In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os
import re
import scipy as sp
import datetime as dt
from sklearn.linear_model import LinearRegression
from datetime import datetime
from sklearn.preprocessing import StandardScaler
from yellowbrick.cluster import KElbowVisualizer
from sklearn.cluster import KMeans
from sklearn.manifold import TSNE
import itertools
import statsmodels.api as sm
from statsmodels.tsa.seasonal import seasonal_decompose
from scipy.optimize import minimize
from tabulate import tabulate
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
df_sales = pd.read_csv('C:\\Users\\Jonathan Jie\\BCG Capstone\\Capstone Sales Analysis R
```

In [3]:
```
df_sales.head()
```

Out[3]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Qty	Total Base Amt	Wg
0	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	156.0	811.20	380
1	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	156.0	811.20	400
2	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1160.0	(9) SKINLESS CHIX BREAST 345G	156.0	811.20	345
3	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1162.0	(3) CRISPY CHIX S/WEED 400G	312.0	1622.40	400
4	1/7/2022	SO00000002	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	15.6	83.46	400

In [4]:
```
df_sales.shape
```

Out[4]:
```
(74935, 10)
```

In [5]:
```
df_sales.describe
```

Out[5]:
```
<bound method NDFrame.describe of       Transaction Date Sales Order No. Customer Code
\
0              1/7/2022      SO000000001          C033
1              1/7/2022      SO000000001          C033
2              1/7/2022      SO000000001          C033
3              1/7/2022      SO000000001          C033
4              1/7/2022      SO000000002          C001
...                 ...              ...           ...
74930       31/05/2023      SO000030685          C021
74931       31/05/2023      SO000030685          C021
74932       31/05/2023      SO000030685          C021
74933       31/05/2023      SO000030685          C021
74934       31/05/2023      SO000030685          C021

                          Customer Name Customer Category Desc  \
0      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
1      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
2      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
3      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
4             COLD STORAGE SUPERMARKETS            SUPERMARKET
...                                 ...                    ...
74930             CASH - STAFF PURCHASE                 HAWKER
74931             CASH - STAFF PURCHASE                 HAWKER
74932             CASH - STAFF PURCHASE                 HAWKER
74933             CASH - STAFF PURCHASE                 HAWKER
74934             CASH - STAFF PURCHASE                 HAWKER

       Inventory Code                Inventory Desc    Qty  Total Base Amt  \
0              1161.0  (4) HAINANESE CHIX THIGH 380G  156.0         811.200
1              1163.0  (1) CRISPY CHIX ORIGINAL 400G  156.0         811.200
2              1160.0  (9) SKINLESS CHIX BREAST 345G  156.0         811.200
3              1162.0    (3) CRISPY CHIX S/WEED 400G  312.0        1622.400
4              1163.0  (1) CRISPY CHIX ORIGINAL 400G   15.6          83.460
...               ...                            ...    ...             ...
74930          1173.0           YAKITORI WITH SAUCE    1.3          15.210
74931          1190.0     N1 CHICKEN NUGGET (10PKT)    2.6          16.900
74932          1191.0        TEMPURA CHICKEN NUGGET    3.9          20.085
74933          1184.0           CHEESE CHICKEN BALL    5.2          22.360
74934          1360.0  COCONUT WATER WITH MEAT 250ML   26.0          39.000

        Wgt.
0       380g
1       400g
2       345g
3       400g
4       400g
...      ...
74930    1kg
74931    1kg
74932    1kg
74933   500g
74934  250ml

[74935 rows x 10 columns]>
```

In [6]:
```
df_sales.nunique()
```

Out[6]:
```
Transaction Date           294
Sales Order No.          28360
Customer Code              598
Customer Name              598
Customer Category Desc      17
Inventory Code             125
Inventory Desc             125
```

```
Qty                          679
Total Base Amt              2414
Wgt.                          33
dtype: int64
```

In [7]:
```
df_sales.isnull().sum()
```

Out[7]:
```
Transaction Date             0
Sales Order No.              0
Customer Code                0
Customer Name                0
Customer Category Desc       0
Inventory Code             170
Inventory Desc             170
Qty                          0
Total Base Amt               0
Wgt.                      2129
dtype: int64
```

In [8]:
```
(df_sales.isnull().sum()/(len(df_sales)))*100
```

Out[8]:
```
Transaction Date          0.000000
Sales Order No.           0.000000
Customer Code             0.000000
Customer Name             0.000000
Customer Category Desc    0.000000
Inventory Code            0.226863
Inventory Desc            0.226863
Qty                       0.000000
Total Base Amt            0.000000
Wgt.                      2.841129
dtype: float64
```

In [9]:
```
df_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 74935 entries, 0 to 74934
Data columns (total 10 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Transaction Date        74935 non-null  object
 1   Sales Order No.         74935 non-null  object
 2   Customer Code           74935 non-null  object
 3   Customer Name           74935 non-null  object
 4   Customer Category Desc  74935 non-null  object
 5   Inventory Code          74765 non-null  float64
 6   Inventory Desc          74765 non-null  object
 7   Qty                     74935 non-null  float64
 8   Total Base Amt          74935 non-null  float64
 9   Wgt.                    72806 non-null  object
dtypes: float64(3), object(7)
memory usage: 5.7+ MB
```

In [10]:
```
# Replace blank Inventory Code with "1001"
df_sales['Inventory Code'].replace({np.nan: 1001.0, '': 1001.0}, inplace=True)

# Replace blank and NaN Inventory Desc with "Containers" or "Stocks"
df_sales['Inventory Desc'] = df_sales['Inventory Desc'].apply(lambda x: 'Containers or S

# Filter rows where Inventory Code is "1001" (as a float or as a string)
filtered_df = df_sales[(df_sales['Inventory Code'] == 1001.0) | (df_sales['Inventory Cod

# Print the filtered DataFrame
print(filtered_df[['Inventory Code', 'Inventory Desc']])
```

```
## Rounding base amount to 2 decimal points
df_sales['Total Base Amt'] = df_sales['Total Base Amt'].astype(float).round(2)
```

```
       Inventory Code        Inventory Desc
1868           1001.0  Containers or Stocks
2274           1001.0  Containers or Stocks
2301           1001.0  Containers or Stocks
3223           1001.0  Containers or Stocks
3296           1001.0  Containers or Stocks
...               ...                   ...
73630          1001.0  Containers or Stocks
73631          1001.0  Containers or Stocks
73632          1001.0  Containers or Stocks
73710          1001.0  Containers or Stocks
74150          1001.0  Containers or Stocks

[170 rows x 2 columns]
```

In [11]:
```
df_sales['Total Base Amt']
```

Out[11]:
```
0         811.20
1         811.20
2         811.20
3        1622.40
4          83.46
          ...
74930      15.21
74931      16.90
74932      20.08
74933      22.36
74934      39.00
Name: Total Base Amt, Length: 74935, dtype: float64
```

In [12]:
```
## Inserting Weights for CHICKEN BONELESS LEG (2KGX6PACK), HONEY CHAR SIEW 5KG , TAKOYAK

bonelessleg = df_sales['Inventory Desc'] == 'CHICKEN BONELESS LEG (2KGX6PACK)'
df_sales.loc[bonelessleg, 'Wgt.'] = '12000g'

honeycharsiew = df_sales['Inventory Desc'] == 'HONEY CHAR SIEW 5KG'
df_sales.loc[honeycharsiew, 'Wgt.'] = '5000g'

takoyaki = df_sales['Inventory Desc'] == 'TAKOYAKI 1KG'
df_sales.loc[takoyaki,'Wgt.'] = '1000g'
```

In [13]:
```
### Standardizing weight to grams.

conversion_factors = {
    'g': 1,
    'kg': 1000,     # 1 kg = 1000 g
    'ml': 1,        # Assuming ml is equivalent to grams (this may not always be true)
    'litre': 1000  # 1 liter = 1000 g
}

def convert_to_grams(weight_str):
    if isinstance(weight_str, str):
        value_str = ""
        unit_str = ""

        for char in weight_str:
            if char.isnumeric() or char == '.':
                value_str += char
            else:
                unit_str += char

        try:
            value = float(value_str)
```

```
        except ValueError:
            return None

        unit = unit_str.strip()

        return value * conversion_factors.get(unit, 1)
    else:
        return None
```

In [14]:
```
##Applying the above function

df_sales['weight_in_grams'] = df_sales['Wgt.'].apply(convert_to_grams)
```

In [15]:
```
##Checking if applied correctly

df_sales
```

Out[15]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Qty	Total Base Amt
0	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	156.0	811.20
1	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	156.0	811.20
2	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1160.0	(9) SKINLESS CHIX BREAST 345G	156.0	811.20
3	1/7/2022	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1162.0	(3) CRISPY CHIX S/WEED 400G	312.0	1622.40
4	1/7/2022	SO00000002	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	15.6	83.46
...	...	...	...	...	...	...	...	...	...
74930	31/05/2023	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1173.0	YAKITORI WITH SAUCE	1.3	15.21
74931	31/05/2023	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1190.0	N1 CHICKEN NUGGET (10PKT)	2.6	16.90
74932	31/05/2023	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1191.0	TEMPURA CHICKEN NUGGET	3.9	20.08
74933	31/05/2023	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1184.0	CHEESE CHICKEN BALL	5.2	22.36

74934	31/05/2023	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1360.0	COCONUT WATER WITH MEAT 250ML	26.0	39.00

74935 rows × 11 columns

In [16]:
```
def categorize_inventory(description):
    # Check if the description contains 'Containers or Stocks'
    if 'Containers or Stocks' in description:
        return ''

    # Check if the description starts with 'SP' specifically
    if description.startswith('SP'):
        return 'Raw'

    # Check if the description contains 'Betagro'
    if 'BETAGRO' in description:
        return 'Ready to Eat (RTE)'

    # If not labeled as 'SP', 'Betagro', or 'Containers or Stocks', consider it 'Ready t
    return 'Ready to Cook (RTC)'
```

In [17]:
```
df_sales['Inventory Category'] = df_sales['Inventory Desc'].apply(categorize_inventory)
```

In [18]:
```
## Rearranging the columns

col = df_sales.pop('Inventory Category')

df_sales.insert(7, col.name, col)
```

In [19]:
```
## Creating price per unit column for Price Elasticity and rounding the results

df_sales['Price Per Unit'] = df_sales['Total Base Amt'] / df_sales['Qty']

df_sales['Price Per Unit'] = df_sales['Price Per Unit'].astype(float).round(2)
```

In [20]:
```
## Create RTC dataframe for Mean Median Mode
RTC_DF = df_sales[df_sales['Inventory Category'] == 'Ready to Cook (RTC)']
```

In [21]:
```
RTC_weight_mean = RTC_DF['weight_in_grams'].mean()
RTC_median_weight = RTC_DF['weight_in_grams'].median()
RTC_mode_weight = RTC_DF['weight_in_grams'].mode().iloc[0]

print("Ready To Cook Mean Weight in G:", RTC_weight_mean)
print("Ready To Cook Median Weight in G:", RTC_median_weight)
print("Ready To Cook Mode Weight in G:", RTC_mode_weight)
```

```
Ready To Cook Mean Weight in G: 1020.2275705909698
Ready To Cook Median Weight in G: 1000.0
Ready To Cook Mode Weight in G: 1000.0
```

In [22]:
```
## Replacing NA for those in RTC and RTC_median_weight

df_sales['Wgt.'] = df_sales.apply(lambda row: RTC_median_weight if row['Inventory Catego
```

In [23]:
```
df_sales['Wgt.'] = df_sales['Wgt.'].replace('-', RTC_median_weight).astype(str)
```

In [24]:
```
### Standardizing weight to grams.

conversion_factors = {
    'g': 1,
```

```
    'kg': 1000,     # 1 kg = 1000 g
    'ml': 1,        # Assuming ml is equivalent to grams (this may not always be true)
    'litre': 1000   # 1 liter = 1000 g
}

def convert_to_grams(weight_str):
    if isinstance(weight_str, str):
        value_str = ""
        unit_str = ""

        for char in weight_str:
            if char.isnumeric() or char == '.':
                value_str += char
            else:
                unit_str += char

        try:
            value = float(value_str)
        except ValueError:
            return None

        unit = unit_str.strip()

        conversion_factor = conversion_factors.get(unit, 1)

        return value * conversion_factor
    else:
        return None
```

In [25]:
```
##Applying the above function

df_sales['weight_in_grams'] = df_sales['Wgt.'].apply(convert_to_grams)
```

In [26]:
```
df_sales.isnull().sum()
```

Out[26]:
```
Transaction Date            0
Sales Order No.             0
Customer Code               0
Customer Name               0
Customer Category Desc      0
Inventory Code              0
Inventory Desc              0
Inventory Category          0
Qty                         0
Total Base Amt              0
Wgt.                        0
weight_in_grams           170
Price Per Unit              0
dtype: int64
```

In [27]:
```
df_sales['Transaction Date']
```

Out[27]:
```
0          1/7/2022
1          1/7/2022
2          1/7/2022
3          1/7/2022
4          1/7/2022
             ...
74930    31/05/2023
74931    31/05/2023
74932    31/05/2023
74933    31/05/2023
74934    31/05/2023
Name: Transaction Date, Length: 74935, dtype: object
```

In [28]:
```
df_sales['Transaction Date'] = pd.to_datetime(df_sales['Transaction Date'], format="%d/%
```

In [29]:
```
df_sales
```

Out[29]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	Q
0	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	Ready to Cook (RTC)	156
1	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	156
2	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1160.0	(9) SKINLESS CHIX BREAST 345G	Ready to Cook (RTC)	156
3	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1162.0	(3) CRISPY CHIX S/WEED 400G	Ready to Cook (RTC)	312
4	2022-07-01	SO00000002	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	15
...	...	...	...	...	...	...	...	...	
74930	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1173.0	YAKITORI WITH SAUCE	Ready to Cook (RTC)	1
74931	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1190.0	N1 CHICKEN NUGGET (10PKT)	Ready to Cook (RTC)	2
74932	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1191.0	TEMPURA CHICKEN NUGGET	Ready to Cook (RTC)	3
74933	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1184.0	CHEESE CHICKEN BALL	Ready to Cook (RTC)	5
74934	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1360.0	COCONUT WATER WITH MEAT 250ML	Ready to Cook (RTC)	26

74935 rows × 13 columns

In [30]:
```
df_sales.shape
```

Out[30]:
```
(74935, 13)
```

In [31]:
```
df_sales.describe
```

Out[31]:
```
<bound method NDFrame.describe of       Transaction Date Sales Order No. Customer Code
\
0           2022-07-01      SO000000001          C033
1           2022-07-01      SO000000001          C033
2           2022-07-01      SO000000001          C033
3           2022-07-01      SO000000001          C033
4           2022-07-01      SO000000002          C001
...                ...              ...           ...
74930       2023-05-31      SO000030685          C021
74931       2023-05-31      SO000030685          C021
74932       2023-05-31      SO000030685          C021
74933       2023-05-31      SO000030685          C021
74934       2023-05-31      SO000030685          C021

                         Customer Name Customer Category Desc  \
0      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
1      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
2      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
3      CMM MARKETING MANAGEMENT PTE LTD            SUPERMARKET
4             COLD STORAGE SUPERMARKETS            SUPERMARKET
...                                 ...                    ...
74930             CASH - STAFF PURCHASE                 HAWKER
74931             CASH - STAFF PURCHASE                 HAWKER
74932             CASH - STAFF PURCHASE                 HAWKER
74933             CASH - STAFF PURCHASE                 HAWKER
74934             CASH - STAFF PURCHASE                 HAWKER

       Inventory Code                 Inventory Desc   Inventory Category  \
0              1161.0  (4) HAINANESE CHIX THIGH 380G  Ready to Cook (RTC)
1              1163.0  (1) CRISPY CHIX ORIGINAL 400G  Ready to Cook (RTC)
2              1160.0  (9) SKINLESS CHIX BREAST 345G  Ready to Cook (RTC)
3              1162.0    (3) CRISPY CHIX S/WEED 400G  Ready to Cook (RTC)
4              1163.0  (1) CRISPY CHIX ORIGINAL 400G  Ready to Cook (RTC)
...               ...                            ...                   ...
74930          1173.0            YAKITORI WITH SAUCE  Ready to Cook (RTC)
74931          1190.0      N1 CHICKEN NUGGET (10PKT)  Ready to Cook (RTC)
74932          1191.0         TEMPURA CHICKEN NUGGET  Ready to Cook (RTC)
74933          1184.0            CHEESE CHICKEN BALL  Ready to Cook (RTC)
74934          1360.0  COCONUT WATER WITH MEAT 250ML  Ready to Cook (RTC)

         Qty  Total Base Amt   Wgt.  weight_in_grams  Price Per Unit
0      156.0          811.20   380g            380.0            5.20
1      156.0          811.20   400g            400.0            5.20
2      156.0          811.20   345g            345.0            5.20
3      312.0         1622.40   400g            400.0            5.20
4       15.6           83.46   400g            400.0            5.35
...      ...             ...    ...              ...             ...
74930    1.3           15.21    1kg           1000.0           11.70
74931    2.6           16.90    1kg           1000.0            6.50
74932    3.9           20.08    1kg           1000.0            5.15
74933    5.2           22.36   500g            500.0            4.30
74934   26.0           39.00  250ml            250.0            1.50

[74935 rows x 13 columns]>
```

In [32]:
```
df_sales.nunique()
```

Out[32]:
```
Transaction Date            294
Sales Order No.           28360
Customer Code               598
Customer Name               598
Customer Category Desc       17
Inventory Code              126
Inventory Desc              126
Inventory Category            4
```

```
Qty                          679
Total Base Amt              2414
Wgt.                          37
weight_in_grams               33
Price Per Unit               216
dtype: int64
```

In [33]:
```
df_sales.isnull().sum()
```

Out[33]:
```
Transaction Date             0
Sales Order No.              0
Customer Code                0
Customer Name                0
Customer Category Desc       0
Inventory Code               0
Inventory Desc               0
Inventory Category           0
Qty                          0
Total Base Amt               0
Wgt.                         0
weight_in_grams            170
Price Per Unit               0
dtype: int64
```

In [34]:
```
(df_sales.isnull().sum()/(len(df_sales)))*100
```

Out[34]:
```
Transaction Date           0.000000
Sales Order No.            0.000000
Customer Code              0.000000
Customer Name              0.000000
Customer Category Desc     0.000000
Inventory Code             0.000000
Inventory Desc             0.000000
Inventory Category         0.000000
Qty                        0.000000
Total Base Amt             0.000000
Wgt.                       0.000000
weight_in_grams            0.226863
Price Per Unit             0.000000
dtype: float64
```

In [35]:
```
df_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 74935 entries, 0 to 74934
Data columns (total 13 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Transaction Date        74935 non-null  datetime64[ns]
 1   Sales Order No.         74935 non-null  object
 2   Customer Code           74935 non-null  object
 3   Customer Name           74935 non-null  object
 4   Customer Category Desc  74935 non-null  object
 5   Inventory Code          74935 non-null  float64
 6   Inventory Desc          74935 non-null  object
 7   Inventory Category      74935 non-null  object
 8   Qty                     74935 non-null  float64
 9   Total Base Amt          74935 non-null  float64
 10  Wgt.                    74935 non-null  object
 11  weight_in_grams         74765 non-null  float64
 12  Price Per Unit          74935 non-null  float64
dtypes: datetime64[ns](1), float64(5), object(7)
memory usage: 7.4+ MB
```

In [36]:
```
total_base_amt_sum = df_sales['Total Base Amt'].sum()
print(total_base_amt_sum)
```

39614447.03

In [37]:
```
# Define the list of specific 'Inventory Desc' values you want to filter
specific_inventory_desc = ['HONEY CHAR SIEW 5KG', 'TAKOYAKI 1KG', 'BW60 BREADED COD FISH

# Filter the DataFrame to show only the rows with these specific 'Inventory Desc' values
filtered_df_idesc = df_sales[df_sales['Inventory Desc'].isin(specific_inventory_desc)]

# Print the result
print(filtered_df_idesc)
```

```
      Transaction Date Sales Order No. Customer Code  \
241         2022-07-01      SO00000073          A001
242         2022-07-01      SO00000074          A022
406         2022-07-02      SO00000143          A001
499         2022-07-02      SO00000184          A001
518         2022-07-02      SO00000192          T029
...                ...             ...           ...
74774       2023-05-31      SO00030615          O013
74815       2023-05-31      SO00030634          A001
74820       2023-05-31      SO00030635          A001
74835       2023-05-31      SO00030643          C058
74891       2023-05-30      SO00030674          C005

                                    Customer Name Customer Category Desc  \
241                          ADVANCE FOOD SYSTEMS                 RETAIL
242                    ALICE - CANBERRA SEC SCHOOL                 SCHOOL
406                          ADVANCE FOOD SYSTEMS                 RETAIL
499                          ADVANCE FOOD SYSTEMS                 RETAIL
518                              THE FRENCH LADLE                   CAFE
...                                           ...                   ...
74774  ONE SMS.PTE.LTD C/O SING MY SONG FAMILY KTV                   CAFE
74815                          ADVANCE FOOD SYSTEMS                 RETAIL
74820                          ADVANCE FOOD SYSTEMS                 RETAIL
74835                                 COMMON FOLKS                   CAFE
74891                               CASH - MAY A/C                 SCHOOL

       Inventory Code             Inventory Desc   Inventory Category   Qty  \
241            1218.0        HONEY CHAR SIEW 5KG  Ready to Cook (RTC)  32.5
242            1605.0  BW60 BREADED COD FISH 60G  Ready to Cook (RTC)   3.9
406            1218.0        HONEY CHAR SIEW 5KG  Ready to Cook (RTC)  26.0
499            1218.0        HONEY CHAR SIEW 5KG  Ready to Cook (RTC)  13.0
518            1605.0  BW60 BREADED COD FISH 60G  Ready to Cook (RTC)  26.0
...               ...                        ...                  ...   ...
74774          1418.0               TAKOYAKI 1KG  Ready to Cook (RTC)   6.5
74815          1218.0        HONEY CHAR SIEW 5KG  Ready to Cook (RTC)   6.5
74820          1218.0        HONEY CHAR SIEW 5KG  Ready to Cook (RTC)  13.0
74835          1605.0  BW60 BREADED COD FISH 60G  Ready to Cook (RTC)  26.0
74891          1605.0  BW60 BREADED COD FISH 60G  Ready to Cook (RTC)   1.3

       Total Base Amt    Wgt.  weight_in_grams  Price Per Unit
241            351.00   5000g           5000.0            10.8
242             13.65  1000.0           1000.0             3.5
406            280.80   5000g           5000.0            10.8
499            140.40   5000g           5000.0            10.8
518             91.00  1000.0           1000.0             3.5
...               ...     ...              ...             ...
74774           55.25   1000g           1000.0             8.5
74815           70.20   5000g           5000.0            10.8
74820          140.40   5000g           5000.0            10.8
74835           91.00  1000.0           1000.0             3.5
74891            4.55  1000.0           1000.0             3.5

[747 rows x 13 columns]
```

In [38]:
```
df_sales.describe()
```

Out[38]:

	Transaction Date	Inventory Code	Qty	Total Base Amt	weight_in_grams	Price Per Unit
count	74935	74935.000000	74935.000000	74935.000000	74765.000000	74935.000000
mean	2022-12-14 13:22:41.932340992	1198.218976	92.102463	528.650791	959.617883	9.445082
min	2022-07-01 00:00:00	1001.000000	1.300000	1.820000	80.000000	0.050000
25%	2022-09-20 00:00:00	1142.000000	7.800000	66.300000	400.000000	5.400000
50%	2022-12-14 00:00:00	1158.000000	15.600000	91.260000	1000.000000	8.500000
75%	2023-03-10 00:00:00	1182.000000	26.000000	180.180000	1000.000000	10.900000
max	2023-05-31 00:00:00	1709.000000	31200.000000	386100.000000	12000.000000	607.200000
std	NaN	141.723191	400.697054	5145.362379	1038.241554	12.487056

In [39]:
```
df_sales.nunique()
```

Out[39]:
```
Transaction Date            294
Sales Order No.           28360
Customer Code               598
Customer Name               598
Customer Category Desc       17
Inventory Code              126
Inventory Desc              126
Inventory Category            4
Qty                         679
Total Base Amt             2414
Wgt.                         37
weight_in_grams              33
Price Per Unit              216
dtype: int64
```

In [40]:
```
df_sales_RFM = df_sales
```

In [41]:
```
df_sales_RFM.shape
```

Out[41]:
```
(74935, 13)
```

In [42]:
```
df_sales_RFM.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 74935 entries, 0 to 74934
Data columns (total 13 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Transaction Date        74935 non-null  datetime64[ns]
 1   Sales Order No.         74935 non-null  object
 2   Customer Code           74935 non-null  object
 3   Customer Name           74935 non-null  object
 4   Customer Category Desc  74935 non-null  object
 5   Inventory Code          74935 non-null  float64
 6   Inventory Desc          74935 non-null  object
 7   Inventory Category      74935 non-null  object
 8   Qty                     74935 non-null  float64
 9   Total Base Amt          74935 non-null  float64
 10  Wgt.                    74935 non-null  object
 11  weight_in_grams         74765 non-null  float64
 12  Price Per Unit          74935 non-null  float64
dtypes: datetime64[ns](1), float64(5), object(7)
memory usage: 7.4+ MB
```

In [43]:
```
#  Create Recency Frequency Monetary (RFM) table

print('Start date:' , df_sales_RFM['Transaction Date'].min())
print('End date:' , df_sales_RFM['Transaction Date'].max())
```

```
Start date: 2022-07-01 00:00:00
End date: 2023-05-31 00:00:00
```

In [44]:
```
# Convert to show date only
df_sales_RFM['Transaction Date'] = df_sales_RFM['Transaction Date'].dt.date
```

In [45]:
```
# The idea is to use this snapshot date as a reference point for measuring how recently
# By adding one day to the maximum date, it's ensuring that all customers are at least o
# being zero for customers who made a purchase on the maximum date.
snapshot_date = max(df_sales_RFM['Transaction Date'])+ dt.timedelta(days=1)
print("Snapshot Date", snapshot_date)
```

```
Snapshot Date 2023-06-01
```

In [46]:
```
# RFM table
# Aggregate data by each customer
rfm = df_sales_RFM.groupby('Customer Code').agg({'Transaction Date': lambda x: (snapshot
rfm['Transaction Date'] = rfm['Transaction Date'].astype(int)

# Rename columns
rfm.rename(columns={'Transaction Date': 'Recency',
                    'Sales Order No.': 'Frequency',
                    'Total Base Amt': 'Monetary'}, inplace=True)
rfm.head(10)
```

Out[46]:

	Customer Code	Recency	Frequency	Monetary
0	A001	1	3082	316950.01
1	A002	14	19	2536.04
2	A003	15	120	7482.41
3	A004	61	113	4414.15
4	A005	3	2803	220783.16
5	A007	1	989	681102.50
6	A008	14	56	5811.26
7	A009	7	34	7098.52
8	A010	111	24	109226.00
9	A011	1	1450	357023.68

In [47]:
```
rfm.shape
```

Out[47]:
```
(598, 4)
```

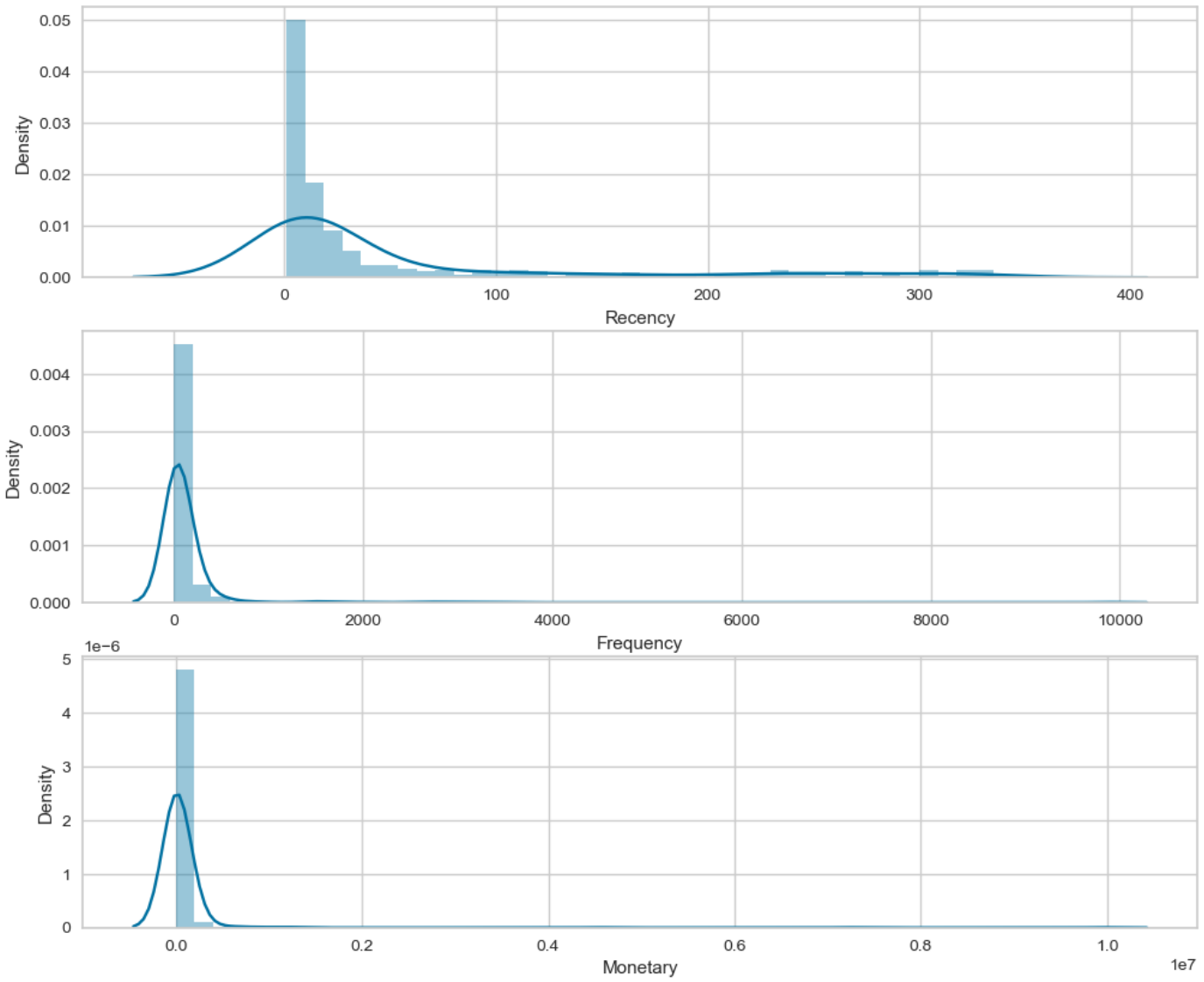
In [48]:
```
df_sales_RFM.shape
```

Out[48]:
```
(74935, 13)
```

In [49]:
```
# Right now, the dataset consists of recency, frequency, and monetary value column.
# But we cannot use the dataset yet because we have to preprocess the data more.

# Manage Skewness and Scaling
# We have to make sure that the data meet these assumptions:
```

```
# The data should meet assumptions where the variables are not skewed and have the same
# Because of that, we have to manage the skewness of the variables. Here are the visuali

# Explore the data for Recency, Frequency,MonetaryValue distributions
```

In [50]:
```
plt.figure(figsize=(12,10))

# Plot recency distribution
plt.subplot(3, 1, 1); sns.distplot(rfm['Recency'])

# Plot frequency distribution
plt.subplot(3, 1, 2); sns.distplot(rfm['Frequency'])

# Plot monetary value distribution
plt.subplot(3, 1, 3); sns.distplot(rfm['Monetary'])

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

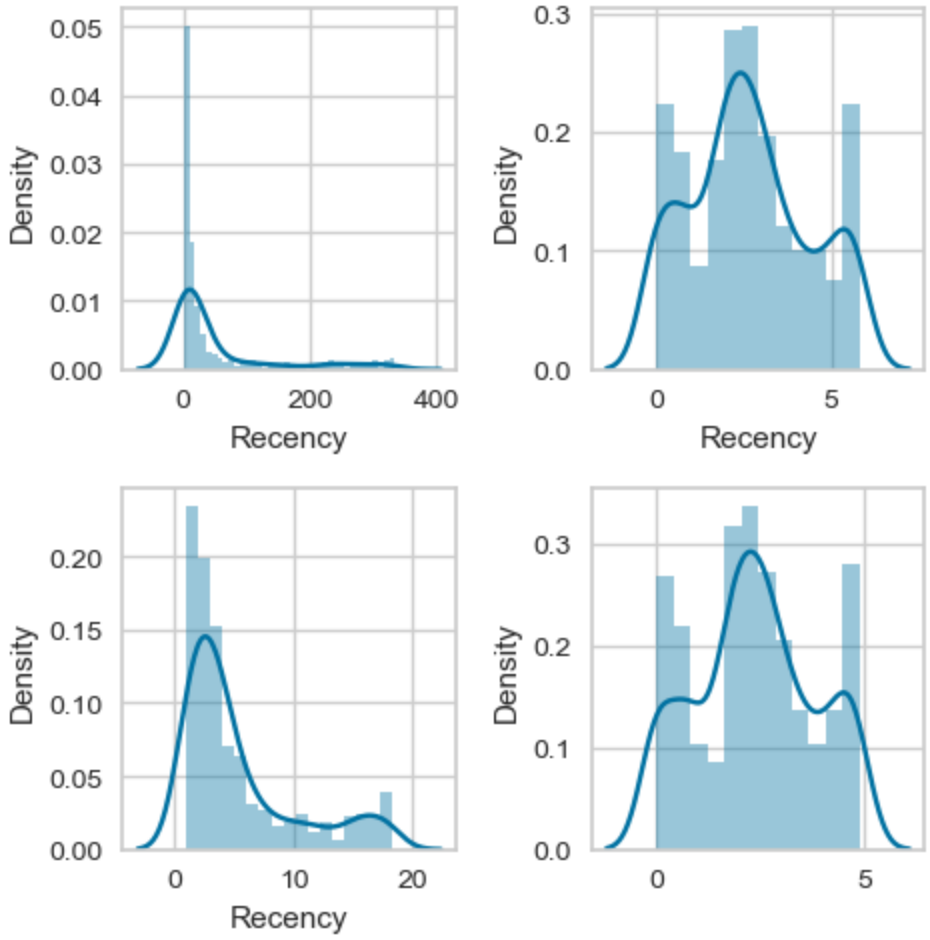


In [51]:
```
#As we can see from above, we have to transform the data, so it has a more symmetrical f
#There are some methods that we can use to manage the skewness:

# log transformation
# square root transformation
# box-cox transformation Note: We can use the transformation if and only if the variable
```

In [52]:
```
from scipy import stats
def analyze_skewness(x):
```

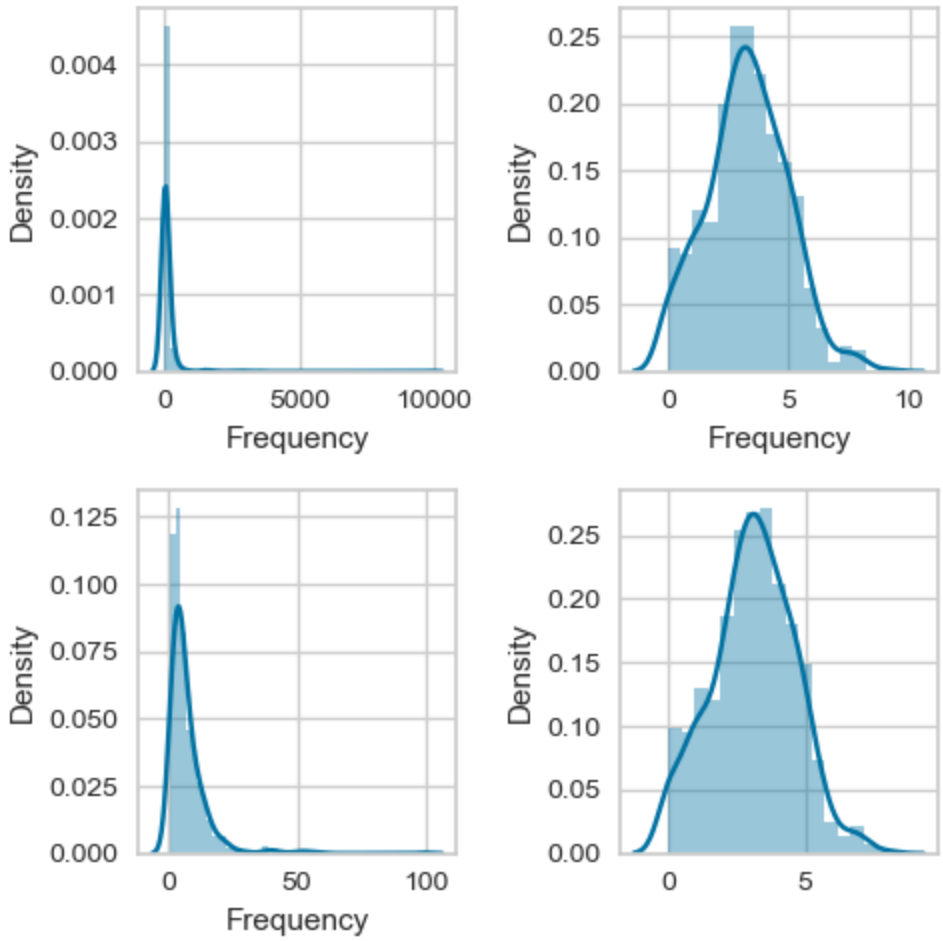
```
    fig, ax = plt.subplots(2, 2, figsize=(5,5))
    sns.distplot(rfm[x], ax=ax[0,0])
    sns.distplot(np.log(rfm[x]), ax=ax[0,1])
    sns.distplot(np.sqrt(rfm[x]), ax=ax[1,0])
    sns.distplot(stats.boxcox(rfm[x])[0], ax=ax[1,1])
    plt.tight_layout()
    plt.show()


    print('Log Transform : The skew coefficient of', rfm[x].skew().round(2), 'to', np.lo
    print('Square Root Transform : The skew coefficient of', rfm[x].skew().round(2), 'to
    print('Box-Cox Transform : The skew coefficient of', rfm[x].skew().round(2), 'to', p
```

In [53]:
```
analyze_skewness('Recency')
```



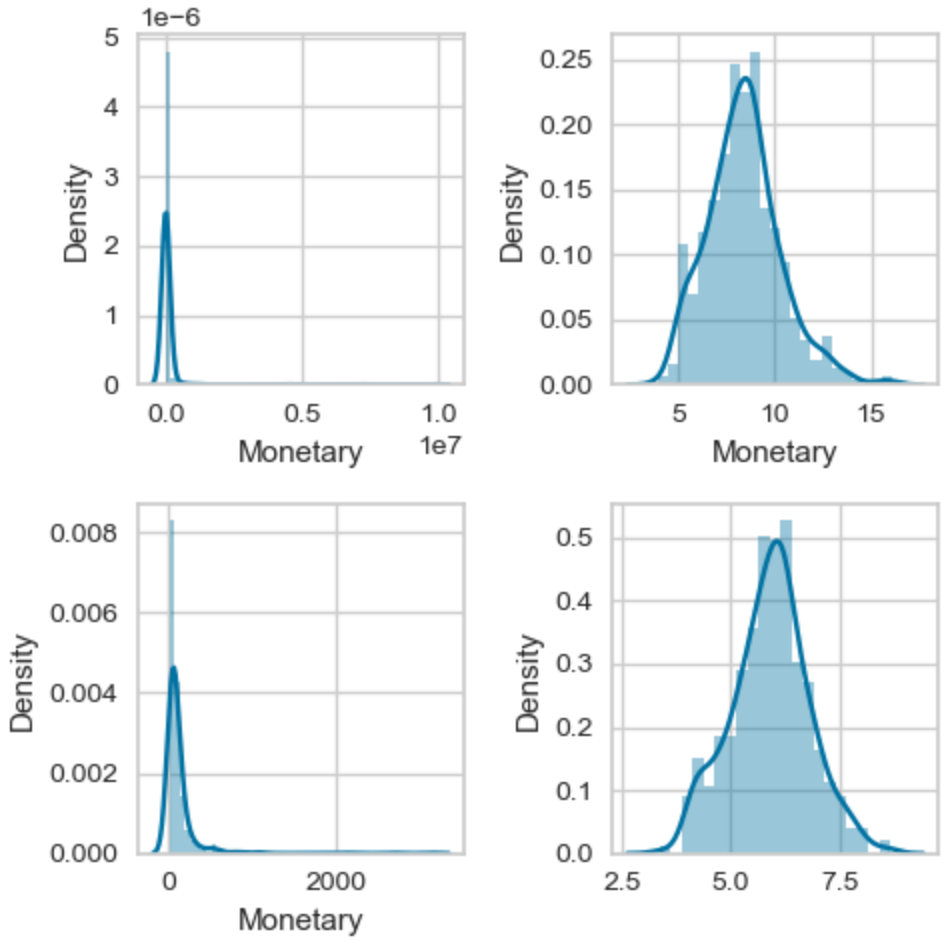

```
Log Transform : The skew coefficient of 2.01 to 0.21
Square Root Transform : The skew coefficient of 2.01 to 1.38
Box-Cox Transform : The skew coefficient of 2.01 to 0.03
```

