customer_data, unique_products_purchased, on='CustomerID')

customer_data.head()
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Average_D
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	

3.4.4. Geographical features

In [ ]:
```
df['Country'].value_counts()
```

Out[ ]:
```
Country
United Kingdom          356110
Germany                   9081
France                    8152
EIRE                      7370
Spain                     2463
Netherlands               2330
Belgium                   1971
Switzerland               1844
Portugal                  1427
Australia                 1256
Norway                    1060
Italy                      783
Channel Islands            752
Finland                    653
Cyprus                     608
Sweden                     436
Austria                    387
Denmark                    375
Japan                      355
Poland                     336
USA                        291
Israel                     247
Unspecified                241
Singapore                  215
Iceland                    182
Canada                     150
Greece                     142
Malta                      123
United Arab Emirates        67
European Community          58
RSA                         57
Lebanon                     45
Lithuania                   35
Brazil                      32
Czech Republic              28
Bahrain                     17
Saudi Arabia                10
Name: count, dtype: int64
```

Most of orders are from UK, so it would be better to separate UK customers from others.

In [ ]:
```
df.groupby(by = 'CustomerID', as_index = False)['Country'].nunique().sort_values(by = 'Country', ascending = False)
```

Out[ ]:

	CustomerID	Country
56	12417	2
20	12370	2
38	12394	2
90	12455	2
69	12431	2
...	...	...
1465	14332	1
1466	14333	1
1467	14334	1
1468	14335	1
4362	18287	1

4363 rows × 2 columns

So, it probably that some Customer purchased in more than 1 country.

In [ ]:
```
customer_main_country = df.groupby(by = ['CustomerID','Country'], as_index = False)\
    .agg(count_order = ('InvoiceNo', 'nunique'))\
        .sort_values(by = 'count_order', ascending = False)\
        .drop_duplicates(subset = 'CustomerID', keep = 'first')

customer_main_country['Is_UK'] = 0
customer_main_country.loc[customer_main_country['Country'] == 'United Kingdom', 'Is_UK'] = 1
```

```
customer_main_country.drop(columns = 'count_order', inplace = True)
customer_main_country
```

Out[ ]:

	CustomerID	Country	Is_UK
1900	14911	EIRE	0
336	12748	United Kingdom	1
4042	17841	United Kingdom	1
1679	14606	United Kingdom	1
574	13089	United Kingdom	1
...	...	...	...
1634	14542	United Kingdom	1
3232	16721	United Kingdom	1
3231	16720	United Kingdom	1
3229	16718	United Kingdom	1
2185	15292	United Kingdom	1

4363 rows × 3 columns

In [ ]:
```
customer_data = customer_data.merge(customer_main_country, on = 'CustomerID', how = 'inner')
customer_data
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Averag
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	
...	...	...	...	...	...	...	
4280	18280	277	1	45	180.60	180.600000	
4281	18281	180	1	54	80.82	80.820000	
4282	18282	7	3	98	176.60	58.866667	
4283	18283	3	16	1355	2039.58	127.473750	
4284	18287	42	3	1586	1837.28	612.426667	

4285 rows × 12 columns

3.4.5. Cancellation behaviour

In [ ]:
```
df['Invoice_Status'].unique()
```

Out[ ]:
```
array(['normal', 'cancel'], dtype=object)
```

In [ ]:
```
customer_total_order = df.groupby(by = 'CustomerID', as_index = False).agg(total_order = ('InvoiceNo', 'nunique'))
customer_cancel_order = df[df['Invoice_Status'] == 'cancel'].groupby(by = 'CustomerID', as_index = False).agg(Cancellation_Fre
```

In [ ]:
```
customer_cancel_rate = customer_total_order.merge(customer_cancel_order, on = 'CustomerID', how = 'left')
customer_cancel_rate['Cancellation_Frequency'].fillna(0, inplace = True)
customer_cancel_rate['Cancel_Rate'] = customer_cancel_rate['Cancellation_Frequency'] / customer_cancel_rate['total_order']
customer_cancel_rate.drop(columns=['total_order'], inplace = True)
customer_cancel_rate
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\3136470079.py:2: FutureWarning:

A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are set
ting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = d
f[col].method(value) instead, to perform the operation inplace on the original object.
```

Out[ ]:

	CustomerID	Cancellation_Frequency	Cancel_Rate
0	12346	1.0	0.500000
1	12347	0.0	0.000000
2	12348	0.0	0.000000
3	12349	0.0	0.000000
4	12350	0.0	0.000000
...	...	...	...
4358	18280	0.0	0.000000
4359	18281	0.0	0.000000
4360	18282	1.0	0.333333
4361	18283	0.0	0.000000
4362	18287	0.0	0.000000

4363 rows × 3 columns

In [ ]:
```
customer_data = customer_data.merge(customer_cancel_rate, on='CustomerID', how='left')
```

In [ ]:
```
customer_data.drop(columns=['Country'], inplace=True)
```

In [ ]:
```
customer_data
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Averag
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	
...	...	...	...	...	...	...	
4280	18280	277	1	45	180.60	180.600000	
4281	18281	180	1	54	80.82	80.820000	
4282	18282	7	3	98	176.60	58.866667	
4283	18283	3	16	1355	2039.58	127.473750	
4284	18287	42	3	1586	1837.28	612.426667	

4285 rows × 13 columns

3.4.6. Seasonality and Trends

Diving deeper into:

- `Monthly_Average_Spend` : the average spend per month. Customers who spend more averagely are more likely to purchase hi-end / pricey products.
- `Monthly_Std_Spend` : the standard deviation of spend per month. A higher value shows that a customer's spending fluctuates a lot. In contrary, a lower value shows that a customer's spending is more stable and consitent.
- `Trending_Slope` : The slope of a linear regression for each customers. A positive slope indicates that the customer's spending is increasing over time. A negative slope indicates that the customer's spending is decreasing over time.

In [ ]:
```
df['Year'] = df['InvoiceDate'].dt.year
df['Month'] = df['InvoiceDate'].dt.month
```

In [ ]:
```
df.head()
```

Out[ ]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	StockCodeLen	StockCodeDigitLen	Invoice
0	536365	85123A	WHITE HANGING HEART T-LIGH...	6	2010-12-01 08:26:00	2.55	17850	United Kingdom	6	5	
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	5	5	
2	536365	84406B	CREAM CUPID HEARTS COAT HA...	8	2010-12-01 08:26:00	2.75	17850	United Kingdom	6	5	
3	536365	84029G	KNITTED UNION FLAG HOT WAT...	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	6	5	
4	536365	84029E	RED WOOLLY HOTTIE WHITE HE...	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	6	5	

Monthly_Average_Spend & Monthly_Std_Spend

In [ ]:
```
df_customer_monthly_spending = df.groupby(by = ['CustomerID','Year','Month'], as_index=False).agg(Total_Spend = ('Total_Spend'

df_customer_monthly_spend_stat = df_customer_monthly_spending.groupby(by = ['CustomerID'], as_index=False).agg(Monthly_Average
                                                                                                               , Monthly_Std_
```

```
df_customer_monthly_spend_stat['Monthly_Std_Spend'].fillna(0, inplace=True)
df_customer_monthly_spend_stat
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\2525997994.py:5: FutureWarning:

A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are set
ting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = d
f[col].method(value) instead, to perform the operation inplace on the original object.
```

Out[ ]:

	CustomerID	Monthly_Average_Spend	Monthly_Std_Spend
0	12346	0.000000	0.000000
1	12347	615.714286	341.070789
2	12348	359.310000	203.875689
3	12349	1457.550000	0.000000
4	12350	294.400000	0.000000
...	...	...	...
4358	18280	180.600000	0.000000
4359	18281	80.820000	0.000000
4360	18282	88.300000	14.792674
4361	18283	203.958000	165.798738
4362	18287	918.640000	216.883792

4363 rows × 3 columns

Trending Slope

In [ ]:
```
from sklearn.linear_model import LinearRegression, SGDRegressor
```

In [ ]:
```
df_customer_monthly_spending['YearMonth'] = df_customer_monthly_spending['Year'] + (df_customer_monthly_spending['Month'] - 1)
```

In [ ]:
```
df_customer_monthly_spending.head()
```

Out[ ]:

	CustomerID	Year	Month	Total_Spend	YearMonth
0	12346	2011	1	0.00	2011.000000
1	12347	2010	12	711.79	2010.916667
2	12347	2011	1	475.39	2011.000000
3	12347	2011	4	636.25	2011.250000
4	12347	2011	6	382.52	2011.416667

In [ ]:
```
LN_model = LinearRegression()
```

In [ ]:
```
def calculate_slope(group):
    X = group['YearMonth'].values.reshape(-1, 1)
    y = group['Total_Spend'].values
    LN_model.fit(X, y)
    return LN_model.coef_[0]
```

In [ ]:
```
customer_trending_slopes = df_customer_monthly_spending.groupby('CustomerID').apply(calculate_slope).reset_index()
customer_trending_slopes.columns = ['CustomerID', 'Trending_Slope']
customer_trending_slopes
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\1420970717.py:1: DeprecationWarning:

DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecated, and in a future version of pandas the gro
uping columns will be excluded from the operation. Either pass `include_groups=False` to exclude the groupings or explicitly se
lect the grouping columns after groupby to silence this warning.
```

Out[ ]:

	CustomerID	Trending_Slope
0	12346	0.000000
1	12347	38.876738
2	12348	-270.587755
3	12349	0.000000
4	12350	0.000000
...	...	...
4358	18280	0.000000
4359	18281	0.000000
4360	18282	-62.760000
4361	18283	206.868759
4362	18287	736.128000

4363 rows × 2 columns

Merge to main dataframe

In [ ]:
```
customer_data = customer_data.merge(df_customer_monthly_spend_stat, on = 'CustomerID', how = 'inner')
customer_data = customer_data.merge(customer_trending_slopes, on = 'CustomerID', how = 'inner')
```