In [54]:
```
analyze_skewness('Frequency')
```




```
Log Transform : The skew coefficient of 13.12 to 0.14
Square Root Transform : The skew coefficient of 13.12 to 4.57
Box-Cox Transform : The skew coefficient of 13.12 to 0.0
```

In [55]:
```
analyze_skewness('Monetary')
```



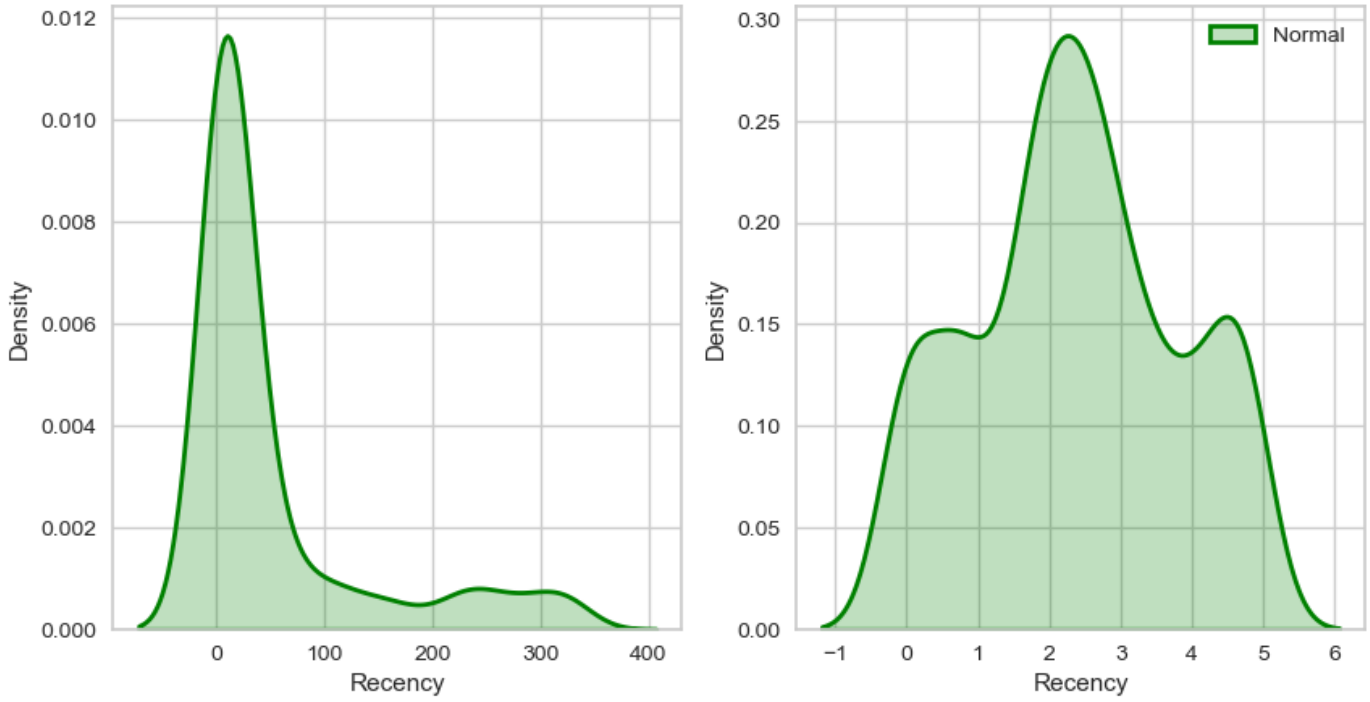

```
Log Transform : The skew coefficient of 14.84 to 0.56
Square Root Transform : The skew coefficient of 14.84 to 8.24
Box-Cox Transform : The skew coefficient of 14.84 to -0.0
```

In [56]:
```
# Based on the results you've provided, it appears that the Box-Cox transform consistent
# values for all three variables (Recency, Frequency, and MonetaryValue). The skew coeff
# are close to 0, which is desirable.

# So, for correcting skewness in data, the Box-Cox transformation is likely the most app
```

In [57]:
```
# Initialize fitted_data and fitted_lambda
fitted_data = pd.DataFrame()
fitted_lambda = {}
# transform training data & save lambda value
fitted_data['Recency'], fitted_lambda['Recency'] = stats.boxcox(rfm['Recency'])

# creating axes to draw plots
fig, ax = plt.subplots(1, 2)

# plotting the original data(non-normal) and
# fitted data (normal)
sns.distplot(rfm['Recency'], hist = False, kde = True,
             kde_kws = {'shade': True, 'linewidth': 2},
             label = "Non-Normal", color ="green", ax = ax[0])

sns.distplot(fitted_data['Recency'], hist = False, kde = True,
             kde_kws = {'shade': True, 'linewidth': 2},
             label = "Normal", color ="green", ax = ax[1])

# adding legends to the subplots
plt.legend(loc = "upper right")

# rescaling the subplots
fig.set_figheight(5)
fig.set_figwidth(10)
```

```
print(f"Lambda value used for Transformation: {fitted_lambda['Recency']}")
```

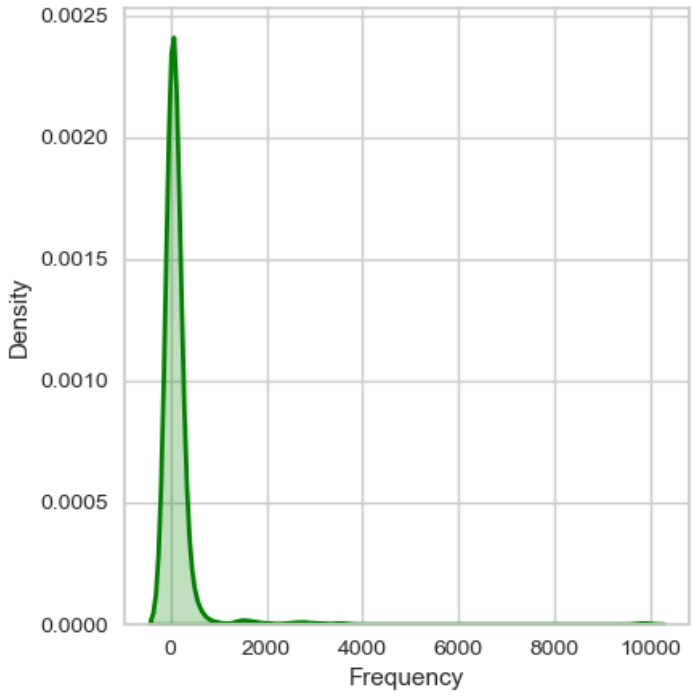
```
Lambda value used for Transformation: -0.06256522187993799
```

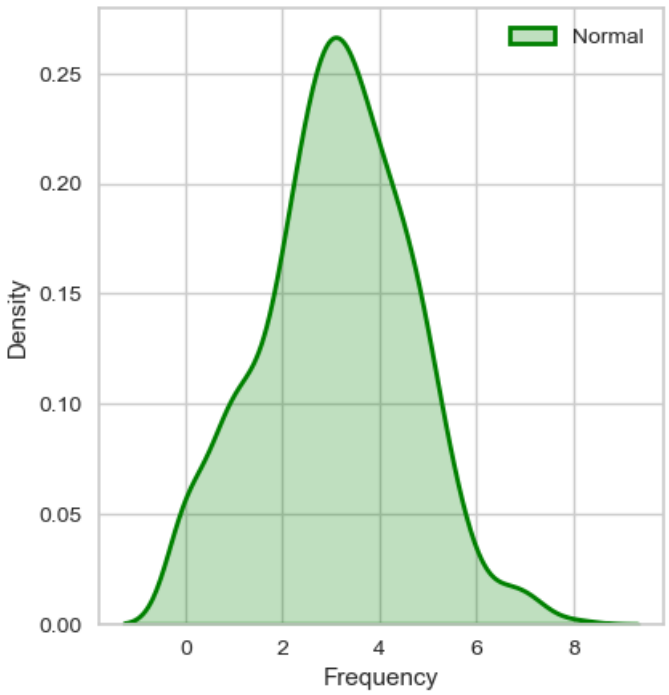


In [58]:

```
# transform training data & save lambda value
fitted_data['Frequency'], fitted_lambda['Frequency'] = stats.boxcox(rfm['Frequency'])

# creating axes to draw plots
fig, ax = plt.subplots(1, 2)

# plotting the original data(non-normal) and
# fitted data (normal)
sns.distplot(rfm['Frequency'], hist = False, kde = True,
             kde_kws = {'shade': True, 'linewidth': 2},
             label = "Non-Normal", color ="green", ax = ax[0])

sns.distplot(fitted_data['Frequency'], hist = False, kde = True,
             kde_kws = {'shade': True, 'linewidth': 2},
             label = "Normal", color ="green", ax = ax[1])

# adding legends to the subplots
plt.legend(loc = "upper right")

# rescaling the subplots
fig.set_figheight(5)
fig.set_figwidth(10)

print(f"Lambda value used for Transformation: {fitted_lambda['Frequency']}")
```

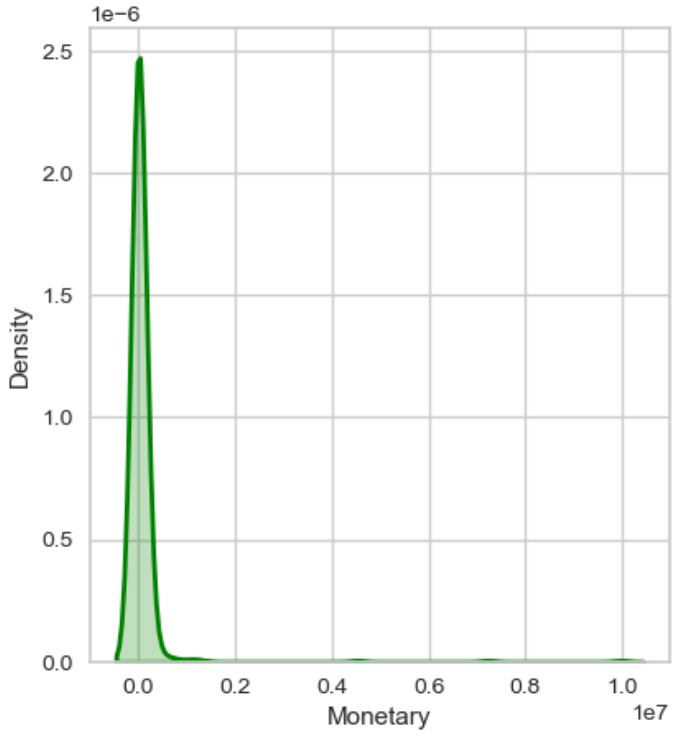
```
Lambda value used for Transformation: -0.029515193030542657
```

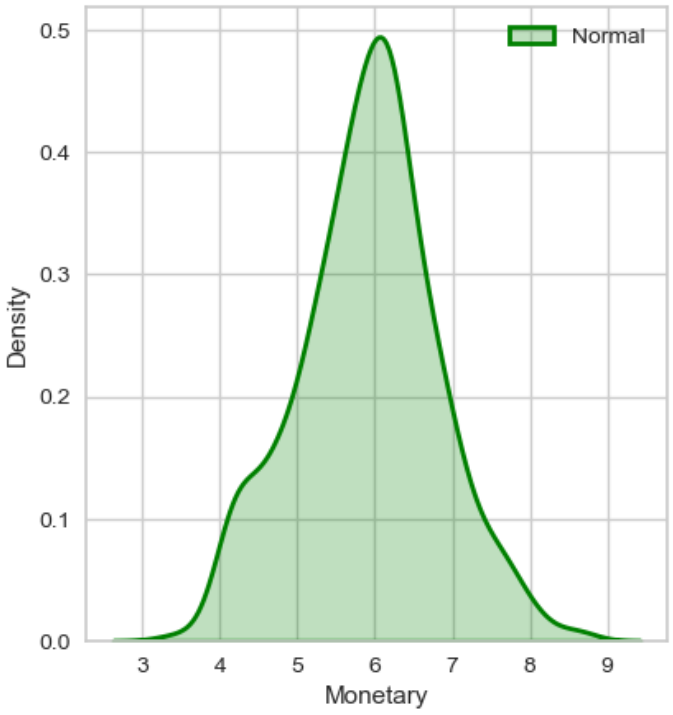




In [59]:
```
# transform training data & save lambda value
fitted_data['Monetary'], fitted_lambda['Monetary'] = stats.boxcox(rfm['Monetary'])

# creating axes to draw plots
fig, ax = plt.subplots(1, 2)

# plotting the original data(non-normal) and
# fitted data (normal)
sns.distplot(rfm['Monetary'], hist = False, kde = True,
             kde_kws = {'shade': True, 'linewidth': 2},
             label = "Non-Normal", color ="green", ax = ax[0])

sns.distplot(fitted_data['Monetary'], hist = False, kde = True,
             kde_kws = {'shade': True, 'linewidth': 2},
             label = "Normal", color ="green", ax = ax[1])

# adding legends to the subplots
plt.legend(loc = "upper right")

# rescaling the subplots
fig.set_figheight(5)
fig.set_figwidth(10)

print(f"Lambda value used for Transformation: {fitted_lambda['Monetary']}")
```

```
Lambda value used for Transformation: -0.08670970030224771
```





In [60]:
```
fitted_data
```

Out[60]:

	Recency	Frequency	Monetary
0	0.000000	7.151973	7.687321
1	2.432698	2.820122	5.688079
2	2.491064	4.464631	6.211465
3	3.624772	4.412401	5.962305
4	1.061706	7.077010	7.564854
...	...	...	...
593	0.678332	4.323189	6.173121
594	4.143174	2.602652	5.416415
595	0.000000	3.954253	6.356148
596	2.978229	3.140606	5.788247
597	3.007786	2.226085	5.266423

598 rows × 3 columns

In [61]:
```
rfm_normalized = fitted_data
```

In [62]:
```
rfm_normalized.describe()
```

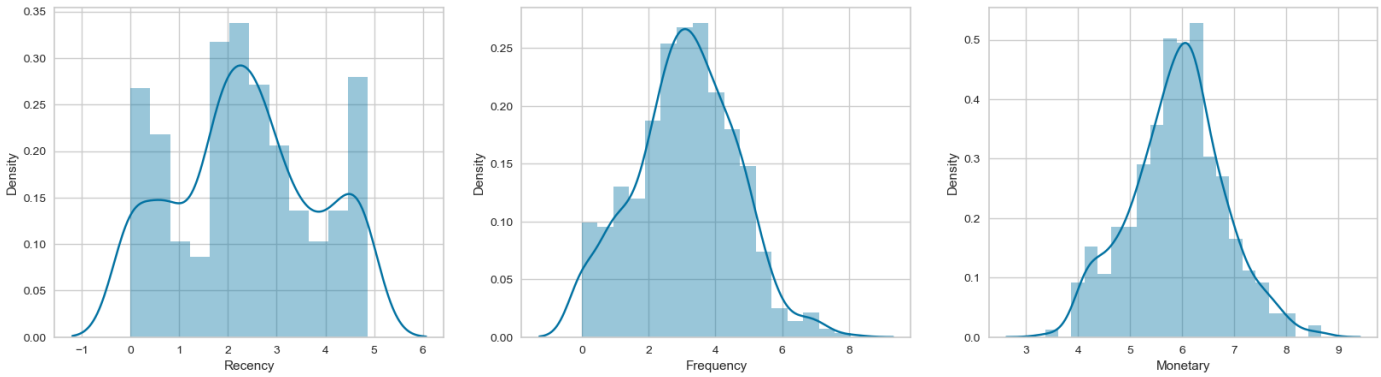
Out[62]:

	Recency	Frequency	Monetary
count	598.000000	598.000000	598.000000
mean	2.366976	3.100621	5.906490
std	1.427565	1.516859	0.891491
min	0.000000	0.000000	3.362994

25%	1.531060	2.127493	5.328984
50%	2.369723	3.140606	5.962760
75%	3.351420	4.161107	6.441942
max	4.874012	8.053448	8.681258

In [63]:
```
#Plot data distribution after boxcox transformation
plt.figure(figsize=(20,5))

plt.subplot(1,3,1)
sns.distplot(rfm_normalized['Recency'])

plt.subplot(1,3,2)
sns.distplot(rfm_normalized['Frequency'])

plt.subplot(1,3,3)
sns.distplot(rfm_normalized['Monetary'])
```

Out[63]:
```
<Axes: xlabel='Monetary', ylabel='Density'>
```



In [64]:
```
# Each variable don't have the same mean and variance. We have to normalize it.
```

In [65]:
```
# Normalization is the process of scaling features to have a mean of 0 and a standard de
# It can be useful in some machine learning algorithms, especially those that are sensit
# features, such as K-Means clustering planning to apply in this anlaysis

# When planning to use K-Means clustering, it's a good practice to normalize your data,
# scale of the features. Normalization ensures that each feature contributes equally to
# In K-Means, the distance between data points is a critical factor in forming clusters,
# different scales, it can lead to suboptimal clustering results.

# Therefore, for K-Means clustering,recommended normalization of data using the Standard
```

In [66]:
```
# Initialize the Object
scaler = StandardScaler()
# Fit and Transform The Data
scaler.fit(rfm_normalized)
rfm_scaled = scaler.transform(rfm_normalized)
# Assert that it has mean 0 and variance 1
print(rfm_scaled.mean(axis = 0).round(2))
print(rfm_scaled.std(axis = 0).round(2))
```

```
[-0.  0.  0.]
[1. 1. 1.]
```

In [67]:
```
rfm_scaled = pd.DataFrame(rfm_scaled, index=rfm_normalized.index, columns= rfm_normalize
rfm_scaled.head()
```

Out[67]:

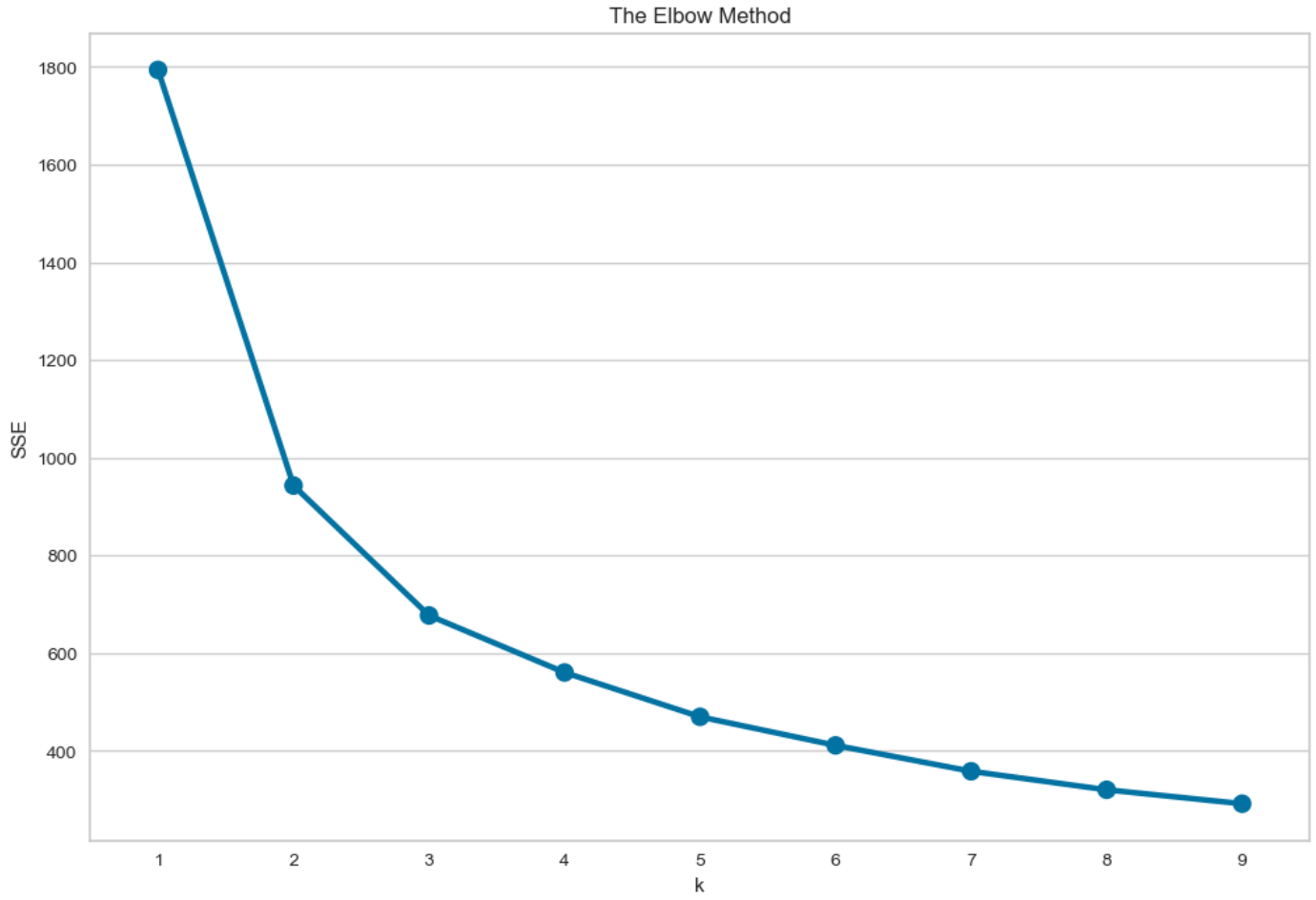
	Recency	Frequency	Monetary

0	-1.659440	2.673118	1.999260
1	0.046076	-0.185076	-0.245201
2	0.086995	0.899986	0.342381
3	0.881816	0.865524	0.062661
4	-0.915099	2.623656	1.861771

In [68]:
```
# Clustering with K-means algorithm
# To make segmentation from the data, we can use the K-Means algorithm to do this.

# K-Means algorithm is an unsupervised learning algorithm that uses the geometrical prin
# determine which cluster belongs to the data. By determine each centroid, we calculate
# Each data belongs to a centroid if it has the smallest distance from the other. It rep
# the distance doesn't have significant changes than before.
```

In [69]:
```
# Determine the Optimal K
# To make our clustering reach its maximum performance, we have to determine which hyper
# To determine which hyperparameter is the best for our model and data, we can use the e
```

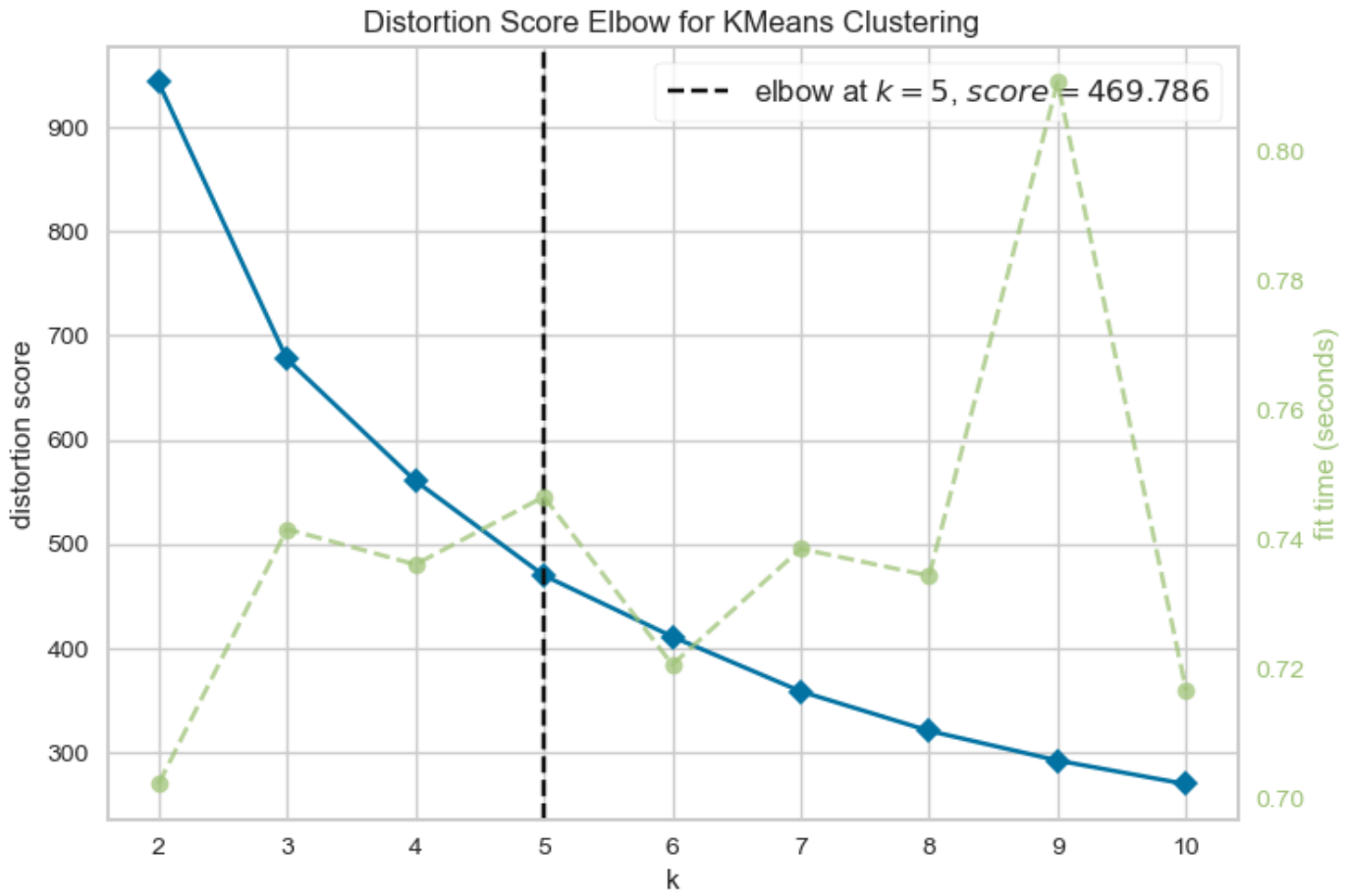
In [70]:
```
# Elbow Method
plt.figure(figsize=(12,8))
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
K = range(1, 10)

distortions = []
inertias = []
mapping1 = {}
sse = {}
for k in K:
    kmeans = KMeans(n_clusters=k, max_iter=1000, random_state=42)
    kmeans.fit(rfm_scaled)

    distortions.append(sum(np.min(cdist(rfm_scaled, kmeans.cluster_centers_,
                                        'euclidean'), axis=1)) / rfm_scaled.shape[0])
    inertias.append(kmeans.inertia_)

    mapping1[k] = sum(np.min(cdist(rfm_scaled, kmeans.cluster_centers_,
                                   'euclidean'), axis=1)) / rfm_scaled.shape[0]

    sse[k] = kmeans.inertia_ # SSE to closest cluster centroid

plt.title('The Elbow Method')
plt.xlabel('k')
plt.ylabel('SSE')
sns.pointplot(x=list(sse.keys()), y=list(sse.values()))
plt.show()
```



In [71]:
```
for key, val in mapping1.items():
    print(f'{key} : {val}')
```

```
1 : 1.5175629009568123
2 : 1.139529798157361
3 : 0.9577685895784434
4 : 0.8770547961374462
5 : 0.7972658558536244
6 : 0.7450723292775535
7 : 0.6925938507597045
8 : 0.6621426144293572
9 : 0.6337106246094812
```

In [72]:
```
# Initialize the KElbowVisualizer with the KMeans estimator and a range of K values
Elbow_M = KElbowVisualizer(KMeans(), k=10)

# Fit the visualizer to the data
Elbow_M.fit(rfm_scaled)

# Display the Elbow Method plot
Elbow_M.show()
```



Out[72]:
```
<Axes: title={'center': 'Distortion Score Elbow for KMeans Clustering'}, xlabel='k', yla
bel='distortion score'>
```

In [73]:
```
# The Within-Cluster Sum of Squares (WSS) graph reveals the elbow point for our dataset.
# From the above plot, we can observe that the elbow point occurs at k=5.
```

In [74]:
```
rfm_scaled_fit = rfm_scaled
```

In [75]:
```
rfm_normalized.shape
```

Out[75]:
```
(598, 3)
```

In [76]:
```
rfm_scaled_fit.shape
```

Out[76]:
```
(598, 3)
```

In [77]:
```
# Kmeans when K=5
kmeans_5 = KMeans(n_clusters=5, random_state=42)
# Fit the KMeans model to your data
yhat_KM_k5 = kmeans_5.fit_predict(rfm_scaled_fit)

# Assign cluster labels to your DataFrame
rfm_scaled_ds_k5 = rfm_scaled_fit
rfm_k5 = rfm
rfm_scaled_ds_k5["Cluster"] = yhat_KM_k5
rfm_scaled_ds_k5["Cluster"] = yhat_KM_k5
```

In [78]:
```
# Kmeans when K=4
kmeans_4 = KMeans(n_clusters=4, random_state=42)
# Fit the KMeans model to your data
yhat_KM_k4 = kmeans_4.fit_predict(rfm_scaled_fit)

# Assign cluster labels to your DataFrame
rfm_scaled_ds_k4 = rfm_scaled_fit
```

```
rfm_k4 = rfm
rfm_scaled_ds_k4["Cluster"] = yhat_KM_k4
rfm_scaled_ds_k4["Cluster"] = yhat_KM_k4
```

In [79]:
```
# Evaluating Methods
```

In [80]:
```
# Sum of Squared Errors (SSE) measures the sum of the squared distances between each dat
# and its assigned centroid. A lower SSE value indicates that the data points within eac
# are closer to their respective centroids, suggesting a better clustering solution.
```

In [81]:
```
sse_k5 = kmeans_5.inertia_
sse_k4 = kmeans_4.inertia_
print(f"When K=5 SSE = {sse_k5}\nWhen K=4 SSE = {sse_k4}")
```

```
When K=5 SSE = 469.7859642702
When K=4 SSE = 676.3867162940801
```

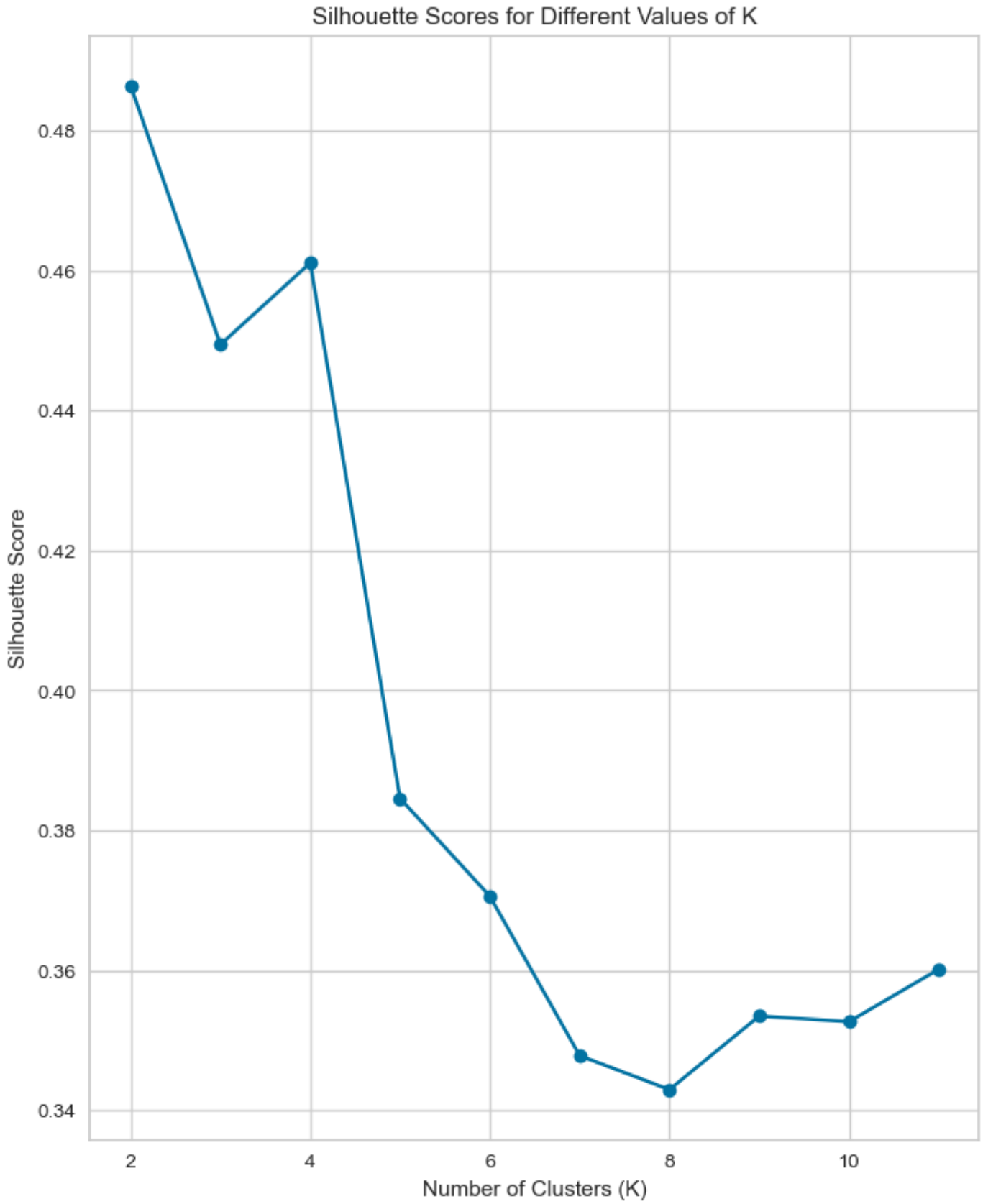
In [82]:
```
# From the above SSE for K=5 is lesser, suggesting best cluster when optimal value, k=5
```

In [83]:
```
# Silhouette Score
```

In [84]:
```
from sklearn.metrics import silhouette_score
k_values = range(2, 12)
silhouette_scores = []

best_k = None
best_silhouette_score = -1

for k in k_values:
    km = KMeans(n_clusters=k, init='k-means++', n_init=10, max_iter=1000, random_state=4
    km.fit(rfm_scaled)
    silhouette_avg = silhouette_score(rfm_scaled, km.labels_)
    silhouette_scores.append((k, silhouette_avg))  # Store k and silhouette score as a t

print("K-values and their Silhouette Scores:")
for k, silhouette_avg in silhouette_scores:
    print(f"K = {k}: Silhouette Score = {silhouette_avg}")

# Find the best k
for k, silhouette_avg in silhouette_scores:
    if silhouette_avg > best_silhouette_score:
        best_k = k
        best_silhouette_score = silhouette_avg

print("\nBest K value:", best_k)
print("Best Silhouette Score:", best_silhouette_score)

# Plot the Silhouette Scores
k_values, silhouette_scores = zip(*silhouette_scores)  # Unzip the k-values and silhouet
plt.figure(figsize=(8, 10))
plt.plot(k_values, silhouette_scores, marker='o')
plt.xlabel("Number of Clusters (K)")
plt.ylabel("Silhouette Score")
plt.title("Silhouette Scores for Different Values of K")
plt.grid(True)
plt.show()
```

```
K-values and their Silhouette Scores:
K = 2: Silhouette Score = 0.4863880036208239
K = 3: Silhouette Score = 0.4494007591227359
K = 4: Silhouette Score = 0.4610727376080494
K = 5: Silhouette Score = 0.38458471011720086
K = 6: Silhouette Score = 0.3706204569049797
K = 7: Silhouette Score = 0.34784266416890103
```

```
K = 8: Silhouette Score = 0.34298925864105073
K = 9: Silhouette Score = 0.3534976346525608
K = 10: Silhouette Score = 0.3526988815527097
K = 11: Silhouette Score = 0.36018284346252927

Best K value: 2
Best Silhouette Score: 0.4863880036208239
```



In [85]:
```
# Select only the numeric columns
numeric_data = rfm.select_dtypes(include=[np.number])

# Calculate quartiles for numeric columns
```

```
quantiles = numeric_data.quantile(q=[0.2, 0.4, 0.6, 0.8])
quantiles = quantiles.to_dict()
```

In [86]:
```
def RScore(x, p, d):
    if x <= d[p][0.20]:
        return 5
    elif x <= d[p][0.40]:
        return 4
    elif x <= d[p][0.60]:
        return 3
    elif x <= d[p][0.80]:
        return 2
    else:
        return 1

def FMScore(x, p, d):
    if x <= d[p][0.20]:
        return 1
    elif x <= d[p][0.40]:
        return 2
    elif x <= d[p][0.60]:
        return 3
    elif x <= d[p][0.80]:
        return 4
    else:
        return 5
```

In [87]:
```
#Calculate Add R, F and M segment value columns in the existing dataset to show R, F and
rfm['R'] = rfm['Recency'].apply(RScore, args=('Recency',quantiles,))
rfm['F'] = rfm['Frequency'].apply(FMScore, args=('Frequency',quantiles,))
rfm['M'] = rfm['Monetary'].apply(FMScore, args=('Monetary',quantiles,))
rfm.head(10)
```

Out[87]:

	Customer Code	Recency	Frequency	Monetary	R	F	M
0	A001	1	3082	316950.01	5	5	5
1	A002	14	19	2536.04	3	3	2
2	A003	15	120	7482.41	3	5	4
3	A004	61	113	4414.15	2	4	3
4	A005	3	2803	220783.16	5	5	5
5	A007	1	989	681102.50	5	5	5
6	A008	14	56	5811.26	3	4	3
7	A009	7	34	7098.52	4	3	4
8	A010	111	24	109226.00	1	3	5
9	A011	1	1450	357023.68	5	5	5

In [88]:
```
# Concat RFM quartile values to create RFM Segments
def join_rfm(x): return str(x['R']) + str(x['F']) + str(x['M'])
rfm['RFM_Segment'] = rfm.apply(join_rfm, axis=1)
# Calculate RFM_Score
rfm['RFM_Score'] = rfm[['R','F','M']].sum(axis=1)
```

In [89]:
```
rfm['RFM_Segment'].unique()
```

Out[89]:
```
array(['555', '332', '354', '243', '343', '434', '135', '333', '123',
       '111', '344', '211', '315', '232', '143', '455', '133', '353',
       '443', '322', '533', '543', '432', '532', '411', '554', '444',
```

```
       '544', '454', '311', '212', '433', '422', '224', '233', '223',
       '132', '215', '414', '513', '511', '244', '255', '144', '355',
       '334', '145', '222', '323', '253', '522', '535', '115', '524',
       '121', '134', '113', '242', '122', '234', '114', '425', '512',
       '221', '435', '523', '245', '335', '534', '423', '225', '325',
       '445', '321', '342', '545', '553', '421', '254', '312', '424',
       '214', '131', '155', '112', '314', '412', '142'], dtype=object)
```

In [90]:
```
rfm.head(20)
```

Out[90]:

	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score
0	A001	1	3082	316950.01	5	5	5	555	15
1	A002	14	19	2536.04	3	3	2	332	8
2	A003	15	120	7482.41	3	5	4	354	12
3	A004	61	113	4414.15	2	4	3	243	9
4	A005	3	2803	220783.16	5	5	5	555	15
5	A007	1	989	681102.50	5	5	5	555	15
6	A008	14	56	5811.26	3	4	3	343	10
7	A009	7	34	7098.52	4	3	4	434	11
8	A010	111	24	109226.00	1	3	5	135	9
9	A011	1	1450	357023.68	5	5	5	555	15
10	A012	13	22	3439.28	3	3	3	333	9
11	A013	211	11	3264.30	1	2	3	123	6
12	A014	303	7	205.92	1	1	1	111	3
13	A015	16	103	6930.04	3	4	4	344	11
14	A016	35	86	5277.22	2	4	3	243	9
15	A017	38	2	167.70	2	1	1	211	4
16	A018	27	55	5834.53	2	4	3	243	9
17	A019	16	3	125897.20	3	1	5	315	9
18	A020	10	65	11228.88	3	4	4	344	11
19	A021	9	28	3088.67	3	3	3	333	9