In [ ]:
```
customer_data
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Averag
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	
...	...	...	...	...	...	...	
4280	18280	277	1	45	180.60	180.600000	
4281	18281	180	1	54	80.82	80.820000	
4282	18282	7	3	98	176.60	58.866667	
4283	18283	3	16	1355	2039.58	127.473750	
4284	18287	42	3	1586	1837.28	612.426667	

4285 rows × 16 columns

3.4.7. Review the afore-mentioned steps

In [ ]:
```
customer_data.columns
```

Out[ ]:
```
Index(['CustomerID', 'Days_Since_Last_Purchase', 'Total_Transactions',
       'Total_Products_Purchased', 'Total_Spend', 'Average_Transaction_Value',
       'Average_Days_Between_Purchases', 'Day_Of_Week', 'Hour',
       'Unique_Products_Purchased', 'Is_UK', 'Cancellation_Frequency',
       'Cancel_Rate', 'Monthly_Average_Spend', 'Monthly_Std_Spend',
       'Trending_Slope'],
      dtype='object')
```

I will create a data dictionary to for later tracks

In [ ]:
```
data_dict = [
    {"Variable": "CustomerID",
     "Description": "Unique identifier for each customer, used to differentiate individual customers."},

    {"Variable": "Days_Since_Last_Purchase",
     "Description": "The number of days that have elapsed since the customer's most recent purchase."},

    {"Variable": "Total_Transactions",
     "Description": "The total count of transactions the customer has completed."},

    {"Variable": "Total_Products_Purchased",
     "Description": "The cumulative number of products the customer has bought across all transactions."},

    {"Variable": "Total_Spend",
     "Description": "The total expenditure by the customer across all transactions."},

    {"Variable": "Average_Transaction_Value",
     "Description": "The average value of the customer's transactions, calculated as total spend divided by the total number o

    {"Variable": "Unique_Products_Purchased",
     "Description": "The count of distinct products the customer has purchased."},

    {"Variable": "Average_Days_Between_Purchases",
     "Description": "The average number of days between consecutive purchases made by the customer."},

    {"Variable": "Day_Of_Week",
     "Description": "The day of the week when the customer most frequently shops, represented numerically (0 for Monday, 6 for

    {"Variable": "Hour",
     "Description": "The hour of the day when the customer most frequently shops, represented in 24-hour format."},

    {"Variable": "Is_UK",
     "Description": "A binary variable indicating whether the customer is based in the UK (1 for UK, 0 for others)."},

    {"Variable": "Cancellation_Frequency",
     "Description": "The total count of transactions the customer has cancelled."},

    {"Variable": "Cancellation_Rate",
     "Description": "The proportion of transactions that the customer has cancelled, calculated as cancellation frequency divi
```

```
    {"Variable": "Monthly_Average_Spend",
     "Description": "The mean monthly expenditure of the customer."},

    {"Variable": "Monthly_Std_Spend",
     "Description": "The standard deviation of the customer's monthly spending, reflecting the variability in their spending p

    {"Variable": "Trending_Slope",
     "Description": "A numerical indicator of the trend in the customer's spending over time. Positive indicates an upward tre
]

df_dict = pd.DataFrame(data_dict)
```

In [ ]:
```
# extend column width
pd.set_option('display.max_colwidth', 200)
```

In [ ]:
```
df_dict
```

Out[ ]:

	Variable	Description
0	CustomerID	Unique identifier for each customer, used to differentiate individual customers.
1	Days_Since_Last_Purchase	The number of days that have elapsed since the customer's most recent purchase.
2	Total_Transactions	The total count of transactions the customer has completed.
3	Total_Products_Purchased	The cumulative number of products the customer has bought across all transactions.
4	Total_Spend	The total expenditure by the customer across all transactions.
5	Average_Transaction_Value	The average value of the customer's transactions, calculated as total spend divided by the total number of transactions.
6	Unique_Products_Purchased	The count of distinct products the customer has purchased.
7	Average_Days_Between_Purchases	The average number of days between consecutive purchases made by the customer.
8	Day_Of_Week	The day of the week when the customer most frequently shops, represented numerically (0 for Monday, 6 for Sunday).
9	Hour	The hour of the day when the customer most frequently shops, represented in 24-hour format.
10	Is_UK	A binary variable indicating whether the customer is based in the UK (1 for UK, 0 for others).
11	Cancellation_Frequency	The total count of transactions the customer has cancelled.
12	Cancellation_Rate	The proportion of transactions that the customer has cancelled, calculated as cancellation frequency divided by total transactions.
13	Monthly_Average_Spend	The mean monthly expenditure of the customer.
14	Monthly_Std_Spend	The standard deviation of the customer's monthly spending, reflecting the variability in their spending pattern.
15	Trending_Slope	A numerical indicator of the trend in the customer's spending over time. Positive indicates an upward trend, negative indicates a downward trend, and values near zero indicate stability.

In [ ]:
```
# extend column width
pd.set_option('display.max_colwidth', 30)
```

3.5. Outliers Detection and Resolvability

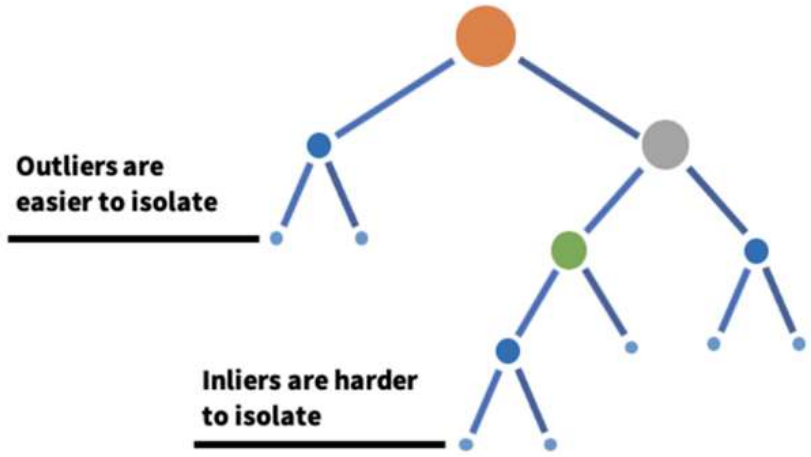
Why:

- The outliers detection seems to be one of the most importaant part of the project. This stems from the fact that when using outliers values, it can lead to a bad clustering, make the model potentially bias and skew our analysis. In K-means, the outliers can affect on the centroid and the distance of the data points. Therefore, it's essential for us to remove them and make the model more accurate and meaningful

Approach:

- I will handle outliers with `IsolationForest Algorithm` due to it's ability to handle outliers in multi-dimensional spaces. This algorithm randomly takes out and isolates a feature from the principle dataset, then choose a random value to split min - max value of the chosen feature.

Image:



In [ ]:
```
from sklearn.ensemble import IsolationForest
```

In [ ]:
```
# ?IsolationForest
```

In [ ]:
```
IF_model = IsolationForest(contamination=0.05, random_state=42)
IF_model
```

Out[ ]:
```
IsolationForest
IsolationForest(contamination=0.05, random_state=42)
```

In [ ]:
```
customer_data_array = customer_data.iloc[:, 1:].to_numpy()
```

In [ ]:
```
customer_data['Outlier_Scores'] = IF_model.fit_predict(customer_data_array)
```

Importance Notes: The output 1 is inliers ; -1 is outliers

In [ ]:
```
customer_data['Is_Outliers'] = customer_data['Outlier_Scores'].map({1: 0, -1: 1})
customer_data
```

Out[ ]:

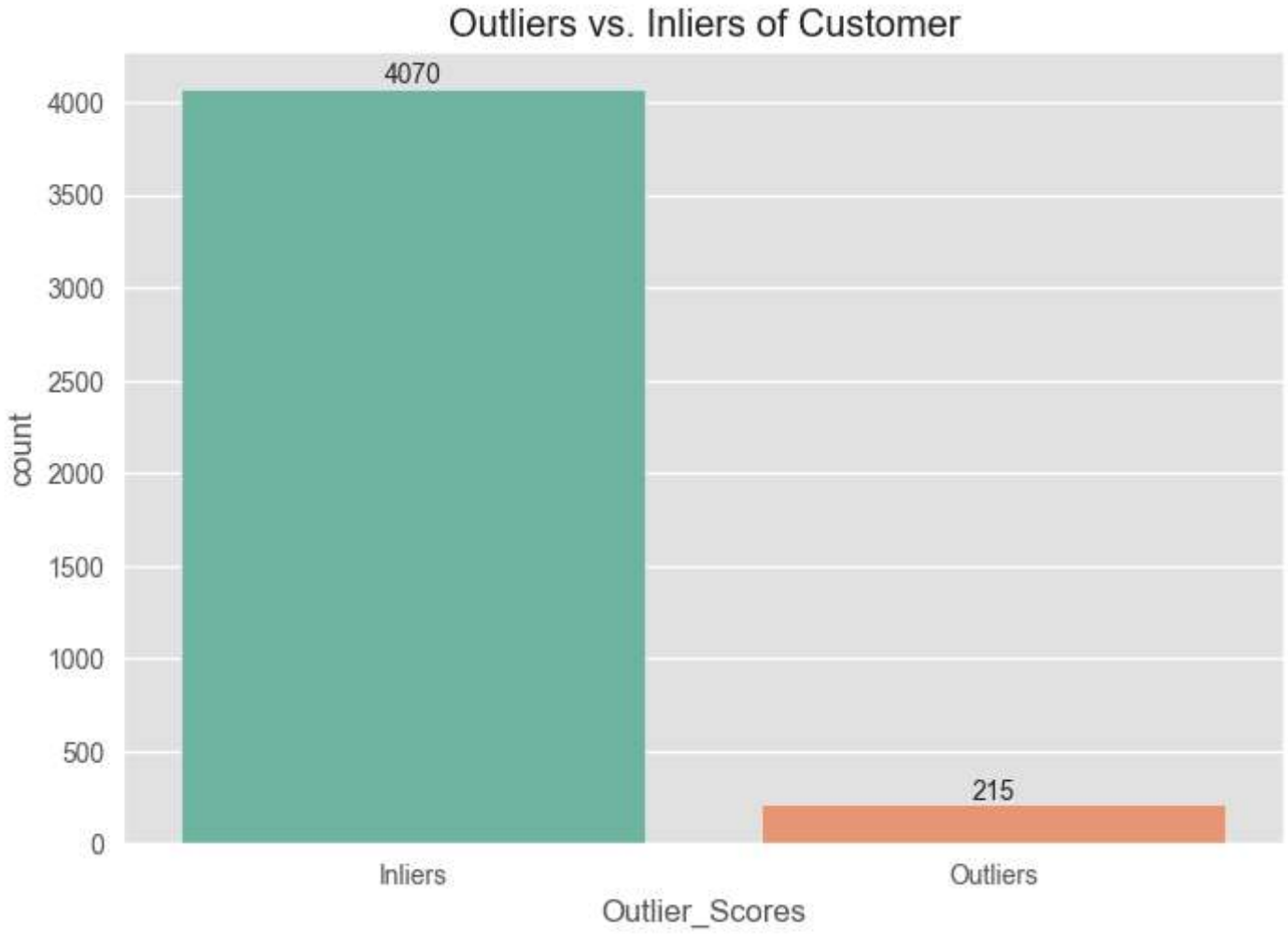
	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Averag
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	
...	...	...	...	...	...	...	
4280	18280	277	1	45	180.60	180.600000	
4281	18281	180	1	54	80.82	80.820000	
4282	18282	7	3	98	176.60	58.866667	
4283	18283	3	16	1355	2039.58	127.473750	
4284	18287	42	3	1586	1837.28	612.426667	

4285 rows × 18 columns

In [ ]:

```
ax = sns.countplot(data = customer_data
                   , x = customer_data['Outlier_Scores'].map({1: 'Inliers', -1: 'Outliers'})
                   , palette = 'Set2'
                   )

for i in ax.containers:
    ax.bar_label(i,)

plt.title('Outliers vs. Inliers of Customer')
plt.show()
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\3306960643.py:1: FutureWarning:


Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and se
t `legend=False` for the same effect.
```



Strategy:

- In this project, outliers contain about ~5% of the data. So `IsolationForest` works well.
- Split outliers into another dataframe, use them later if needed.
- Continue further analysis for the inliers.
- Drop 2 columns: *Outlier_Scores* and *Is_Outliers*

In [ ]:
```
customer_data_ouliers = customer_data[customer_data['Is_Outliers'] == 1]
customer_data_cleaned = customer_data[customer_data['Is_Outliers'] == 0]

customer_data_ouliers.drop(columns=[ 'Is_Outliers','Outlier_Scores'], inplace=True)
customer_data_cleaned.drop(columns=[ 'Is_Outliers','Outlier_Scores'], inplace=True)
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\3389971279.py:4: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\3389971279.py:5: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

In [ ]:
```
customer_data_cleaned.head()
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Average_D
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	

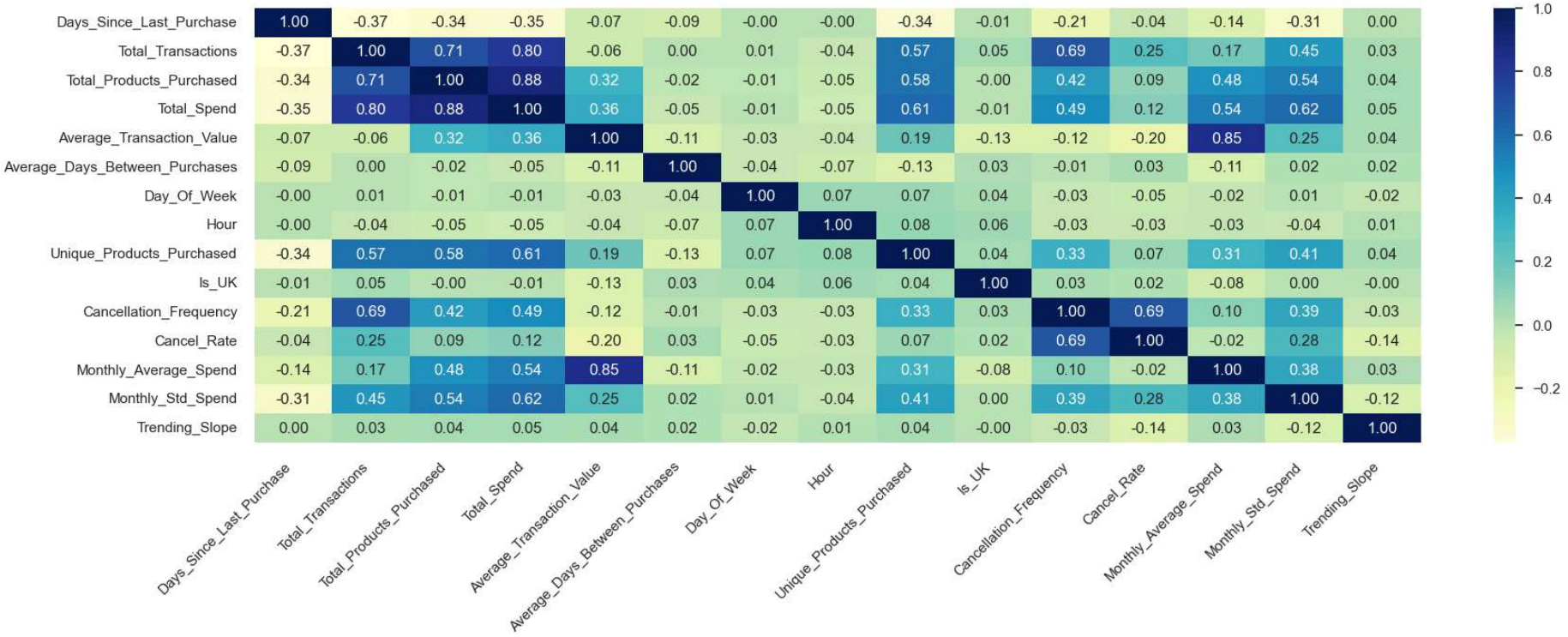
Let's see how many customers left ^^

In [ ]:
```
print('Remaining customers:' , customer_data_cleaned.shape[0])
```

```
Remaining customers: 4070
```

3.6. Correlation Analysis

In [ ]:
```
sns.set_style("whitegrid")
sns.set_context("notebook")


plt.figure(figsize = (20,6))
sns.heatmap(data = customer_data_cleaned.corr(numeric_only=True), annot = True, cmap = "YlGnBu", fmt = '.2f')

plt.xticks(rotation = 45
           , ha = 'right')

plt.show()
```



There are some pairs of features that have high correlations:

- Monthly_Average_Spend and Average_Transaction_Value
- Total_Spend and Uniqe_Products_Purchased
- ... and so on

These variables move closely together, implying a possibility of **Multicollinearity**.

3.7. Feature Scailing

In [ ]:
```
from sklearn.preprocessing import StandardScaler
```

In [ ]:
```
customer_data_cleaned.columns
```

Out[ ]:
```
Index(['CustomerID', 'Days_Since_Last_Purchase', 'Total_Transactions',
       'Total_Products_Purchased', 'Total_Spend', 'Average_Transaction_Value',
       'Average_Days_Between_Purchases', 'Day_Of_Week', 'Hour',
       'Unique_Products_Purchased', 'Is_UK', 'Cancellation_Frequency',
       'Cancel_Rate', 'Monthly_Average_Spend', 'Monthly_Std_Spend',
       'Trending_Slope'],
      dtype='object')
```

In [ ]:
```
pre_scaled_columns = [ 'Days_Since_Last_Purchase', 'Total_Transactions',
       'Total_Products_Purchased', 'Total_Spend', 'Average_Transaction_Value',
       'Average_Days_Between_Purchases', 'Hour',
       'Unique_Products_Purchased', 'Cancellation_Frequency',
       'Cancel_Rate', 'Monthly_Average_Spend', 'Monthly_Std_Spend',
       'Trending_Slope']
```

In [ ]:
```
scaler = StandardScaler()
scaler
```

Out[ ]:
```
StandardScaler
StandardScaler()
```

In [ ]:
```
customer_data_scaled = customer_data_cleaned.copy()

customer_data_scaled[pre_scaled_columns] = scaler.fit_transform(customer_data_scaled[pre_scaled_columns])
customer_data_scaled
```

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Averag
0	12346	2.364696	-0.477864	-0.769735	-0.824326	-1.305773	
1	12347	-0.901766	0.662978	2.021179	2.351817	1.499599	
2	12348	-0.163526	-0.021527	1.878113	0.234811	0.331347	
3	12349	-0.739960	-0.706033	-0.054407	0.249778	5.335245	
4	12350	2.213003	-0.706033	-0.547189	-0.607376	0.035598	
...	...	...	...	...	...	...	
4280	18280	1.879277	-0.706033	-0.718640	-0.691238	-0.482907	
4281	18281	0.898327	-0.706033	-0.708421	-0.764768	-0.937533	
4282	18282	-0.851202	-0.249696	-0.658462	-0.694185	-1.037559	
4283	18283	-0.891653	2.716494	0.768787	0.678690	-0.724966	
4284	18287	-0.497251	-0.249696	1.031074	0.529610	1.484619	

4070 rows × 16 columns

In [ ]:

3.8. Dimensional Reduction

Why:

- Multicollinearity detected in previous step
- Better Clustering with K-means since K-means is a distance based algorithm then too much features can cause the less informative of the dataset. Less features may help K-means to cluster better.
- Noise Reduction by focusing on some importance features

- Enhance visualizations
- Imporve computing speed

Which:

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)` for the following reasons:

- Good Starting Point: PCA is effective at capturing linear relationships in the data, which is important given the multicollinearity we found. It helps reduce the number of features while retaining most of the important information, making our clustering analysis potentially more accurate and easier to interpret.
- Computational Efficiency: PCA is computationally efficient, so it won't significantly increase processing time.

However, we're keeping our options open. If PCA doesn't capture enough variance, meaning it might miss out on important information, we might explore other non-linear methods. These methods could provide a more detailed view of complex patterns in the data, but they may require more computation and time.

In [ ]:
```
from sklearn.decomposition import PCA
```

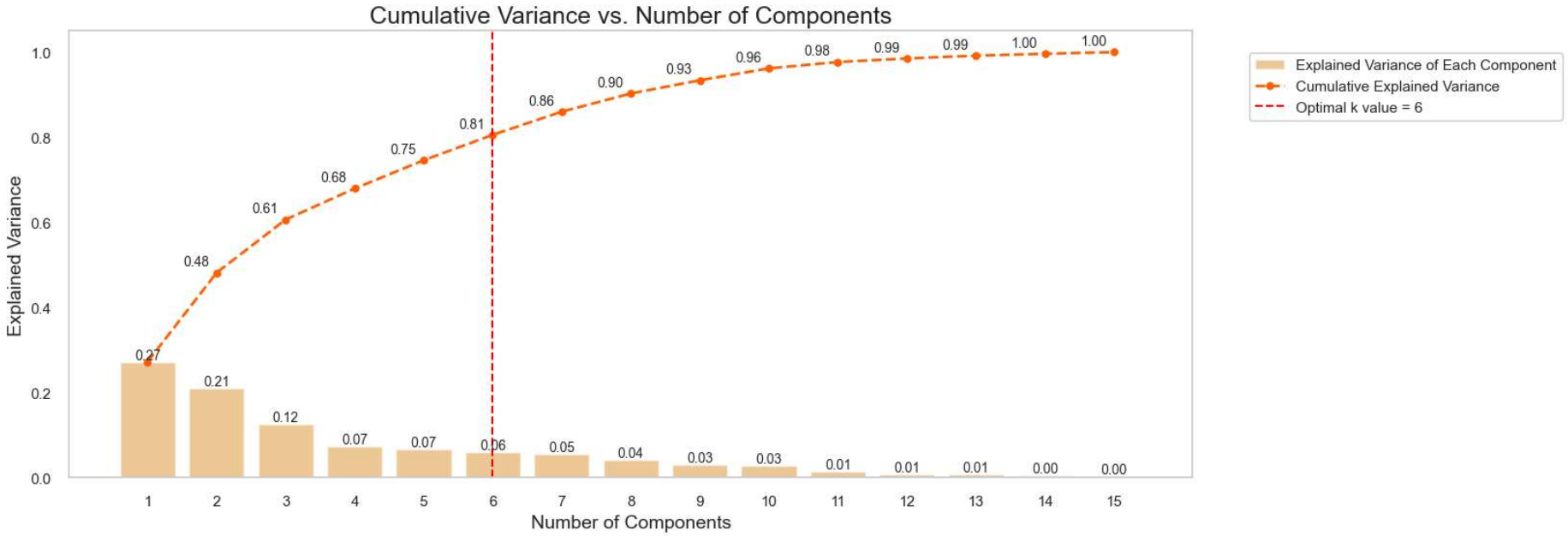
In [ ]:
```
# Setting CustomerID as the index column
customer_data_scaled.set_index('CustomerID', inplace=True)