In [91]:
```
# Create human friendly RFM labels
segt_map = {

    r'11': 'Hibernating',
    r'12': 'Hibernating',
    r'13': 'Hibernating',
    r'14': 'Hibernating',
    r'15': 'Need attention',
    r'21': 'Hibernating',
    r'22': 'Hibernating',
    r'23': 'Hibernating',
    r'24': 'Hibernating',
    r'25': 'Need attention',
    r'31': 'Hibernating',
    r'32': 'Hibernating',
    r'33': 'Need attention',
    r'34': 'Need attention',
```

```
    r'35': 'Need attention',
    r'41': 'Hibernating',
    r'42': 'Hibernating',
    r'43': 'Promising',
    r'44': 'Loyal',
    r'45': 'Loyal',
    r'51': 'Promising',
    r'52': 'Promising',
    r'53': 'Loyal',
    r'54': 'Loyal',
    r'55': 'Champions'

}

# rfm['Segment'] = rfm['R'].map(str) + rfm['F'].map(str)+ rfm['M'].map(str)
rfm['Segment'] = rfm['R'].map(str) + rfm['F'].map(str)
rfm['Segment'] = rfm['Segment'].replace(segt_map, regex=True)

rfm['Score'] = 'Green'  # Default to 'Green'
# Update Score based on RFM Score values
rfm.loc[rfm['RFM_Score'] > 5, 'Score'] = 'Bronze'
rfm.loc[rfm['RFM_Score'] > 7, 'Score'] = 'Silver'
rfm.loc[rfm['RFM_Score'] > 9, 'Score'] = 'Gold'
rfm.loc[rfm['RFM_Score'] > 14, 'Score'] = 'Platinum'



# View the DataFrame with both Segment and Score columns
rfm.head(10)
```

Out[91]:

	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score
0	A001	1	3082	316950.01	5	5	5	555	15	Champions	Platinum
1	A002	14	19	2536.04	3	3	2	332	8	Need attention	Silver
2	A003	15	120	7482.41	3	5	4	354	12	Need attention	Gold
3	A004	61	113	4414.15	2	4	3	243	9	Hibernating	Silver
4	A005	3	2803	220783.16	5	5	5	555	15	Champions	Platinum
5	A007	1	989	681102.50	5	5	5	555	15	Champions	Platinum
6	A008	14	56	5811.26	3	4	3	343	10	Need attention	Gold
7	A009	7	34	7098.52	4	3	4	434	11	Promising	Gold
8	A010	111	24	109226.00	1	3	5	135	9	Hibernating	Silver
9	A011	1	1450	357023.68	5	5	5	555	15	Champions	Platinum

In [92]:

```
segment_monetary_total = rfm.groupby('Segment')['Monetary'].sum()

# Reset the index to make the result more accessible
segment_monetary_total = segment_monetary_total.reset_index()

# Print the total MonetaryValue for each segment
print(segment_monetary_total)
```

```
          Segment     Monetary
0       Champions  29758375.04
1     Hibernating   2050564.23
```

```
2           Loyal   3610555.40
3  Need attention   1538043.82
4       Promising   2656908.54
```

In [93]:
```
unique_segments = rfm['Segment'].unique()
print(unique_segments)
```

```
['Champions' 'Need attention' 'Hibernating' 'Promising' 'Loyal']
```

In [94]:
```
rfm
```

Out[94]:

	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score
0	A001	1	3082	316950.01	5	5	5	555	15	Champions	Platinum
1	A002	14	19	2536.04	3	3	2	332	8	Need attention	Silver
2	A003	15	120	7482.41	3	5	4	354	12	Need attention	Gold
3	A004	61	113	4414.15	2	4	3	243	9	Hibernating	Silver
4	A005	3	2803	220783.16	5	5	5	555	15	Champions	Platinum
...	...	...	...	...	...	...	...	...	...	...	...
593	Z004	2	102	6887.79	5	4	4	544	13	Loyal	Gold
594	Z005	121	15	1501.76	1	2	2	122	5	Hibernating	Green
595	Z006	1	67	10282.74	5	4	4	544	13	Loyal	Gold
596	Z008	27	27	3095.56	2	3	3	233	8	Hibernating	Silver
597	Z009	28	10	1135.68	2	2	2	222	6	Hibernating	Bronze

598 rows × 11 columns

In [95]:
```
#Validate the data for RFM_Segment = 111
rfm[rfm['RFM_Segment']=='111'].sort_values('Monetary', ascending=False).reset_index().he
```

Out[95]:

	index	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score
0	546	T030	170	1	891.80	1	1	1	111	3	Hibernating	Green
1	62	B010	114	5	881.40	1	1	1	111	3	Hibernating	Green
2	257	I013	320	1	879.84	1	1	1	111	3	Hibernating	Green
3	366	M027	76	7	874.90	1	1	1	111	3	Hibernating	Green
4	69	B019	140	2	861.90	1	1	1	111	3	Hibernating	Green

In [96]:
```
#Validate the data for RFM_Segment = 555
rfm[rfm['RFM_Segment']=='555'].sort_values('Monetary', ascending=False).reset_index().he
```

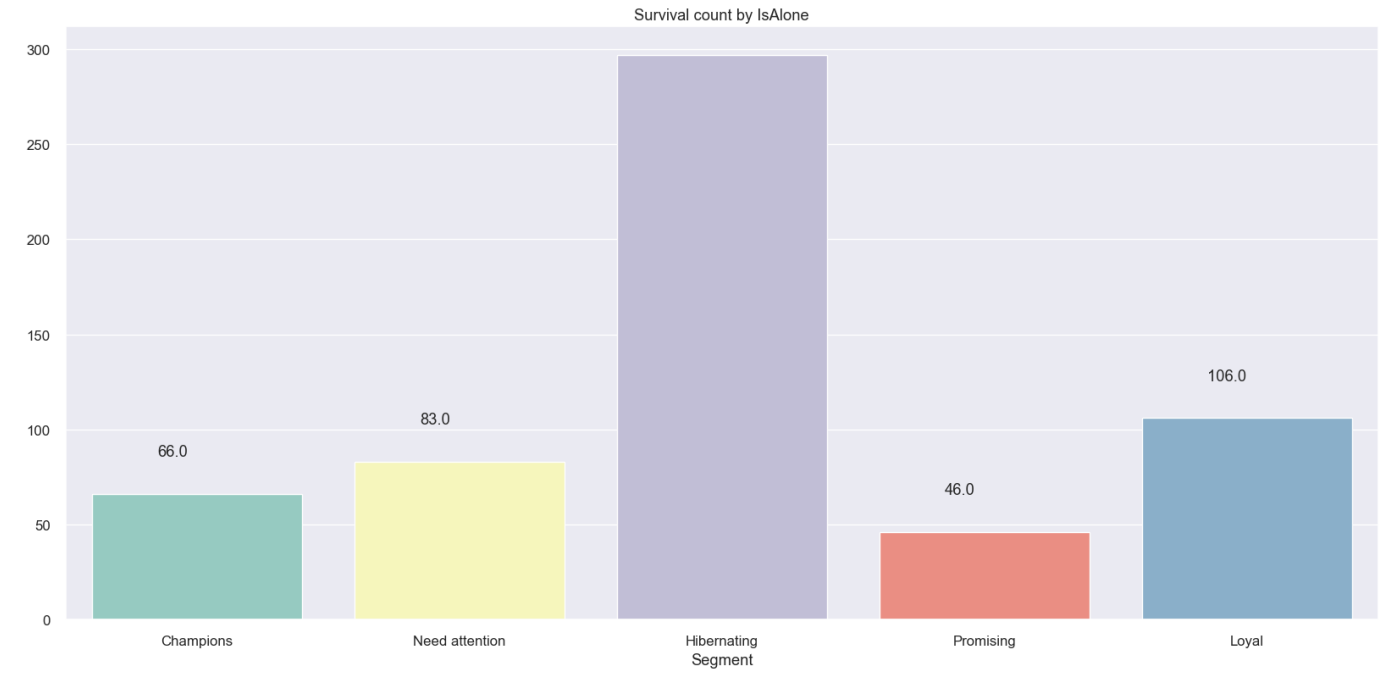
Out[96]:

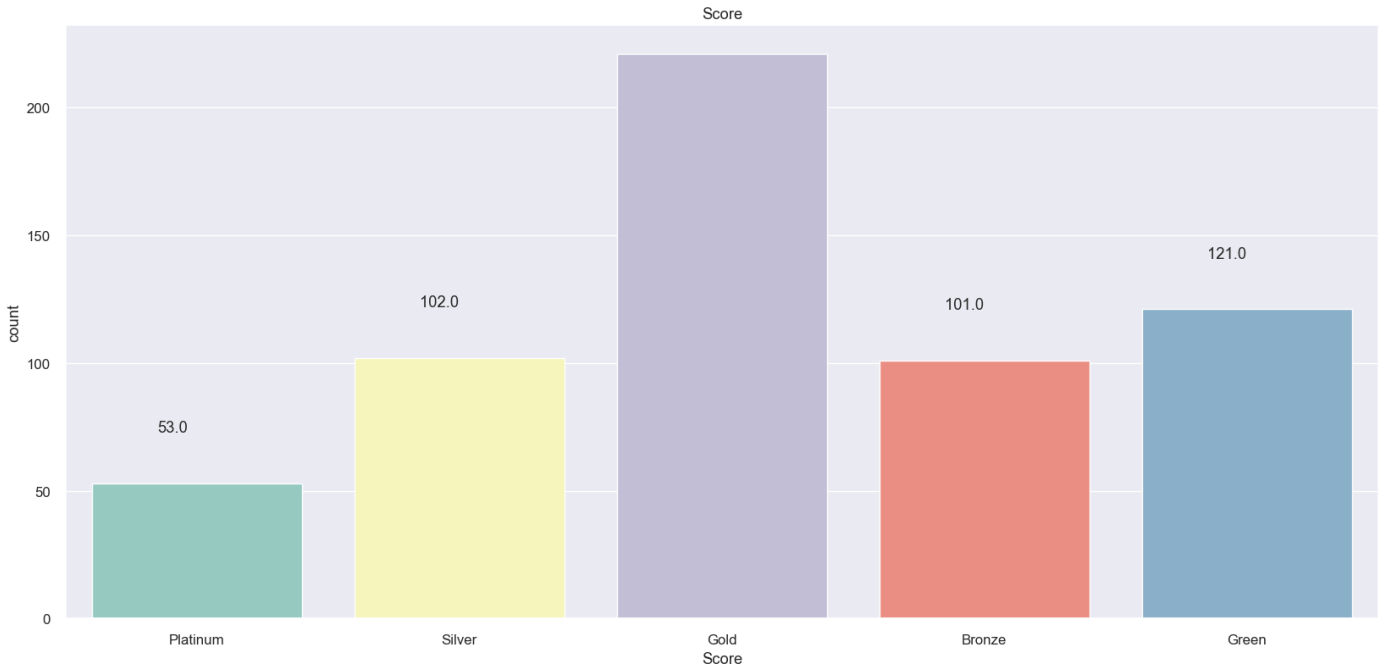
	index	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score
0	391	N002	3	548	9972019.59	5	5	5	555	15	Champions	Platinum
1	78	C001	1	9855	7224920.09	5	5	5	555	15	Champions	Platinum
2	482	S017	1	3526	4529597.75	5	5	5	555	15	Champions	Platinum

3	324	L016	2	847	845845.52	5	5	5	555	15	Champions	Platinum
4	5	A007	1	989	681102.50	5	5	5	555	15	Champions	Platinum

In [97]:
```
fig1 = rfm.groupby(['Segment']).count().unstack().fillna(0)
fig2 = rfm.groupby(['Score']).count().unstack().fillna(0)
```

In [98]:
```
sns.set(font_scale=1.1)

# Pie Chart
f, ax = plt.subplots(2, 1, figsize=(20, 20))
sns.countplot(x='Segment', data=rfm, ax=ax[0], palette="Set3")
ax[0].set_title('Segment')
ax[0].set_ylabel('')
ax[0].set_title('Survival count by IsAlone')

# Bar chart - count
sns.countplot(x='Score', data=rfm, ax=ax[1], palette="Set3")
ax[1].set_title('Score')

for p in ax[0].patches:
    ax[0].annotate('{:.1f}'.format(p.get_height()), (p.get_x() + 0.25, p.get_height() +

for p in ax[1].patches:
    ax[1].annotate('{:.1f}'.format(p.get_height()), (p.get_x() + 0.25, p.get_height() +

plt.show()
```



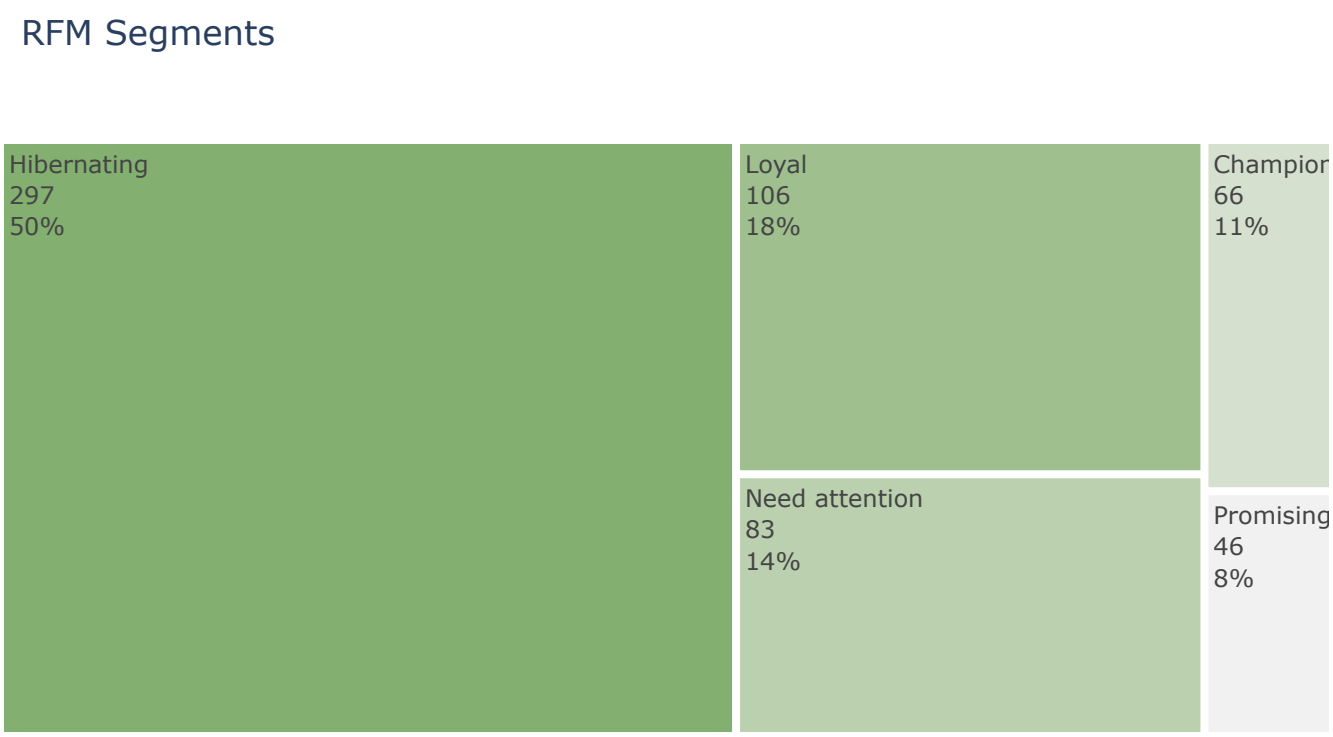


In [99]:
```
# Aggregate data by each customer
fig3 = rfm.groupby('Segment').agg({'Customer Code': lambda x: len(x)}).reset_index()

# Rename columns
fig3.rename(columns={'Customer Code': 'Count'}, inplace=True)
fig3['percent'] = (fig3['Count'] / fig3['Count'].sum()) * 100
fig3['percent'] = fig3['percent'].round(1)

fig3
```

Out[99]:

	Segment	Count	percent
0	Champions	66	11.0
1	Hibernating	297	49.7
2	Loyal	106	17.7
3	Need attention	83	13.9
4	Promising	46	7.7

In [100…
```
#Creating a Treemap with Plotly Express
# colors=['#fae588','#f79d65','#f9dc5c','#e8ac65','#e76f51','#ef233c','#b7094c'] #color
colors=['#83af70','#9fbf8f','#bad0af','#d5e0cf','#f1f1f1','#f1d4d4','#f0b8b8','#ec9c9d']
import plotly.express as px

fig = px.treemap(fig3, path=['Segment'],values='Count'
                 , width=800, height=400
                 ,title="RFM Segments")

fig.update_layout(
    treemapcolorway = colors, #defines the colors in the treemap
    margin = dict(t=50, l=25, r=25, b=25))

fig.data[0].textinfo = 'label+text+value+percent root'
fig.show()
```



In [101…
```
#### Customer Segments and their interpretation


# Champions: Bought recently, buy often and spend the most
# Loyal customers: Buy on a regular basis. Responsive to promotions
# Potential Loyalist: Recent customers with average frequency
# Recent customers: Bought most recently, but not often
# Promising: Recent shoppers, but haven't spent much
# Needs attention: Above average recency, frequency and monetary values. May not have bo
# About to sleep: Below average recency and frequency. Will lose them if not reactivated
# At risk: Some time since theyve purchased. Need to bring them back!
# Cant lost them: Used to purchase frequently but havent returned for a long time
# Hibernating: Last purchase was long back and low number of orders. May be lost
```

In [102…
```
# Aggregate data by each customer
fig4 = rfm.groupby('Score').agg({'Customer Code': lambda x: len(x)}).reset_index()


# Rename columns
fig4.rename(columns={'Customer Code': 'Count'}, inplace=True)
fig4['percent'] = (fig4['Count'] / fig4['Count'].sum()) * 100
fig4['percent'] = fig4['percent'].round(1)
```

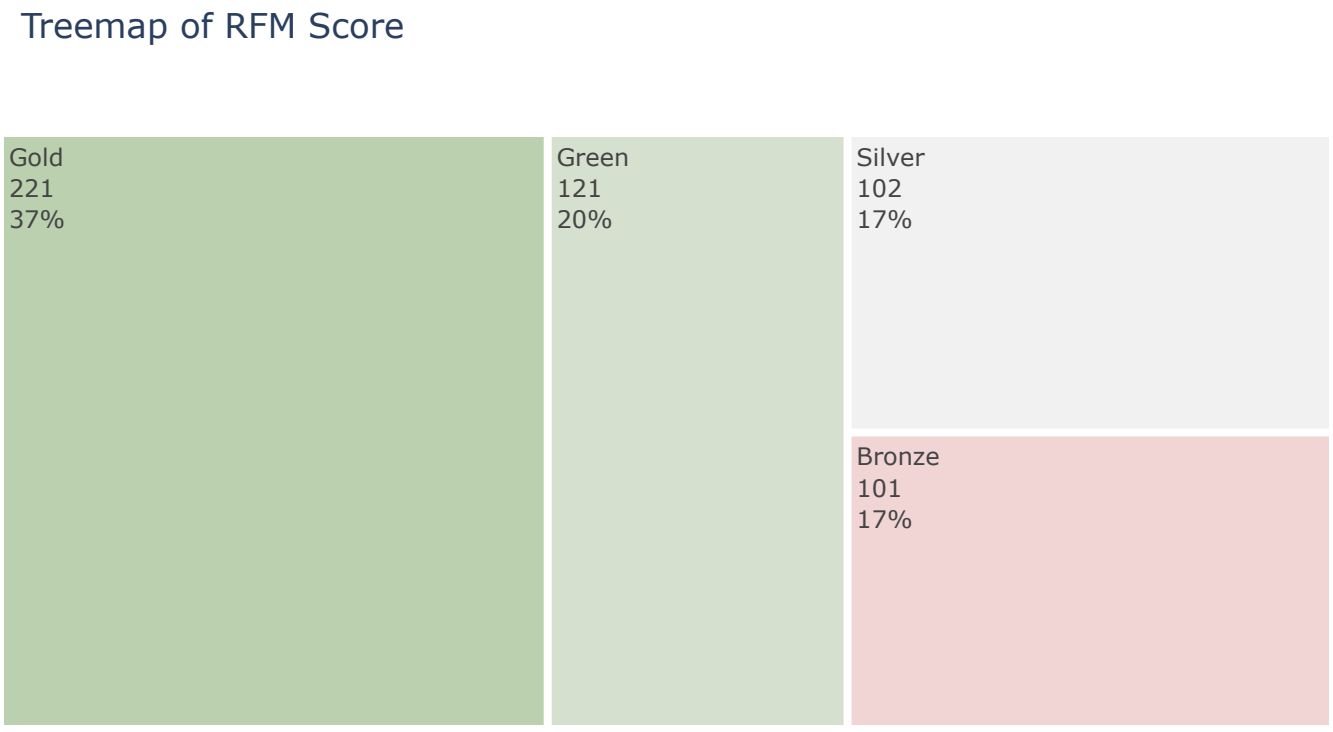
```
fig4
```

Out[102]:

	Score	Count	percent
0	Bronze	101	16.9
1	Gold	221	37.0
2	Green	121	20.2
3	Platinum	53	8.9
4	Silver	102	17.1

In [103…

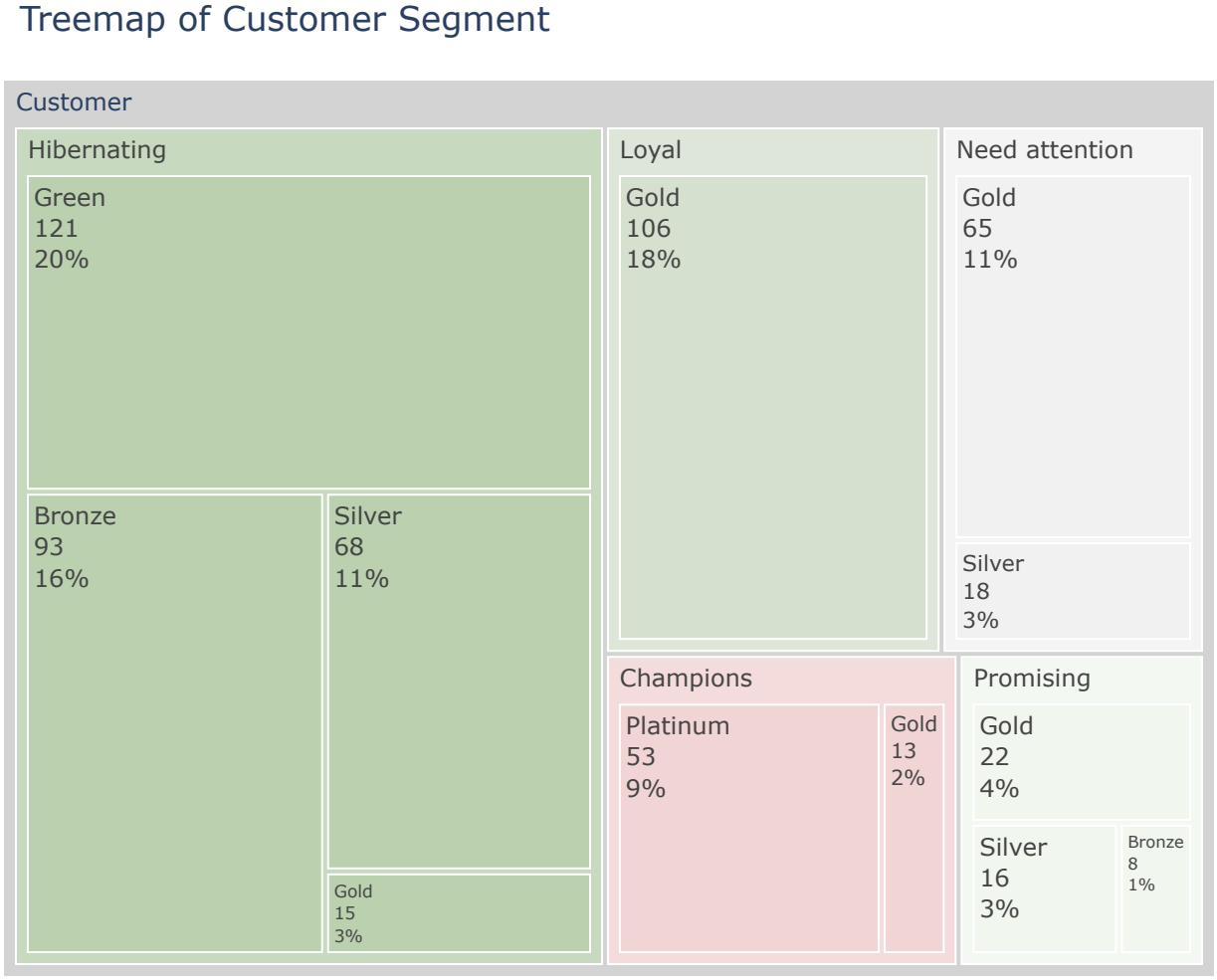
```
#Creating a Treemap with Plotly Express
# colors=['#fae588','#f79d65','#f9dc5c','#e8ac65','#e76f51','#ef233c','#b7094c'] #color
colors=['#bad0af','#d5e0cf','#f1f1f1','#f1d4d4'] #color palette
import plotly.express as px

fig = px.treemap(fig4, path=['Score'],values='Count'
                 , width=800, height=400
                 ,title="Treemap of RFM Score")

fig.update_layout(
    treemapcolorway = colors, #defines the colors in the treemap
    margin = dict(t=50, l=25, r=25, b=25))

fig.data[0].textinfo = 'label+text+value+percent root'
fig.show()
```



In [104…

```
# Aggregate data by each customer
fig5 = rfm.groupby(['Segment', 'Score']).agg({'Customer Code': lambda x: len(x)}).reset_

# Rename columns
fig5.rename(columns={'Customer Code': 'Count'}, inplace=True)
fig5['percent'] = (fig5['Count'] / fig5['Count'].sum()) * 100
fig5['percent'] = fig5['percent'].round(1)
```

```
fig5.head()
colors=['#bad0af','#d5e0cf','#f1f1f1','#f1d4d4'] #color palette
import plotly.express as px
fig = px.treemap(fig5, path=[px.Constant("Customer"), 'Segment', 'Score'], values='Count
                ,title="Treemap of Customer Segment")
fig.update_traces(root_color="lightgrey")
fig.update_layout(margin = dict(t=50, l=25, r=25, b=25))

fig.update_layout(
    treemapcolorway = colors, #defines the colors in the treemap
    margin = dict(t=50, l=25, r=25, b=25))

fig.data[0].textinfo = 'label+text+value+percent root'
fig.show()
```



In [105…

```
# Aggregate data by each customer
rfm1 = rfm.groupby(['Score','Segment'],).agg({'Customer Code': lambda x: len(x)}).reset_

# Rename columns
rfm1.rename(columns={'Customer Code': 'Count'}, inplace=True)
rfm1['percent'] = (rfm1['Count'] / rfm1.groupby('Score')['Count'].transform('sum')) * 10
rfm1['percent'] = rfm1['percent'].round(1)

rfm1
```

Out[105]:

	Score	Segment	Count	percent
0	Bronze	Hibernating	93	92.1
1	Bronze	Promising	8	7.9
2	Gold	Champions	13	5.9

3	Gold	Hibernating	15	6.8
4	Gold	Loyal	106	48.0
5	Gold	Need attention	65	29.4
6	Gold	Promising	22	10.0
7	Green	Hibernating	121	100.0
8	Platinum	Champions	53	100.0
9	Silver	Hibernating	68	66.7
10	Silver	Need attention	18	17.6
11	Silver	Promising	16	15.7

In [106…
```
rfm
```

Out[106]:

	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score
0	A001	1	3082	316950.01	5	5	5	555	15	Champions	Platinum
1	A002	14	19	2536.04	3	3	2	332	8	Need attention	Silver
2	A003	15	120	7482.41	3	5	4	354	12	Need attention	Gold
3	A004	61	113	4414.15	2	4	3	243	9	Hibernating	Silver
4	A005	3	2803	220783.16	5	5	5	555	15	Champions	Platinum
...	...	...	...	...	...	...	...	...	...	...	...
593	Z004	2	102	6887.79	5	4	4	544	13	Loyal	Gold
594	Z005	121	15	1501.76	1	2	2	122	5	Hibernating	Green
595	Z006	1	67	10282.74	5	4	4	544	13	Loyal	Gold
596	Z008	27	27	3095.56	2	3	3	233	8	Hibernating	Silver
597	Z009	28	10	1135.68	2	2	2	222	6	Hibernating	Bronze

598 rows × 11 columns

In [107…
```
rfm.to_csv('RFM.csv', index=False)
```

In [108…
```
# Assuming your data is stored in a DataFrame called 'data'
segment_descriptions = rfm.groupby('Segment').describe()

# Loop through segments and print descriptions
for segment, description in segment_descriptions.groupby(level=0):
    print(f"Description for segment '{segment}':")
    display(description)
    print("\n")
```

```
Description for segment 'Champions':
```

	Recency								Frequency		...	M		
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count
Segment														
Champions	66.0	1.575758	0.702974	1.0	1.0	1.0	2.0	3.0	66.0	710.287879	...	5.0	5.0	66.0

1

1 rows × 56 columns

```
Description for segment 'Hibernating':
```

	Recency								Frequency		...	M		
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	cour
Segment														
Hibernating	297.0	97.740741	104.196881	5.0	20.0	41.0	162.0	335.0	297.0	16.892256	...	3.0	5.0	297.

1 rows × 56 columns

```
Description for segment 'Loyal':
```

	Recency								Frequency		...	M		
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count
Segment														
Loyal	106.0	4.084906	2.426512	1.0	2.0	3.0	6.0	8.0	106.0	136.349057	...	5.0	5.0	106.0	12.3

1 rows × 56 columns

```
Description for segment 'Need attention':
```

	Recency								Frequency		...	M		
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count
Segment														
Need attention	83.0	17.86747	19.079277	9.0	10.0	13.0	16.0	118.0	83.0	92.626506	...	4.0	5.0	83.0	10

1 rows × 56 columns

```
Description for segment 'Promising':
```

	Recency								Frequency		...	M		
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	
Segment														
Promising	46.0	4.217391	2.723666	1.0	2.0	4.0	7.0	8.0	46.0	19.521739	...	3.0	5.0	46.0	9.36

1 rows × 56 columns

In [109…
```
rfm.describe()
```

Out[109]:

	Recency	Frequency	Monetary	R	F	M	RFM_Score
count	598.000000	598.000000	5.980000e+02	598.000000	598.000000	598.00000	598.000000
mean	52.245819	125.309365	6.624489e+04	3.045151	2.988294	3.00000	9.033445
std	86.614636	513.728369	5.454539e+05	1.456103	1.427722	1.41658	3.667808
min	1.000000	1.000000	5.330000e+01	1.000000	1.000000	1.00000	3.000000

25%	5.000000	9.000000	1.275007e+03	2.000000	2.000000	2.00000	6.000000
50%	13.000000	27.000000	4.418310e+03	3.000000	3.000000	3.00000	9.000000
75%	43.000000	84.750000	1.246856e+04	4.000000	4.000000	4.00000	12.000000
max	335.000000	9855.000000	9.972020e+06	5.000000	5.000000	5.00000	15.000000

In [110…
```
# From both the above Methods, best K value is 5.
# So we choose K value to be 5 for further analysis.
```

In [111…
```
# Fit the model with 5 clusters
```

In [112…
```
#Build K Means clustering model using K=5
kmean_model = KMeans(n_clusters=5, init='k-means++', max_iter=1000, random_state=42)
kmean_model.fit(rfm_scaled)

#find the clusters for the records given in our dataset
rfm_scaled['Cluster'] = kmean_model.labels_+1

#check the datset
rfm_scaled
```

Out[112]:

	Recency	Frequency	Monetary	Cluster
0	-1.659440	2.673118	1.999260	4
1	0.046076	-0.185076	-0.245201	3
2	0.086995	0.899986	0.342381	1
3	0.881816	0.865524	0.062661	1
4	-0.915099	2.623656	1.861771	4
...	...	...	...	...
593	-1.183875	0.806661	0.299334	5
594	1.245257	-0.328565	-0.550186	3
595	-1.659440	0.563234	0.504810	5
596	0.428537	0.026382	-0.132747	3
597	0.449259	-0.577027	-0.718575	3

598 rows × 4 columns

In [113…
```
fitted_lambda
```

Out[113]:
```
{'Recency': -0.06256522187993799,
 'Frequency': -0.029515193030542657,
 'Monetary': -0.08670970030224771}
```

In [114…
```
rfm[['Recency', 'Frequency', 'Monetary']]
```

Out[114]:

	Recency	Frequency	Monetary
0	1	3082	316950.01
1	14	19	2536.04
2	15	120	7482.41
3	61	113	4414.15

4	3	2803	220783.16
...	...	...	...
593	2	102	6887.79
594	121	15	1501.76
595	1	67	10282.74
596	27	27	3095.56
597	28	10	1135.68

598 rows × 3 columns

In [115…
```
import numpy as np
import pandas as pd
from scipy import stats

def reverse_boxcox(transformed, lambda_value):
    if lambda_value == 0:
        return np.exp(transformed)
    else:
        return (np.power((transformed * lambda_value) + 1, 1 / lambda_value))

# Reverse the Box-Cox transformation for Recency, Frequency, and MonetaryValue
rfm_reversed = pd.DataFrame()
rfm_reversed["Recency"] = reverse_boxcox(rfm_normalized["Recency"], fitted_lambda["Recen
rfm_reversed["Frequency"] = reverse_boxcox(rfm_normalized["Frequency"], fitted_lambda["F
rfm_reversed["Monetary"] = reverse_boxcox(rfm_normalized["Monetary"], fitted_lambda["Mon
rfm_reversed['Cluster'] = rfm_scaled['Cluster']
rfm_reversed
```

Out[115]:

	Recency	Frequency	Monetary	Cluster
0	1.0	3082.0	316950.01	4
1	14.0	19.0	2536.04	3
2	15.0	120.0	7482.41	1
3	61.0	113.0	4414.15	1
4	3.0	2803.0	220783.16	4
...	...	...	...	...
593	2.0	102.0	6887.79	5
594	121.0	15.0	1501.76	3
595	1.0	67.0	10282.74	5
596	27.0	27.0	3095.56	3
597	28.0	10.0	1135.68	3

598 rows × 4 columns

In [116…
```
rfm_reversed.describe()
```

Out[116]:

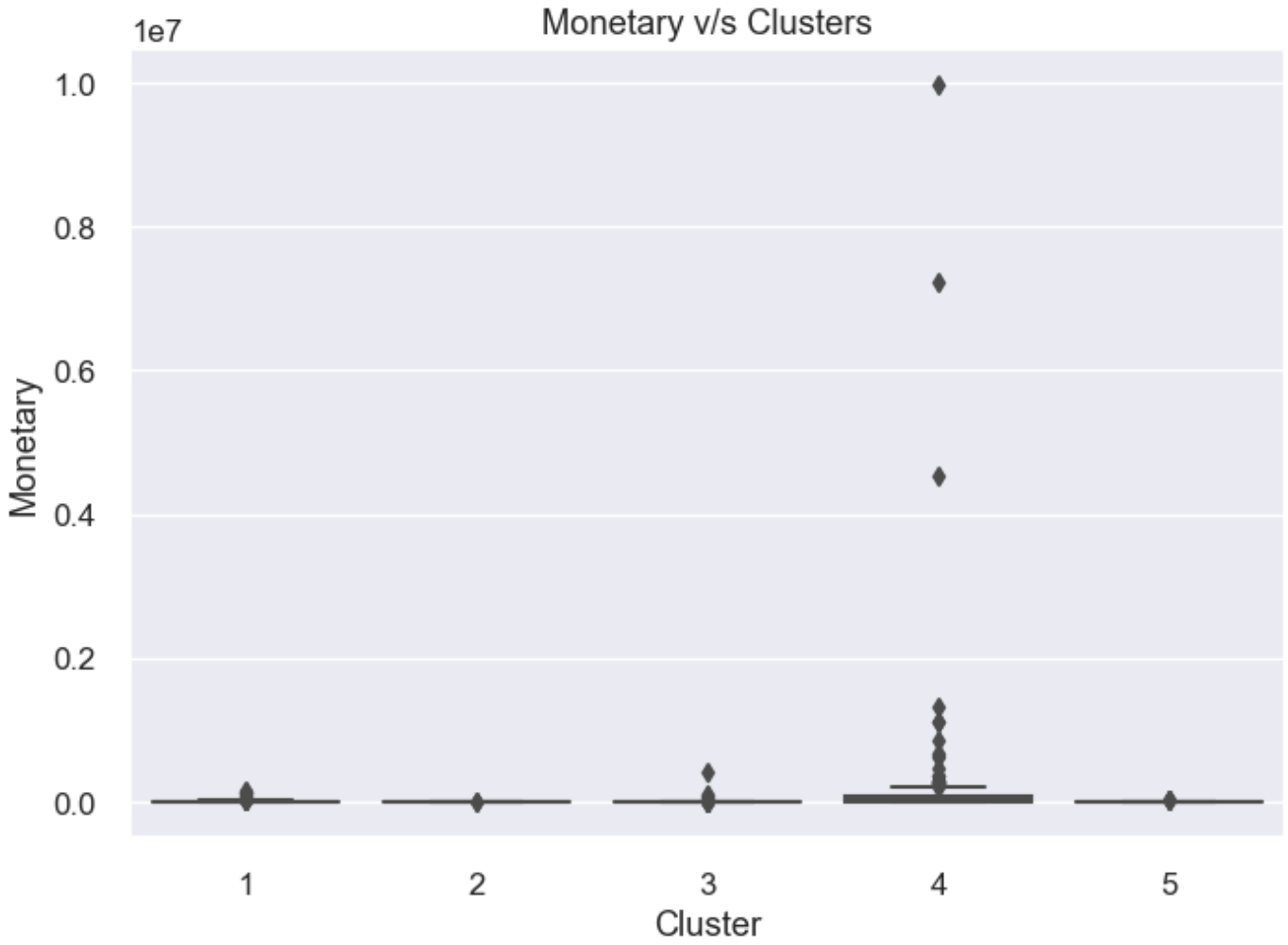
	Recency	Frequency	Monetary	Cluster
count	598.000000	598.000000	5.980000e+02	598.000000
mean	52.245819	125.309365	6.624489e+04	2.956522

std	86.614636	513.728369	5.454539e+05	1.373883
min	1.000000	1.000000	5.330000e+01	1.000000
25%	5.000000	9.000000	1.275008e+03	2.000000
50%	13.000000	27.000000	4.418310e+03	3.000000
75%	43.000000	84.750000	1.246856e+04	4.000000
max	335.000000	9855.000000	9.972020e+06	5.000000

In [117…
```
# K-means with RFM Visualization
```

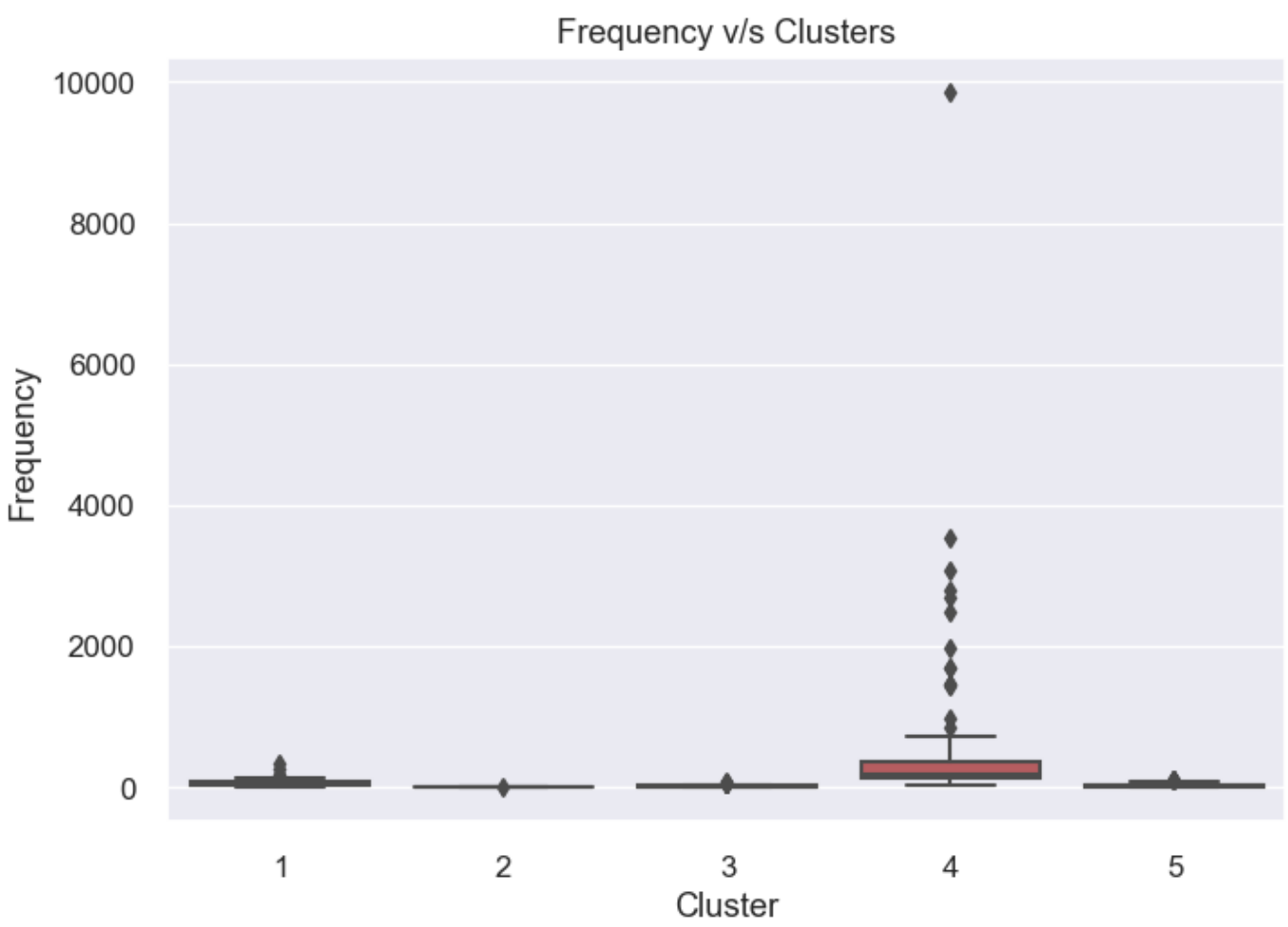
In [118…
```
sns.boxplot(x='Cluster',y='Monetary', data=rfm_reversed).set_title("Monetary v/s Cluster
```

Out[118]:
```
Text(0.5, 1.0, 'Monetary v/s Clusters')
```



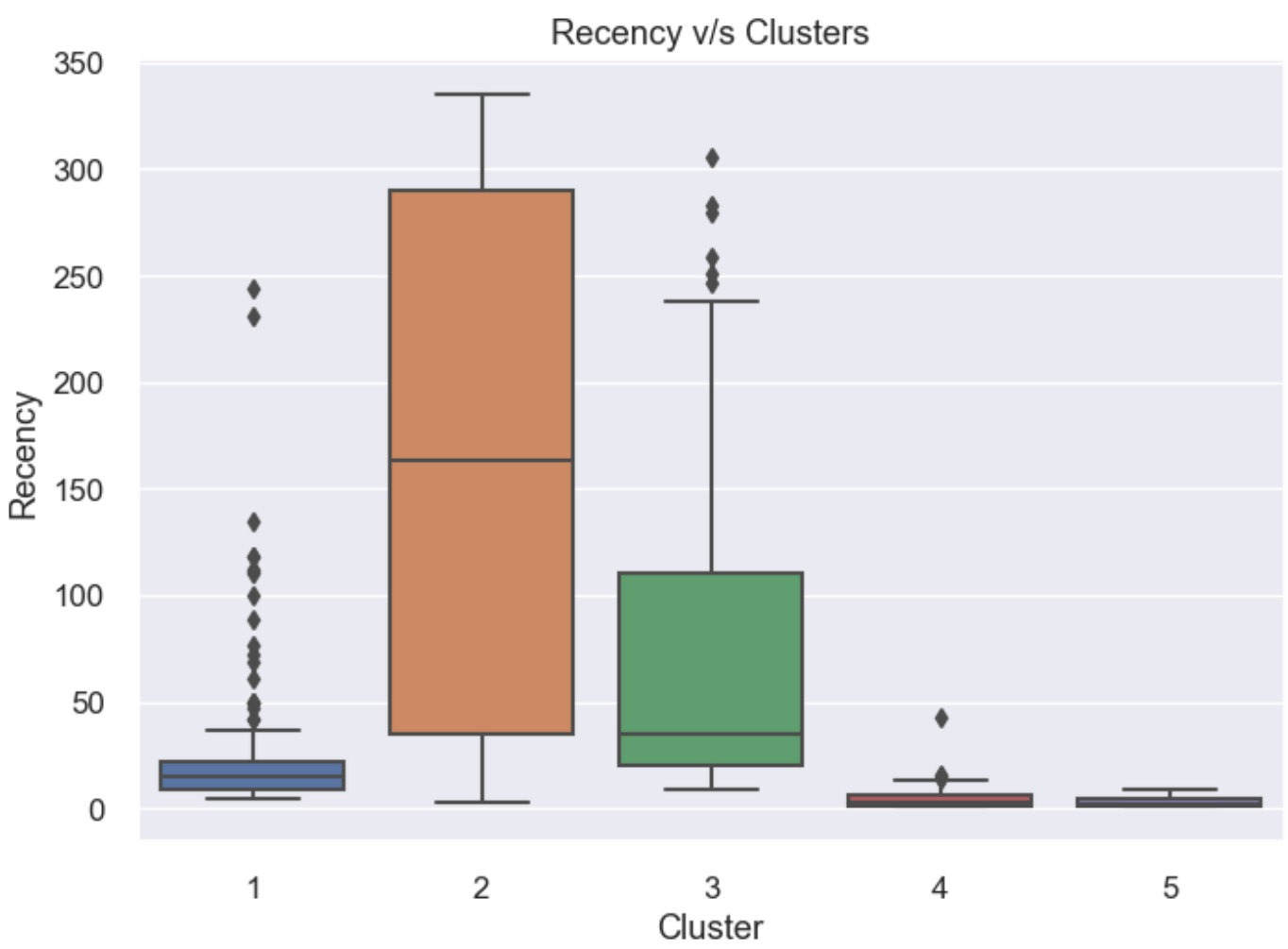
In [119…
```
sns.boxplot(x='Cluster',y='Frequency', data=rfm_reversed).set_title("Frequency v/s Clust
```

Out[119]:
```
Text(0.5, 1.0, 'Frequency v/s Clusters')
```



In [120…
```
sns.boxplot(x='Cluster',y='Recency', data=rfm_reversed).set_title("Recency v/s Clusters"
```

Out[120]:
```
Text(0.5, 1.0, 'Recency v/s Clusters')
```

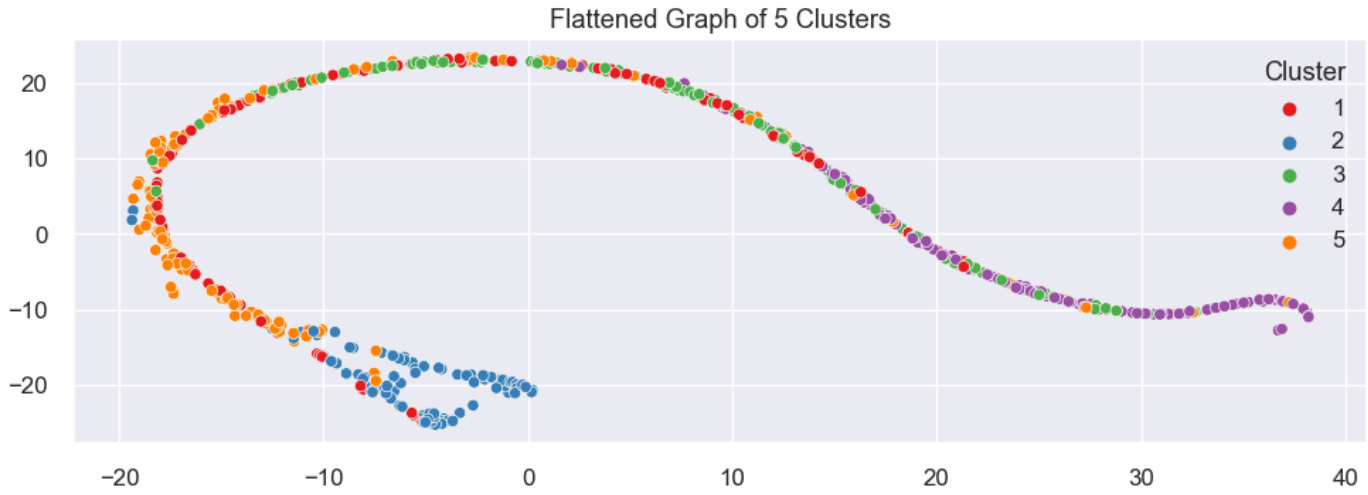


In [121…
```
#centriods
kmean_model.cluster_centers_
```

Out[121]:
```
array([[ 1.57481185e-01,  3.72249269e-01,  3.92855001e-01,
         1.01574803e+00],
       [ 1.06525357e+00, -1.46746154e+00, -1.51838572e+00,
         3.00000000e+00],
       [ 6.90674340e-01, -3.77207336e-01, -2.90848491e-01,
         1.00000000e+00],
       [-9.12120034e-01,  1.26227235e+00,  1.16667186e+00,
         2.00000000e+00],
       [-1.08129730e+00, -1.31273142e-01, -1.08011173e-01,
        -4.44089210e-16]])
```

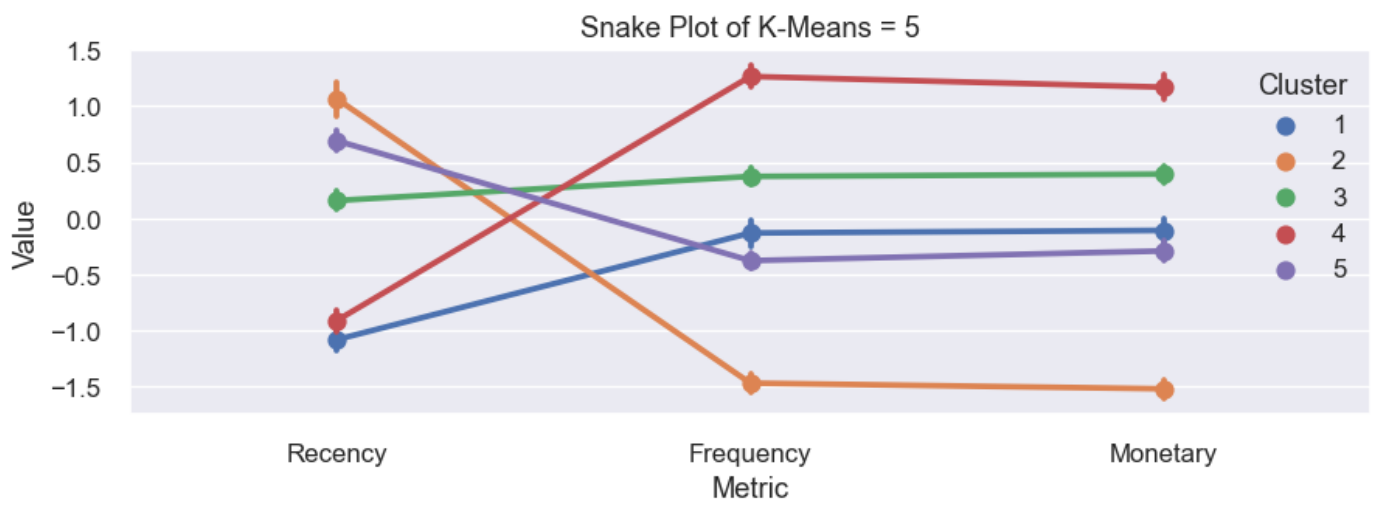
In [122…
```
def kmeans(normalised_df_rfm, clusters_number, original_df_rfm):
    kmeans = KMeans(n_clusters=clusters_number, random_state=42)
    kmeans.fit(normalised_df_rfm)

    # Extract cluster labels
    cluster_labels = kmeans.labels_

    # Create a cluster label column in the original dataset and adjust labels to start f
    df_new = original_df_rfm.assign(Cluster=cluster_labels + 1)

    # Remove non-numeric columns
    numeric_columns = df_new.select_dtypes(include='number')

    # Initialize t-SNE
    model = TSNE(random_state=42)
    transformed = model.fit_transform(numeric_columns)

    # Plot t-SNE using cluster labels as colors
    plt.title('Flattened Graph of {} Clusters'.format(clusters_number))
```

```
    sns.scatterplot(x=transformed[:, 0], y=transformed[:, 1], hue=df_new['Cluster'], pal

    return df_new

plt.figure(figsize=(10, 10))

plt.subplot(3, 1, 2)
df_rfm_k5 = kmeans(rfm_scaled, 5, rfm)
plt.tight_layout()
```



In [123…

```
def snake_plot(normalised_df_rfm, df_rfm_kmeans, df_rfm_original):
    normalised_df_rfm = pd.DataFrame(normalised_df_rfm,
                                     index=rfm.index,
                                     columns=rfm.columns)

    # Adjust cluster labels to start from 1
    cluster_labels = df_rfm_kmeans['Cluster']
    normalised_df_rfm['Cluster'] = cluster_labels

    # Drop the 'level_0' column if it exists
    if 'level_0' in normalised_df_rfm.columns:
        normalised_df_rfm = normalised_df_rfm.drop('level_0', axis=1)

    # Melt data into long format
    df_melt = pd.melt(normalised_df_rfm.reset_index(),
                      id_vars=['Customer Code', 'Cluster'],
                      value_vars=['Recency', 'Frequency', 'Monetary'],
                      var_name='Metric',
                      value_name='Value')

    plt.xlabel('Metric')
    plt.ylabel('Value')
    sns.pointplot(data=df_melt, x='Metric', y='Value', hue='Cluster')

    return

# Use the function with adjusted cluster labels
plt.figure(figsize=(9, 9))

plt.subplot(3, 1, 2)
plt.title('Snake Plot of K-Means = 5')
snake_plot(rfm_scaled, df_rfm_k5, rfm)

plt.tight_layout()
```



In [124…
```
# Relative importance of attributes by cluster
```

In [125…
```
# We can also use the following method to understand the relative importance of segments

# 1.Calculate average values of each cluster
# 2.Calculate average values of population
# 3.Calculate importance score by dividing them and subtracting 1 (ensures 0 is returned
```

In [126…
```
cluster_avg = rfm_reversed[['Cluster','Recency','Frequency','Monetary']].groupby(['Clust
cluster_avg
```

Out[126]:

Cluster	Recency	Frequency	Monetary
1	27.149606	61.653543	14728.023071
2	163.959184	3.051020	416.741531
3	74.662069	17.193103	7586.638345
4	4.607692	472.115385	277796.047308
5	3.081633	29.979592	4995.905306

In [127…
```
population_avg = rfm_reversed[['Recency','Frequency','Monetary']].head().mean()
population_avg
```

Out[127]:
```
Recency          18.800
Frequency      1227.400
Monetary     110433.154
dtype: float64
```

In [128…
```
# As the final step in this analysis, we can extract this information now for each custo
# customer with their relative importance by the company:
```

In [129…
```
relative_imp = cluster_avg / population_avg - 1
relative_imp.round(2)
```

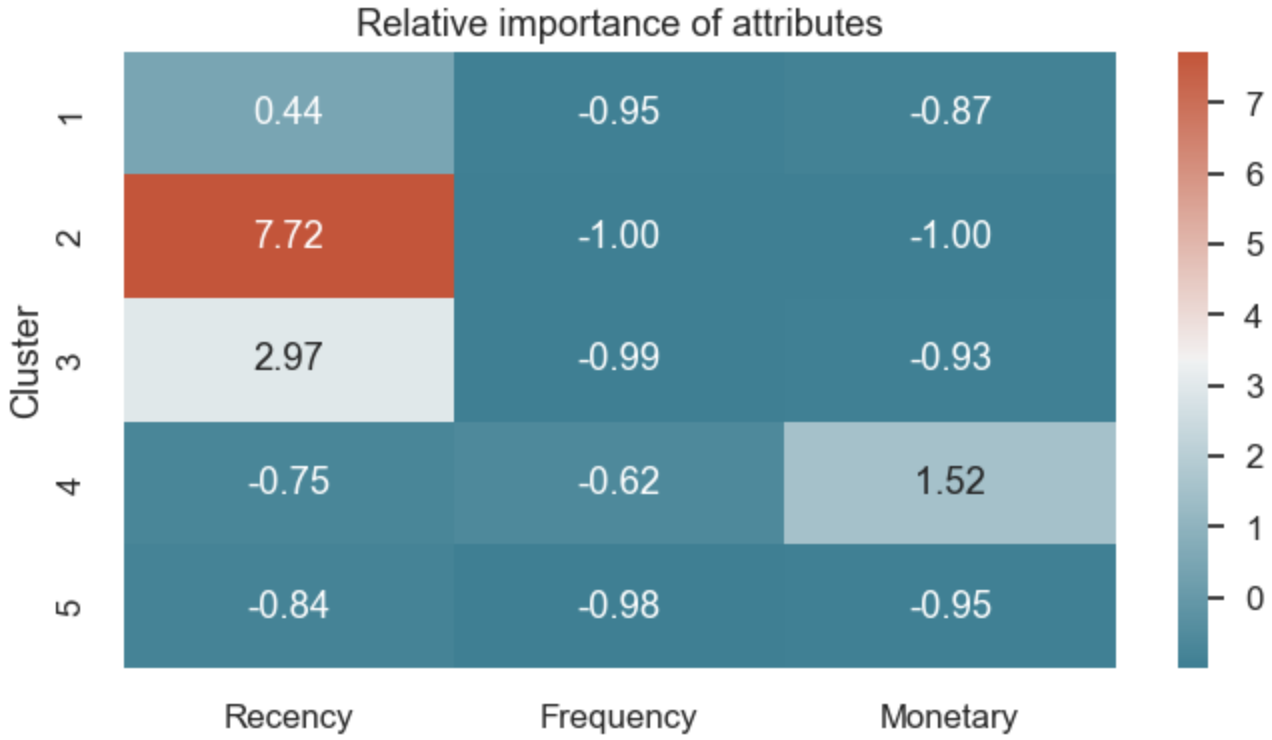
Out[129]:

Cluster	Recency	Frequency	Monetary
1	0.44	-0.95	-0.87
2	7.72	-1.00	-1.00
3	2.97	-0.99	-0.93

4	-0.75	-0.62	1.52
5	-0.84	-0.98	-0.95

In [130…

```
cmap = sns.diverging_palette(220, 20, as_cmap=True)
# Plot heatmap
plt.figure(figsize=(8, 4))
plt.title('Relative importance of attributes')
sns.heatmap(data=relative_imp, annot=True, fmt='.2f', cmap=cmap)
plt.show()
```



In [131…

```
# Create a new column for RFM Customer Segments
rfm['RFM_kMeans_Segment'] = ''

# Assign RFM segments based on the RFM score
rfm.loc[rfm['RFM_Score'] >= 9, 'RFM_kMeans_Segment'] = 'Champions'
rfm.loc[(rfm['RFM_Score'] >= 6) & (rfm['RFM_Score'] < 9), 'RFM_kMeans_Segment'] = 'Poten
rfm.loc[(rfm['RFM_Score'] >= 5) & (rfm['RFM_Score'] < 6), 'RFM_kMeans_Segment'] = 'At Ri
rfm.loc[(rfm['RFM_Score'] >= 4) & (rfm['RFM_Score'] < 5), 'RFM_kMeans_Segment'] = "Can't
rfm.loc[(rfm['RFM_Score'] >= 3) & (rfm['RFM_Score'] < 4), 'RFM_kMeans_Segment'] = "Lost"

# Reset the index so that 'Customer Code' becomes a regular column
#rfm.reset_index(inplace=True)

# Now you can access the 'Customer Code' column
rfm[['Customer Code', 'Score', 'RFM_kMeans_Segment']]
```

Out[131]:

	Customer Code	Score	RFM_kMeans_Segment
0	A001	Platinum	Champions
1	A002	Silver	Potential Loyalists
2	A003	Gold	Champions
3	A004	Silver	Champions
4	A005	Platinum	Champions
…	…	…	…

593	Z004	Gold	Champions
594	Z005	Green	At Risk Customers
595	Z006	Gold	Champions
596	Z008	Silver	Potential Loyalists
597	Z009	Bronze	Potential Loyalists

598 rows × 3 columns

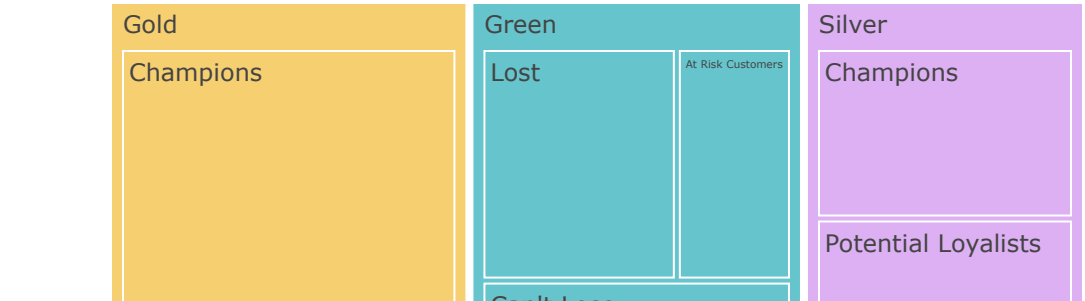
In [132…
```
rfm
```

Out[132]:

	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score	RFM
0	A001	1	3082	316950.01	5	5	5	555	15	Champions	Platinum	
1	A002	14	19	2536.04	3	3	2	332	8	Need attention	Silver	
2	A003	15	120	7482.41	3	5	4	354	12	Need attention	Gold	
3	A004	61	113	4414.15	2	4	3	243	9	Hibernating	Silver	
4	A005	3	2803	220783.16	5	5	5	555	15	Champions	Platinum	
...	...	...	...	...	...	...	...	...	...	...	...	
593	Z004	2	102	6887.79	5	4	4	544	13	Loyal	Gold	
594	Z005	121	15	1501.76	1	2	2	122	5	Hibernating	Green	
595	Z006	1	67	10282.74	5	4	4	544	13	Loyal	Gold	
596	Z008	27	27	3095.56	2	3	3	233	8	Hibernating	Silver	
597	Z009	28	10	1135.68	2	2	2	222	6	Hibernating	Bronze	

598 rows × 12 columns

In [133…
```
seg_product_counts = rfm.groupby(['Score','RFM_kMeans_Segment']).size().reset_index(name
seg_product_counts = seg_product_counts.sort_values('Count', ascending=False)
fig_treemap_seg_product = px.treemap(seg_product_counts, path=['Score', 'RFM_kMeans_Segm
                                     values='Count', color='Score', color_discrete_seque
                                     title='RFM kMeans Segments by Value')
fig_treemap_seg_product.show()
```



In [134…

```python
import plotly.express as px
import numpy as np

# Round the 'Cluster' values to the nearest integer
df_rfm_k5['Cluster'] = np.round(df_rfm_k5['Cluster']).astype(int)

fig = px.scatter_3d(
    df_rfm_k5,
    x='Recency',
    y='Frequency',
    z='Monetary',
    color='Cluster',
    size_max=10,
    opacity=0.7,
    labels={'Recency': 'Recency', 'Frequency': 'Frequency', 'Monetary': 'Monetary'},
    title='3D Scatter Plot of RFM Data',
    template='plotly_dark'  # You can change the template to your preference
)

# Customize cluster labels without decimal points
fig.update_traces(marker=dict(size=5))
fig.update_traces(textposition='top center', textfont_size=12)

# Customize axis formatting to display whole numbers without rounding or prefixes
fig.update_layout(scene=dict(
    xaxis_title='Recency',
    yaxis_title='Frequency',
    zaxis_title='Monetary',
    xaxis_tickformat=',d',  # Format as integer (whole number)
    yaxis_tickformat=',d',  # Format as integer (whole number)
    zaxis_tickformat=',d',  # Format as integer (whole number)
))

# Update the colorbar ticks to display whole numbers
fig.update_layout(coloraxis_colorbar=dict(
    tickmode='array',
    tickvals=np.arange(1, 6),
    ticktext=np.arange(1, 6).astype(str),
    title='Cluster'
))

fig.show()
```

Price Elasticity and Cross Price Elasticity

Setting up the Dataframe for Supermarket and Retail

In [ ]:

In [135…
```
sp_df = df_sales[df_sales['Customer Category Desc'] == 'SUPERMARKET']
rt_df = df_sales[df_sales['Customer Category Desc'] == 'RETAIL']
```

In [136…
```
rte_sp_df = sp_df[sp_df['Inventory Category'] == 'Ready to Eat (RTE)']
rtc_sp_df = sp_df[sp_df['Inventory Category'] == 'Ready to Cook (RTC)']
raw_sp_df = sp_df[sp_df['Inventory Category'] == 'Raw']
```

In [137…
```
rte_sp_5_df = rte_sp_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Invento

rte_sp_top5_df = rte_sp_5_df.sort_values(by='Total Base Amt', ascending=False).head(5)

rte_sp_top5_df['Total Base Amt'] = rte_sp_top5_df['Total Base Amt'].round(2)

print(rte_sp_top5_df)
```

```
                Total Base Amt                    Inventory Desc
Inventory Code
1332.0                1128903.36               BETAGRO BREAST HERBS
1331.0                1070612.40       BETAGRO BREAST GARLIC PEPPER
1330.0                 902175.04         BETAGRO BREAST HOT & SPICY
1333.0                 354590.08           BETAGRO BREAST PINK SALT
1334.0                  11774.88  BETAGRO BREAST BLACK PEPPER CK CUT
```

In [138…
```
rtc_sp_5_df = rtc_sp_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Invento

rtc_sp_top5_df = rtc_sp_5_df.sort_values(by='Total Base Amt', ascending=False).head(5)
```

```
rtc_sp_top5_df['Total Base Amt'] = rtc_sp_top5_df['Total Base Amt'].round(2)

print(rtc_sp_top5_df)
```

```
                Total Base Amt                   Inventory Desc
Inventory Code
1158.0                1201016.96  (33) CRISPY CHICKEN SEAWEED 1KG
1162.0                 766534.80      (3) CRISPY CHIX S/WEED 400G
1163.0                 300917.36    (1) CRISPY CHIX ORIGINAL 400G
1166.0                 295099.62       (2) SUPER CRISPY CHIX 400G
1160.0                 289467.66    (9) SKINLESS CHIX BREAST 345G
```

In [139…
```
raw_sp_5_df = raw_sp_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Invento

raw_sp_top5_df = raw_sp_5_df.sort_values(by='Total Base Amt', ascending=False).head(5)

raw_sp_top5_df['Total Base Amt'] = raw_sp_top5_df['Total Base Amt'].round(2)

print(raw_sp_top5_df)
```

```
                Total Base Amt             Inventory Desc
Inventory Code
1030.0                1768119.60  SP01 SKINLESS CHIX BREAST
1032.0                1130651.80        SP03 CHICKEN FILLET
1037.0                 798677.10      SP08 CHIX BONLESS LEG
1036.0                 762357.96    SP07 CHIX BONE IN THIGH
1033.0                 723603.76     SP04 CHICKEN DRUMSTICK
```

In [140…
```
rte_rt_df = rt_df[rt_df['Inventory Category'] == 'Ready to Eat (RTE)']
rtc_rt_df = rt_df[rt_df['Inventory Category'] == 'Ready to Cook (RTC)']
raw_rt_df = rt_df[rt_df['Inventory Category'] == 'Raw']
```

In [141…
```
rte_rt_5_df = rte_rt_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Invento

rte_rt_top5_df = rte_rt_5_df.sort_values(by='Total Base Amt', ascending=False).head(5)

rte_rt_top5_df['Total Base Amt'] = rte_rt_top5_df['Total Base Amt'].round(2)

print(rte_rt_top5_df)
```

```
                Total Base Amt                Inventory Desc
Inventory Code
1332.0                 15331.68          BETAGRO BREAST HERBS
1331.0                 15294.24  BETAGRO BREAST GARLIC PEPPER
1333.0                  7899.84      BETAGRO BREAST PINK SALT
```

In [142…
```
rtc_rt_5_df = rtc_rt_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Invento

rtc_rt_top5_df = rtc_rt_5_df.sort_values(by='Total Base Amt', ascending=False).head(5)

rtc_rt_top5_df['Total Base Amt'] = rtc_rt_top5_df['Total Base Amt'].round(2)

print(rtc_rt_top5_df)
```

```
                Total Base Amt                    Inventory Desc
Inventory Code
1155.0                171494.83  (99) S.LESS CHICKEN BREAST 1.15KG
1316.0                100504.30          GRILLED CHIX STEAK (5PKT)
1158.0                 79015.43  (33) CRISPY CHICKEN SEAWEED 1KG
1217.0                 74945.00                HONEY PORK RIBS
1218.0                 68487.12             HONEY CHAR SIEW 5KG
```

In [143…
```
raw_rt_5_df = raw_rt_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Invento

raw_rt_top5_df = raw_rt_5_df.sort_values(by='Total Base Amt', ascending=False).head(5)
```

```
raw_rt_top5_df['Total Base Amt'] = raw_rt_top5_df['Total Base Amt'].round(2)

print(raw_rt_top5_df)
```

```
                Total Base Amt             Inventory Desc
Inventory Code
1030.0                177624.72  SP01 SKINLESS CHIX BREAST
1036.0                 53557.92    SP07 CHIX BONE IN THIGH
1032.0                 43243.20        SP03 CHICKEN FILLET
1031.0                 21314.28           SP02 CHIX BREAST
```

Ready to Eat(RTE) Supermarket Cross PED and Price Optimization

In [144…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rte_sp_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rte_sp_df = rte_sp_df[rte_sp_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rte_sp_df = filtered_rte_sp_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [145…
```
rte_sp_df
```

Out[145]:

	Transaction Date	Inventory Code	Qty	Price Per Unit
0	2022-07-01	1331.0	2870.4	1.7
1	2022-07-01	1332.0	2641.6	1.7
2	2022-07-01	1333.0	988.0	1.7
3	2022-07-04	1331.0	3764.8	1.7
4	2022-07-04	1332.0	3504.8	1.7
…	…	…	…	…
824	2023-05-31	1330.0	2953.6	1.7
825	2023-05-31	1331.0	3151.2	1.7
826	2023-05-31	1332.0	3057.6	1.7
827	2023-05-31	1333.0	852.8	1.7
828	2023-05-31	1334.0	2849.6	1.7

829 rows × 4 columns

In [146…
```
#Using mode of each inventory code to replace missing data in dataframe

# Assuming your DataFrame is named rte_sp_df
# Step 1: Identify unique inventory codes for each transaction date
unique_inventory_codes = rte_sp_df.groupby('Transaction Date')['Inventory Code'].unique(

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = rte_sp_df.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(cal

# Step 3 and 4: Check and add missing rows
```

```
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(rte_sp_df['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
rte_sp_df = pd.concat([rte_sp_df, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
rte_sp_df = rte_sp_df.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
rte_sp_df.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(rte_sp_df)
```

```
     Transaction Date  Inventory Code     Qty  Price Per Unit
0          2022-07-01          1330.0  3016.0             1.7
1          2022-07-01          1331.0  2870.4             1.7
2          2022-07-01          1332.0  2641.6             1.7
3          2022-07-01          1333.0   988.0             1.7
4          2022-07-01          1334.0    20.8             1.7
...               ...             ...     ...             ...
1135       2023-05-31          1330.0  2953.6             1.7
1136       2023-05-31          1331.0  3151.2             1.7
1137       2023-05-31          1332.0  3057.6             1.7
1138       2023-05-31          1333.0   852.8             1.7
1139       2023-05-31          1334.0  2849.6             1.7

[1140 rows x 4 columns]
```

In [147…

```
# Create a list of unique inventory codes
unique_inventory_codes = rte_sp_df['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1330.0, 1331.0)
Inventory Code Pair: (1330.0, 1332.0)
Inventory Code Pair: (1330.0, 1333.0)
Inventory Code Pair: (1330.0, 1334.0)
Inventory Code Pair: (1331.0, 1332.0)
Inventory Code Pair: (1331.0, 1333.0)
Inventory Code Pair: (1331.0, 1334.0)
Inventory Code Pair: (1332.0, 1333.0)
Inventory Code Pair: (1332.0, 1334.0)
Inventory Code Pair: (1333.0, 1334.0)
```

In [148…

```
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rte_sp_df[rte_sp_df['Inventory Code'] == code1]
```

```
    data_code2 = rte_sp_df[rte_sp_df['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:      Transaction Date  Inventory Code     Qty  Price Per Unit
3           2022-07-01          1333.0   988.0             1.7
8           2022-07-04          1333.0  1372.8             1.7
13          2022-07-05          1333.0  1196.0             1.7
18          2022-07-06          1333.0   717.6             1.7
23          2022-07-07          1333.0  1092.0             1.7
...                ...             ...     ...             ...
1118        2023-05-25          1333.0  1123.2             1.7
1123        2023-05-26          1333.0  1008.8             1.7
1128        2023-05-29          1333.0  1622.4             1.7
1133        2023-05-30          1333.0  1237.6             1.7
1138        2023-05-31          1333.0   852.8             1.7

[228 rows x 4 columns]
Data for Code Pair 2:      Transaction Date  Inventory Code     Qty  Price Per Unit
4           2022-07-01          1334.0    20.8             1.7
9           2022-07-04          1334.0    20.8             1.7
14          2022-07-05          1334.0    20.8             1.7
19          2022-07-06          1334.0    20.8             1.7
24          2022-07-07          1334.0    20.8             1.7
...                ...             ...     ...             ...
1119        2023-05-25          1334.0    41.6             1.7
1124        2023-05-26          1334.0    83.2             1.7
1129        2023-05-29          1334.0   447.2             1.7
1134        2023-05-30          1334.0  3203.2             1.7
1139        2023-05-31          1334.0  2849.6             1.7

[228 rows x 4 columns]
```

In [149…

```
#Creating the cross reference for each pair on a daily basis
# Create an empty DataFrame to store the results
rte_sp_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1',

# Create a list of unique inventory codes
unique_inventory_codes = rte_sp_df['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rte_sp_df[rte_sp_df['Inventory Code'] == code1]
    data_code2 = rte_sp_df[rte_sp_df['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    rte_sp_elasticity_df = pd.concat([rte_sp_elasticity_df, results])

# Print or inspect the elasticity results
print(rte_sp_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1330.0                1331.0
1         2022-07-04                1330.0                1331.0
2         2022-07-05                1330.0                1331.0
```

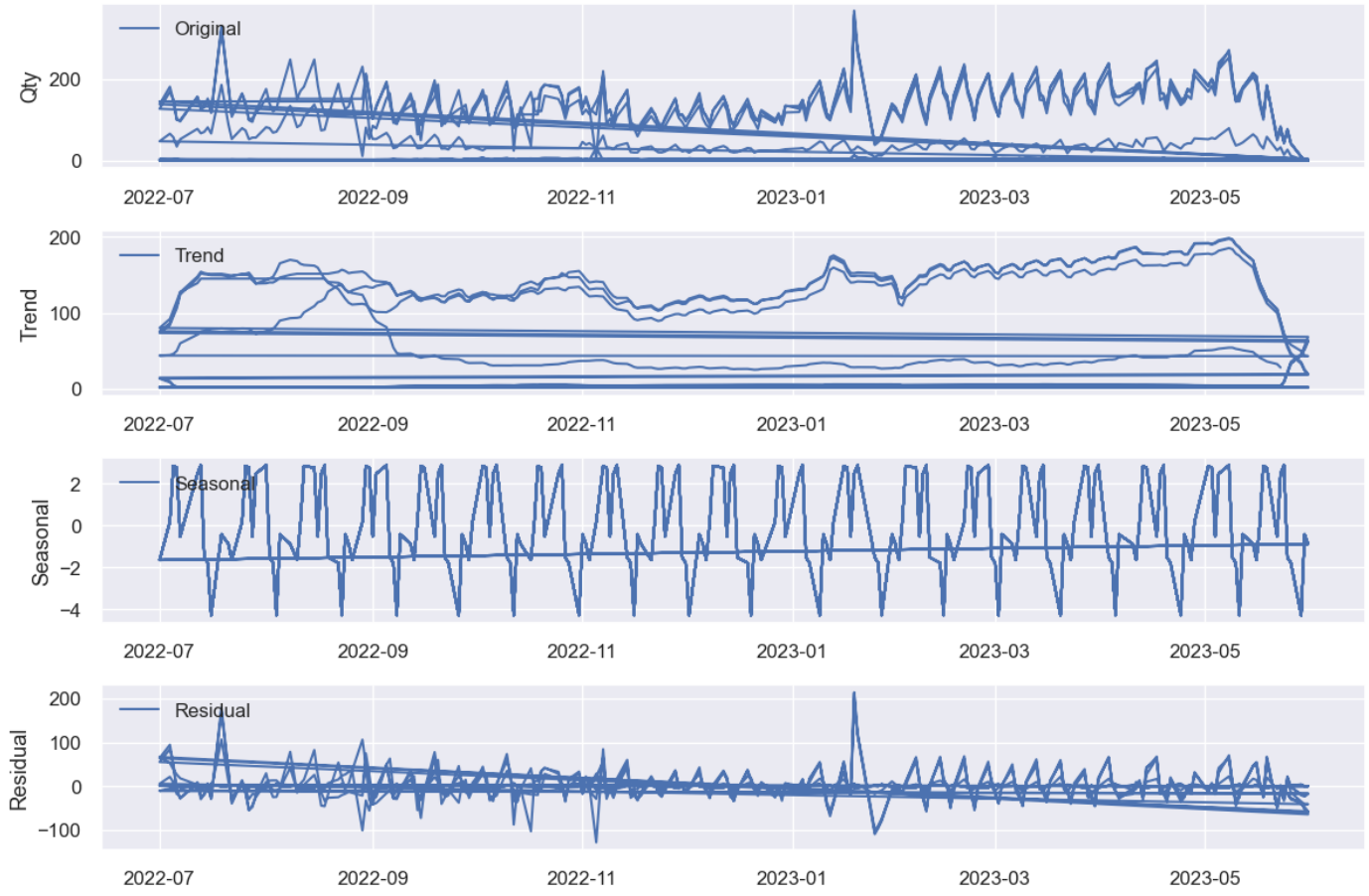
```
3          2022-07-06                1330.0                1331.0
4          2022-07-07                1330.0                1331.0
..                ...                   ...                   ...
223        2023-05-25                1333.0                1334.0
224        2023-05-26                1333.0                1334.0
225        2023-05-29                1333.0                1334.0
226        2023-05-30                1333.0                1334.0
227        2023-05-31                1333.0                1334.0

     Relative_Price  Relative_Volume
0               1.0         1.050725
1               1.0         0.801105
2               1.0         0.979730
3               1.0         1.457286
4               1.0         1.457286
..              ...              ...
223             1.0        27.000000
224             1.0        12.125000
225             1.0         3.627907
226             1.0         0.386364
227             1.0         0.299270

[2280 rows x 5 columns]
```

In [150…

```
## Creating Log Relative Price and Log Relative Volume in columns

rte_sp_elasticity_df.loc[:, 'Log Relative Price'] = np.log(rte_sp_elasticity_df['Relativ
rte_sp_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(rte_sp_elasticity_df['Relati
```

In [151…

```
#Seasonality
rte_sp_elasticity_df.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rte_sp_elasticity_df['Relative_Volume']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [152…

```
#Copying Seasonal Value to Data Frame and resetting the index

rte_sp_elasticity_df = rte_sp_elasticity_df.copy()  # Create a copy of the DataFrame
rte_sp_elasticity_df['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the c

rte_sp_elasticity_df.reset_index(inplace=True)
```

In [153…

```
## Setting Weekend and Holiday Binary Values

# Convert 'Transaction Date' to datetime64 format
rte_sp_elasticity_df['Transaction Date'] = pd.to_datetime(rte_sp_elasticity_df['Transact

# Convert 'Transaction Date' to date format
rte_sp_elasticity_df['Transaction Date'] = rte_sp_elasticity_df['Transaction Date'].dt.d

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rte_sp_elasticity_df['IsWeekend'] = (rte_sp_elasticity_df['Transaction Date'].apply(lamb
rte_sp_elasticity_df['IsHoliday'] = rte_sp_elasticity_df['Transaction Date'].isin(holida
```

In [154…

```
# Group the data by unique product pairs
unique_product_pairs = rte_sp_elasticity_df[['Inventory Code_code1', 'Inventory Code_cod

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = rte_sp_elasticity_df[(rte_sp_elasticity_df['Inventory Code_code1'] == inven