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

Out[ ]:
```
PCA()
```

In [ ]:
```
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_explained_variance = np.cumsum(explained_variance_ratio)
```

```python
optimal_k = 6

plt.figure(figsize=(15, 6))

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

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

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

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

# Customize ticks and legend
plt.xticks(range(0, len(cumulative_explained_variance)))
plt.legend(handles=[barplot.patches[0], lineplot, optimal_k_line],
           labels=['Explained Variance of Each Component', 'Cumulative Explained Variance', f'Optimal k value = {optimal_k}'],
           loc=(0.62, 0.1),
           bbox_to_anchor=(1.05, 0.8),
           frameon=True,
           framealpha=1.0)

# Display the variance values for both graphs on the plots
x_offset = -0.3
y_offset = 0.01
for i, (ev_ratio, cum_ev_ratio) in enumerate(zip(explained_variance_ratio, cumulative_explained_variance)):
    plt.text(i, ev_ratio, f"{ev_ratio:.2f}", ha="center", va="bottom", fontsize=10)
    if i > 0:
        plt.text(i + x_offset, cum_ev_ratio + y_offset, f"{cum_ev_ratio:.2f}", ha="center", va="bottom", fontsize=10)
```

```
plt.grid(False)
plt.show()
```



To determine the optimal number of components, we typically look for the "elbow point" on the plot.

From the plot, it appears that the rate of increase in cumulative variance begins to taper off **after the 6th component, which accounts for approximately 81%** of the total variance.

In the context of customer segmentation, it's important to keep enough information to accurately identify distinct customer groups. Therefore, retaining the first 6 components seems like a reasonable choice. They collectively capture a significant portion of the variance while reducing the dataset's dimensionality.

In [ ]:
```
# Creating a PCA object with 6 components
pca = PCA(n_components=6, random_state = 42)

# 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
```

In [ ]:
```
customer_data_pca
```

Out[ ]:

CustomerID	PC1	PC2	PC3	PC4	PC5	PC6
12346	-2.008651	-1.737604	-2.367384	-1.703024	-0.287408	1.976157
12347	3.293514	-1.443549	1.928351	0.752986	-0.169012	-0.786635
12348	0.611355	0.568745	0.722391	0.919500	-0.815395	-1.737688
12349	1.628160	-2.730849	5.585956	-1.941364	1.089819	0.704577
12350	-2.026492	-0.502494	0.667558	-0.277420	-1.495086	0.216499
...	...	...	...	...	...	...
18280	-2.203774	-2.647653	0.135626	-0.408042	0.249111	1.369353
18281	-2.287979	3.364751	-0.395960	-0.067892	0.561250	0.998503
18282	-1.110251	1.357503	-1.971440	-0.022772	0.449942	-0.210959
18283	2.576528	0.772925	-0.640374	2.581534	-1.157880	-0.769577
18287	1.463886	-0.628566	2.463847	-0.220318	0.950722	0.620463

4070 rows × 6 columns

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

# Create the PCA component DataFrame and apply the highlighting function
pc_df = pd.DataFrame(pca.components_.T, columns=['PC{}'.format(i+1) for i in range(pca.n_components_)],
                     index=customer_data_scaled.columns)
```

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

Out[ ]:

	PC1	PC2	PC3	PC4	PC5	PC6
Days_Since_Last_Purchase	-0.221049	-0.009334	0.049851	-0.319647	-0.193325	0.417757
Total_Transactions	0.387273	0.011258	-0.236155	0.205270	-0.033873	0.076605
Total_Products_Purchased	0.412052	-0.000058	0.070321	0.126818	0.030131	0.032508
Total_Spend	0.440168	0.004655	0.067332	0.086599	0.015723	0.054097
Average_Transaction_Value	0.170688	-0.019587	0.571220	-0.252436	0.065162	0.027196
Average_Days_Between_Purchases	-0.024721	-0.037041	-0.127882	0.181771	0.711480	-0.233762
Day_Of_Week	-0.012281	0.994802	-0.009097	-0.041270	0.069702	0.046267
Hour	-0.020824	0.057162	0.002894	0.174766	-0.612335	-0.472961
Unique_Products_Purchased	0.334294	0.056073	0.023221	0.215679	-0.214521	-0.091850
Is_UK	-0.000066	0.006326	-0.021146	0.017094	-0.009844	-0.011291
Cancellation_Frequency	0.300131	-0.018734	-0.411575	-0.190264	-0.079350	0.200952
Cancel_Rate	0.138938	-0.035332	-0.442915	-0.467287	-0.042006	0.142832
Monthly_Average_Spend	0.269412	-0.010873	0.461547	-0.279460	0.028936	0.061940
Monthly_Std_Spend	0.337002	0.006327	-0.012208	-0.195160	0.116529	-0.183408
Trending_Slope	0.002199	-0.012943	0.098996	0.534365	-0.038312	0.653903

4. K-Means Clustering

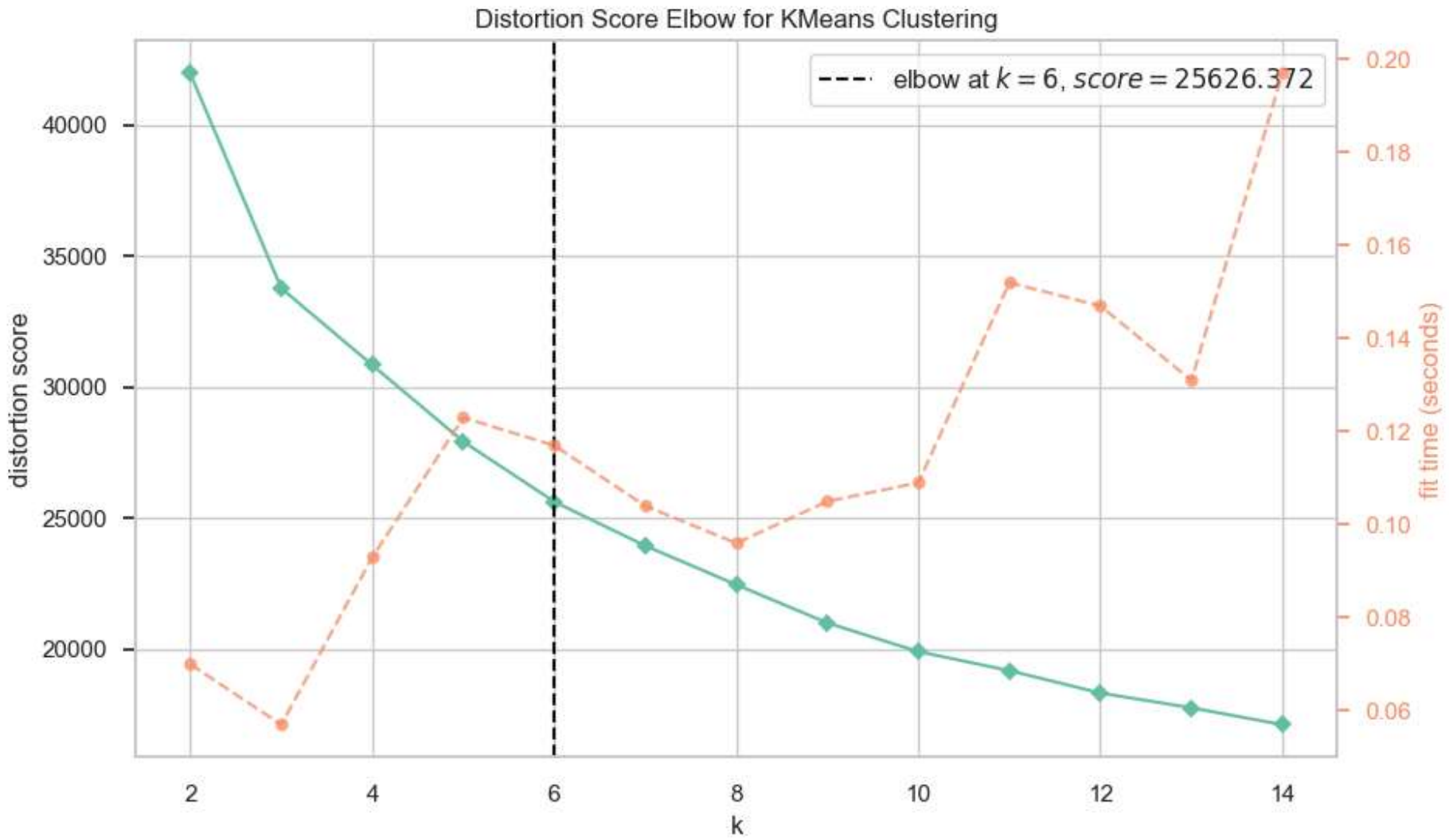
In [ ]:

```
from yellowbrick.cluster import KElbowVisualizer, SilhouetteVisualizer
from sklearn.cluster import KMeans
import matplotlib.gridspec as gridspec
```

4.1. Elbow method

In [ ]:

```
# Set palette
sns.set_palette("Set2")

# Instantiate the clustering model with the specified parameters
km = KMeans(init='k-means++', n_init=10, max_iter=100, random_state=42)

# Create a figure and axis with the desired size
fig, ax = plt.subplots(figsize=(10, 6))

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

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

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



Out[ ]: <Axes: title={'center': 'Distortion Score Elbow for KMeans Clustering'}, xlabel='k', ylabel='distortion score'>

4.2. Silhouette calculation

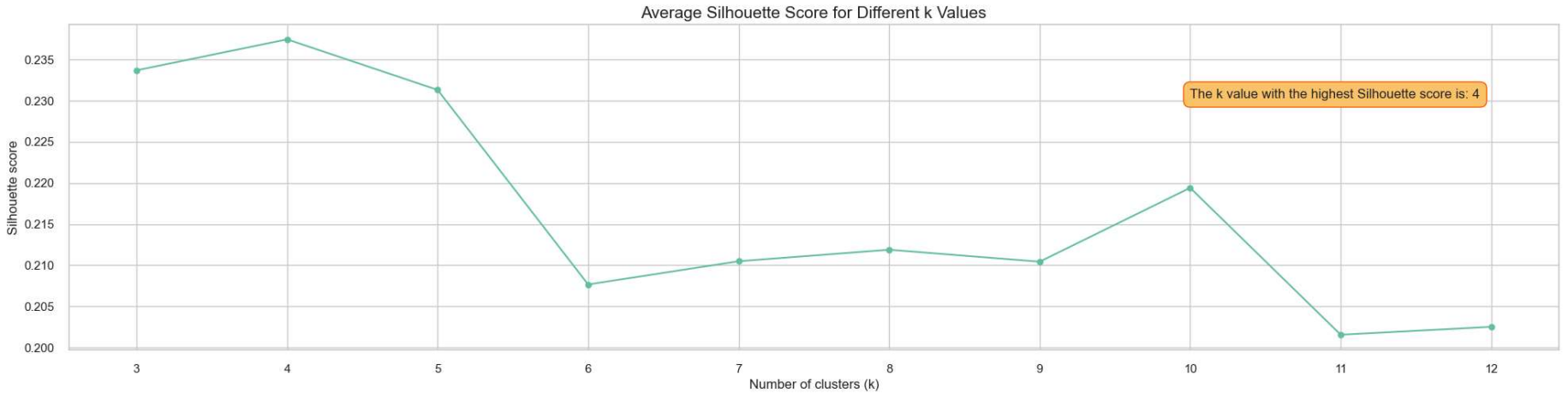
In [ ]:
```
from sklearn.metrics import silhouette_score, calinski_harabasz_score, davies_bouldin_score
```

In [ ]:

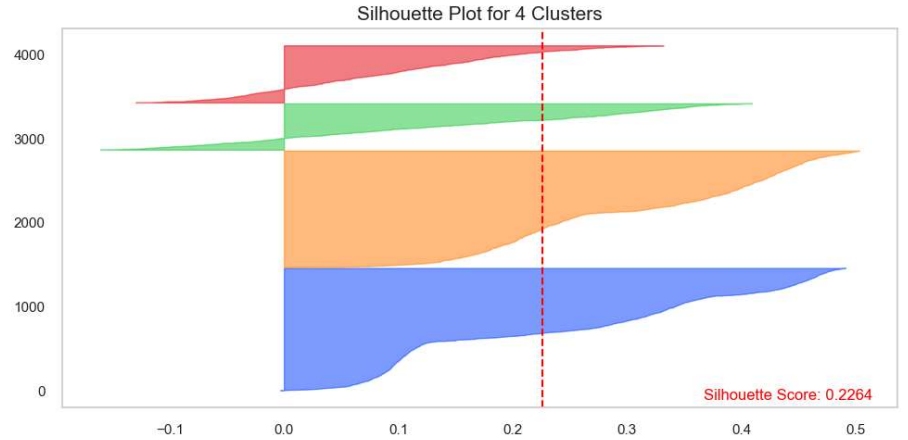
```
def silhouette_analysis(df, start_k, stop_k, figsize=(15, 16)):
    """
    Perform Silhouette analysis for a range of k values and visualize the results.
    """

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

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

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

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

    silhouette_scores = []

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

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

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

    # Add the optimal k value text to the plot
    optimal_k_text = f'The k value with the highest Silhouette score is: {best_k}'
    plt.text(10, 0.23, optimal_k_text, fontsize=12, verticalalignment='bottom',
             horizontalalignment='left', bbox=dict(facecolor='#fcc36d', edgecolor='#ff6200', boxstyle='round, pad=0.5'))
```

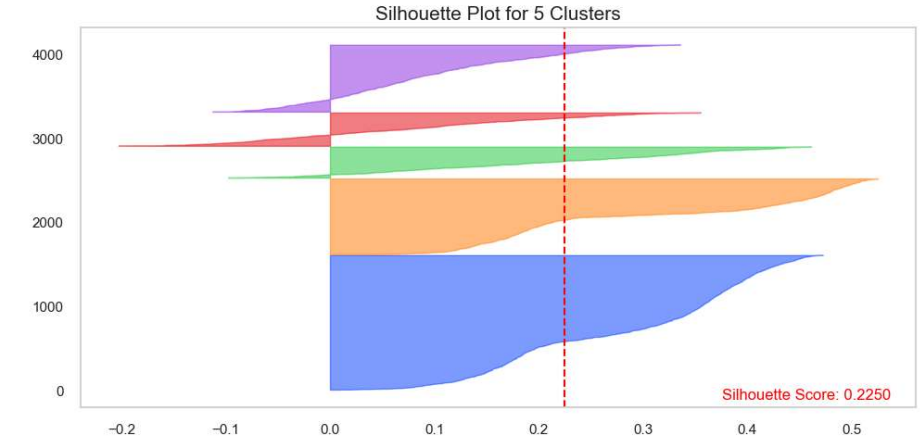
```
    # Second plot (subplot): Silhouette plots for each k value
    colors = sns.color_palette("bright")

    for i in range(start_k, stop_k + 1):
        km = KMeans(n_clusters=i, init='k-means++', n_init=10, max_iter=100, random_state=42)
        row_idx, col_idx = divmod(i - start_k, 2)

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

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

        # Add the Silhouette score text to the plot
        score = silhouette_score(df, km.labels_)
        ax.text(0.97, 0.02, f'Silhouette Score: {score:.4f}', fontsize=12, \
                ha='right', transform=ax.transAxes, color='red')

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

        ax.grid(False)
    plt.tight_layout()
    plt.show()
```

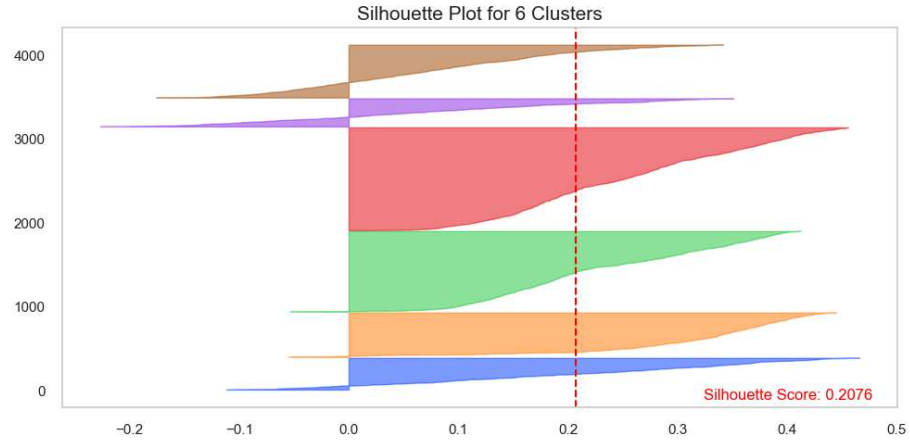
In [ ]:
```
silhouette_analysis(customer_data_pca, 3, 12, figsize=(20, 50))
```

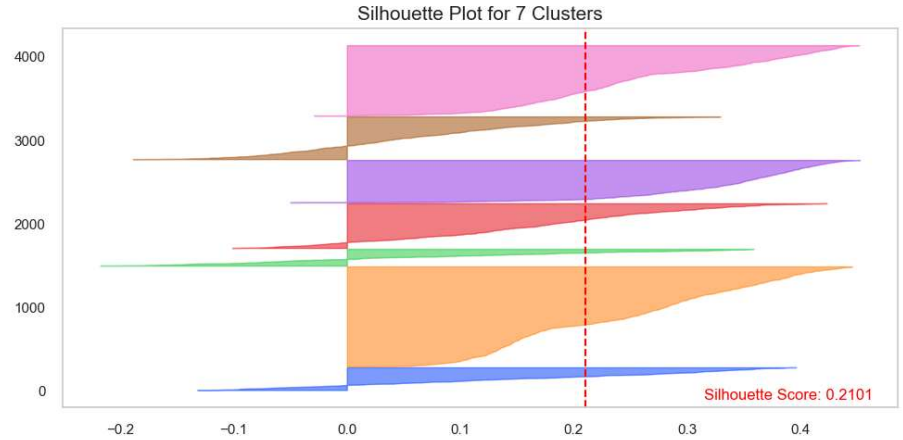

Average Silhouette Score for Different k Values
Silhouette score
Number of clusters (k)
The k value with the highest Silhouette score is: 4