    X = sm.add_constant(subset[['Log Relative Price', 'Seasonal', 'IsWeekend', 'IsHolida
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        coefficients_seasonal.append(model.params['Seasonal'])
        p_values_seasonal.append(model.pvalues['Seasonal'])
        coefficients_is_weekend.append(model.params['IsWeekend'])
        p_values_is_weekend.append(model.pvalues['IsWeekend'])
        coefficients_is_holiday.append(model.params['IsHoliday'])
        p_values_is_holiday.append(model.pvalues['IsHoliday'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

rte_sp_table_df = pd.DataFrame(table_data)

rte_sp_table_df['Type'] = rte_sp_table_df['Coefficient (Log Relative Price)'].apply(lamb

rte_sp_table_df['Significance (Log Relative Price)'] = rte_sp_table_df['P-Value (Log Rel
rte_sp_table_df['Significance (Seasonal)'] = rte_sp_table_df['P-Value (Seasonal)'].apply
rte_sp_table_df['Significance (IsWeekend)'] = rte_sp_table_df['P-Value (IsWeekend)'].app
rte_sp_table_df['Significance (IsHoliday)'] = rte_sp_table_df['P-Value (IsHoliday)'].app

# Create a DataFrame to store the OLS results
rte_sp_results_df = pd.DataFrame({
```

```
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(rte_sp_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)	
0	(1330.0, 1331.0)	0.0	NaN	-0.001664	0.852613	1.929158	3.433447e-17	0.0	NaN	Cc
1	(1330.0, 1332.0)	0.0	NaN	-0.015755	0.039742	0.277588	1.231948e-01	0.0	NaN	Cc
2	(1330.0, 1333.0)	0.0	NaN	0.000060	0.996959	0.004905	9.894077e-01	0.0	NaN	Cc
3	(1330.0, 1334.0)	0.0	NaN	0.026031	0.149958	0.255246	5.481608e-01	0.0	NaN	Cc
4	(1331.0, 1332.0)	0.0	NaN	-0.014091	0.034661	-1.651570	1.759485e-21	0.0	NaN	Cc
5	(1331.0, 1333.0)	0.0	NaN	0.001724	0.900770	-1.924252	1.220816e-08	0.0	NaN	Cc
6	(1331.0, 1334.0)	0.0	NaN	0.027695	0.127694	-1.673911	1.159823e-04	0.0	NaN	Cc
7	(1332.0, 1333.0)	0.0	NaN	0.015815	0.266153	-0.272682	4.153346e-01	0.0	NaN	Cc
8	(1332.0, 1334.0)	0.0	NaN	0.041786	0.019309	-0.022342	9.573859e-01	0.0	NaN	Cc
9	(1333.0, 1334.0)	0.0	NaN	0.025971	0.219668	0.250341	6.149385e-01	0.0	NaN	Cc

In [155…

```
# Initialize lists to store the optimization results
optimized_prices_list = []
maximized_revenue_list = []

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Iterate over rows in the DataFrame
for _, row in rte_sp_table_df.iterrows():
    # Extract the cross-price elasticity coefficient from the table
    cross_price_elasticity_coefficient = row['Coefficient (Log Relative Price)']

    # Initial guess for prices (you may need to adjust this based on your specific conte
    initial_prices = [1.0, 1.0]  # Assuming you are optimizing prices for a pair of prod

    # Define bounds on prices
    bounds = [(0, None), (0, None)]  # Separate bounds for each item in the pair
```

```
    # Run the optimization
    result = minimize(objective_function, initial_prices, args=(cross_price_elasticity_c

    # Extract the optimized prices and maximized revenue
    optimized_prices = result.x.tolist()
    maximized_revenue = -result.fun  # Negate to get the actual maximum revenue

    # Append the results to the lists
    optimized_prices_list.append(optimized_prices)
    maximized_revenue_list.append(maximized_revenue)

# Add new columns to the DataFrame
rte_sp_table_df['Optimized Prices'] = optimized_prices_list
rte_sp_table_df['Maximized Revenue'] = maximized_revenue_list

# Display the updated DataFrame
display(rte_sp_table_df[['Inventory Code Pair', 'Optimized Prices', 'Maximized Revenue']
```

	Inventory Code Pair	Optimized Prices	Maximized Revenue
0	(1330.0, 1331.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
1	(1330.0, 1332.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
2	(1330.0, 1333.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
3	(1330.0, 1334.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
4	(1331.0, 1332.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
5	(1331.0, 1333.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
6	(1331.0, 1334.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
7	(1332.0, 1333.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
8	(1332.0, 1334.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
9	(1333.0, 1334.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf

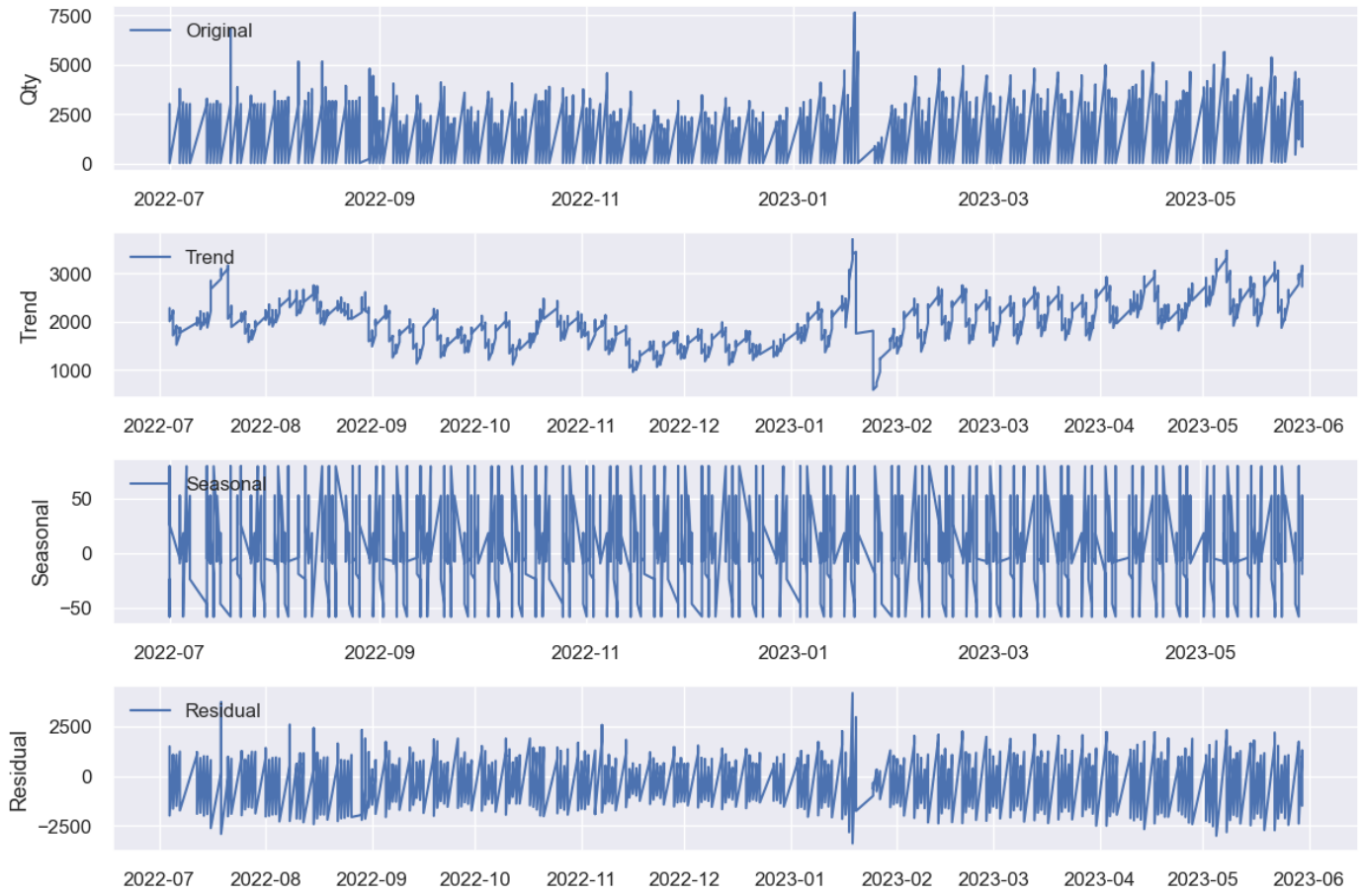
Method 2 - Seperate Cross Price in 2 Entries

In [156…
```
rte_sp_df2 = rte_sp_df
```

In [157…
```
## Creating Log Relative Price and Log Relative Volume in columns

rte_sp_df2.loc[:, 'Log Price'] = np.log(rte_sp_df2['Price Per Unit'])
rte_sp_df2.loc[:, 'Log Qty'] = np.log(rte_sp_df2['Qty'])
```

In [158…
```
rte_sp_df2.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rte_sp_df2['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
```

```
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [159…

```
#Copying Seasonal Value to Data Frame and resetting the index

rte_sp_df2 = rte_sp_df2.copy()  # Create a copy of the DataFrame
rte_sp_df2['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rte_sp_df2.reset_index(inplace=True)
```

In [160…

```
rte_sp_df2['Transaction Date'] = pd.to_datetime(rte_sp_df2['Transaction Date'])

# Convert 'Transaction Date' to date format
rte_sp_df2['Transaction Date'] = rte_sp_df2['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
```

```
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rte_sp_df2['IsWeekend'] = (rte_sp_df2['Transaction Date'].apply(lambda x: x.weekday()) >
rte_sp_df2['IsHoliday'] = rte_sp_df2['Transaction Date'].isin(holiday_dates).astype(int)
```

In [161…

```
# Assuming you have separate demand functions for each product
products = rte_sp_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rte_sp_df2[(rte_sp_df2['Inventory Code'] == products[i]) | (rte_sp_d

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_1.params['Seasonal'],
            'P-Value (Seasonality)': model_1.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_1.params['IsWeekend'],
            'P-Value (IsWeekend)': model_1.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_1.params['IsHoliday'],
            'P-Value (IsHoliday)': model_1.pvalues['IsHoliday'],
        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_2.params['Seasonal'],
            'P-Value (Seasonality)': model_2.pvalues['Seasonal'],
```

```
            'Coefficient (IsWeekend)': model_2.params['IsWeekend'],
            'P-Value (IsWeekend)': model_2.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_2.params['IsHoliday'],
            'P-Value (IsHoliday)': model_2.pvalues['IsHoliday'],
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[161]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonality)	P-Value (Seasonality)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)
0	1330.0 - 1331.0	1330.0	14.877901	1.911974e-306	0.000085	0.891427	0.117685	6.376411e-01	0.0
1	1330.0 - 1331.0	1331.0	15.053695	4.355655e-309	0.000402	0.512887	-1.817651	2.721737e-12	0.0
2	1330.0 - 1332.0	1330.0	14.877901	1.911974e-306	0.000085	0.891427	0.117685	6.376411e-01	0.0
3	1330.0 - 1332.0	1332.0	15.029312	4.691139e-313	0.001096	0.063960	-0.180154	4.452905e-01	0.0
4	1330.0 - 1333.0	1330.0	14.877901	1.911974e-306	0.000085	0.891427	0.117685	6.376411e-01	0.0
5	1330.0 - 1333.0	1333.0	12.683422	9.024150e-254	0.000239	0.793743	0.108877	7.660896e-01	0.0
6	1330.0 - 1334.0	1330.0	14.877901	1.911974e-306	0.000085	0.891427	0.117685	6.376411e-01	0.0
7	1330.0 - 1334.0	1334.0	5.878520	4.250490e-176	-0.000632	0.506380	-0.102130	7.884391e-01	0.0
8	1331.0 - 1332.0	1331.0	15.053695	4.355655e-309	0.000402	0.512887	-1.817651	2.721737e-12	0.0
9	1331.0 - 1332.0	1332.0	15.029312	4.691139e-313	0.001096	0.063960	-0.180154	4.452905e-01	0.0
10	1331.0 - 1333.0	1331.0	15.053695	4.355655e-309	0.000402	0.512887	-1.817651	2.721737e-12	0.0
11	1331.0 - 1333.0	1333.0	12.683422	9.024150e-254	0.000239	0.793743	0.108877	7.660896e-01	0.0
12	1331.0 - 1334.0	1331.0	15.053695	4.355655e-309	0.000402	0.512887	-1.817651	2.721737e-12	0.0
13	1331.0	1334.0	5.878520	4.250490e-	-0.000632	0.506380	-0.102130	7.884391e-	0.0

	1334.0			176				01	
14	1332.0 - 1333.0	1332.0	15.029312	4.691139e-313	0.001096	0.063960	-0.180154	4.452905e-01	0.0
15	1332.0 - 1333.0	1333.0	12.683422	9.024150e-254	0.000239	0.793743	0.108877	7.660896e-01	0.0
16	1332.0 - 1334.0	1332.0	15.029312	4.691139e-313	0.001096	0.063960	-0.180154	4.452905e-01	0.0
17	1332.0 - 1334.0	1334.0	5.878520	4.250490e-176	-0.000632	0.506380	-0.102130	7.884391e-01	0.0
18	1333.0 - 1334.0	1333.0	12.683422	9.024150e-254	0.000239	0.793743	0.108877	7.660896e-01	0.0
19	1333.0 - 1334.0	1334.0	5.878520	4.250490e-176	-0.000632	0.506380	-0.102130	7.884391e-01	0.0

In [162…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rte_sp_df.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rte_sp_min_max_data2 = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rte_sp_min_max_data2.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', '

# Display the result
print(rte_sp_min_max_data2)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1330.0        1.7        1.7    239.2   7415.2
1          1331.0        1.7        1.7     93.6   6999.2
2          1332.0        1.7        1.7    509.6   7633.6
3          1333.0        1.7        1.7    187.2   4784.0
4          1334.0        1.7        1.7     20.8   3203.2
```

In [163…

```
# Assuming you have separate demand functions for each product
products = rte_sp_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Initialize lists to store mean price and quantity
mean_price_list = []
mean_quantity_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rte_sp_df2[(rte_sp_df2['Inventory Code'] == products[i]) | (rte_sp_d
```

```
        # Calculate mean price and quantity for each product in the pair
        mean_price_i = pair_data[pair_data['Inventory Code'] == products[i]]['Price Per
        mean_quantity_i = pair_data[pair_data['Inventory Code'] == products[i]]['Qty'].m

        mean_price_j = pair_data[pair_data['Inventory Code'] == products[j]]['Price Per
        mean_quantity_j = pair_data[pair_data['Inventory Code'] == products[j]]['Qty'].m

        # Store the mean price and quantity in the lists
        mean_price_list.extend([mean_price_i, mean_price_j])
        mean_quantity_list.extend([mean_quantity_i, mean_quantity_j])

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Mean Price': mean_price_i,
            'Mean Quantity': mean_quantity_i,

        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Mean Price': mean_price_j,
            'Mean Quantity': mean_quantity_j,

        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[163]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Mean Price	Mean Quantity
0	1330.0 - 1331.0	1330.0	14.877901	1.911974e-306	1.7	2830.259649
1	1330.0 - 1331.0	1331.0	15.053695	4.355655e-309	1.7	3066.221053
2	1330.0 - 1332.0	1330.0	14.877901	1.911974e-306	1.7	2830.259649
3	1330.0 - 1332.0	1332.0	15.029312	4.691139e-313	1.7	3056.870175
4	1330.0 - 1333.0	1330.0	14.877901	1.911974e-306	1.7	2830.259649

5	1330.0 - 1333.0	1333.0	12.683422	9.024150e-254	1.7	973.266667
6	1330.0 - 1334.0	1330.0	14.877901	1.911974e-306	1.7	2830.259649
7	1330.0 - 1334.0	1334.0	5.878520	4.250490e-176	1.7	50.084211
8	1331.0 - 1332.0	1331.0	15.053695	4.355655e-309	1.7	3066.221053
9	1331.0 - 1332.0	1332.0	15.029312	4.691139e-313	1.7	3056.870175
10	1331.0 - 1333.0	1331.0	15.053695	4.355655e-309	1.7	3066.221053
11	1331.0 - 1333.0	1333.0	12.683422	9.024150e-254	1.7	973.266667
12	1331.0 - 1334.0	1331.0	15.053695	4.355655e-309	1.7	3066.221053
13	1331.0 - 1334.0	1334.0	5.878520	4.250490e-176	1.7	50.084211
14	1332.0 - 1333.0	1332.0	15.029312	4.691139e-313	1.7	3056.870175
15	1332.0 - 1333.0	1333.0	12.683422	9.024150e-254	1.7	973.266667
16	1332.0 - 1334.0	1332.0	15.029312	4.691139e-313	1.7	3056.870175
17	1332.0 - 1334.0	1334.0	5.878520	4.250490e-176	1.7	50.084211
18	1333.0 - 1334.0	1333.0	12.683422	9.024150e-254	1.7	973.266667
19	1333.0 - 1334.0	1334.0	5.878520	4.250490e-176	1.7	50.084211

In [164…

```
#No bounds
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in results_df
for index, row in results_df.iterrows():
    inventory_code = row['Product']
    mean_price = row['Mean Price']
    mean_quantity = row['Mean Quantity']
    elasticity = row['Coefficient (Log Price)']

    # Set bounds for optimization
    bounds = [(0, None)]  # Optimize only price

    # Perform optimization
    result = minimize(calculate_revenue, x0=[mean_price], args=(elasticity, mean_quantit

    # Extract optimized price
    optimized_price = result.x[0]

    optimized_prices.append(optimized_price)

# Add optimized prices to the results DataFrame
results_df['Optimized Price'] = optimized_prices

# Display the updated results_df
print(results_df[['Pair', 'Product', 'Optimized Price']])
```

```
               Pair  Product  Optimized Price
0   1330.0 - 1331.0  1330.0              0.0
1   1330.0 - 1331.0  1331.0              0.0
2   1330.0 - 1332.0  1330.0              0.0
3   1330.0 - 1332.0  1332.0              0.0
4   1330.0 - 1333.0  1330.0              0.0
5   1330.0 - 1333.0  1333.0              0.0
6   1330.0 - 1334.0  1330.0              0.0
```

```
7   1330.0 - 1334.0  1334.0              0.0
8   1331.0 - 1332.0  1331.0              0.0
9   1331.0 - 1332.0  1332.0              0.0
10  1331.0 - 1333.0  1331.0              0.0
11  1331.0 - 1333.0  1333.0              0.0
12  1331.0 - 1334.0  1331.0              0.0
13  1331.0 - 1334.0  1334.0              0.0
14  1332.0 - 1333.0  1332.0              0.0
15  1332.0 - 1333.0  1333.0              0.0
16  1332.0 - 1334.0  1332.0              0.0
17  1332.0 - 1334.0  1334.0              0.0
18  1333.0 - 1334.0  1333.0              0.0
19  1333.0 - 1334.0  1334.0              0.0
```

Self Price Elasticity Ready To Eat Supermarket

In [165…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rte_sp_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rte_sp_df = rte_sp_df[rte_sp_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rte_sp_df = filtered_rte_sp_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

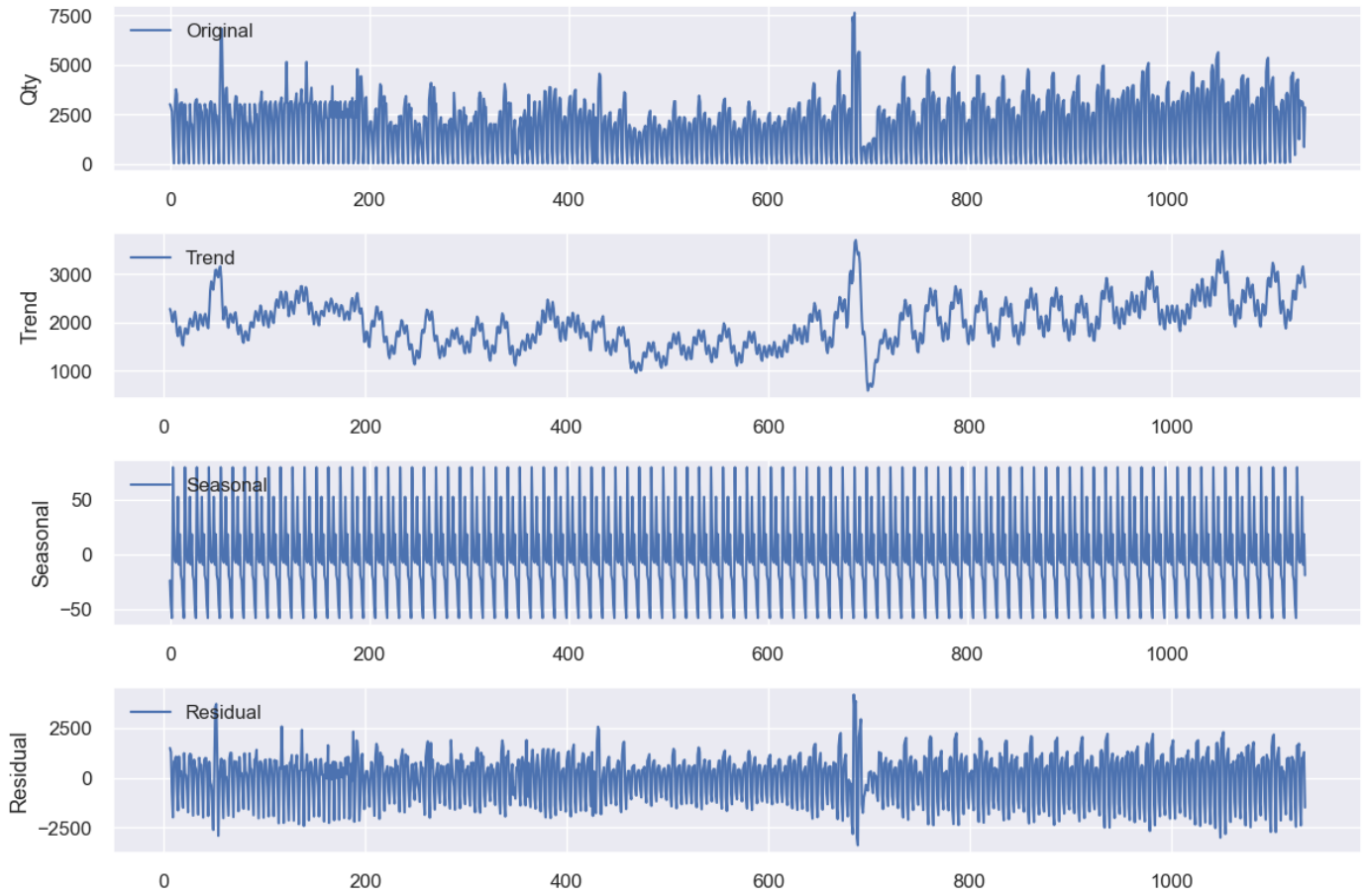
In [166…
```
rte_sp_df3 = rte_sp_df
```

In [167…
```
## Creating Log Price and Log Quantity in columns

rte_sp_df3.loc[:, 'Log Price'] = np.log(rte_sp_df3['Price Per Unit'])
rte_sp_df3.loc[:, 'Log Qty'] = np.log(rte_sp_df3['Qty'])
```

In [168…
```
rte_sp_df3.reset_index(inplace=True)
```

In [169…
```
# Perform seasonal decomposition
time_series = rte_sp_df3['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')
```

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



In [170…

```
#Copying Seasonal Value to Data Frame and resetting the index

rte_sp_df3 = rte_sp_df3.copy()  # Create a copy of the DataFrame
rte_sp_df3['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rte_sp_df3.reset_index(inplace=True)
```

In [171…

```
## Setting Weekend and Holiday Binary Values

rte_sp_df3['Transaction Date'] = pd.to_datetime(rte_sp_df3['Transaction Date'])

# Convert 'Transaction Date' to date format
rte_sp_df3['Transaction Date'] = rte_sp_df3['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
```

```
rte_sp_df3['IsWeekend'] = (rte_sp_df3['Transaction Date'].apply(lambda x: x.weekday()) >
rte_sp_df3['IsHoliday'] = rte_sp_df3['Transaction Date'].isin(holiday_dates).astype(int)
```

In [172…

```
# Linear Regression for Self Price PED RTE Supermarket

# Get unique product codes from the 'Inventory Code' column
unique_inventory = rte_sp_df3['Inventory Code'].unique()

# Initialize lists to store results
inventory_codes = []
coefficients_price = []
p_values_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []

# Loop through the unique product codes
for inventory_code in unique_inventory:
    inventory_data = rte_sp_df3[rte_sp_df3['Inventory Code'] == inventory_code]

    # Create features and target variables
    X = sm.add_constant(inventory_data[['Log Price', 'IsWeekend', 'IsHoliday', 'Seasonal

    model = sm.OLS(inventory_data['Log Qty'], X).fit()

    # Append results to lists
    inventory_codes.append(inventory_code)
    coefficients_price.append(model.params['Log Price'])
    p_values_price.append(model.pvalues['Log Price'])
    coefficients_seasonal.append(model.params['Seasonal'])
    p_values_seasonal.append(model.pvalues['Seasonal'])
    coefficients_is_weekend.append(model.params['IsWeekend'])
    p_values_is_weekend.append(model.pvalues['IsWeekend'])
    coefficients_is_holiday.append(model.params['IsHoliday'])
    p_values_is_holiday.append(model.pvalues['IsHoliday'])

# Create a DataFrame with the aggregated coefficients and p-values
table_data = {
    'Inventory Code': inventory_codes,
    'Coefficient (Log Price)': coefficients_price,
    'P-Value (Log Price)': p_values_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

own_price_rte_sp_df = pd.DataFrame(table_data)

# Apply the significance labels
for column in own_price_rte_sp_df.columns:
    if 'P-Value' in column:
        significance_column = 'Significance' + column.split('P-Value', 1)[1]
        own_price_rte_sp_df[significance_column] = own_price_rte_sp_df[column].apply(lam

# Display the table in the Jupyter Notebook
print("Ready to Eat Supermarket Aggregated Coefficients and Significance Table:")
display(own_price_rte_sp_df)
```

```
Ready to Eat Supermarket Aggregated Coefficients and Significance Table:
```

	Inventory Code	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	1330.0	14.877901	1.911974e-306	0.000085	0.891427	0.117685	6.376411e-01	0.0	NaN
1	1331.0	15.053695	4.355655e-309	0.000402	0.512887	-1.817651	2.721737e-12	0.0	NaN
2	1332.0	15.029312	4.691139e-313	0.001096	0.063960	-0.180154	4.452905e-01	0.0	NaN
3	1333.0	12.683422	9.024150e-254	0.000239	0.793743	0.108877	7.660896e-01	0.0	NaN
4	1334.0	5.878520	4.250490e-176	-0.000632	0.506380	-0.102130	7.884391e-01	0.0	NaN

In [173…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rte_sp_df.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rte_sp_min_max_data = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rte_sp_min_max_data.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', 'M

# Display the result
print(rte_sp_min_max_data)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1330.0        1.7        1.7    239.2   7415.2
1          1331.0        1.7        1.7     93.6   6999.2
2          1332.0        1.7        1.7    509.6   7633.6
3          1333.0        1.7        1.7    187.2   4784.0
4          1334.0        1.7        1.7     20.8   3203.2
```

In [174…

```
## Price Optimization Using Min & Max Qty & Price as bounds

from scipy.optimize import minimize

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rte_sp_df, rte_sp_min_max_data, on='Inventory Code')

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, qty, coefficient_price):
    return -price * qty * coefficient_price  # Negative sign because scipy.optimize mini

# Function to be minimized (negative revenue for maximization)
def objective_function(prices):
    total_revenue = 0
    for index, row in merged_data.iterrows():
        min_price, max_price = row['Min Price'], row['Max Price']
        min_qty, max_qty = row['Min Qty'], row['Max Qty']
        coefficient_price = row['Coefficient (Log Price)']

        # Ensure prices are within bounds
        price = max(min(max_price, prices[index]), min_price)

        # Calculate revenue
```

```
        qty = max(min(max_qty, prices[len(merged_data) + index]), min_qty)
        total_revenue += calculate_revenue(price, qty, coefficient_price)

    return total_revenue

# Initial guess for prices
initial_prices = merged_data['Min Price'].values.tolist() + merged_data['Min Qty'].value

# Constraints
price_constraints = [(min_price, max_price) for min_price, max_price in zip(merged_data[
qty_constraints = [(min_qty, max_qty) for min_qty, max_qty in zip(merged_data['Min Qty']
constraints = price_constraints + qty_constraints

# Optimization
result = minimize(objective_function, initial_prices, constraints={'type': 'ineq', 'fun'
                  bounds=constraints)

# Extract optimized prices and quantities
optimized_prices = result.x[:len(merged_data)]
optimized_quantities = result.x[len(merged_data):]

# Display optimized prices and quantities
optimized_data = merged_data.copy()
optimized_data['Optimized Price'] = optimized_prices
optimized_data['Optimized Quantity'] = optimized_quantities

print("Optimized Prices and Quantities:")
print(optimized_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Optimized Prices and Quantities:
   Inventory Code  Optimized Price  Optimized Quantity
0          1330.0              1.7              7415.2
1          1331.0              1.7              6999.2
2          1332.0              1.7              7633.6
3          1333.0              1.7              4784.0
4          1334.0              1.7              3203.2
```

In [175…

```
## Price Optimization Using Min Price as bounds

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rte_sp_df, rte_sp_min_max_data, on='Inventory Code')

# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in merged_data
for index, row in merged_data.iterrows():
    inventory_code = row['Inventory Code']
    min_price = row['Min Price']

    # Get elasticity from own_price_rte_sp_df based on the inventory code
    elasticity = own_price_rte_sp_df.loc[own_price_rte_sp_df['Inventory Code'] == invent

    # Set bounds for optimization
    bounds = [(min_price, None), (None, None)]  # Remove lower bound for quantity
```

```
    # Perform optimization
    result = minimize(objective_function, x0=[min_price, row['Min Qty']], args=(elastici

    # Extract optimized price and quantity
    optimized_price, optimized_quantity = result.x

    # Append results to lists
    optimized_prices.append(optimized_price)
    optimized_quantities.append(optimized_quantity)

# Add optimized prices and quantities to the DataFrame
merged_data['Optimized Price'] = optimized_prices
merged_data['Optimized Quantity'] = optimized_quantities

# Display the DataFrame
print(merged_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

	Inventory Code	Optimized Price	Optimized Quantity
0	1330.0	1.085237e+148	1.549028e+149
1	1331.0	5.686708e+147	7.076480e+148
2	1332.0	2.476027e+148	3.768191e+149
3	1333.0	1.240861e+147	1.511198e+148
4	1334.0	7.684547e+153	1.724754e+153

Ready to Cook (RTC) Supermarket Cross PED and Price Optimization

In [176…

```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rtc_sp_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rtc_sp_df = rtc_sp_df[rtc_sp_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rtc_sp_df = filtered_rtc_sp_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [177…

```
#Using mode of each inventory code to replace missing data in dataframe

import pandas as pd

# Assuming your DataFrame is named rte_sp_df
# Step 1: Identify unique inventory codes for each transaction date
unique_inventory_codes = rtc_sp_df.groupby('Transaction Date')['Inventory Code'].unique(

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = rtc_sp_df.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(cal

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(rtc_sp_df['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
```

```
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
rtc_sp_df = pd.concat([rtc_sp_df, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
rtc_sp_df = rtc_sp_df.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
rtc_sp_df.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(rtc_sp_df)
```

```
     Transaction Date  Inventory Code    Qty  Price Per Unit
0          2022-07-01          1158.0  143.0       11.330000
1          2022-07-01          1160.0  171.6        5.275000
2          2022-07-01          1162.0  546.0        5.177778
3          2022-07-01          1163.0  249.6        5.228571
4          2022-07-01          1166.0   78.0        5.280000
...               ...             ...    ...             ...
1330       2023-05-31          1158.0   78.0       11.700000
1331       2023-05-31          1160.0   15.6        5.100000
1332       2023-05-31          1162.0   31.2        5.520000
1333       2023-05-31          1163.0   46.8        5.520000
1334       2023-05-31          1166.0   15.6        5.520000

[1335 rows x 4 columns]
```

In [178…

```
# Create a list of unique inventory codes
unique_inventory_codes = rtc_sp_df['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1158.0, 1160.0)
Inventory Code Pair: (1158.0, 1162.0)
Inventory Code Pair: (1158.0, 1163.0)
Inventory Code Pair: (1158.0, 1166.0)
Inventory Code Pair: (1160.0, 1162.0)
Inventory Code Pair: (1160.0, 1163.0)
Inventory Code Pair: (1160.0, 1166.0)
Inventory Code Pair: (1162.0, 1163.0)
Inventory Code Pair: (1162.0, 1166.0)
Inventory Code Pair: (1163.0, 1166.0)
```

In [179…

```
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rtc_sp_df[rtc_sp_df['Inventory Code'] == code1]
    data_code2 = rtc_sp_df[rtc_sp_df['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:      Transaction Date  Inventory Code     Qty  Price Per Unit
3          2022-07-01          1163.0   249.6        5.228571
8          2022-07-02          1163.0    62.4        5.350000
13         2022-07-04          1163.0  1263.6        5.350000
```

```
18          2022-07-05          1163.0    31.2        5.520000
23          2022-07-06          1163.0    93.6        5.350000
...                ...             ...     ...             ...
1313        2023-05-26          1163.0    15.6        5.500000
1318        2023-05-27          1163.0    31.2        5.520000
1323        2023-05-29          1163.0  1092.0        5.900000
1328        2023-05-30          1163.0    62.4        5.520000
1333        2023-05-31          1163.0    46.8        5.520000

[267 rows x 4 columns]
Data for Code Pair 2:      Transaction Date  Inventory Code     Qty  Price Per Unit
4           2022-07-01          1166.0    78.0           5.280
9           2022-07-02          1166.0    78.0           5.300
14          2022-07-04          1166.0  1107.6           5.425
19          2022-07-05          1166.0    46.8           5.350
24          2022-07-06          1166.0    78.0           5.350
...                ...             ...     ...             ...
1314        2023-05-26          1166.0    78.0           5.516
1319        2023-05-27          1166.0    15.6           5.520
1324        2023-05-29          1166.0  1092.0           5.900
1329        2023-05-30          1166.0    31.2           5.520
1334        2023-05-31          1166.0    15.6           5.520

[267 rows x 4 columns]
```

In [180…

```python
#Creating the cross reference for each pair on a daily basis

# Create an empty DataFrame to store the results
rtc_sp_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1',

# Create a list of unique inventory codes
unique_inventory_codes = rtc_sp_df['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rtc_sp_df[rtc_sp_df['Inventory Code'] == code1]
    data_code2 = rtc_sp_df[rtc_sp_df['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    rtc_sp_elasticity_df = pd.concat([rtc_sp_elasticity_df, results])

# Print or inspect the elasticity results
print(rtc_sp_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1158.0                1160.0
1         2022-07-02                1158.0                1160.0
2         2022-07-04                1158.0                1160.0
3         2022-07-05                1158.0                1160.0
4         2022-07-06                1158.0                1160.0
..               ...                   ...                   ...
262       2023-05-26                1163.0                1166.0
263       2023-05-27                1163.0                1166.0
264       2023-05-29                1163.0                1166.0
265       2023-05-30                1163.0                1166.0
266       2023-05-31                1163.0                1166.0
```

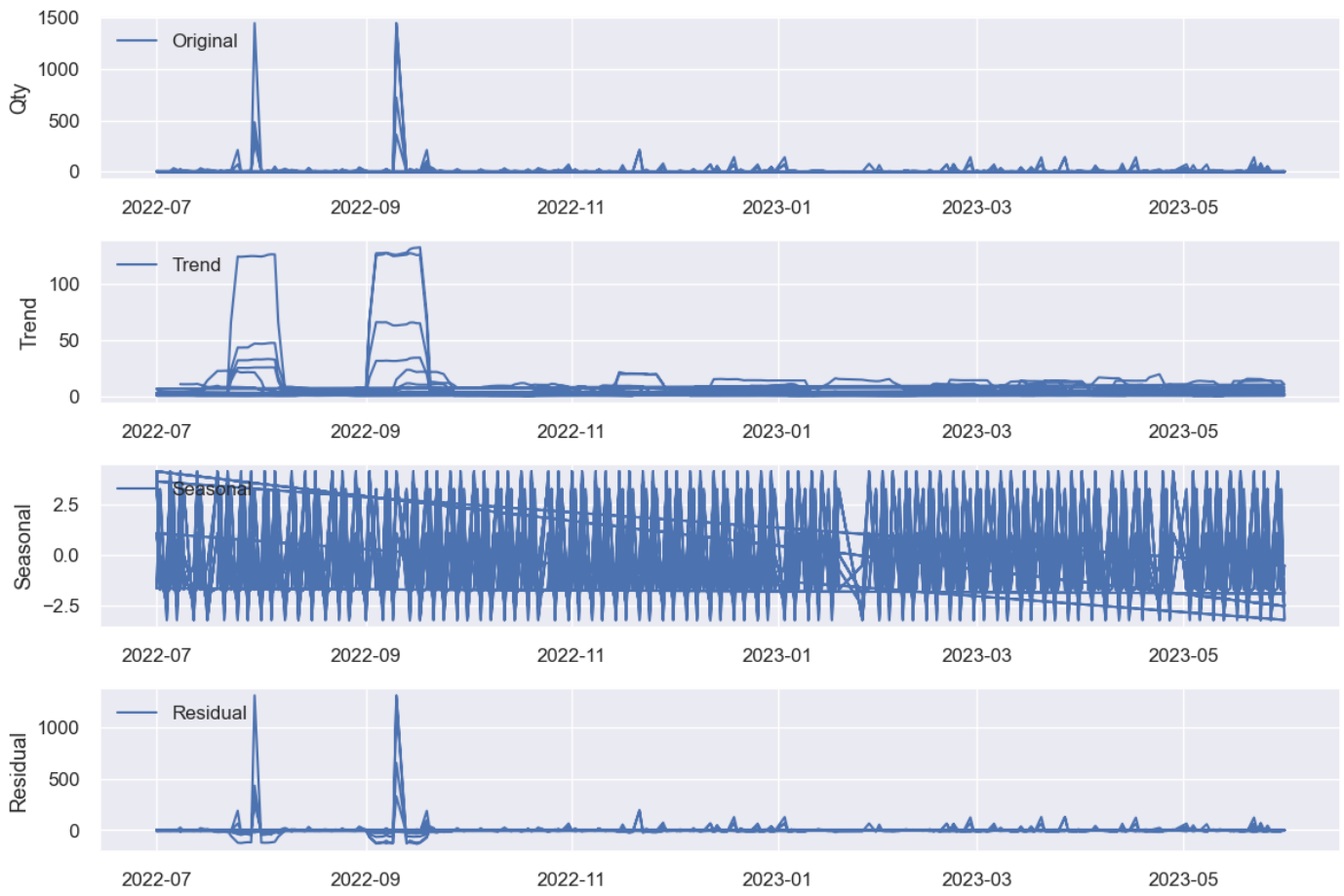
```
     Relative_Price  Relative_Volume
0          2.147867         0.833333
1          2.117757         3.333333
2          1.942056         0.541667
3          2.221569         5.000000
4          2.281947        34.166667
..              ...              ...
262        0.997099         0.200000
263        1.000000         2.000000
264        1.000000         1.000000
265        1.000000         2.000000
266        1.000000         3.000000

[2670 rows x 5 columns]
```

In [181…

```
## Creating Log Relative Price and Log Relative Volume in columns

rtc_sp_elasticity_df.loc[:, 'Log Relative Price'] = np.log(rtc_sp_elasticity_df['Relativ
rtc_sp_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(rtc_sp_elasticity_df['Relati
```

In [182…

```
#Seasonality
rtc_sp_elasticity_df.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rtc_sp_elasticity_df['Relative_Volume']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [183…

```
#Copying Seasonal Value to Data Frame and resetting the index

rtc_sp_elasticity_df = rtc_sp_elasticity_df.copy()  # Create a copy of the DataFrame
rtc_sp_elasticity_df['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the c

rtc_sp_elasticity_df.reset_index(inplace=True)
```

In [184…

```
## Setting Weekend and Holiday Binary Values

# Convert 'Transaction Date' to datetime64 format
rtc_sp_elasticity_df['Transaction Date'] = pd.to_datetime(rtc_sp_elasticity_df['Transact

# Convert 'Transaction Date' to date format
rtc_sp_elasticity_df['Transaction Date'] = rtc_sp_elasticity_df['Transaction Date'].dt.d

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rtc_sp_elasticity_df['IsWeekend'] = (rtc_sp_elasticity_df['Transaction Date'].apply(lamb
rtc_sp_elasticity_df['IsHoliday'] = rtc_sp_elasticity_df['Transaction Date'].isin(holida
```

In [185…

```
# Group the data by unique product pairs
unique_product_pairs = rtc_sp_elasticity_df[['Inventory Code_code1', 'Inventory Code_cod

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = rtc_sp_elasticity_df[(rtc_sp_elasticity_df['Inventory Code_code1'] == inven

    X = sm.add_constant(subset[['Log Relative Price', 'Seasonal', 'IsWeekend', 'IsHolida
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        coefficients_seasonal.append(model.params['Seasonal'])
        p_values_seasonal.append(model.pvalues['Seasonal'])
        coefficients_is_weekend.append(model.params['IsWeekend'])
        p_values_is_weekend.append(model.pvalues['IsWeekend'])
        coefficients_is_holiday.append(model.params['IsHoliday'])
        p_values_is_holiday.append(model.pvalues['IsHoliday'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

rtc_sp_table_df = pd.DataFrame(table_data)

rtc_sp_table_df['Type'] = rtc_sp_table_df['Coefficient (Log Relative Price)'].apply(lamb
rtc_sp_table_df['Significance (Log Relative Price)'] = rtc_sp_table_df['P-Value (Log Rel
rtc_sp_table_df['Significance (Seasonal)'] = rtc_sp_table_df['P-Value (Seasonal)'].apply
rtc_sp_table_df['Significance (IsWeekend)'] = rtc_sp_table_df['P-Value (IsWeekend)'].app
rtc_sp_table_df['Significance (IsHoliday)'] = rtc_sp_table_df['P-Value (IsHoliday)'].app

# Create a DataFrame to store the OLS results
rtc_sp_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
```

```
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(rtc_sp_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	(1158.0, 1160.0)	27.385779	5.670962e-32	0.003285	0.937423	0.148099	0.579775	-1.716838	0.299659
1	(1158.0, 1162.0)	19.452902	1.255872e-07	0.038780	0.435867	0.459322	0.151785	-2.591717	0.190329
2	(1158.0, 1163.0)	15.724694	1.634929e-06	-0.042467	0.348561	0.098442	0.735363	-0.347664	0.845538
3	(1158.0, 1166.0)	19.239597	3.546982e-09	0.061520	0.178021	0.060122	0.837909	-2.789818	0.124657
4	(1160.0, 1162.0)	10.957303	2.526315e-10	-0.055730	0.048488	0.168579	0.347746	0.145213	0.895255
5	(1160.0, 1163.0)	16.010214	1.911085e-20	-0.015944	0.550392	-0.164898	0.335714	2.657056	0.012696
6	(1160.0, 1166.0)	13.851573	1.326377e-15	-0.051892	0.064105	-0.179418	0.318195	-0.013604	0.990151
7	(1162.0, 1163.0)	23.146643	1.518317e-13	0.026956	0.319831	-0.353440	0.041983	2.202186	0.041536
8	(1162.0, 1166.0)	17.264457	6.745319e-07	0.050793	0.099818	-0.371696	0.059316	-0.027761	0.981657
9	(1163.0, 1166.0)	15.452906	9.228471e-08	-0.027243	0.336288	-0.017162	0.924608	-2.895445	0.010427

In [186…

```
# Initialize lists to store the optimization results
optimized_prices_list = []
maximized_revenue_list = []

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Iterate over rows in the DataFrame
for _, row in rtc_sp_table_df.iterrows():
    # Extract the cross-price elasticity coefficient from the table
    cross_price_elasticity_coefficient = row['Coefficient (Log Relative Price)']

    # Initial guess for prices (you may need to adjust this based on your specific conte
    initial_prices = [1.0, 1.0]  # Assuming you are optimizing prices for a pair of prod

    # Define bounds on prices
    bounds = [(0, None), (0, None)]  # Separate bounds for each item in the pair

    # Run the optimization
```

```
    result = minimize(objective_function, initial_prices, args=(cross_price_elasticity_c

    # Extract the optimized prices and maximized revenue
    optimized_prices = result.x.tolist()
    maximized_revenue = -result.fun  # Negate to get the actual maximum revenue

    # Append the results to the lists
    optimized_prices_list.append(optimized_prices)
    maximized_revenue_list.append(maximized_revenue)

# Add new columns to the DataFrame
rtc_sp_table_df['Optimized Prices'] = optimized_prices_list
rtc_sp_table_df['Maximized Revenue'] = maximized_revenue_list

# Display the updated DataFrame
display(rtc_sp_table_df[['Inventory Code Pair', 'Optimized Prices', 'Maximized Revenue']
```

	Inventory Code Pair	Optimized Prices	Maximized Revenue
0	(1158.0, 1160.0)	[2.2826949548201004e+150, 2.2826949548201004e+...	inf
1	(1158.0, 1162.0)	[2.3512588493796958e+148, 2.3512588493796958e+...	inf
2	(1158.0, 1163.0)	[1.419826126394558e+147, 1.419826126394558e+147]	inf
3	(1158.0, 1166.0)	[2.0312823588413932e+148, 2.0312823588413932e+...	inf
4	(1160.0, 1162.0)	[1.3235687874480614e+145, 1.3235687874480614e+...	inf
5	(1160.0, 1163.0)	[1.7978198268288332e+147, 1.7978198268288332e+...	inf
6	(1160.0, 1166.0)	[2.7109044333830237e+146, 2.7109044333830237e+...	inf
7	(1162.0, 1163.0)	[53058410.832539305, 53058410.832539305]	6.797751e+16
8	(1162.0, 1166.0)	[122982075.89612988, 122982075.89612988]	2.762424e+17
9	(1163.0, 1166.0)	[1.1298521436691999e+147, 1.1298521436691999e+...	inf

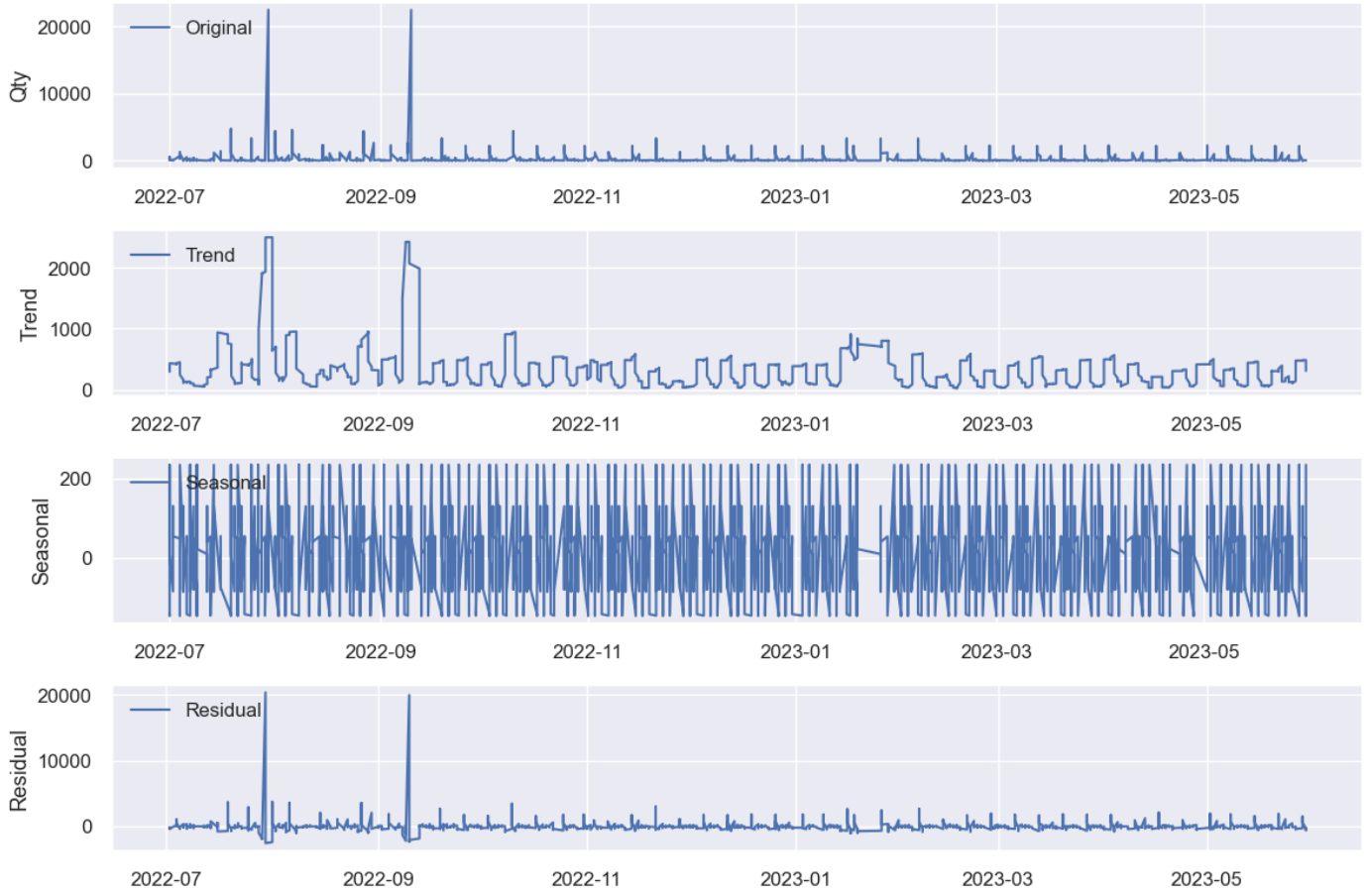
Method 2 - Seperate Cross Price in 2 Entries

In [187…
```
rtc_sp_df2 = rtc_sp_df
```

In [188…
```
## Creating Log Relative Price and Log Relative Volume in columns

rtc_sp_df2.loc[:, 'Log Price'] = np.log(rtc_sp_df2['Price Per Unit'])
rtc_sp_df2.loc[:, 'Log Qty'] = np.log(rtc_sp_df2['Qty'])
```

In [189…
```
rtc_sp_df2.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rtc_sp_df2['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
```

```
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [190…

```
#Copying Seasonal Value to Data Frame and resetting the index

rtc_sp_df2 = rtc_sp_df2.copy()  # Create a copy of the DataFrame
rtc_sp_df2['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rtc_sp_df2.reset_index(inplace=True)
```

In [191…

```
rtc_sp_df2['Transaction Date'] = pd.to_datetime(rtc_sp_df2['Transaction Date'])

# Convert 'Transaction Date' to date format
rtc_sp_df2['Transaction Date'] = rtc_sp_df2['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
```

```
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rtc_sp_df2['IsWeekend'] = (rtc_sp_df2['Transaction Date'].apply(lambda x: x.weekday()) >
rtc_sp_df2['IsHoliday'] = rtc_sp_df2['Transaction Date'].isin(holiday_dates).astype(int)
```

In [192…

```
# Assuming you have separate demand functions for each product
products = rtc_sp_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rtc_sp_df2[(rtc_sp_df2['Inventory Code'] == products[i]) | (rtc_sp_d

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_1.params['Seasonal'],
            'P-Value (Seasonality)': model_1.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_1.params['IsWeekend'],
            'P-Value (IsWeekend)': model_1.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_1.params['IsHoliday'],
            'P-Value (IsHoliday)': model_1.pvalues['IsHoliday'],
        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_2.params['Seasonal'],
            'P-Value (Seasonality)': model_2.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_2.params['IsWeekend'],
```

```
            'P-Value (IsWeekend)': model_2.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_2.params['IsHoliday'],
            'P-Value (IsHoliday)': model_2.pvalues['IsHoliday'],
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[192]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonality)	P-Value (Seasonality)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)
0	1158.0 - 1160.0	1158.0	-1.993080	5.333545e-01	0.000746	0.295986	-0.009342	0.963754	-1.184839
1	1158.0 - 1160.0	1160.0	27.487107	1.229973e-59	-0.000035	0.947420	-0.081010	0.598155	0.257002
2	1158.0 - 1162.0	1158.0	-1.993080	5.333545e-01	0.000746	0.295986	-0.009342	0.963754	-1.184839
3	1158.0 - 1162.0	1162.0	14.301918	5.124822e-06	0.002661	0.001125	-0.392518	0.096225	1.545538
4	1158.0 - 1163.0	1158.0	-1.993080	5.333545e-01	0.000746	0.295986	-0.009342	0.963754	-1.184839
5	1158.0 - 1163.0	1163.0	12.607377	2.441207e-06	-0.000769	0.269448	-0.081689	0.684727	-0.974115
6	1158.0 - 1166.0	1158.0	-1.993080	5.333545e-01	0.000746	0.295986	-0.009342	0.963754	-1.184839
7	1158.0 - 1166.0	1166.0	16.801735	1.647265e-09	-0.000184	0.797934	-0.043785	0.834770	1.667493
8	1160.0 - 1162.0	1160.0	27.487107	1.229973e-59	-0.000035	0.947420	-0.081010	0.598155	0.257002
9	1160.0 - 1162.0	1162.0	14.301918	5.124822e-06	0.002661	0.001125	-0.392518	0.096225	1.545538
10	1160.0 - 1163.0	1160.0	27.487107	1.229973e-59	-0.000035	0.947420	-0.081010	0.598155	0.257002
11	1160.0 - 1163.0	1163.0	12.607377	2.441207e-06	-0.000769	0.269448	-0.081689	0.684727	-0.974115
12	1160.0 - 1166.0	1160.0	27.487107	1.229973e-59	-0.000035	0.947420	-0.081010	0.598155	0.257002
13	1160.0 -	1166.0	16.801735	1.647265e-09	-0.000184	0.797934	-0.043785	0.834770	1.667493

	1166.0								
14	1162.0 - 1163.0	1162.0	14.301918	5.124822e-06	0.002661	0.001125	-0.392518	0.096225	1.545538
15	1162.0 - 1163.0	1163.0	12.607377	2.441207e-06	-0.000769	0.269448	-0.081689	0.684727	-0.974115
16	1162.0 - 1166.0	1162.0	14.301918	5.124822e-06	0.002661	0.001125	-0.392518	0.096225	1.545538
17	1162.0 - 1166.0	1166.0	16.801735	1.647265e-09	-0.000184	0.797934	-0.043785	0.834770	1.667493
18	1163.0 - 1166.0	1163.0	12.607377	2.441207e-06	-0.000769	0.269448	-0.081689	0.684727	-0.974115
19	1163.0 - 1166.0	1166.0	16.801735	1.647265e-09	-0.000184	0.797934	-0.043785	0.834770	1.667493

In [193…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rtc_sp_df2.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rtc_sp_min_max_data2 = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rtc_sp_min_max_data2.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', '

# Display the result
print(rtc_sp_min_max_data2)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1158.0      10.39  15.488333     13.0  22503.0
1          1160.0       5.10   5.900000     15.6   3276.0
2          1162.0       5.10   5.900000     15.6   4726.8
3          1163.0       5.10   5.900000     15.6   1341.6
4          1166.0       5.10   5.900000     15.6   1248.0
```

In [194…

```
# Assuming you have separate demand functions for each product
products = rtc_sp_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Initialize lists to store mean price and quantity
mean_price_list = []
mean_quantity_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rtc_sp_df2[(rtc_sp_df2['Inventory Code'] == products[i]) | (rtc_sp_d
```

```
        # Calculate mean price and quantity for each product in the pair
        mean_price_i = pair_data[pair_data['Inventory Code'] == products[i]]['Price Per
        mean_quantity_i = pair_data[pair_data['Inventory Code'] == products[i]]['Qty'].m

        mean_price_j = pair_data[pair_data['Inventory Code'] == products[j]]['Price Per
        mean_quantity_j = pair_data[pair_data['Inventory Code'] == products[j]]['Qty'].m

        # Store the mean price and quantity in the lists
        mean_price_list.extend([mean_price_i, mean_price_j])
        mean_quantity_list.extend([mean_quantity_i, mean_quantity_j])

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Mean Price': mean_price_i,
            'Mean Quantity': mean_quantity_i,

        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Mean Price': mean_price_j,
            'Mean Quantity': mean_quantity_j,

        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[194]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Mean Price	Mean Quantity
0	1158.0 - 1160.0	1158.0	-1.993080	5.333545e-01	11.613744	347.645318
1	1158.0 - 1160.0	1160.0	27.487107	1.229973e-59	5.223752	199.235955
2	1158.0 - 1162.0	1158.0	-1.993080	5.333545e-01	11.613744	347.645318
3	1158.0 - 1162.0	1162.0	14.301918	5.124822e-06	5.468619	508.139326
4	1158.0 - 1163.0	1158.0	-1.993080	5.333545e-01	11.613744	347.645318

5	1158.0 - 1163.0	1163.0	12.607377	2.441207e-06	5.451401	203.267416
6	1158.0 - 1166.0	1158.0	-1.993080	5.333545e-01	11.613744	347.645318
7	1158.0 - 1166.0	1166.0	16.801735	1.647265e-09	5.485882	196.197753
8	1160.0 - 1162.0	1160.0	27.487107	1.229973e-59	5.223752	199.235955
9	1160.0 - 1162.0	1162.0	14.301918	5.124822e-06	5.468619	508.139326
10	1160.0 - 1163.0	1160.0	27.487107	1.229973e-59	5.223752	199.235955
11	1160.0 - 1163.0	1163.0	12.607377	2.441207e-06	5.451401	203.267416
12	1160.0 - 1166.0	1160.0	27.487107	1.229973e-59	5.223752	199.235955
13	1160.0 - 1166.0	1166.0	16.801735	1.647265e-09	5.485882	196.197753
14	1162.0 - 1163.0	1162.0	14.301918	5.124822e-06	5.468619	508.139326
15	1162.0 - 1163.0	1163.0	12.607377	2.441207e-06	5.451401	203.267416
16	1162.0 - 1166.0	1162.0	14.301918	5.124822e-06	5.468619	508.139326
17	1162.0 - 1166.0	1166.0	16.801735	1.647265e-09	5.485882	196.197753
18	1163.0 - 1166.0	1163.0	12.607377	2.441207e-06	5.451401	203.267416
19	1163.0 - 1166.0	1166.0	16.801735	1.647265e-09	5.485882	196.197753

In [195…

```python
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in results_df
for index, row in results_df.iterrows():
    inventory_code = row['Product']
    mean_price = row['Mean Price']
    mean_quantity = row['Mean Quantity']
    elasticity = row['Coefficient (Log Price)']

    # Set bounds for optimization
    bounds = [(0, None)]  # Optimize only price

    # Perform optimization
    result = minimize(calculate_revenue, x0=[mean_price], args=(elasticity, mean_quantit

    # Extract optimized price
    optimized_price = result.x[0]

    optimized_prices.append(optimized_price)

# Add optimized prices to the results DataFrame
results_df['Optimized Price'] = optimized_prices

# Display the updated results_df
print(results_df[['Pair', 'Product', 'Optimized Price']])
```

```
                Pair  Product  Optimized Price
0   1158.0 - 1160.0  1158.0     2.385805e+19
1   1158.0 - 1160.0  1160.0     0.000000e+00
2   1158.0 - 1162.0  1158.0     2.385805e+19
3   1158.0 - 1162.0  1162.0     0.000000e+00
4   1158.0 - 1163.0  1158.0     2.385805e+19
5   1158.0 - 1163.0  1163.0     0.000000e+00
6   1158.0 - 1166.0  1158.0     2.385805e+19
7   1158.0 - 1166.0  1166.0     0.000000e+00
8   1160.0 - 1162.0  1160.0     0.000000e+00
```

```
9   1160.0 - 1162.0  1162.0     0.000000e+00
10  1160.0 - 1163.0  1160.0     0.000000e+00
11  1160.0 - 1163.0  1163.0     0.000000e+00
12  1160.0 - 1166.0  1160.0     0.000000e+00
13  1160.0 - 1166.0  1166.0     0.000000e+00
14  1162.0 - 1163.0  1162.0     0.000000e+00
15  1162.0 - 1163.0  1163.0     0.000000e+00
16  1162.0 - 1166.0  1162.0     0.000000e+00
17  1162.0 - 1166.0  1166.0     0.000000e+00
18  1163.0 - 1166.0  1163.0     0.000000e+00
19  1163.0 - 1166.0  1166.0     0.000000e+00
```

Ready to Cook(RTC) Supermarket Self PED and Price Optimization

In [196…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rtc_sp_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rtc_sp_df = rtc_sp_df[rtc_sp_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rtc_sp_df = filtered_rtc_sp_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

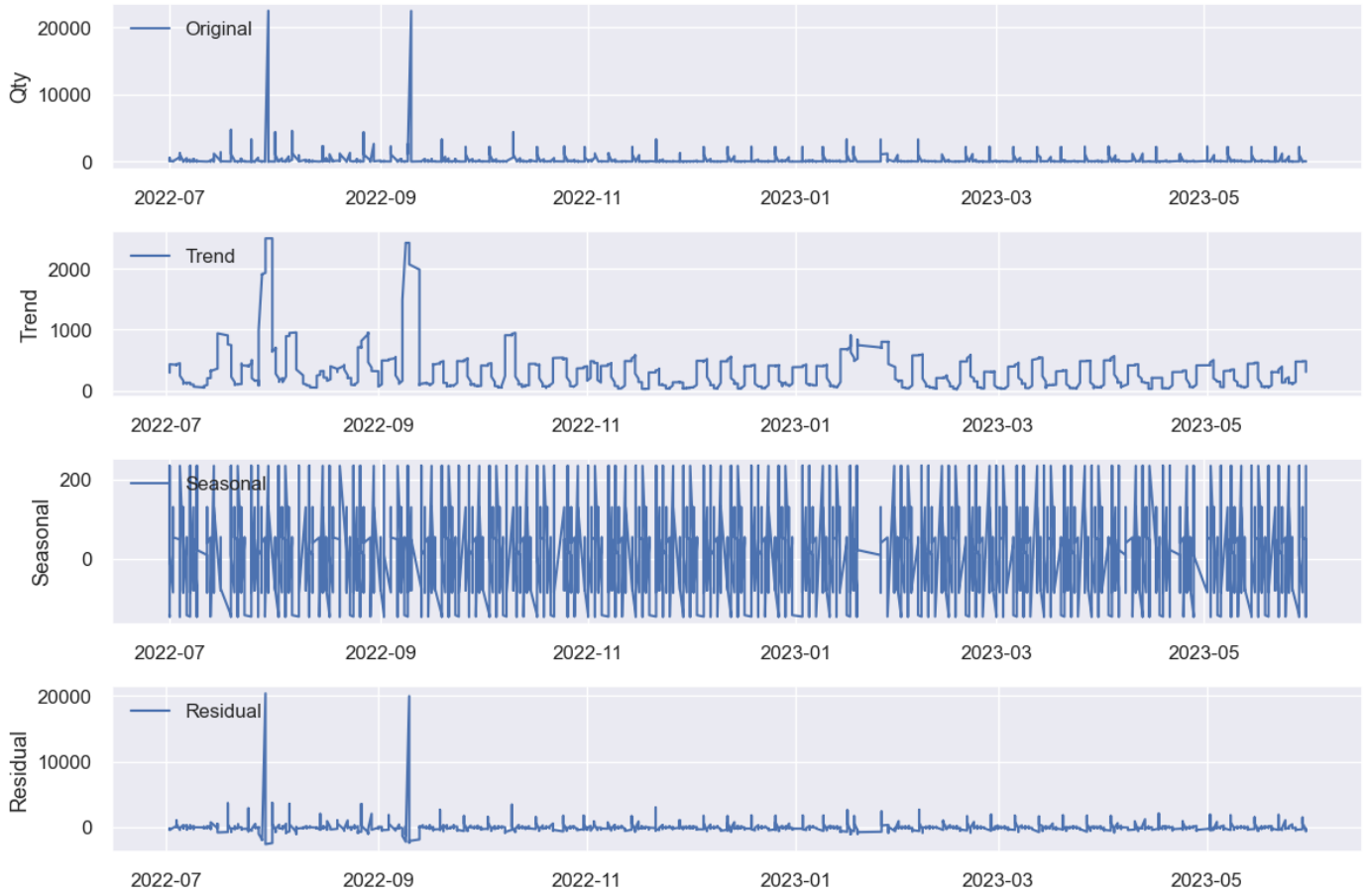
In [197…
```
rtc_sp_df3 = rtc_sp_df
```

In [198…
```
## Creating Log Price and Log Quantity in columns

rtc_sp_df3.loc[:, 'Log Price'] = np.log(rtc_sp_df3['Price Per Unit'])
rtc_sp_df3.loc[:, 'Log Qty'] = np.log(rtc_sp_df3['Qty'])
```

In [199…
```
rtc_sp_df3.reset_index(inplace=True)
```

In [200…
```
## Seasonality Values

rtc_sp_df3.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rtc_sp_df3['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
```

```
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [201…

```
#Copying Seasonal Value to Data Frame and resetting the index

rtc_sp_df3 = rtc_sp_df3.copy()  # Create a copy of the DataFrame
rtc_sp_df3['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rtc_sp_df3.reset_index(inplace=True)
```

In [202…

```
## Setting Weekend and Holiday Binary Values

rtc_sp_df3['Transaction Date'] = pd.to_datetime(rtc_sp_df3['Transaction Date'])

# Convert 'Transaction Date' to date format
rtc_sp_df3['Transaction Date'] = rtc_sp_df3['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
```

```
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rtc_sp_df3['IsWeekend'] = (rtc_sp_df3['Transaction Date'].apply(lambda x: x.weekday()) >
rtc_sp_df3['IsHoliday'] = rtc_sp_df3['Transaction Date'].isin(holiday_dates).astype(int)
```

In [203…

```
# Linear Regression for Self Price PED RTC Supermarket

# Get unique product codes from the 'Inventory Code' column
unique_inventory = rtc_sp_df3['Inventory Code'].unique()

# Initialize lists to store results
inventory_codes = []
coefficients_price = []
p_values_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []

# Loop through the unique product codes
for inventory_code in unique_inventory:
    inventory_data = rtc_sp_df3[rtc_sp_df3['Inventory Code'] == inventory_code]

    # Create features and target variables
    X = sm.add_constant(inventory_data[['Log Price', 'IsWeekend', 'IsHoliday', 'Seasonal

    model = sm.OLS(inventory_data['Log Qty'], X).fit()

    # Append results to lists
    inventory_codes.append(inventory_code)
    coefficients_price.append(model.params['Log Price'])
    p_values_price.append(model.pvalues['Log Price'])
    coefficients_seasonal.append(model.params['Seasonal'])
    p_values_seasonal.append(model.pvalues['Seasonal'])
    coefficients_is_weekend.append(model.params['IsWeekend'])
    p_values_is_weekend.append(model.pvalues['IsWeekend'])
    coefficients_is_holiday.append(model.params['IsHoliday'])
    p_values_is_holiday.append(model.pvalues['IsHoliday'])

# Create a DataFrame with the aggregated coefficients and p-values
table_data = {
    'Inventory Code': inventory_codes,
    'Coefficient (Log Price)': coefficients_price,
    'P-Value (Log Price)': p_values_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

own_price_rtc_sp_df = pd.DataFrame(table_data)

# Apply the significance labels
for column in own_price_rtc_sp_df.columns:
    if 'P-Value' in column:
        significance_column = 'Significance' + column.split('P-Value', 1)[1]
        own_price_rtc_sp_df[significance_column] = own_price_rtc_sp_df[column].apply(lam

# Display the table in the Jupyter Notebook
```

```
print("Ready to Cook Supermarket Aggregated Coefficients and Significance Table:")
display(own_price_rtc_sp_df)
```

Ready to Cook Supermarket Aggregated Coefficients and Significance Table:

	Inventory Code	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	1158.0	-1.993080	5.333545e-01	0.000746	0.295986	-0.009342	0.963754	-1.184839	0.345690
1	1160.0	27.487107	1.229973e-59	-0.000035	0.947420	-0.081010	0.598155	0.257002	0.782676
2	1162.0	14.301918	5.124822e-06	0.002661	0.001125	-0.392518	0.096225	1.545538	0.288932
3	1163.0	12.607377	2.441207e-06	-0.000769	0.269448	-0.081689	0.684727	-0.974115	0.427654
4	1166.0	16.801735	1.647265e-09	-0.000184	0.797934	-0.043785	0.834770	1.667493	0.192383

In [204…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rtc_sp_df3.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rtc_sp_min_max_data = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rtc_sp_min_max_data.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', 'M

# Display the result
print(rtc_sp_min_max_data)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1158.0      10.39  15.488333     13.0  22503.0
1          1160.0       5.10   5.900000     15.6   3276.0
2          1162.0       5.10   5.900000     15.6   4726.8
3          1163.0       5.10   5.900000     15.6   1341.6
4          1166.0       5.10   5.900000     15.6   1248.0
```

In [205…

```
## Price Optimization Using Min & Max Qty & Price as bounds


# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rtc_sp_df, rtc_sp_min_max_data, on='Inventory Code')

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, qty, coefficient_price):
    return -price * qty * coefficient_price  # Negative sign because scipy.optimize mini

# Function to be minimized (negative revenue for maximization)
def objective_function(prices):
    total_revenue = 0
    for index, row in merged_data.iterrows():
        min_price, max_price = row['Min Price'], row['Max Price']
        min_qty, max_qty = row['Min Qty'], row['Max Qty']
        coefficient_price = row['Coefficient (Log Price)']

        # Ensure prices are within bounds
```

```
        price = max(min(max_price, prices[index]), min_price)

        # Calculate revenue
        qty = max(min(max_qty, prices[len(merged_data) + index]), min_qty)
        total_revenue += calculate_revenue(price, qty, coefficient_price)

    return total_revenue

# Initial guess for prices
initial_prices = merged_data['Min Price'].values.tolist() + merged_data['Min Qty'].value

# Constraints
price_constraints = [(min_price, max_price) for min_price, max_price in zip(merged_data[
qty_constraints = [(min_qty, max_qty) for min_qty, max_qty in zip(merged_data['Min Qty']
constraints = price_constraints + qty_constraints

# Optimization
result = minimize(objective_function, initial_prices, constraints={'type': 'ineq', 'fun'
                  bounds=constraints)

# Extract optimized prices and quantities
optimized_prices = result.x[:len(merged_data)]
optimized_quantities = result.x[len(merged_data):]

# Display optimized prices and quantities
optimized_data = merged_data.copy()
optimized_data['Optimized Price'] = optimized_prices
optimized_data['Optimized Quantity'] = optimized_quantities

print("Optimized Prices and Quantities:")
print(optimized_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Optimized Prices and Quantities:
   Inventory Code  Optimized Price  Optimized Quantity
0          1158.0            10.39                13.0
1          1160.0             5.90              3276.0
2          1162.0             5.90              4726.8
3          1163.0             5.90              1341.6
4          1166.0             5.90              1248.0
```

In [206…

```
## Price Optimization Using Min Price as bounds

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rtc_sp_df, rtc_sp_min_max_data, on='Inventory Code')

# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in merged_data
for index, row in merged_data.iterrows():
    inventory_code = row['Inventory Code']
    min_price = row['Min Price']

    # Get elasticity from own_price_rte_sp_df based on the inventory code
    elasticity = own_price_rtc_sp_df.loc[own_price_rtc_sp_df['Inventory Code'] == invent
```

```
    # Set bounds for optimization
    bounds = [(min_price, None), (None, None)]  # Remove lower bound for quantity

    # Perform optimization
    result = minimize(objective_function, x0=[min_price, row['Min Qty']], args=(elastici

    # Extract optimized price and quantity
    optimized_price, optimized_quantity = result.x

    # Append results to lists
    optimized_prices.append(optimized_price)
    optimized_quantities.append(optimized_quantity)

# Add optimized prices and quantities to the DataFrame
merged_data['Optimized Price'] = optimized_prices
merged_data['Optimized Quantity'] = optimized_quantities

# Display the DataFrame
print(merged_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

	Inventory Code	Optimized Price	Optimized Quantity
0	1158.0	1.959778e+137	-4.582113e+162
1	1160.0	1.309007e+151	3.657266e+151
2	1162.0	2.361329e+147	6.148192e+147
3	1163.0	4.660943e+146	1.192003e+147
4	1166.0	1.918251e+148	5.099181e+148

Raw Supermarket Cross PED and Price Optimization

In [207…

```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = raw_sp_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_raw_sp_df = raw_sp_df[raw_sp_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
raw_sp_df = filtered_raw_sp_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [208…

```
#Using mode of each inventory code to replace missing data in dataframe

# Assuming your DataFrame is named rte_sp_df
# Step 1: Identify unique inventory codes for each transaction date
unique_inventory_codes = raw_sp_df.groupby('Transaction Date')['Inventory Code'].unique(

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = raw_sp_df.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(cal

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(raw_sp_df['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
```

```
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
raw_sp_df = pd.concat([raw_sp_df, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
raw_sp_df = raw_sp_df.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
raw_sp_df.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(raw_sp_df)
```

```
     Transaction Date  Inventory Code     Qty  Price Per Unit
0          2022-07-01          1030.0  2964.0            4.40
1          2022-07-01          1032.0  1731.6            4.60
2          2022-07-01          1033.0  1482.0            3.75
3          2022-07-01          1036.0  1185.6            5.10
4          2022-07-01          1037.0  1294.8            4.95
...               ...             ...     ...             ...
1290       2023-05-31          1030.0   530.4            5.40
1291       2023-05-31          1032.0   374.4            5.40
1292       2023-05-31          1033.0   327.6            4.60
1293       2023-05-31          1036.0   249.6            5.80
1294       2023-05-31          1037.0   280.8            5.80

[1295 rows x 4 columns]
```

In [209…

```
# Create a list of unique inventory codes
unique_inventory_codes = raw_sp_df['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1030.0, 1032.0)
Inventory Code Pair: (1030.0, 1033.0)
Inventory Code Pair: (1030.0, 1036.0)
Inventory Code Pair: (1030.0, 1037.0)
Inventory Code Pair: (1032.0, 1033.0)
Inventory Code Pair: (1032.0, 1036.0)
Inventory Code Pair: (1032.0, 1037.0)
Inventory Code Pair: (1033.0, 1036.0)
Inventory Code Pair: (1033.0, 1037.0)
Inventory Code Pair: (1036.0, 1037.0)
```

In [210…

```
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = raw_sp_df[raw_sp_df['Inventory Code'] == code1]
    data_code2 = raw_sp_df[raw_sp_df['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:      Transaction Date  Inventory Code     Qty  Price Per Unit
3          2022-07-01          1036.0  1185.6             5.1
8          2022-07-02          1036.0   312.0             5.8
```

```
13          2022-07-04          1036.0   936.0             5.1
18          2022-07-06          1036.0  1185.6             5.1
23          2022-07-07          1036.0   312.0             5.8
...                ...             ...     ...             ...
1273        2023-05-26          1036.0   452.4             5.8
1278        2023-05-27          1036.0   468.0             5.8
1283        2023-05-29          1036.0   608.4             5.8
1288        2023-05-30          1036.0   140.4             5.8
1293        2023-05-31          1036.0   249.6             5.8

[259 rows x 4 columns]
Data for Code Pair 2:      Transaction Date  Inventory Code     Qty  Price Per Unit
4           2022-07-01          1037.0  1294.8            4.95
9           2022-07-02          1037.0   468.0            5.80
14          2022-07-04          1037.0   889.2            4.95
19          2022-07-06          1037.0  1326.0            4.95
24          2022-07-07          1037.0   468.0            5.80
...                ...             ...     ...             ...
1274        2023-05-26          1037.0   452.4            5.80
1279        2023-05-27          1037.0   468.0            5.80
1284        2023-05-29          1037.0   592.8            5.80
1289        2023-05-30          1037.0   202.8            5.80
1294        2023-05-31          1037.0   280.8            5.80

[259 rows x 4 columns]
```

In [211…

```python
#Creating the cross reference for each pair on a daily basis

# Create an empty DataFrame to store the results
raw_sp_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1',

# Create a list of unique inventory codes
unique_inventory_codes = raw_sp_df['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = raw_sp_df[raw_sp_df['Inventory Code'] == code1]
    data_code2 = raw_sp_df[raw_sp_df['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    raw_sp_elasticity_df = pd.concat([raw_sp_elasticity_df, results])

# Print or inspect the elasticity results
print(raw_sp_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1030.0                1032.0
1         2022-07-02                1030.0                1032.0
2         2022-07-04                1030.0                1032.0
3         2022-07-06                1030.0                1032.0
4         2022-07-07                1030.0                1032.0
..               ...                   ...                   ...
254       2023-05-26                1036.0                1037.0
255       2023-05-27                1036.0                1037.0
256       2023-05-29                1036.0                1037.0
257       2023-05-30                1036.0                1037.0
258       2023-05-31                1036.0                1037.0
```

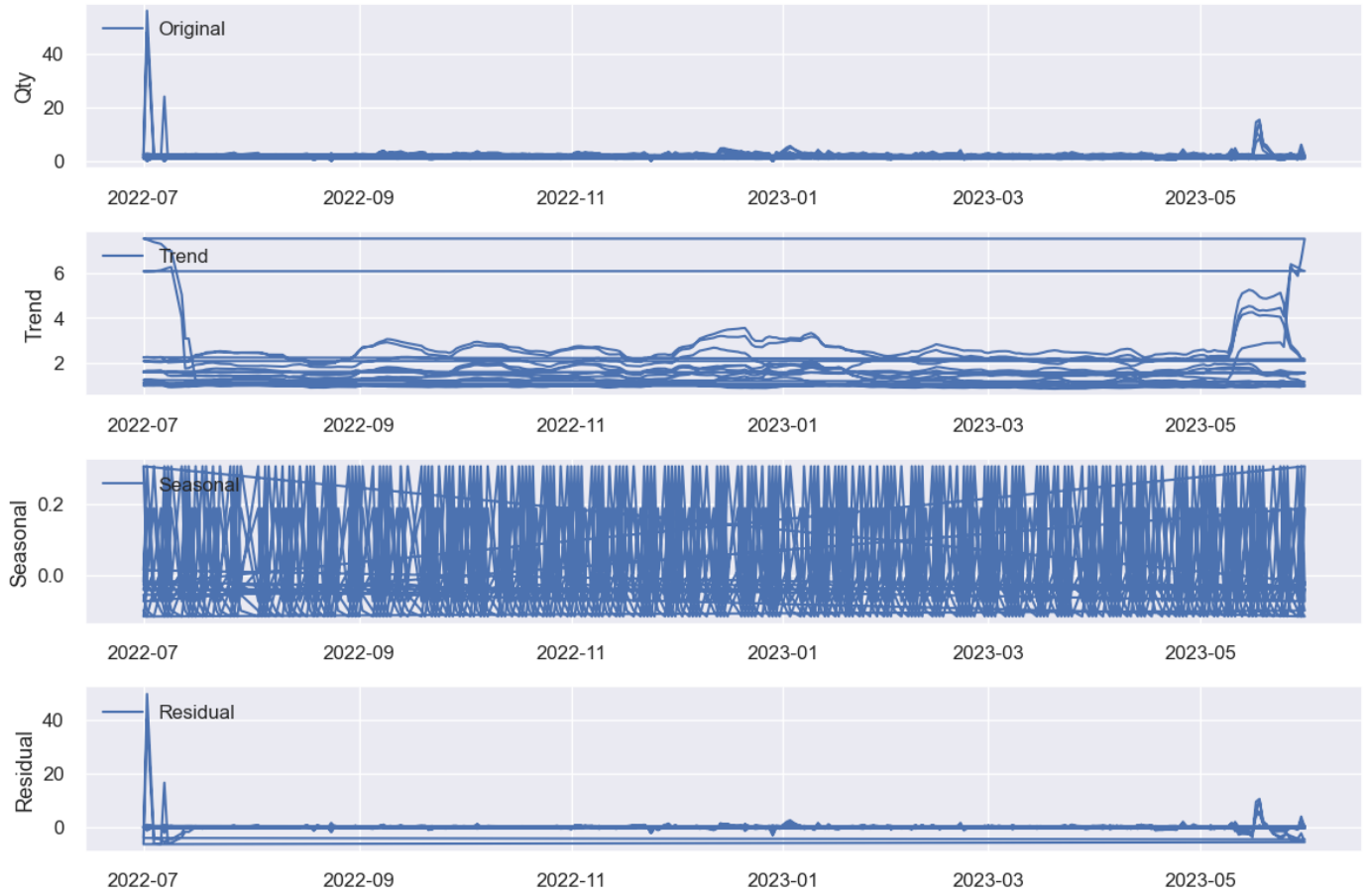
```
     Relative_Price  Relative_Volume
0          0.956522         1.711712
1          1.000000         1.166667
2          0.956522         1.617978
3          0.956522         1.523810
4          0.888889         0.020833
..              ...              ...
254        1.000000         1.000000
255        1.000000         1.000000
256        1.000000         1.026316
257        1.000000         0.692308
258        1.000000         0.888889

[2590 rows x 5 columns]
```

In [212…

```
## Creating Log Relative Price and Log Relative Volume in columns

raw_sp_elasticity_df.loc[:, 'Log Relative Price'] = np.log(raw_sp_elasticity_df['Relativ
raw_sp_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(raw_sp_elasticity_df['Relati
```

In [213…