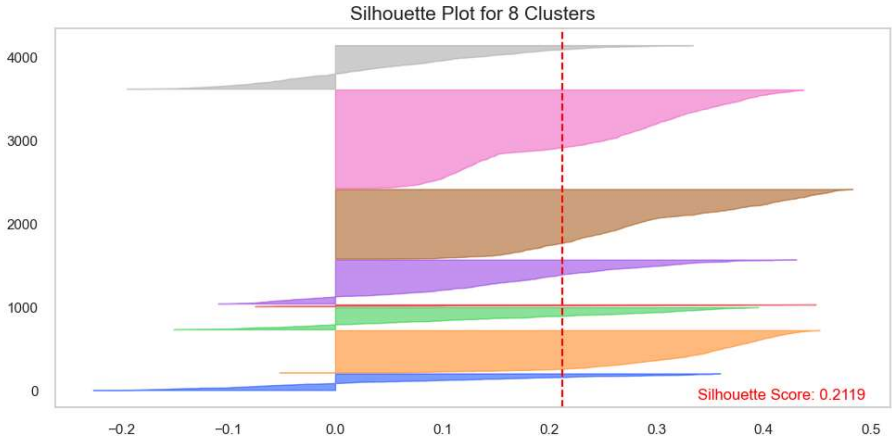

Silhouette Plot for 3 Clusters
Silhouette Score: 0.2338

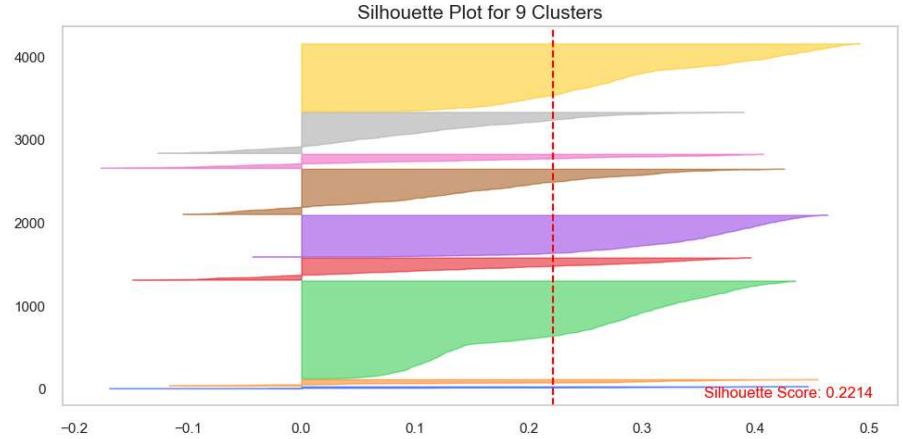

Silhouette Plot for 4 Clusters
Silhouette Score: 0.2264

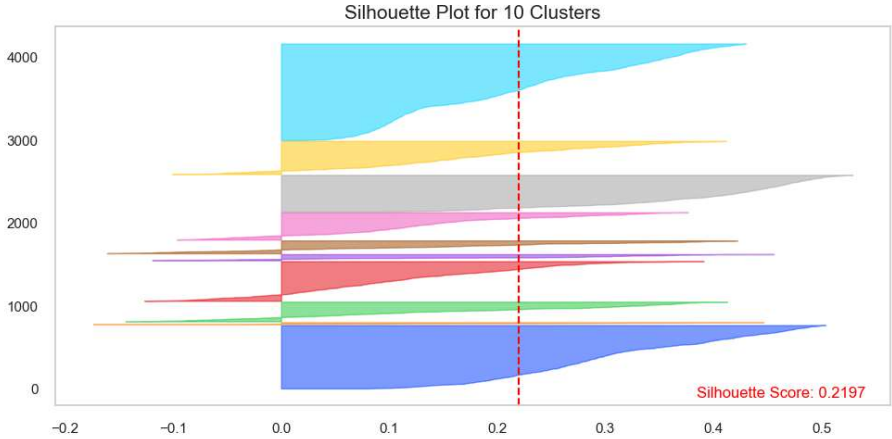

Silhouette Plot for 5 Clusters
Silhouette Score: 0.2250

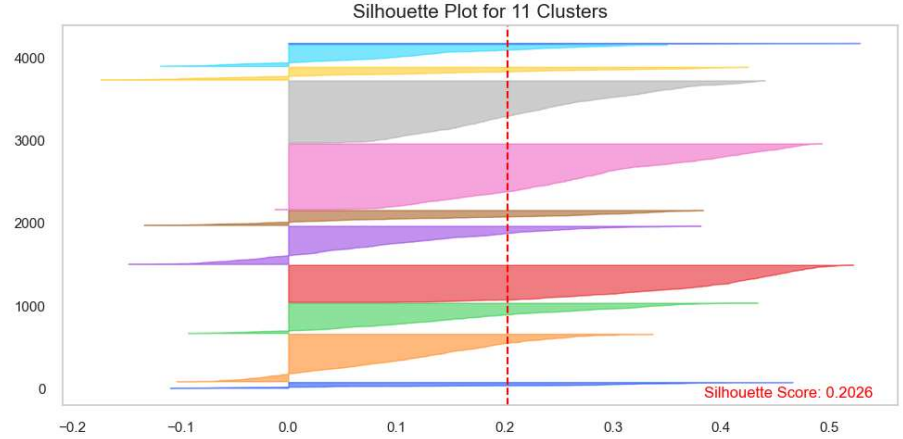

Silhouette Plot for 6 Clusters
Silhouette Score: 0.2076

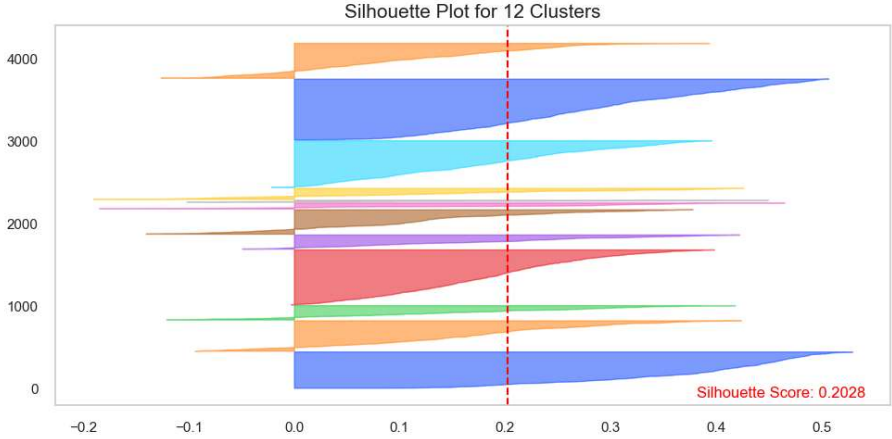

Silhouette Plot for 7 Clusters
Silhouette Score: 0.2101


Silhouette Plot for 8 Clusters
Silhouette Score: 0.2119


Silhouette Plot for 9 Clusters
Silhouette Score: 0.2214


Silhouette Plot for 10 Clusters
Silhouette Score: 0.2197


Silhouette Plot for 11 Clusters
Silhouette Score: 0.2026


Silhouette Plot for 12 Clusters
Silhouette Score: 0.2028

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.

In [ ]:
```
from collections import Counter
```

In [ ]:
```
# Apply KMeans clustering using the optimal k
kmeans = KMeans(n_clusters=3, init='k-means++', n_init=10, max_iter=100, random_state=42)
kmeans.fit(customer_data_pca)

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

cluster_frequencies
```

Out[ ]:
```
Counter({0: 1681, 1: 1671, 2: 718})
```

In [ ]:
```
cluster_frequencies.most_common()
```

Out[ ]:
```
[(0, 1681), (1, 1671), (2, 718)]
```

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

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

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

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

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

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\1829448208.py:14: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

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

Out[ ]:

	CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Average_D
0	12346	325	2	0	0.00	0.000000	
1	12347	2	7	2458	4310.00	615.714286	
2	12348	75	4	2332	1437.24	359.310000	
3	12349	18	1	630	1457.55	1457.550000	
4	12350	310	1	196	294.40	294.400000	

4.3. Model Evaluation

4.3.1. 3D ScatterPlot of PCA

In [ ]:
```
import plotly.graph_objects as go
```

In [ ]:
```
colors = ['#e8000b', '#1ac938', '#023eff']
```

In [ ]:
```
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]
```

In [ ]:
```
customer_data_pca
```

Out[ ]:

CustomerID	PC1	PC2	PC3	PC4	PC5	PC6	cluster
12346	-2.008651	-1.737604	-2.367384	-1.703024	-0.287408	1.976157	0
12347	3.293514	-1.443549	1.928351	0.752986	-0.169012	-0.786635	2
12348	0.611355	0.568745	0.722391	0.919500	-0.815395	-1.737688	1
12349	1.628160	-2.730849	5.585956	-1.941364	1.089819	0.704577	0
12350	-2.026492	-0.502494	0.667558	-0.277420	-1.495086	0.216499	0
...	...	...	...	...	...	...	...
18280	-2.203774	-2.647653	0.135626	-0.408042	0.249111	1.369353	0
18281	-2.287979	3.364751	-0.395960	-0.067892	0.561250	0.998503	1
18282	-1.110251	1.357503	-1.971440	-0.022772	0.449942	-0.210959	1
18283	2.576528	0.772925	-0.640374	2.581534	-1.157880	-0.769577	2
18287	1.463886	-0.628566	2.463847	-0.220318	0.950722	0.620463	2

4070 rows × 7 columns

In [ ]:
```
fig = go.Figure()

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'))

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
)

fig.show()
```

4.3.2. Cluster Distribution

In [ ]:
```
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)
cluster_percentage
```

Out[ ]:

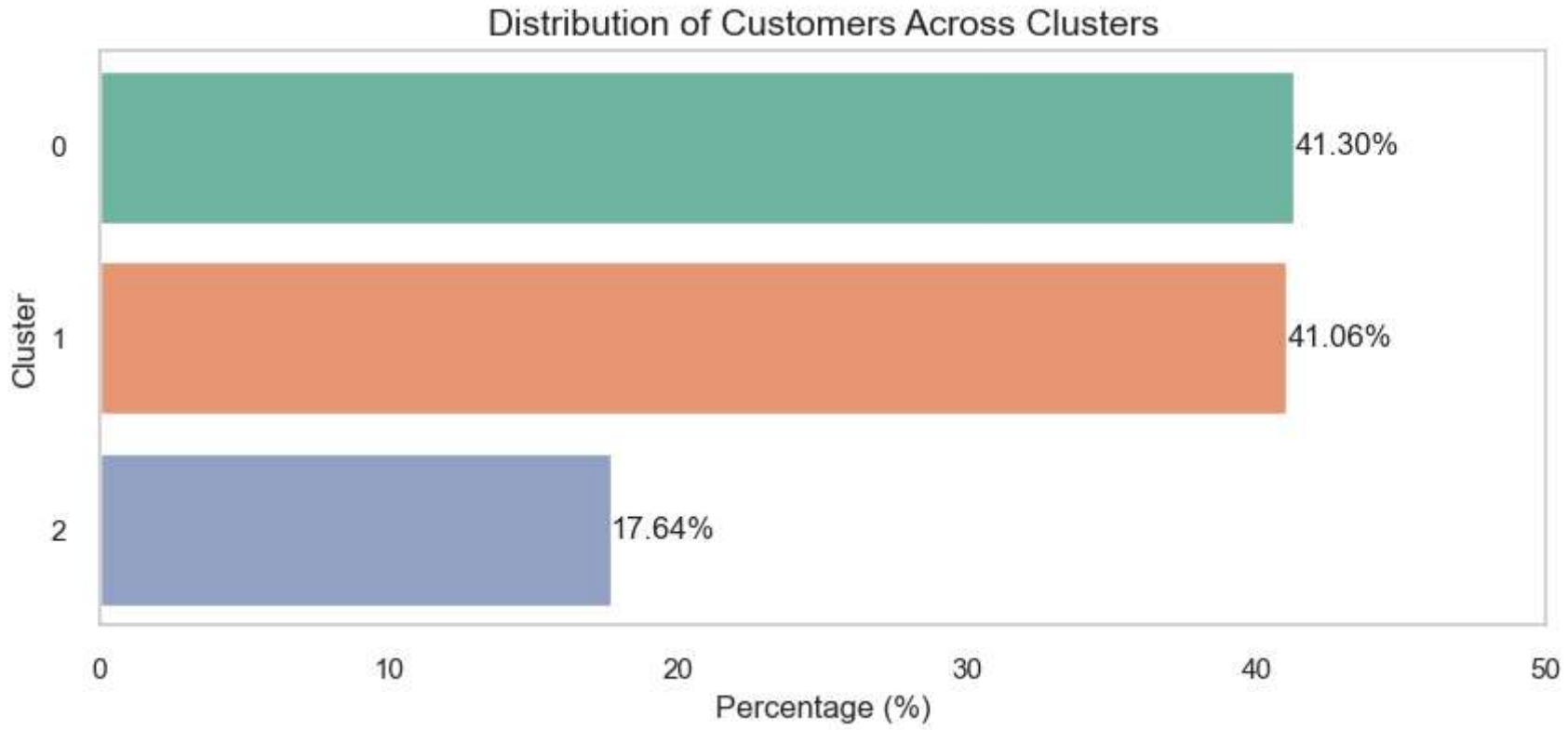
	Cluster	Percentage
0	0	41.302211
1	1	41.056511
2	2	17.641278

In [ ]:
```
cluster_percentage['Cluster'] = cluster_percentage['Cluster'].astype('str')
```

In [ ]:
```
# Create a horizontal bar plot
plt.figure(figsize=(10, 4))
ax = sns.barplot(data=cluster_percentage
            , y= 'Cluster'
            , x= 'Percentage'
            , palette='Set2')

for i in ax.containers:
    ax.bar_label(i, fmt = '%.2f%%')

plt.title('Distribution of Customers Across Clusters', fontsize=14)
plt.xlabel('Percentage (%)')
plt.xlim(0,50)
```

```
plt.grid(False)

plt.show()
```

```
C:\Users\TuNguyen\AppData\Local\Temp\ipykernel_17040\1374789756.py:3: FutureWarning:


Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and se
t `legend=False` for the same effect.
```



4.3.3. Evaluation Metrics

Evaluating Clustering Effectiveness with Silhouette, Calinski-Harabasz, and Davies-Bouldin Scores

The metrics `silhouette_score`, `calinski_harabasz_score`, and `davies_bouldin_score` are commonly used measures to evaluate the effectiveness of clustering methods in machine learning. Below are the meanings of each metric:

1. Silhouette Score

- **Meaning**: This metric measures how similar a data point is to other points in the same cluster compared to points in different clusters.
- **Value**:
  - The silhouette score ranges from -1 to 1.
  - A score closer to 1 indicates that the data points in that cluster are close to each other and far from points in other clusters, which means good clustering.
  - A score close to 0 indicates that the clusters have a high degree of overlap.
  - A negative score means that the data points are assigned to the wrong cluster.
- **Usage**: Used to evaluate the degree of separation between clusters. A well-isolated cluster will have a high silhouette score.

2. Calinski-Harabasz Score (Variance Ratio Criterion)

- **Meaning**: This metric measures the ratio of the total variance between clusters to the total variance within clusters.
- **Value**:
  - A higher value means the clusters are more distinct.
  - A high Calinski-Harabasz score indicates that the clusters are well-separated and that data points within each cluster are close together.
- **Usage**: This is a good measure for clustering and the performance of clustering methods, especially when you want to find the optimal number of clusters.

3. Davies-Bouldin Score

- **Meaning**: This metric measures the average similarity between each cluster and the nearest cluster, relative to the size of the cluster.
- **Value**:
  - A lower value indicates that the clustering is better, as the clusters are far apart and the data points within the same cluster are close together.
  - A higher value indicates that the clusters may overlap and the clustering is not good.
- **Usage**: Commonly used to determine the quality of clustering. This score is the inverse of the other scores, so a lower value is better.

In Summary:

- **Silhouette Score**: Evaluates the overall clustering of data points.
- **Calinski-Harabasz Score**: Measures the ratio of variance between clusters to variance within clusters, assessing the separation of clusters.
- **Davies-Bouldin Score**: Evaluates the similarity between clusters, with a lower value indicating better clustering.

In [ ]:
```
from tabulate import tabulate
from sklearn.metrics import silhouette_score, calinski_harabasz_score, davies_bouldin_score
```

In [ ]:
```
num_observations = len(customer_data_pca)

X = customer_data_pca.drop('cluster', axis=1)
clusters = customer_data_pca['cluster']

sil_score = silhouette_score(X, clusters)
calinski_score = calinski_harabasz_score(X, clusters)
davies_score = davies_bouldin_score(X, clusters)

table_data = [
    ["Number of Observations", num_observations],
    ["Silhouette Score", sil_score],
    ["Calinski Harabasz Score", calinski_score],
    ["Davies Bouldin Score", davies_score]
]

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