```
#Seasonality
raw_sp_elasticity_df.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = raw_sp_elasticity_df['Relative_Volume']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [214…

```
#Copying Seasonal Value to Data Frame and resetting the index

raw_sp_elasticity_df = raw_sp_elasticity_df.copy()  # Create a copy of the DataFrame
raw_sp_elasticity_df['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the c

raw_sp_elasticity_df.reset_index(inplace=True)
```

In [215…

```
## Setting Weekend and Holiday Binary Values


raw_sp_elasticity_df['Transaction Date'] = pd.to_datetime(raw_sp_elasticity_df['Transact

# Convert 'Transaction Date' to date format
raw_sp_elasticity_df['Transaction Date'] = raw_sp_elasticity_df['Transaction Date'].dt.d

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
raw_sp_elasticity_df['IsWeekend'] = (raw_sp_elasticity_df['Transaction Date'].apply(lamb
raw_sp_elasticity_df['IsHoliday'] = raw_sp_elasticity_df['Transaction Date'].isin(holida
```

In [216…

```
# Group the data by unique product pairs
unique_product_pairs = raw_sp_elasticity_df[['Inventory Code_code1', 'Inventory Code_cod

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = raw_sp_elasticity_df[(raw_sp_elasticity_df['Inventory Code_code1'] == inven

    X = sm.add_constant(subset[['Log Relative Price', 'Seasonal', 'IsWeekend', 'IsHolida
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        coefficients_seasonal.append(model.params['Seasonal'])
        p_values_seasonal.append(model.pvalues['Seasonal'])
        coefficients_is_weekend.append(model.params['IsWeekend'])
        p_values_is_weekend.append(model.pvalues['IsWeekend'])
        coefficients_is_holiday.append(model.params['IsHoliday'])
        p_values_is_holiday.append(model.pvalues['IsHoliday'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

raw_sp_table_df = pd.DataFrame(table_data)

raw_sp_table_df['Type'] = raw_sp_table_df['Coefficient (Log Relative Price)'].apply(lamb
raw_sp_table_df['Significance (Log Relative Price)'] = raw_sp_table_df['P-Value (Log Rel
raw_sp_table_df['Significance (Seasonal)'] = raw_sp_table_df['P-Value (Seasonal)'].apply
raw_sp_table_df['Significance (IsWeekend)'] = raw_sp_table_df['P-Value (IsWeekend)'].app
raw_sp_table_df['Significance (IsHoliday)'] = raw_sp_table_df['P-Value (IsHoliday)'].app

# Create a DataFrame to store the OLS results
raw_sp_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
```

```
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(raw_sp_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	(1030.0, 1032.0)	-1.864208	1.528135e-02	0.120328	0.538174	0.085586	0.165169	0.105243	0.458248
1	(1030.0, 1033.0)	-2.734785	1.764463e-03	-0.071358	0.750594	0.151722	0.034156	-0.034508	0.833210
2	(1030.0, 1036.0)	-1.152736	2.042526e-01	0.166903	0.551851	0.105174	0.239432	0.167179	0.414103
3	(1030.0, 1037.0)	0.350087	7.481598e-01	0.365918	0.245384	0.084830	0.386354	0.049506	0.826359
4	(1032.0, 1033.0)	1.529247	1.336087e-01	0.458093	0.028199	0.050449	0.445657	-0.120396	0.426437
5	(1032.0, 1036.0)	-0.442256	6.940633e-01	0.020354	0.916254	0.012297	0.841263	0.066093	0.637349
6	(1032.0, 1037.0)	0.042905	9.686441e-01	0.069620	0.729732	0.001707	0.978698	0.001723	0.990638
7	(1033.0, 1036.0)	0.376022	6.665098e-01	-0.059915	0.799033	-0.057961	0.433852	0.205984	0.226143
8	(1033.0, 1037.0)	2.095975	4.781747e-02	0.108186	0.665996	-0.072813	0.362263	0.163716	0.368968
9	(1036.0, 1037.0)	7.292388	5.575798e-18	-0.198796	0.027193	-0.008450	0.766743	-0.025139	0.701179

In [217…

```
# Initialize lists to store the optimization results
optimized_prices_list = []
maximized_revenue_list = []

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Iterate over rows in the DataFrame
for _, row in raw_sp_table_df.iterrows():
    # Extract the cross-price elasticity coefficient from the table
    cross_price_elasticity_coefficient = row['Coefficient (Log Relative Price)']

    # Initial guess for prices (you may need to adjust this based on your specific conte
    initial_prices = [1.0, 1.0]  # Assuming you are optimizing prices for a pair of prod

    # Define bounds on prices
    bounds = [(0, None), (0, None)]  # Separate bounds for each item in the pair

    # Run the optimization
```

```
    result = minimize(objective_function, initial_prices, args=(cross_price_elasticity_c

    # Extract the optimized prices and maximized revenue
    optimized_prices = result.x.tolist()
    maximized_revenue = -result.fun  # Negate to get the actual maximum revenue

    # Append the results to the lists
    optimized_prices_list.append(optimized_prices)
    maximized_revenue_list.append(maximized_revenue)

# Add new columns to the DataFrame
raw_sp_table_df['Optimized Prices'] = optimized_prices_list
raw_sp_table_df['Maximized Revenue'] = maximized_revenue_list

# Display the updated DataFrame
display(raw_sp_table_df[['Inventory Code Pair', 'Optimized Prices', 'Maximized Revenue']
```

	Inventory Code Pair	Optimized Prices	Maximized Revenue
0	(1030.0, 1032.0)	[1.7479095542041793e-09, 1.7479095542041793e-09]	-2.640319e-18
1	(1030.0, 1033.0)	[0.0, 0.0]	-0.000000e+00
2	(1030.0, 1036.0)	[1.969994991058728e-09, 1.969994991058728e-09]	-5.927483e-19
3	(1030.0, 1037.0)	[2.1208443430346155e+152, 2.1208443430346155e+...	inf
4	(1032.0, 1033.0)	[49919685.58517369, 49919685.58517369]	6.302820e+15
5	(1032.0, 1036.0)	[2.4491614308582057e+146, 2.4491614308582057e+...	inf
6	(1032.0, 1037.0)	[3.83638741737666e+150, 3.83638741737666e+150]	inf
7	(1033.0, 1036.0)	[2.8525503500562805e+152, 2.8525503500562805e+...	inf
8	(1033.0, 1037.0)	[3.0445329160041957e+147, 3.0445329160041957e+...	inf
9	(1036.0, 1037.0)	[8.146330349591323e+142, 8.146330349591323e+142]	inf

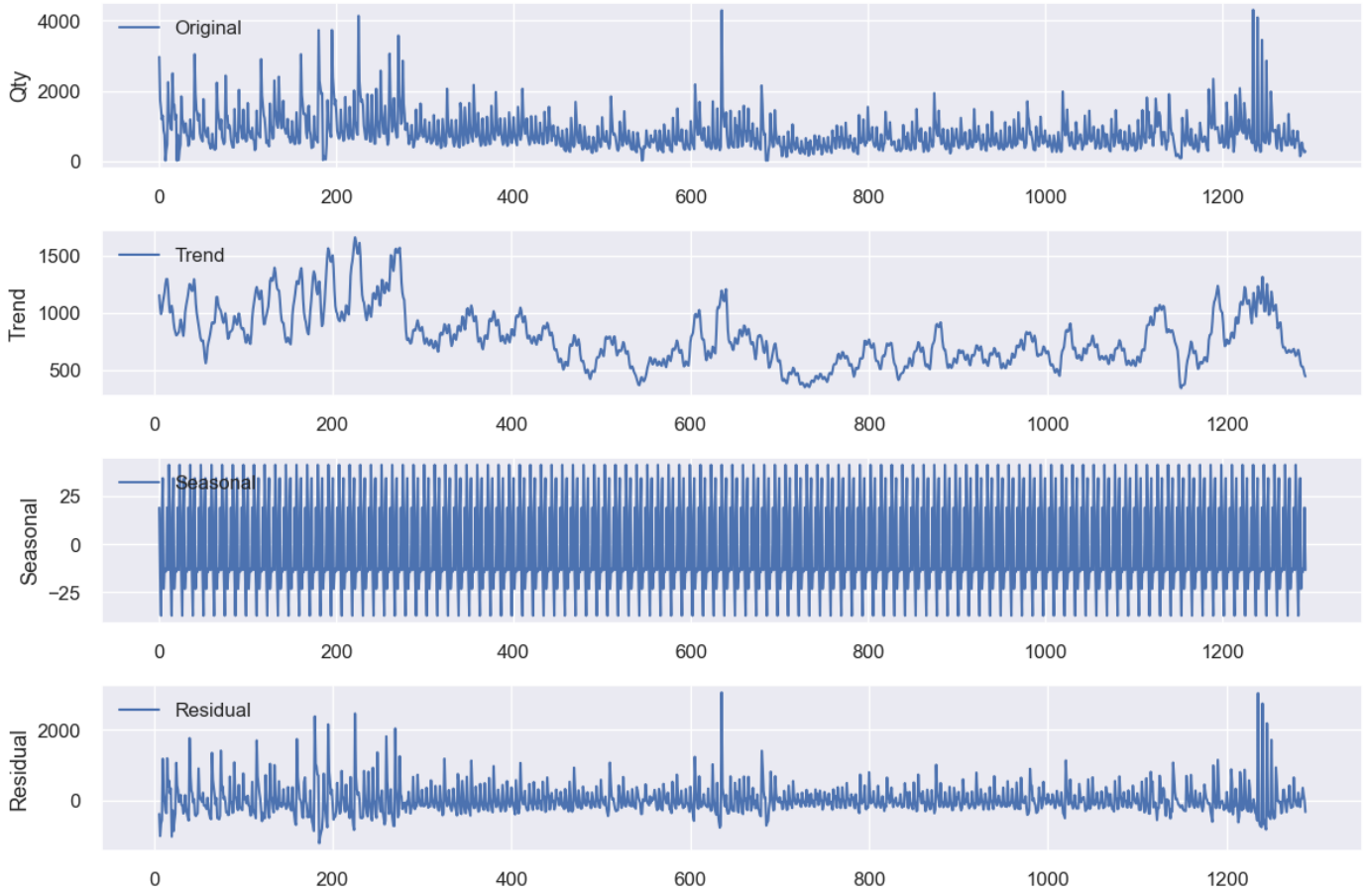
Method 2 - Seperate Cross Price in 2 Entries

In [218…
```
raw_sp_df2 = raw_sp_df
```

In [219…
```
## Creating Log Relative Price and Log Relative Volume in columns

raw_sp_df2.loc[:, 'Log Price'] = np.log(raw_sp_df2['Price Per Unit'])
raw_sp_df2.loc[:, 'Log Qty'] = np.log(raw_sp_df2['Qty'])
```

In [220…
```
rtc_sp_df2.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = raw_sp_df2['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
```

```
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [221…

```
#Copying Seasonal Value to Data Frame and resetting the index

raw_sp_df2 = raw_sp_df2.copy()  # Create a copy of the DataFrame
raw_sp_df2['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

raw_sp_df2.reset_index(inplace=True)
```

In [222…

```
raw_sp_df2['Transaction Date'] = pd.to_datetime(raw_sp_df2['Transaction Date'])

# Convert 'Transaction Date' to date format
raw_sp_df2['Transaction Date'] = raw_sp_df2['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
```

```
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
raw_sp_df2['IsWeekend'] = (raw_sp_df2['Transaction Date'].apply(lambda x: x.weekday()) >
raw_sp_df2['IsHoliday'] = raw_sp_df2['Transaction Date'].isin(holiday_dates).astype(int)
```

In [223…

```
products = raw_sp_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = raw_sp_df2[(raw_sp_df2['Inventory Code'] == products[i]) | (raw_sp_d

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_1.params['Seasonal'],
            'P-Value (Seasonality)': model_1.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_1.params['IsWeekend'],
            'P-Value (IsWeekend)': model_1.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_1.params['IsHoliday'],
            'P-Value (IsHoliday)': model_1.pvalues['IsHoliday'],
        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_2.params['Seasonal'],
            'P-Value (Seasonality)': model_2.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_2.params['IsWeekend'],
            'P-Value (IsWeekend)': model_2.pvalues['IsWeekend'],
```

```
            'Coefficient (IsHoliday)': model_2.params['IsHoliday'],
            'P-Value (IsHoliday)': model_2.pvalues['IsHoliday'],
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[223]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonality)	P-Value (Seasonality)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)
0	1030.0 - 1032.0	1030.0	-1.861197	1.447949e-03	0.004611	0.009128	0.265400	0.016725	-0.051182
1	1030.0 - 1032.0	1032.0	-1.863228	1.326130e-03	0.001565	0.214564	0.136851	0.089325	-0.208475
2	1030.0 - 1033.0	1030.0	-1.861197	1.447949e-03	0.004611	0.009128	0.265400	0.016725	-0.051182
3	1030.0 - 1033.0	1033.0	-2.070260	7.813732e-05	-0.000437	0.769899	0.081993	0.384223	-0.054431
4	1030.0 - 1036.0	1030.0	-1.861197	1.447949e-03	0.004611	0.009128	0.265400	0.016725	-0.051182
5	1030.0 - 1036.0	1036.0	-3.154435	6.882544e-07	-0.000090	0.940548	0.122849	0.109638	-0.239883
6	1030.0 - 1037.0	1030.0	-1.861197	1.447949e-03	0.004611	0.009128	0.265400	0.016725	-0.051182
7	1030.0 - 1037.0	1037.0	-2.679994	1.870554e-08	0.000843	0.440729	0.125432	0.071569	-0.180302
8	1032.0 - 1033.0	1032.0	-1.863228	1.326130e-03	0.001565	0.214564	0.136851	0.089325	-0.208475
9	1032.0 - 1033.0	1033.0	-2.070260	7.813732e-05	-0.000437	0.769899	0.081993	0.384223	-0.054431
10	1032.0 - 1036.0	1032.0	-1.863228	1.326130e-03	0.001565	0.214564	0.136851	0.089325	-0.208475
11	1032.0 - 1036.0	1036.0	-3.154435	6.882544e-07	-0.000090	0.940548	0.122849	0.109638	-0.239883
12	1032.0 - 1037.0	1032.0	-1.863228	1.326130e-03	0.001565	0.214564	0.136851	0.089325	-0.208475
13	1032.0 - 1037.0	1037.0	-2.679994	1.870554e-08	0.000843	0.440729	0.125432	0.071569	-0.180302

14	1033.0 - 1036.0	1033.0	-2.070260	7.813732e-05	-0.000437	0.769899	0.081993	0.384223	-0.054431
15	1033.0 - 1036.0	1036.0	-3.154435	6.882544e-07	-0.000090	0.940548	0.122849	0.109638	-0.239883
16	1033.0 - 1037.0	1033.0	-2.070260	7.813732e-05	-0.000437	0.769899	0.081993	0.384223	-0.054431
17	1033.0 - 1037.0	1037.0	-2.679994	1.870554e-08	0.000843	0.440729	0.125432	0.071569	-0.180302
18	1036.0 - 1037.0	1036.0	-3.154435	6.882544e-07	-0.000090	0.940548	0.122849	0.109638	-0.239883
19	1036.0 - 1037.0	1037.0	-2.679994	1.870554e-08	0.000843	0.440729	0.125432	0.071569	-0.180302

In [224…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = raw_sp_df2.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
raw_sp_min_max_data2 = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
raw_sp_min_max_data2.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', '

# Display the result
print(raw_sp_min_max_data2)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1030.0       4.40        5.4     15.6   4305.6
1          1032.0       4.60        5.4     31.2   2386.8
2          1033.0       3.75        4.7     15.6   2106.0
3          1036.0       5.10        5.8     46.8   1903.2
4          1037.0       4.95        5.8     93.6   1934.4
```

In [225…

```
products = raw_sp_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Initialize lists to store mean price and quantity
mean_price_list = []
mean_quantity_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = raw_sp_df2[(raw_sp_df2['Inventory Code'] == products[i]) | (raw_sp_d

        # Calculate mean price and quantity for each product in the pair
        mean_price_i = pair_data[pair_data['Inventory Code'] == products[i]]['Price Per
        mean_quantity_i = pair_data[pair_data['Inventory Code'] == products[i]]['Qty'].m
```

```
        mean_price_j = pair_data[pair_data['Inventory Code'] == products[j]]['Price Per
        mean_quantity_j = pair_data[pair_data['Inventory Code'] == products[j]]['Qty'].m

        # Store the mean price and quantity in the lists
        mean_price_list.extend([mean_price_i, mean_price_j])
        mean_quantity_list.extend([mean_quantity_i, mean_quantity_j])

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Mean Price': mean_price_i,
            'Mean Quantity': mean_quantity_i,

        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Mean Price': mean_price_j,
            'Mean Quantity': mean_quantity_j,

        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[225]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Mean Price	Mean Quantity
0	1030.0 - 1032.0	1030.0	-1.861197	1.447949e-03	5.220257	1344.310425
1	1030.0 - 1032.0	1032.0	-1.863228	1.326130e-03	5.266885	849.667954
2	1030.0 - 1033.0	1030.0	-1.861197	1.447949e-03	5.220257	1344.310425
3	1030.0 - 1033.0	1033.0	-2.070260	7.813732e-05	4.452564	638.274903
4	1030.0 - 1036.0	1030.0	-1.861197	1.447949e-03	5.220257	1344.310425
5	1030.0 - 1036.0	1036.0	-3.154435	6.882544e-07	5.669498	527.448649
6	1030.0 - 1037.0	1030.0	-1.861197	1.447949e-03	5.220257	1344.310425

7	1030.0 - 1037.0	1037.0	-2.679994	1.870554e-08	5.650000	558.648649
8	1032.0 - 1033.0	1032.0	-1.863228	1.326130e-03	5.266885	849.667954
9	1032.0 - 1033.0	1033.0	-2.070260	7.813732e-05	4.452564	638.274903
10	1032.0 - 1036.0	1032.0	-1.863228	1.326130e-03	5.266885	849.667954
11	1032.0 - 1036.0	1036.0	-3.154435	6.882544e-07	5.669498	527.448649
12	1032.0 - 1037.0	1032.0	-1.863228	1.326130e-03	5.266885	849.667954
13	1032.0 - 1037.0	1037.0	-2.679994	1.870554e-08	5.650000	558.648649
14	1033.0 - 1036.0	1033.0	-2.070260	7.813732e-05	4.452564	638.274903
15	1033.0 - 1036.0	1036.0	-3.154435	6.882544e-07	5.669498	527.448649
16	1033.0 - 1037.0	1033.0	-2.070260	7.813732e-05	4.452564	638.274903
17	1033.0 - 1037.0	1037.0	-2.679994	1.870554e-08	5.650000	558.648649
18	1036.0 - 1037.0	1036.0	-3.154435	6.882544e-07	5.669498	527.448649
19	1036.0 - 1037.0	1037.0	-2.679994	1.870554e-08	5.650000	558.648649

In [226…

```python
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in results_df
for index, row in results_df.iterrows():
    inventory_code = row['Product']
    mean_price = row['Mean Price']
    mean_quantity = row['Mean Quantity']
    elasticity = row['Coefficient (Log Price)']

    # Set bounds for optimization
    bounds = [(0, None)]  # Optimize only price

    # Perform optimization
    result = minimize(calculate_revenue, x0=[mean_price], args=(elasticity, mean_quantit

    # Extract optimized price
    optimized_price = result.x[0]

    optimized_prices.append(optimized_price)

# Add optimized prices to the results DataFrame
results_df['Optimized Price'] = optimized_prices

# Display the updated results_df
print(results_df[['Pair', 'Product', 'Optimized Price']])
```

```
                Pair Product  Optimized Price
0   1030.0 - 1032.0  1030.0     2.289312e+37
1   1030.0 - 1032.0  1032.0     2.242412e+50
2   1030.0 - 1033.0  1030.0     2.289312e+37
3   1030.0 - 1033.0  1033.0     2.338715e+37
4   1030.0 - 1036.0  1030.0     2.289312e+37
5   1030.0 - 1036.0  1036.0     6.898565e+46
6   1030.0 - 1037.0  1030.0     2.289312e+37
7   1030.0 - 1037.0  1037.0     8.201147e+07
8   1032.0 - 1033.0  1032.0     2.242412e+50
9   1032.0 - 1033.0  1033.0     2.338715e+37
10  1032.0 - 1036.0  1032.0     2.242412e+50
11  1032.0 - 1036.0  1036.0     6.898565e+46
12  1032.0 - 1037.0  1032.0     2.242412e+50
```

```
13  1032.0 - 1037.0  1037.0     8.201147e+07
14  1033.0 - 1036.0  1033.0     2.338715e+37
15  1033.0 - 1036.0  1036.0     6.898565e+46
16  1033.0 - 1037.0  1033.0     2.338715e+37
17  1033.0 - 1037.0  1037.0     8.201147e+07
18  1036.0 - 1037.0  1036.0     6.898565e+46
19  1036.0 - 1037.0  1037.0     8.201147e+07
```

Raw Supermarket Self PED and Price Optimization

In [227…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = raw_sp_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_raw_sp_df = raw_sp_df[raw_sp_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
raw_sp_df = filtered_raw_sp_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [228…
```
raw_sp_df3 = raw_sp_df
```

In [229…
```
## Creating Log Price and Log Quantity in columns

raw_sp_df.loc[:, 'Log Price'] = np.log(raw_sp_df['Price Per Unit'])
raw_sp_df.loc[:, 'Log Qty'] = np.log(raw_sp_df['Qty'])
```

In [230…
```
raw_sp_df
```

Out[230]:

	Transaction Date	Inventory Code	Qty	Price Per Unit	Log Price	Log Qty
0	2022-07-01	1030.0	2964.0	4.40	1.481605	7.994295
1	2022-07-01	1032.0	1731.6	4.60	1.526056	7.456801
2	2022-07-01	1033.0	1482.0	3.75	1.321756	7.301148
3	2022-07-01	1036.0	1185.6	5.10	1.629241	7.078004
4	2022-07-01	1037.0	1294.8	4.95	1.599388	7.166112
...	...	...	...	...	...	...
1290	2023-05-31	1030.0	530.4	5.40	1.686399	6.273631
1291	2023-05-31	1032.0	374.4	5.40	1.686399	5.925325
1292	2023-05-31	1033.0	327.6	4.60	1.526056	5.791793
1293	2023-05-31	1036.0	249.6	5.80	1.757858	5.519860
1294	2023-05-31	1037.0	280.8	5.80	1.757858	5.637643

1295 rows × 6 columns

In [231…
```
raw_sp_df3.reset_index(inplace=True)
```

In [232…
```
## Seasonality Values

raw_sp_df3.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = raw_sp_df3['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou
```

```
# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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

In [233…

```
#Copying Seasonal Value to Data Frame and resetting the index

raw_sp_df3 = raw_sp_df3.copy()  # Create a copy of the DataFrame
raw_sp_df3['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

raw_sp_df3.reset_index(inplace=True)
```

In [234…

```
## Setting Weekend and Holiday Binary Values
```

```
raw_sp_df3['Transaction Date'] = pd.to_datetime(raw_sp_df3['Transaction Date'])

# Convert 'Transaction Date' to date format
raw_sp_df3['Transaction Date'] = raw_sp_df3['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
raw_sp_df3['IsWeekend'] = (raw_sp_df3['Transaction Date'].apply(lambda x: x.weekday()) >
raw_sp_df3['IsHoliday'] = raw_sp_df3['Transaction Date'].isin(holiday_dates).astype(int)
```

In [235…

```
# Linear Regression for Self Price PED RAW Supermarket

# Get unique product codes from the 'Inventory Code' column
unique_inventory = raw_sp_df3['Inventory Code'].unique()

# Initialize lists to store results
inventory_codes = []
coefficients_price = []
p_values_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []

# Loop through the unique product codes
for inventory_code in unique_inventory:
    inventory_data = raw_sp_df3[raw_sp_df3['Inventory Code'] == inventory_code]

    # Create features and target variables
    X = sm.add_constant(inventory_data[['Log Price', 'IsWeekend', 'IsHoliday', 'Seasonal

    model = sm.OLS(inventory_data['Log Qty'], X).fit()

    # Append results to lists
    inventory_codes.append(inventory_code)
    coefficients_price.append(model.params['Log Price'])
    p_values_price.append(model.pvalues['Log Price'])
    coefficients_seasonal.append(model.params['Seasonal'])
    p_values_seasonal.append(model.pvalues['Seasonal'])
    coefficients_is_weekend.append(model.params['IsWeekend'])
    p_values_is_weekend.append(model.pvalues['IsWeekend'])
    coefficients_is_holiday.append(model.params['IsHoliday'])
    p_values_is_holiday.append(model.pvalues['IsHoliday'])

# Create a DataFrame with the aggregated coefficients and p-values
table_data = {
```

```
    'Inventory Code': inventory_codes,
    'Coefficient (Log Price)': coefficients_price,
    'P-Value (Log Price)': p_values_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

own_price_raw_sp_df = pd.DataFrame(table_data)

# Apply the significance labels
for column in own_price_raw_sp_df.columns:
    if 'P-Value' in column:
        significance_column = 'Significance' + column.split('P-Value', 1)[1]
        own_price_raw_sp_df[significance_column] = own_price_raw_sp_df[column].apply(lam

# Display the table in the Jupyter Notebook
print("Raw Supermarket Aggregated Coefficients and Significance Table:")
display(own_price_raw_sp_df)
```

Raw Supermarket Aggregated Coefficients and Significance Table:

	Inventory Code	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	1030.0	-1.861197	1.447949e-03	0.004611	0.009128	0.265400	0.016725	-0.051182	0.839263
1	1032.0	-1.863228	1.326130e-03	0.001565	0.214564	0.136851	0.089325	-0.208475	0.257508
2	1033.0	-2.070260	7.813732e-05	-0.000437	0.769899	0.081993	0.384223	-0.054431	0.801241
3	1036.0	-3.154435	6.882544e-07	-0.000090	0.940548	0.122849	0.109638	-0.239883	0.175236
4	1037.0	-2.679994	1.870554e-08	0.000843	0.440729	0.125432	0.071569	-0.180302	0.259643

In [236…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = raw_sp_df.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
raw_sp_min_max_data = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
raw_sp_min_max_data.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', 'M

# Display the result
print(raw_sp_min_max_data)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1030.0       4.40        5.4     15.6   4305.6
1          1032.0       4.60        5.4     31.2   2386.8
2          1033.0       3.75        4.7     15.6   2106.0
3          1036.0       5.10        5.8     46.8   1903.2
4          1037.0       4.95        5.8     93.6   1934.4
```

In [237…

```
## Price Optimization Using Min & Max Qty & Price as bounds

from scipy.optimize import minimize

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_raw_sp_df, raw_sp_min_max_data, on='Inventory Code')

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, qty, coefficient_price):
    return -price * qty * coefficient_price  # Negative sign because scipy.optimize mini

# Function to be minimized (negative revenue for maximization)
def objective_function(prices):
    total_revenue = 0
    for index, row in merged_data.iterrows():
        min_price, max_price = row['Min Price'], row['Max Price']
        min_qty, max_qty = row['Min Qty'], row['Max Qty']
        coefficient_price = row['Coefficient (Log Price)']

        # Ensure prices are within bounds
        price = max(min(max_price, prices[index]), min_price)

        # Calculate revenue
        qty = max(min(max_qty, prices[len(merged_data) + index]), min_qty)
        total_revenue += calculate_revenue(price, qty, coefficient_price)

    return total_revenue

# Initial guess for prices
initial_prices = merged_data['Min Price'].values.tolist() + merged_data['Min Qty'].value

# Constraints
price_constraints = [(min_price, max_price) for min_price, max_price in zip(merged_data[
qty_constraints = [(min_qty, max_qty) for min_qty, max_qty in zip(merged_data['Min Qty']
constraints = price_constraints + qty_constraints

# Optimization
result = minimize(objective_function, initial_prices, constraints={'type': 'ineq', 'fun'
                  bounds=constraints)

# Extract optimized prices and quantities
optimized_prices = result.x[:len(merged_data)]
optimized_quantities = result.x[len(merged_data):]

# Display optimized prices and quantities
optimized_data = merged_data.copy()
optimized_data['Optimized Price'] = optimized_prices
optimized_data['Optimized Quantity'] = optimized_quantities

print("Optimized Prices and Quantities:")
print(optimized_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Optimized Prices and Quantities:
   Inventory Code  Optimized Price  Optimized Quantity
0          1030.0             4.40                15.6
1          1032.0             4.60                31.2
2          1033.0             3.75                15.6
3          1036.0             5.10                46.8
4          1037.0             4.95                93.6
```

In [238…

```
## Price Optimization Using Min Price as bounds

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)
```

```
# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_raw_sp_df, raw_sp_min_max_data, on='Inventory Code')

# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in merged_data
for index, row in merged_data.iterrows():
    inventory_code = row['Inventory Code']
    min_price = row['Min Price']

    # Get elasticity from own_price_rte_sp_df based on the inventory code
    elasticity = own_price_raw_sp_df.loc[own_price_raw_sp_df['Inventory Code'] == invent

    # Set bounds for optimization
    bounds = [(min_price, None), (None, None)]  # Remove lower bound for quantity

    # Perform optimization
    result = minimize(objective_function, x0=[min_price, row['Min Qty']], args=(elastici

    # Extract optimized price and quantity
    optimized_price, optimized_quantity = result.x

    # Append results to lists
    optimized_prices.append(optimized_price)
    optimized_quantities.append(optimized_quantity)

# Add optimized prices and quantities to the DataFrame
merged_data['Optimized Price'] = optimized_prices
merged_data['Optimized Quantity'] = optimized_quantities

# Display the DataFrame
print(merged_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
   Inventory Code  Optimized Price  Optimized Quantity
0          1030.0    4.008584e+136      -1.733509e+155
1          1032.0    8.802461e+140      -5.065684e+149
2          1033.0    1.455373e+142      -1.038419e+151
3          1036.0    4.646313e+136      -5.393006e+153
4          1037.0    4.509657e+136      -5.575671e+153
```

Ready to Eat(RTE) Retail Cross PED and Price Optimization

In [239…

```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rte_rt_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rte_rt_df = rte_rt_df[rte_rt_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rte_rt_df = filtered_rte_rt_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [240…

```
#Using mode of each inventory code to replace missing data in dataframe
```

```
import pandas as pd

# Assuming your DataFrame is named rte_sp_df
# Step 1: Identify unique inventory codes for each transaction date
unique_inventory_codes = rte_rt_df.groupby('Transaction Date')['Inventory Code'].unique(

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = rte_rt_df.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(cal

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(rte_rt_df['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
rte_rt_df = pd.concat([rte_rt_df, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
rte_rt_df = rte_rt_df.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
rte_rt_df.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(rte_rt_df)
```

```
    Transaction Date  Inventory Code    Qty  Price Per Unit
0         2022-07-01          1331.0  124.8             1.8
1         2022-07-01          1332.0  145.6             1.8
2         2022-07-01          1333.0   83.2             1.8
3         2022-07-02          1331.0   52.0             1.8
4         2022-07-02          1332.0   41.6             1.8
..               ...             ...    ...             ...
340       2023-05-19          1332.0   10.4             1.8
341       2023-05-19          1333.0   20.8             1.8
342       2023-05-20          1331.0   41.6             1.8
343       2023-05-20          1332.0   41.6             1.8
344       2023-05-20          1333.0   20.8             1.8

[345 rows x 4 columns]
```

In [241…

```
# Create a list of unique inventory codes
unique_inventory_codes = rte_rt_df['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1331.0, 1332.0)
```

```
Inventory Code Pair: (1331.0, 1333.0)
Inventory Code Pair: (1332.0, 1333.0)
```

In [242…

```python
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rte_rt_df[rte_rt_df['Inventory Code'] == code1]
    data_code2 = rte_rt_df[rte_rt_df['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:     Transaction Date  Inventory Code    Qty  Price Per Unit
1          2022-07-01          1332.0  145.6             1.8
4          2022-07-02          1332.0   41.6             1.8
7          2022-07-05          1332.0   52.0             1.8
10         2022-07-06          1332.0   10.4             1.8
13         2022-07-07          1332.0   20.8             1.8
..                ...             ...    ...             ...
331        2023-04-27          1332.0   41.6             1.8
334        2023-05-05          1332.0   41.6             1.8
337        2023-05-16          1332.0   10.4             1.8
340        2023-05-19          1332.0   10.4             1.8
343        2023-05-20          1332.0   41.6             1.8

[115 rows x 4 columns]
Data for Code Pair 2:     Transaction Date  Inventory Code   Qty  Price Per Unit
2          2022-07-01          1333.0  83.2             1.8
5          2022-07-02          1333.0  20.8             1.8
8          2022-07-05          1333.0  20.8             1.8
11         2022-07-06          1333.0  20.8             1.8
14         2022-07-07          1333.0  20.8             1.8
..                ...             ...   ...             ...
332        2023-04-27          1333.0  20.8             1.8
335        2023-05-05          1333.0  20.8             1.8
338        2023-05-16          1333.0  20.8             1.8
341        2023-05-19          1333.0  20.8             1.8
344        2023-05-20          1333.0  20.8             1.8

[115 rows x 4 columns]
```

In [243…

```python
#Creating the cross reference for each pair on a daily basis
# Assuming your DataFrame is named rte_sp_df
# Create an empty DataFrame to store the results
rte_rt_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1',

# Create a list of unique inventory codes
unique_inventory_codes = rte_rt_df['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rte_rt_df[rte_rt_df['Inventory Code'] == code1]
    data_code2 = rte_rt_df[rte_rt_df['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    rte_rt_elasticity_df = pd.concat([rte_rt_elasticity_df, results])
```

```
# Print or inspect the elasticity results
print(rte_rt_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1331.0                1332.0
1         2022-07-02                1331.0                1332.0
2         2022-07-05                1331.0                1332.0
3         2022-07-06                1331.0                1332.0
4         2022-07-07                1331.0                1332.0
..               ...                   ...                   ...
110       2023-04-27                1332.0                1333.0
111       2023-05-05                1332.0                1333.0
112       2023-05-16                1332.0                1333.0
113       2023-05-19                1332.0                1333.0
114       2023-05-20                1332.0                1333.0

     Relative_Price  Relative_Volume
0               1.0         0.857143
1               1.0         1.250000
2               1.0         1.200000
3               1.0         1.000000
4               1.0         0.500000
..              ...              ...
110             1.0         2.000000
111             1.0         2.000000
112             1.0         0.500000
113             1.0         0.500000
114             1.0         2.000000

[345 rows x 5 columns]
```

In [244…

```
## Creating Log Relative Price and Log Relative Volume in columns

rte_rt_elasticity_df.loc[:, 'Log Relative Price'] = np.log(rte_rt_elasticity_df['Relativ
rte_rt_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(rte_rt_elasticity_df['Relati
```

In [245…

```
#Seasonality
rte_rt_elasticity_df.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rte_rt_elasticity_df['Relative_Volume']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
```

```
ax4.legend(loc='upper left')

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

In [246…

```
#Copying Seasonal Value to Data Frame and resetting the index

rte_rt_elasticity_df = rte_rt_elasticity_df.copy()  # Create a copy of the DataFrame
rte_rt_elasticity_df['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the c

rte_rt_elasticity_df.reset_index(inplace=True)
```

In [247…

```
## Setting Weekend and Holiday Binary Values

# Assuming rte_elasticity_df['Transaction Date'] is in string format
# Convert 'Transaction Date' to datetime64 format
rte_rt_elasticity_df['Transaction Date'] = pd.to_datetime(rte_rt_elasticity_df['Transact

# Convert 'Transaction Date' to date format
rte_rt_elasticity_df['Transaction Date'] = rte_rt_elasticity_df['Transaction Date'].dt.d

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]
```

```
# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rte_rt_elasticity_df['IsWeekend'] = (rte_rt_elasticity_df['Transaction Date'].apply(lamb
rte_rt_elasticity_df['IsHoliday'] = rte_rt_elasticity_df['Transaction Date'].isin(holida
```

In [248…

```
import pandas as pd
import statsmodels.api as sm

# Group the data by unique product pairs
unique_product_pairs = rte_rt_elasticity_df[['Inventory Code_code1', 'Inventory Code_cod

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = rte_rt_elasticity_df[(rte_rt_elasticity_df['Inventory Code_code1'] == inven

    X = sm.add_constant(subset[['Log Relative Price', 'Seasonal', 'IsWeekend', 'IsHolida
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        coefficients_seasonal.append(model.params['Seasonal'])
        p_values_seasonal.append(model.pvalues['Seasonal'])
        coefficients_is_weekend.append(model.params['IsWeekend'])
        p_values_is_weekend.append(model.pvalues['IsWeekend'])
        coefficients_is_holiday.append(model.params['IsHoliday'])
        p_values_is_holiday.append(model.pvalues['IsHoliday'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

rte_rt_table_df = pd.DataFrame(table_data)
```

```
rte_rt_table_df['Type'] = rte_rt_table_df['Coefficient (Log Relative Price)'].apply(lamb
rte_rt_table_df['Significance (Log Relative Price)'] = rte_rt_table_df['P-Value (Log Rel
rte_rt_table_df['Significance (Seasonal)'] = rte_rt_table_df['P-Value (Seasonal)'].apply
rte_rt_table_df['Significance (IsWeekend)'] = rte_rt_table_df['P-Value (IsWeekend)'].app
rte_rt_table_df['Significance (IsHoliday)'] = rte_rt_table_df['P-Value (IsHoliday)'].app

# Create a DataFrame to store the OLS results
rte_rt_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(rte_rt_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)	
0	(1331.0, 1332.0)	6.762399e-17	0.369232	-0.175681	0.358112	0.093968	0.565135	0.0	NaN	
1	(1331.0, 1333.0)	-5.862705e-15	0.003766	0.784934	0.003778	-0.103159	0.652020	0.0	NaN	C
2	(1332.0, 1333.0)	-2.126462e-15	0.085877	0.431095	0.081831	-0.105540	0.610359	0.0	NaN	C

In [249…

```
# Initialize lists to store the optimization results
optimized_prices_list = []
maximized_revenue_list = []

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Iterate over rows in the DataFrame
for _, row in rte_rt_table_df.iterrows():
    # Extract the cross-price elasticity coefficient from the table
    cross_price_elasticity_coefficient = row['Coefficient (Log Relative Price)']

    # Initial guess for prices (you may need to adjust this based on your specific conte
    initial_prices = [1.0, 1.0]  # Assuming you are optimizing prices for a pair of prod

    # Define bounds on prices
    bounds = [(0, None), (0, None)]  # Separate bounds for each item in the pair

    # Run the optimization
    result = minimize(objective_function, initial_prices, args=(cross_price_elasticity_c

    # Extract the optimized prices and maximized revenue
    optimized_prices = result.x.tolist()
    maximized_revenue = -result.fun  # Negate to get the actual maximum revenue

    # Append the results to the lists
    optimized_prices_list.append(optimized_prices)
    maximized_revenue_list.append(maximized_revenue)
```

```
# Add new columns to the DataFrame
rte_rt_table_df['Optimized Prices'] = optimized_prices_list
rte_rt_table_df['Maximized Revenue'] = maximized_revenue_list

# Display the updated DataFrame
display(rte_rt_table_df[['Inventory Code Pair', 'Optimized Prices', 'Maximized Revenue']]
```