```
+--------------------------+--------------------+
|          Metric          |       Value        |
+--------------------------+--------------------+
| Number of Observations   |        4070        |
|    Silhouette Score      | 0.23378522469278054 |
| Calinski Harabasz Score  | 1226.0713321635212 |
|  Davies Bouldin Score    |  1.393369017152799 |
+--------------------------+--------------------+
```

Analysis:

- **Silhouette Score: 0.23**

The Silhouette Score of 0.23 is **close to 0**, indicating that the clusters have **significant overlap** and the clustering **is not very distinct**.

This may suggest that the data is challenging to cluster, or that the current number of clusters is not optimal.

- **Calinski-Harabasz Score: 1226.07**

  The Calinski-Harabasz Score of 1226.07 is relatively high, which indicates that the clusters are **well separated** and that the data points within each cluster are **relatively close to each other**.

  This suggests that the overall clustering is **reasonably good**.

- **Davies-Bouldin Score: 1.39**

  The Davies-Bouldin Score of 1.39 is relatively low, indicating that the clusters are **well separated** from each other.

  A lower value typically suggests that the clustering has succeeded in keeping the **clusters distinct from one another**.

Conclusion:

Although the Silhouette Score is not high, both the Calinski-Harabasz and Davies-Bouldin Scores indicate that the clustering has a reasonable degree of separation.

However, we may want to reconsider the number of clusters or the clustering method to achieve better separation, especially if the goal is to have more clearly defined clusters.

5. Cluster Profiling and Analyzing

5.1. Approach 1: Radar Chart

In [ ]:
```
import plotly.express as px
from plotly.subplots import make_subplots
```

In [ ]:
```
df_customer = customer_data_cleaned.set_index('CustomerID')
```

In [ ]:
```
scaler
```

Out[ ]:
```
StandardScaler()
```

In [ ]:
```
df_customer_standardized = scaler.fit_transform(df_customer.drop(columns=['cluster'], axis=1))
df_customer_standardized = pd.DataFrame(df_customer_standardized, columns=df_customer.columns[:-1], index=df_customer.index)
df_customer_standardized['cluster'] = df_customer['cluster']
df_customer_standardized.head()
```

Out[ ]:

CustomerID	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Average_Days
12346	2.364696	-0.477864	-0.769735	-0.824326	-1.305773	
12347	-0.901766	0.662978	2.021179	2.351817	1.499599	
12348	-0.163526	-0.021527	1.878113	0.234811	0.331347	
12349	-0.739960	-0.706033	-0.054407	0.249778	5.335245	
12350	2.213003	-0.706033	-0.547189	-0.607376	0.035598	

In [ ]:
```
cluster_centroids = df_customer_standardized.groupby('cluster').mean().reset_index()
cluster_centroids
```

Out[ ]:

	cluster	Days_Since_Last_Purchase	Total_Transactions	Total_Products_Purchased	Total_Spend	Average_Transaction_Value	Average_Days_B
0	0	0.119876	-0.306976	-0.331183	-0.353680	-0.123891	
1	1	0.149605	-0.312234	-0.333788	-0.351408	-0.120785	
2	2	-0.628831	1.445360	1.552199	1.645877	0.571159	

In [ ]:
```
cluster_centroids_melted = cluster_centroids.melt(id_vars=['cluster'], var_name='Metric', value_name='Value')
cluster_centroids_melted.head(6)
```

Out[ ]:

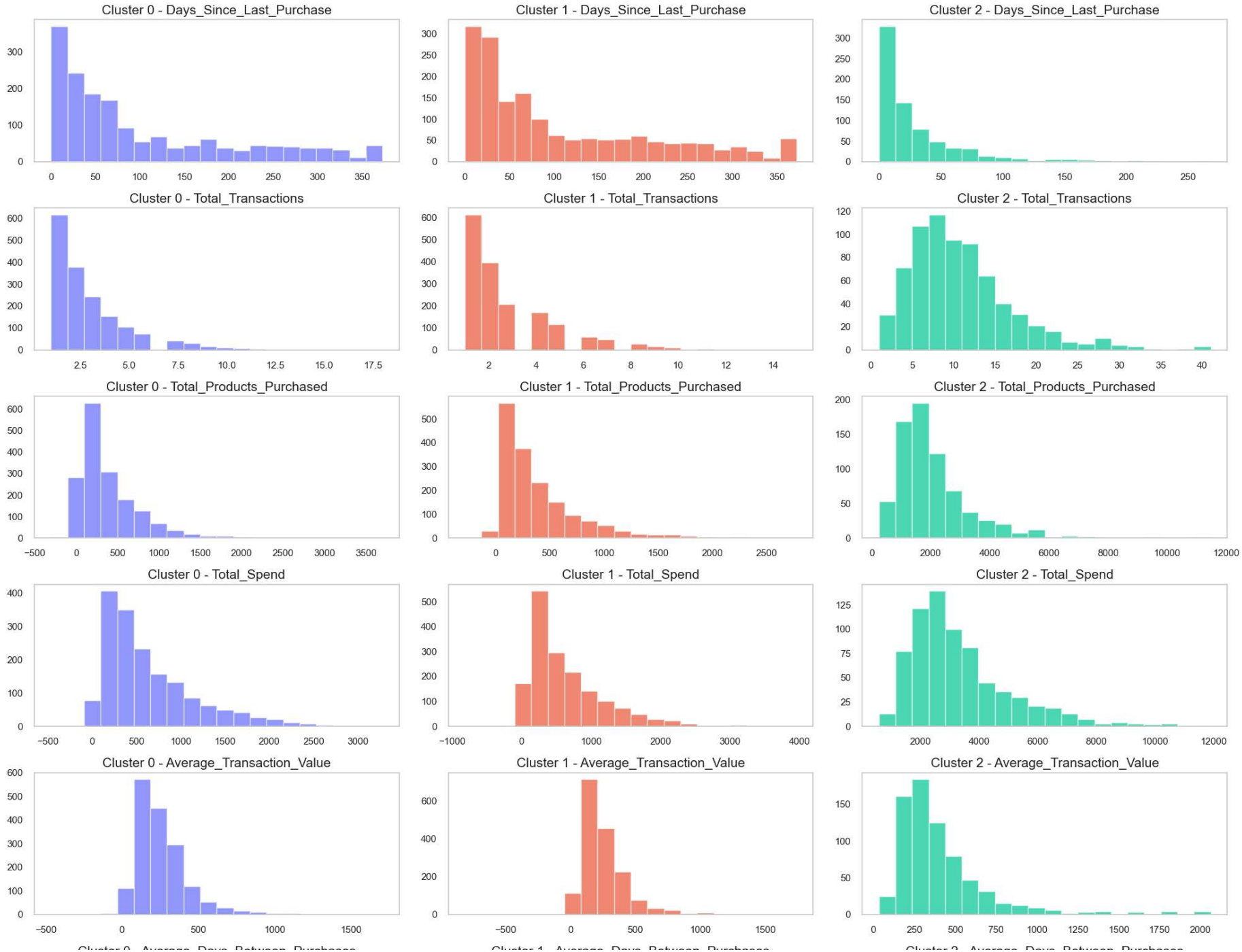
	cluster	Metric	Value
0	0	Days_Since_Last_Purchase	0.119876
1	1	Days_Since_Last_Purchase	0.149605
2	2	Days_Since_Last_Purchase	-0.628831
3	0	Total_Transactions	-0.306976
4	1	Total_Transactions	-0.312234
5	2	Total_Transactions	1.445360

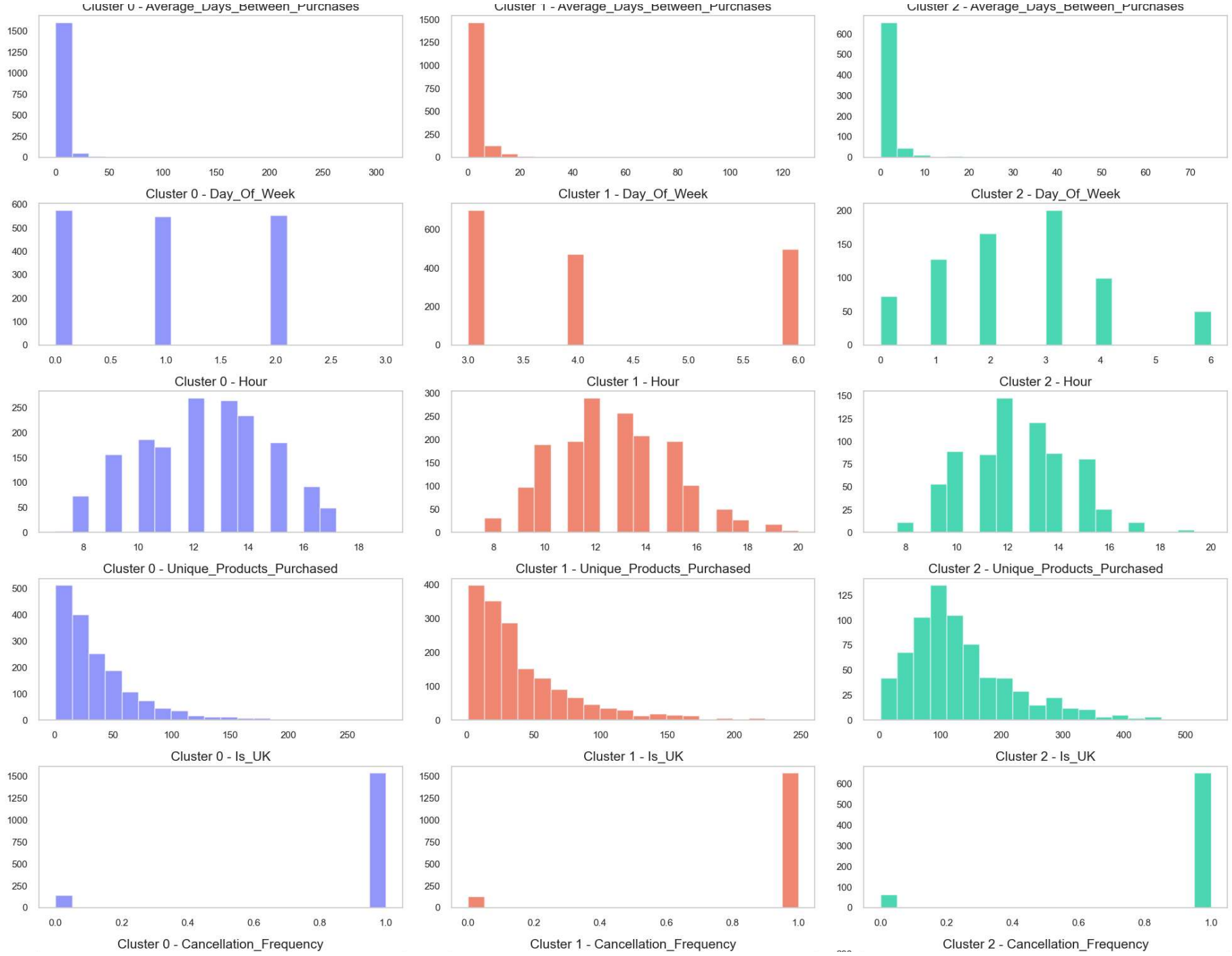
In [ ]:
```
colors = ['#636EFA', '#EF553B', '#00CC96']

# Create radar charts for each cluster
for i, cluster_id in enumerate(cluster_centroids['cluster'].unique()):
    df_cluster = cluster_centroids_melted[cluster_centroids_melted['cluster'] == cluster_id]

    fig = px.line_polar(df_cluster, r='Value', theta='Metric', line_close=True,
                        title=f'Radar Chart for Cluster {cluster_id}')

    fig.update_traces(fill='toself', line=dict(color=colors[i]))
    fig.update_layout(
        polar=dict(
            radialaxis=dict(visible=True,  tickfont=dict(size=8)),
            angularaxis=dict(tickfont=dict(size=10))
        ),
        height=400,
        width=800,
        margin=dict(l=20, r=20, b=20, t=40)  # Adjust margins
    )
    fig.show()
```

5.2. Histogram chart

In [ ]:

```
features = customer_data_cleaned.columns[1:-1]
clusters = customer_data_cleaned['cluster'].unique()
clusters.sort()

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

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('')
        axes[i, j].grid(False)

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 - 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 - Unique_Products_Purchased
Cluster 1 - Unique_Products_Purchased
Cluster 2 - Unique_Products_Purchased
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 - Cancel_Rate
Cluster 1 - Cancel_Rate
Cluster 2 - Cancel_Rate
Cluster 0 - Monthly_Average_Spend
Cluster 1 - Monthly_Average_Spend
Cluster 2 - Monthly_Average_Spend
Cluster 0 - Monthly_Std_Spend
Cluster 1 - Monthly_Std_Spend
Cluster 2 - Monthly_Std_Spend
Cluster 0 - Trending_Slope
Cluster 1 - Trending_Slope
Cluster 2 - Trending_Slope

5.3. Customer Profiles:

`Cluster 0: The Big Busy Spenders`

- Mainly live in the UK
- High spending trend over time
- A moderate purchaser, they tend to spend more than average but not so frequent
- Often do late shopping in the day
- The days between purchases & the average purchase value are so high. They seems to purchase a high order value in each transaction to save the time they spent in the store.
- Cancelled orders and the cancellation rate is quite high.

`Cluster 1: The Small Weekend Shoppers`

- Tend to spend less with a lower number of transactions and products purchased.
- Prefer to shop on weekends.
- Spending trend is slowly upwards and quite stable, not much fluctuation over months.
- Low order's cancel rate and not engaged in many cancellations.
- Average purchase value is quite low, they tend to spend less in each transactions.

`Cluster 2: The Grumpy Champions`

- Spend a lot, purchase many products and have a high number of transactions.
- They are so loyal thus they engage frequently but seems grumpy as their cancellations are higher than others.
- Love shopping early in the day and the days between purchases is low.

- Monthly spending variance is high, seems like their spending pattern is quite unpredictable. If we can push their emotion, they will spend more.
- On top of that, their spending trend is very low, which means they might not spending as much as now in the near future.

THE END 🤗