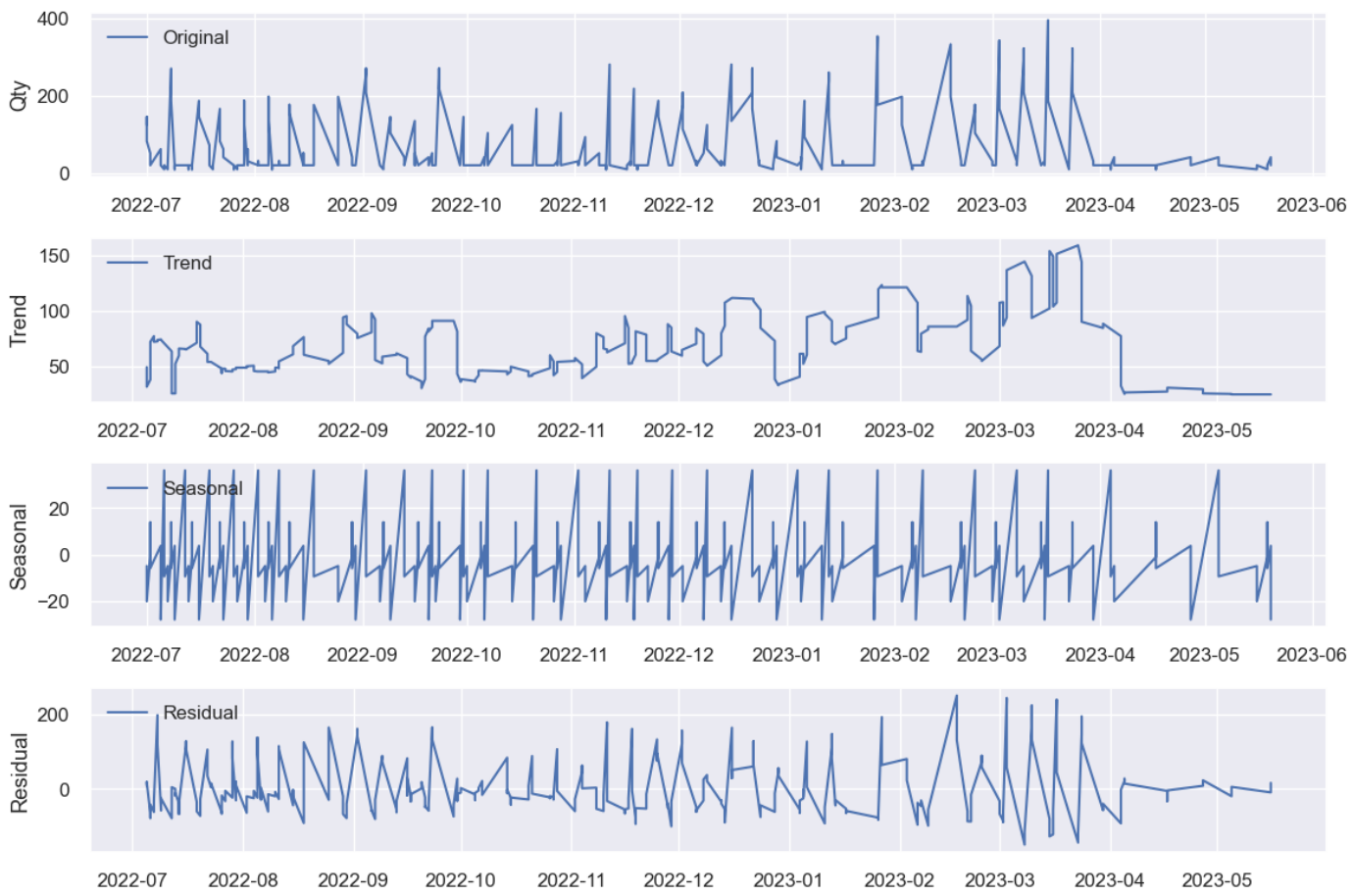
	Inventory Code Pair	Optimized Prices	Maximized Revenue
0	(1331.0, 1332.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
1	(1331.0, 1333.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf
2	(1332.0, 1333.0)	[2.0000000507111565e+150, 2.0000000507111565e+...	inf

Method 2 - Seperate Cross Price in 2 Entries

In [250…
```
rte_rt_df2 = rte_rt_df
```

In [251…
```
## Creating Log Relative Price and Log Relative Volume in columns

rte_rt_df2.loc[:, 'Log Price'] = np.log(rte_rt_df2['Price Per Unit'])
rte_rt_df2.loc[:, 'Log Qty'] = np.log(rte_rt_df2['Qty'])
```

In [252…
```
rte_rt_df2.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rte_rt_df2['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [253…
```
#Copying Seasonal Value to Data Frame and resetting the index

rte_rt_df2 = rte_rt_df2.copy()  # Create a copy of the DataFrame
rte_rt_df2['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rte_rt_df2.reset_index(inplace=True)
```

In [254…
```
rte_rt_df2['Transaction Date'] = pd.to_datetime(rte_rt_df2['Transaction Date'])

# Convert 'Transaction Date' to date format
rte_rt_df2['Transaction Date'] = rte_rt_df2['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rte_rt_df2['IsWeekend'] = (rte_rt_df2['Transaction Date'].apply(lambda x: x.weekday()) >
rte_rt_df2['IsHoliday'] = rte_rt_df2['Transaction Date'].isin(holiday_dates).astype(int)
```

In [255…
```
import statsmodels.api as sm
import pandas as pd
```

```
# Assuming rte_sp_df2 is your DataFrame
# Assuming you have separate demand functions for each product
products = rte_rt_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rte_rt_df2[(rte_rt_df2['Inventory Code'] == products[i]) | (rte_rt_d

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_1.params['Seasonal'],
            'P-Value (Seasonality)': model_1.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_1.params['IsWeekend'],
            'P-Value (IsWeekend)': model_1.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_1.params['IsHoliday'],
            'P-Value (IsHoliday)': model_1.pvalues['IsHoliday'],
        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_2.params['Seasonal'],
            'P-Value (Seasonality)': model_2.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_2.params['IsWeekend'],
            'P-Value (IsWeekend)': model_2.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_2.params['IsHoliday'],
            'P-Value (IsHoliday)': model_2.pvalues['IsHoliday'],
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[255]:

Pair	Product	Coefficient	P-Value	Coefficient	P-Value	Coefficient	P-Value	Coefficient

			(Log Price)	(Log Price)	(Seasonality)	(Seasonality)	(IsWeekend)	(IsWeekend)	(IsHoliday)	(I
0	1331.0 - 1332.0	1331.0	1.599494	2.444196e-57	0.012001	0.050342	-0.287397	0.354276	0.0	
1	1331.0 - 1332.0	1332.0	1.613490	1.275551e-58	0.008892	0.128463	-0.345586	0.274585	0.0	
2	1331.0 - 1333.0	1331.0	1.599494	2.444196e-57	0.012001	0.050342	-0.287397	0.354276	0.0	
3	1331.0 - 1333.0	1333.0	1.616669	3.910446e-44	0.010203	0.261058	-0.232403	0.353478	0.0	
4	1332.0 - 1333.0	1332.0	1.613490	1.275551e-58	0.008892	0.128463	-0.345586	0.274585	0.0	
5	1332.0 - 1333.0	1333.0	1.616669	3.910446e-44	0.010203	0.261058	-0.232403	0.353478	0.0	

In [256…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rte_rt_df2.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rte_rt_min_max_data2 = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rte_rt_min_max_data2.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', '

# Display the result
print(rte_rt_min_max_data2)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1331.0        1.8        1.8     10.4    384.8
1          1332.0        1.8        1.8     10.4    395.2
2          1333.0        1.8        1.8     10.4    218.4
```

In [257…

```
import statsmodels.api as sm
import pandas as pd

# Assuming rte_rt_df2 is your DataFrame
# Assuming you have separate demand functions for each product
products = rte_rt_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Initialize lists to store mean price and quantity
mean_price_list = []
mean_quantity_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
```

```
        # Subset the DataFrame for the current pair
        pair_data = rte_rt_df2[(rte_rt_df2['Inventory Code'] == products[i]) | (rte_rt_d

        # Calculate mean price and quantity for each product in the pair
        mean_price_i = pair_data[pair_data['Inventory Code'] == products[i]]['Price Per
        mean_quantity_i = pair_data[pair_data['Inventory Code'] == products[i]]['Qty'].m

        mean_price_j = pair_data[pair_data['Inventory Code'] == products[j]]['Price Per
        mean_quantity_j = pair_data[pair_data['Inventory Code'] == products[j]]['Qty'].m

        # Store the mean price and quantity in the lists
        mean_price_list.extend([mean_price_i, mean_price_j])
        mean_quantity_list.extend([mean_quantity_i, mean_quantity_j])

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Mean Price': mean_price_i,
            'Mean Quantity': mean_quantity_i,

        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Mean Price': mean_price_j,
            'Mean Quantity': mean_quantity_j,

        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[257]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Mean Price	Mean Quantity
0	1331.0 - 1332.0	1331.0	1.599494	2.444196e-57	1.8	76.598261
1	1331.0 - 1332.0	1332.0	1.613490	1.275551e-58	1.8	75.151304
2	1331.0 - 1333.0	1331.0	1.599494	2.444196e-57	1.8	76.598261

3	1331.0 - 1333.0	1333.0	1.616669	3.910446e-44	1.8	52.090435
4	1332.0 - 1333.0	1332.0	1.613490	1.275551e-58	1.8	75.151304
5	1332.0 - 1333.0	1333.0	1.616669	3.910446e-44	1.8	52.090435

In [258…
```
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in results_df
for index, row in results_df.iterrows():
    inventory_code = row['Product']
    mean_price = row['Mean Price']
    mean_quantity = row['Mean Quantity']
    elasticity = row['Coefficient (Log Price)']

    # Set bounds for optimization
    bounds = [(0, None)]  # Optimize only price

    # Perform optimization
    result = minimize(calculate_revenue, x0=[mean_price], args=(elasticity, mean_quantit

    # Extract optimized price
    optimized_price = result.x[0]

    optimized_prices.append(optimized_price)

# Add optimized prices to the results DataFrame
results_df['Optimized Price'] = optimized_prices

# Display the updated results_df
print(results_df[['Pair', 'Product', 'Optimized Price']])
```

```
               Pair Product  Optimized Price
0  1331.0 - 1332.0  1331.0              0.0
1  1331.0 - 1332.0  1332.0              0.0
2  1331.0 - 1333.0  1331.0              0.0
3  1331.0 - 1333.0  1333.0              0.0
4  1332.0 - 1333.0  1332.0              0.0
5  1332.0 - 1333.0  1333.0              0.0
```

Ready to Eat(RTE) Retail Self PED and Price Optimization

In [259…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rte_rt_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rte_rt_df = rte_rt_df[rte_rt_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rte_rt_df = filtered_rte_rt_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [260…
```
rte_rt_df3 = rte_rt_df
```

In [261…
```
## Creating Log Price and Log Quantity in columns

rte_rt_df3.loc[:, 'Log Price'] = np.log(rte_rt_df3['Price Per Unit'])
rte_rt_df3.loc[:, 'Log Qty'] = np.log(rte_rt_df3['Qty'])
```

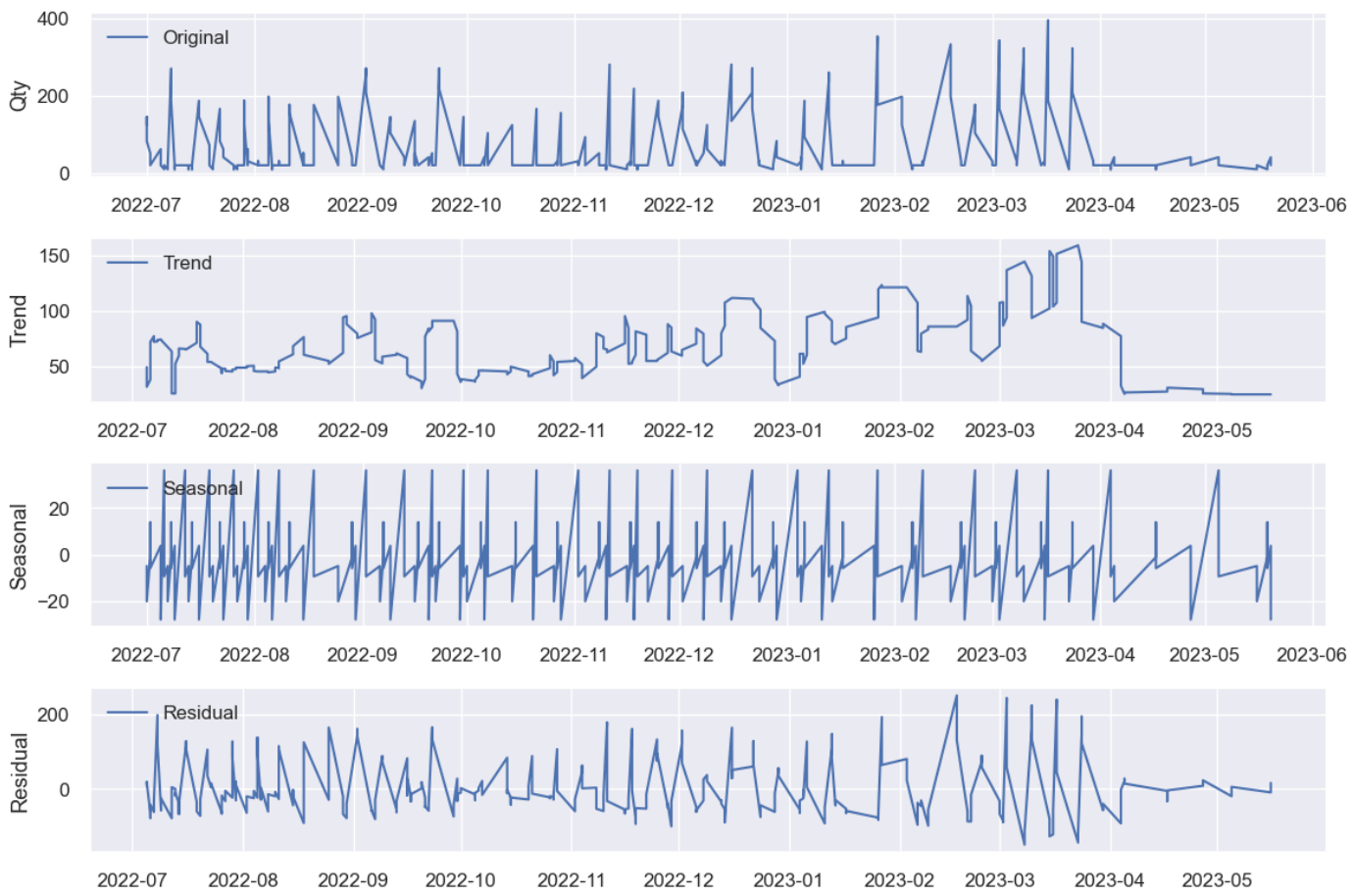
In [262…
```
rte_rt_df
```

Out[262]:

	Transaction Date	Inventory Code	Qty	Price Per Unit	Log Price	Log Qty
0	2022-07-01	1331.0	124.8	1.8	0.587787	4.826712
1	2022-07-01	1332.0	145.6	1.8	0.587787	4.980863
2	2022-07-01	1333.0	83.2	1.8	0.587787	4.421247
3	2022-07-02	1331.0	52.0	1.8	0.587787	3.951244
4	2022-07-02	1332.0	41.6	1.8	0.587787	3.728100
...	...	...	...	...	...	...
340	2023-05-19	1332.0	10.4	1.8	0.587787	2.341806
341	2023-05-19	1333.0	20.8	1.8	0.587787	3.034953
342	2023-05-20	1331.0	41.6	1.8	0.587787	3.728100
343	2023-05-20	1332.0	41.6	1.8	0.587787	3.728100
344	2023-05-20	1333.0	20.8	1.8	0.587787	3.034953

345 rows × 6 columns

In [263…
```
## Seasonality Values
import statsmodels.api as sm
from statsmodels.tsa.seasonal import seasonal_decompose

rte_rt_df3.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rte_rt_df3['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [264…

```
#Copying Seasonal Value to Data Frame and resetting the index

rte_rt_df3 = rte_rt_df3.copy()  # Create a copy of the DataFrame
rte_rt_df3['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rte_rt_df3.reset_index(inplace=True)
```

In [265…

```
## Setting Weekend and Holiday Binary Values

rte_rt_df3['Transaction Date'] = pd.to_datetime(rte_rt_df3['Transaction Date'])

# Convert 'Transaction Date' to date format
rte_rt_df3['Transaction Date'] = rte_rt_df3['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rte_rt_df3['IsWeekend'] = (rte_rt_df3['Transaction Date'].apply(lambda x: x.weekday()) >
rte_rt_df3['IsHoliday'] = rte_rt_df3['Transaction Date'].isin(holiday_dates).astype(int)
```

In [266…

```
# Linear Regression for Self Price PED RTE Retail

# Get unique product codes from the 'Inventory Code' column
unique_inventory = rte_rt_df3['Inventory Code'].unique()

# Initialize lists to store results
inventory_codes = []
coefficients_price = []
p_values_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []

# Loop through the unique product codes
for inventory_code in unique_inventory:
    inventory_data = rte_rt_df3[rte_rt_df3['Inventory Code'] == inventory_code]

    # Create features and target variables
    X = sm.add_constant(inventory_data[['Log Price', 'IsWeekend', 'IsHoliday', 'Seasonal

    model = sm.OLS(inventory_data['Log Qty'], X).fit()

    # Append results to lists
    inventory_codes.append(inventory_code)
    coefficients_price.append(model.params['Log Price'])
    p_values_price.append(model.pvalues['Log Price'])
    coefficients_seasonal.append(model.params['Seasonal'])
    p_values_seasonal.append(model.pvalues['Seasonal'])
    coefficients_is_weekend.append(model.params['IsWeekend'])
    p_values_is_weekend.append(model.pvalues['IsWeekend'])
    coefficients_is_holiday.append(model.params['IsHoliday'])
    p_values_is_holiday.append(model.pvalues['IsHoliday'])

# Create a DataFrame with the aggregated coefficients and p-values
table_data = {
    'Inventory Code': inventory_codes,
    'Coefficient (Log Price)': coefficients_price,
    'P-Value (Log Price)': p_values_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

own_price_rte_rt_df = pd.DataFrame(table_data)

# Apply the significance labels
for column in own_price_rte_rt_df.columns:
    if 'P-Value' in column:
        significance_column = 'Significance' + column.split('P-Value', 1)[1]
        own_price_rte_rt_df[significance_column] = own_price_rte_rt_df[column].apply(lam

# Display the table in the Jupyter Notebook
print("Ready to Eat Retail Aggregated Coefficients and Significance Table:")
display(own_price_rte_rt_df)
```

```
Ready to Eat Retail Aggregated Coefficients and Significance Table:
```

	Inventory Code	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)

0	1331.0	1.599494	2.444196e-57	0.012001	0.050342	-0.287397	0.354276	0.0	NaN
1	1332.0	1.613490	1.275551e-58	0.008892	0.128463	-0.345586	0.274585	0.0	NaN
2	1333.0	6.295977	3.910446e-44	0.010203	0.261058	-0.232403	0.353478	0.0	NaN

In [267…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rte_rt_df.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rte_rt_min_max_data = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rte_rt_min_max_data.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', 'M

# Display the result
print(rte_rt_min_max_data)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1331.0        1.8        1.8     10.4    384.8
1          1332.0        1.8        1.8     10.4    395.2
2          1333.0        1.8        1.8     10.4    218.4
```

In [268…

```
## Price Optimization Using Min & Max Qty & Price as bounds

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rte_rt_df, rte_rt_min_max_data, on='Inventory Code')

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, qty, coefficient_price):
    return -price * qty * coefficient_price  # Negative sign because scipy.optimize mini

# Function to be minimized (negative revenue for maximization)
def objective_function(prices):
    total_revenue = 0
    for index, row in merged_data.iterrows():
        min_price, max_price = row['Min Price'], row['Max Price']
        min_qty, max_qty = row['Min Qty'], row['Max Qty']
        coefficient_price = row['Coefficient (Log Price)']

        # Ensure prices are within bounds
        price = max(min(max_price, prices[index]), min_price)

        # Calculate revenue
        qty = max(min(max_qty, prices[len(merged_data) + index]), min_qty)
        total_revenue += calculate_revenue(price, qty, coefficient_price)

    return total_revenue

# Initial guess for prices
initial_prices = merged_data['Min Price'].values.tolist() + merged_data['Min Qty'].value

# Constraints
price_constraints = [(min_price, max_price) for min_price, max_price in zip(merged_data[
qty_constraints = [(min_qty, max_qty) for min_qty, max_qty in zip(merged_data['Min Qty']
constraints = price_constraints + qty_constraints
```

```
# Optimization
result = minimize(objective_function, initial_prices, constraints={'type': 'ineq', 'fun'
                  bounds=constraints)

# Extract optimized prices and quantities
optimized_prices = result.x[:len(merged_data)]
optimized_quantities = result.x[len(merged_data):]

# Display optimized prices and quantities
optimized_data = merged_data.copy()
optimized_data['Optimized Price'] = optimized_prices
optimized_data['Optimized Quantity'] = optimized_quantities

print("Optimized Prices and Quantities:")
print(optimized_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Optimized Prices and Quantities:
   Inventory Code  Optimized Price  Optimized Quantity
0          1331.0              1.8               384.8
1          1332.0              1.8               395.2
2          1333.0              1.8               218.4
```

In [269…

```
## Price Optimization Using Min Price as bounds

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rte_rt_df, rte_rt_min_max_data, on='Inventory Code')

# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in merged_data
for index, row in merged_data.iterrows():
    inventory_code = row['Inventory Code']
    min_price = row['Min Price']

    # Get elasticity from own_price_rte_sp_df based on the inventory code
    elasticity = own_price_rte_rt_df.loc[own_price_rte_rt_df['Inventory Code'] == invent

    # Set bounds for optimization
    bounds = [(min_price, None), (None, None)]  # Remove lower bound for quantity

    # Perform optimization
    result = minimize(objective_function, x0=[min_price, row['Min Qty']], args=(elastici

    # Extract optimized price and quantity
    optimized_price, optimized_quantity = result.x

    # Append results to lists
    optimized_prices.append(optimized_price)
    optimized_quantities.append(optimized_quantity)

# Add optimized prices and quantities to the DataFrame
merged_data['Optimized Price'] = optimized_prices
merged_data['Optimized Quantity'] = optimized_quantities
```

```
# Display the DataFrame
print("Ready to Eat Retail Aggregated Coefficients and Significance Table:")
print(merged_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Ready to Eat Retail Aggregated Coefficients and Significance Table:
   Inventory Code  Optimized Price  Optimized Quantity
0          1331.0    1.855070e+147       9.701064e+146
1          1332.0    2.009991e+147       1.047590e+147
2          1333.0    3.906230e+142       1.289388e+143
```

Ready to Cook(RTC) Retail Cross PED and Price Optimization

In [270…

```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rtc_rt_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rtc_rt_df = rtc_rt_df[rtc_rt_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rtc_rt_df = filtered_rtc_rt_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [271…

```
#Using mode of each inventory code to replace missing data in dataframe

# Step 1: Identify unique inventory codes for each transaction date
unique_inventory_codes = rtc_rt_df.groupby('Transaction Date')['Inventory Code'].unique(

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = rtc_rt_df.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(cal

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(rtc_rt_df['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
rtc_rt_df = pd.concat([rtc_rt_df, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
rtc_rt_df = rtc_rt_df.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
rtc_rt_df.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(rtc_rt_df)
```

```
     Transaction Date  Inventory Code   Qty  Price Per Unit
0          2022-07-01          1155.0  46.8            10.9
1          2022-07-01          1158.0   2.6            11.5
```

```
2          2022-07-01          1217.0   2.6           100.0
3          2022-07-01          1218.0  32.5            10.8
4          2022-07-01          1316.0  26.0            19.0
...               ...             ...   ...             ...
1390       2023-05-31          1155.0  23.4            11.3
1391       2023-05-31          1158.0   1.3            11.5
1392       2023-05-31          1217.0   2.6           100.0
1393       2023-05-31          1218.0  19.5            10.8
1394       2023-05-31          1316.0  14.3            19.0

[1395 rows x 4 columns]
```

In [272…
```python
# Create a list of unique inventory codes
unique_inventory_codes = rtc_rt_df['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1155.0, 1158.0)
Inventory Code Pair: (1155.0, 1217.0)
Inventory Code Pair: (1155.0, 1218.0)
Inventory Code Pair: (1155.0, 1316.0)
Inventory Code Pair: (1158.0, 1217.0)
Inventory Code Pair: (1158.0, 1218.0)
Inventory Code Pair: (1158.0, 1316.0)
Inventory Code Pair: (1217.0, 1218.0)
Inventory Code Pair: (1217.0, 1316.0)
Inventory Code Pair: (1218.0, 1316.0)
```

In [273…
```python
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rtc_rt_df[rtc_rt_df['Inventory Code'] == code1]
    data_code2 = rtc_rt_df[rtc_rt_df['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:      Transaction Date  Inventory Code   Qty  Price Per Unit
3          2022-07-01          1218.0  32.5            10.8
8          2022-07-02          1218.0  39.0            10.8
13         2022-07-04          1218.0  19.5            10.8
18         2022-07-05          1218.0  19.5            10.8
23         2022-07-06          1218.0  13.0            10.8
...               ...             ...   ...             ...
1373       2023-05-26          1218.0  26.0            10.8
1378       2023-05-27          1218.0  32.5            10.8
1383       2023-05-29          1218.0  19.5            10.8
1388       2023-05-30          1218.0   6.5            10.8
1393       2023-05-31          1218.0  19.5            10.8

[279 rows x 4 columns]
Data for Code Pair 2:      Transaction Date  Inventory Code   Qty  Price Per Unit
4          2022-07-01          1316.0  26.0            19.0
9          2022-07-02          1316.0  32.5            19.0
14         2022-07-04          1316.0  32.5            19.0
19         2022-07-05          1316.0  10.4            19.0
24         2022-07-06          1316.0   6.5            19.0
...               ...             ...   ...             ...
1374       2023-05-26          1316.0  19.5            19.0
1379       2023-05-27          1316.0  19.5            19.0
```

```
1384       2023-05-29          1316.0  19.5            19.0
1389       2023-05-30          1316.0  13.0            19.0
1394       2023-05-31          1316.0  14.3            19.0

[279 rows x 4 columns]
```

In [274…

```python
#Creating the cross reference for each pair on a daily basis

# Create an empty DataFrame to store the results
rtc_rt_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1',

# Create a list of unique inventory codes
unique_inventory_codes = rtc_rt_df['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = rtc_rt_df[rtc_rt_df['Inventory Code'] == code1]
    data_code2 = rtc_rt_df[rtc_rt_df['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    rtc_rt_elasticity_df = pd.concat([rtc_rt_elasticity_df, results])

# Print or inspect the elasticity results
print(rtc_rt_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1155.0                1158.0
1         2022-07-02                1155.0                1158.0
2         2022-07-04                1155.0                1158.0
3         2022-07-05                1155.0                1158.0
4         2022-07-06                1155.0                1158.0
..               ...                   ...                   ...
274       2023-05-26                1218.0                1316.0
275       2023-05-27                1218.0                1316.0
276       2023-05-29                1218.0                1316.0
277       2023-05-30                1218.0                1316.0
278       2023-05-31                1218.0                1316.0

     Relative_Price  Relative_Volume
0          0.947826        18.000000
1          0.982609        15.000000
2          0.957627         9.000000
3          0.982609        36.000000
4          0.982609        18.000000
..              ...              ...
274        0.568421         1.333333
275        0.568421         1.666667
276        0.568421         1.000000
277        0.568421         0.500000
278        0.568421         1.363636

[2790 rows x 5 columns]
```

In [275…

```python
## Creating Log Relative Price and Log Relative Volume in columns

rtc_rt_elasticity_df.loc[:, 'Log Relative Price'] = np.log(rtc_rt_elasticity_df['Relativ
rtc_rt_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(rtc_rt_elasticity_df['Relati
```

In [276…

```
#Seasonality
rtc_rt_elasticity_df.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rtc_rt_elasticity_df['Relative_Volume']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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

In [277…

```
#Copying Seasonal Value to Data Frame and resetting the index
```

```
rtc_rt_elasticity_df = rtc_rt_elasticity_df.copy()  # Create a copy of the DataFrame
rtc_rt_elasticity_df['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the c

rtc_rt_elasticity_df.reset_index(inplace=True)
```

In [278…

```
## Setting Weekend and Holiday Binary Values

rtc_rt_elasticity_df['Transaction Date'] = pd.to_datetime(rtc_rt_elasticity_df['Transact

# Convert 'Transaction Date' to date format
rtc_rt_elasticity_df['Transaction Date'] = rtc_rt_elasticity_df['Transaction Date'].dt.d

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rtc_rt_elasticity_df['IsWeekend'] = (rtc_rt_elasticity_df['Transaction Date'].apply(lamb
rtc_rt_elasticity_df['IsHoliday'] = rtc_rt_elasticity_df['Transaction Date'].isin(holida
```

In [279…

```
# Group the data by unique product pairs
unique_product_pairs = rtc_rt_elasticity_df[['Inventory Code_code1', 'Inventory Code_cod

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = rtc_rt_elasticity_df[(rtc_rt_elasticity_df['Inventory Code_code1'] == inven

    X = sm.add_constant(subset[['Log Relative Price', 'Seasonal', 'IsWeekend', 'IsHolida
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        coefficients_seasonal.append(model.params['Seasonal'])
```

```
        p_values_seasonal.append(model.pvalues['Seasonal'])
        coefficients_is_weekend.append(model.params['IsWeekend'])
        p_values_is_weekend.append(model.pvalues['IsWeekend'])
        coefficients_is_holiday.append(model.params['IsHoliday'])
        p_values_is_holiday.append(model.pvalues['IsHoliday'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

rtc_rt_table_df = pd.DataFrame(table_data)

rtc_rt_table_df['Type'] = rtc_rt_table_df['Coefficient (Log Relative Price)'].apply(lamb
rtc_rt_table_df['Significance (Log Relative Price)'] = rtc_rt_table_df['P-Value (Log Rel
rtc_rt_table_df['Significance (Seasonal)'] = rtc_rt_table_df['P-Value (Seasonal)'].apply
rtc_rt_table_df['Significance (IsWeekend)'] = rtc_rt_table_df['P-Value (IsWeekend)'].app
rtc_rt_table_df['Significance (IsHoliday)'] = rtc_rt_table_df['P-Value (IsHoliday)'].app

# Create a DataFrame to store the OLS results
rtc_rt_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(rtc_rt_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	(1155.0, 1158.0)	-18.747860	2.255866e-04	-0.017697	0.452752	-0.163211	3.670935e-01	0.264725	0.562841
1	(1155.0, 1217.0)	50.594878	2.735060e-21	0.033398	0.051499	-0.925453	5.871185e-12	-0.604084	0.061237
2	(1155.0, 1218.0)	52.046529	1.776344e-24	0.018804	0.232438	-0.986691	1.666018e-14	-0.416428	0.177481
3	(1155.0, 1316.0)	54.838247	4.758065e-26	0.020794	0.202175	-0.674492	1.073337e-07	-0.292164	0.344028
4	(1158.0, 1217.0)	-0.419512	9.714471e-01	0.084231	0.009470	-0.974244	1.090925e-04	-1.016708	0.107935
5	(1158.0, 1218.0)	11.117897	3.435342e-01	0.120541	0.000294	-0.903539	4.098600e-04	-0.835231	0.187021
6	(1158.0, 1316.0)	5.733809	6.387385e-01	0.061980	0.065076	-0.753882	3.735394e-03	-0.647654	0.323794
7	(1217.0, 1218.0)	-0.960351	6.106197e-159	-0.000819	0.943809	-0.028762	7.478526e-01	0.167266	0.454300

8	(1217.0, 1316.0)	-1.252669	1.691726e-154	-0.004810	0.677895	0.253884	4.701179e-03	0.307516	0.175790
9	(1218.0, 1316.0)	-0.024969	8.562858e-02	-0.006874	0.529030	0.274412	1.188276e-03	0.141777	0.499375

In [280…

```python
# Initialize lists to store the optimization results
optimized_prices_list = []
maximized_revenue_list = []

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Iterate over rows in the DataFrame
for _, row in rtc_rt_table_df.iterrows():
    # Extract the cross-price elasticity coefficient from the table
    cross_price_elasticity_coefficient = row['Coefficient (Log Relative Price)']

    # Initial guess for prices (you may need to adjust this based on your specific conte
    initial_prices = [1.0, 1.0]  # Assuming you are optimizing prices for a pair of prod

    # Define bounds on prices
    bounds = [(0, None), (0, None)]  # Separate bounds for each item in the pair

    # Run the optimization
    result = minimize(objective_function, initial_prices, args=(cross_price_elasticity_c

    # Extract the optimized prices and maximized revenue
    optimized_prices = result.x.tolist()
    maximized_revenue = -result.fun  # Negate to get the actual maximum revenue

    # Append the results to the lists
    optimized_prices_list.append(optimized_prices)
    maximized_revenue_list.append(maximized_revenue)

# Add new columns to the DataFrame
rtc_rt_table_df['Optimized Prices'] = optimized_prices_list
rtc_rt_table_df['Maximized Revenue'] = maximized_revenue_list

# Display the updated DataFrame
display(rtc_rt_table_df[['Inventory Code Pair', 'Optimized Prices', 'Maximized Revenue']
```

	Inventory Code Pair	Optimized Prices	Maximized Revenue
0	(1155.0, 1158.0)	[0.0, 0.0]	-0.000000e+00
1	(1155.0, 1217.0)	[1.8716407916651918e+142, 1.8716407916651918e+…	inf
2	(1155.0, 1218.0)	[62629301.53675717, 62629301.53675717]	2.080713e+17
3	(1155.0, 1316.0)	[5.22181781279691e+142, 5.22181781279691e+142]	inf
4	(1158.0, 1217.0)	[4.525694143643627e+146, 4.525694143643627e+146]	inf
5	(1158.0, 1218.0)	[1.593743968271354e+145, 1.593743968271354e+145]	inf
6	(1158.0, 1316.0)	[3.0482096988089986e+152, 3.0482096988089986e+…	inf
7	(1217.0, 1218.0)	[1.5375720844134222e+146, 1.5375720844134222e+…	inf

8	(1217.0, 1316.0)	[7.30658790937384e-08, 7.30658790937384e-08]	-1.348906e-15
9	(1218.0, 1316.0)	[1.3516008760652766e+150, 1.3516008760652766e+...	inf

Method 2 - Seperate Cross Price in 2 Entries

In [281…
```
rtc_rt_df2 = rtc_rt_df
```

In [282…
```
## Creating Log Relative Price and Log Relative Volume in columns

rtc_rt_df2.loc[:, 'Log Price'] = np.log(rtc_rt_df2['Price Per Unit'])
rtc_rt_df2.loc[:, 'Log Qty'] = np.log(rtc_rt_df2['Qty'])
```

In [283…
```
rtc_rt_df2.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rtc_rt_df2['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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

In [284…
```
#Copying Seasonal Value to Data Frame and resetting the index

rtc_rt_df2 = rtc_rt_df2.copy()  # Create a copy of the DataFrame
rtc_rt_df2['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rtc_rt_df2.reset_index(inplace=True)
```

In [285…
```
rtc_rt_df2['Transaction Date'] = pd.to_datetime(rte_rt_df2['Transaction Date'])

# Convert 'Transaction Date' to date format
rtc_rt_df2['Transaction Date'] = rtc_rt_df2['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rtc_rt_df2['IsWeekend'] = (rtc_rt_df2['Transaction Date'].apply(lambda x: x.weekday()) >
rtc_rt_df2['IsHoliday'] = rtc_rt_df2['Transaction Date'].isin(holiday_dates).astype(int)
```

In [286…
```
# Assuming you have separate demand functions for each product
products = rtc_rt_df2['Inventory Code'].unique()
```

```
# Initialize a list to store results
results_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rtc_rt_df2[(rtc_rt_df2['Inventory Code'] == products[i]) | (rtc_rt_d

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_1.params['Seasonal'],
            'P-Value (Seasonality)': model_1.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_1.params['IsWeekend'],
            'P-Value (IsWeekend)': model_1.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_1.params['IsHoliday'],
            'P-Value (IsHoliday)': model_1.pvalues['IsHoliday'],
        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_2.params['Seasonal'],
            'P-Value (Seasonality)': model_2.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_2.params['IsWeekend'],
            'P-Value (IsWeekend)': model_2.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_2.params['IsHoliday'],
            'P-Value (IsHoliday)': model_2.pvalues['IsHoliday'],
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[286]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonality)	P-Value (Seasonality)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)
0	1155.0 -	1155.0	54.250532	4.886975e-34	0.007018	0.175002	0.034725	0.892253	0.0

	1158.0								
1	1155.0 - 1158.0	1158.0	1.143317	9.220474e-01	0.030381	0.015652	-0.555540	0.420576	0.0
2	1155.0 - 1217.0	1155.0	54.250532	4.886975e-34	0.007018	0.175002	0.034725	0.892253	0.0
3	1155.0 - 1217.0	1217.0	0.174650	2.134208e-70	0.009946	0.034837	0.047619	0.816519	0.0
4	1155.0 - 1218.0	1155.0	54.250532	4.886975e-34	0.007018	0.175002	0.034725	0.892253	0.0
5	1155.0 - 1218.0	1218.0	1.061146	1.027439e-201	0.006983	0.131842	0.302792	0.097006	0.0
6	1155.0 - 1316.0	1155.0	54.250532	4.886975e-34	0.007018	0.175002	0.034725	0.892253	0.0
7	1155.0 - 1316.0	1316.0	0.872238	1.173989e-231	-0.001357	0.692450	0.283043	0.038058	0.0
8	1158.0 - 1217.0	1158.0	1.143317	9.220474e-01	0.030381	0.015652	-0.555540	0.420576	0.0
9	1158.0 - 1217.0	1217.0	0.174650	2.134208e-70	0.009946	0.034837	0.047619	0.816519	0.0
10	1158.0 - 1218.0	1158.0	1.143317	9.220474e-01	0.030381	0.015652	-0.555540	0.420576	0.0
11	1158.0 - 1218.0	1218.0	1.061146	1.027439e-201	0.006983	0.131842	0.302792	0.097006	0.0
12	1158.0 - 1316.0	1158.0	1.143317	9.220474e-01	0.030381	0.015652	-0.555540	0.420576	0.0
13	1158.0 - 1316.0	1316.0	0.872238	1.173989e-231	-0.001357	0.692450	0.283043	0.038058	0.0
14	1217.0 - 1218.0	1217.0	0.174650	2.134208e-70	0.009946	0.034837	0.047619	0.816519	0.0
15	1217.0 - 1218.0	1218.0	1.061146	1.027439e-201	0.006983	0.131842	0.302792	0.097006	0.0
16	1217.0 - 1316.0	1217.0	0.174650	2.134208e-70	0.009946	0.034837	0.047619	0.816519	0.0
17	1217.0 - 1316.0	1316.0	0.872238	1.173989e-231	-0.001357	0.692450	0.283043	0.038058	0.0

18	1218.0 - 1316.0	1218.0	1.061146	1.027439e-201	0.006983	0.131842	0.302792	0.097006	0.0
19	1218.0 - 1316.0	1316.0	0.872238	1.173989e-231	-0.001357	0.692450	0.283043	0.038058	0.0

In [287…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rtc_rt_df2.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rtc_rt_min_max_data2 = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rtc_rt_min_max_data2.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', '

# Display the result
print(rtc_rt_min_max_data2)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1155.0       10.9       12.0      7.8    780.0
1          1158.0       11.4       11.8      1.3    715.0
2          1217.0      100.0      100.0      1.3     10.4
3          1218.0       10.8       10.8      6.5     71.5
4          1316.0       19.0       19.0      3.9     57.2
```

In [288…

```
# Assuming you have separate demand functions for each product
products = rtc_rt_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Initialize lists to store mean price and quantity
mean_price_list = []
mean_quantity_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = rtc_rt_df2[(rtc_rt_df2['Inventory Code'] == products[i]) | (rtc_rt_d

        # Calculate mean price and quantity for each product in the pair
        mean_price_i = pair_data[pair_data['Inventory Code'] == products[i]]['Price Per
        mean_quantity_i = pair_data[pair_data['Inventory Code'] == products[i]]['Qty'].m

        mean_price_j = pair_data[pair_data['Inventory Code'] == products[j]]['Price Per
        mean_quantity_j = pair_data[pair_data['Inventory Code'] == products[j]]['Qty'].m

        # Store the mean price and quantity in the lists
        mean_price_list.extend([mean_price_i, mean_price_j])
        mean_quantity_list.extend([mean_quantity_i, mean_quantity_j])

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']
```

```
        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Mean Price': mean_price_i,
            'Mean Quantity': mean_quantity_i,

        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Mean Price': mean_price_j,
            'Mean Quantity': mean_quantity_j,

        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[288]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Mean Price	Mean Quantity
0	1155.0 - 1158.0	1155.0	54.250532	4.886975e-34	11.328620	62.800717
1	1155.0 - 1158.0	1158.0	1.143317	9.220474e-01	11.528315	25.347670
2	1155.0 - 1217.0	1155.0	54.250532	4.886975e-34	11.328620	62.800717
3	1155.0 - 1217.0	1217.0	0.174650	2.134208e-70	100.000000	2.765412
4	1155.0 - 1218.0	1155.0	54.250532	4.886975e-34	11.328620	62.800717
5	1155.0 - 1218.0	1218.0	1.061146	1.027439e-201	10.800000	23.055197
6	1155.0 - 1316.0	1155.0	54.250532	4.886975e-34	11.328620	62.800717
7	1155.0 - 1316.0	1316.0	0.872238	1.173989e-231	19.000000	19.322939
8	1158.0 - 1217.0	1158.0	1.143317	9.220474e-01	11.528315	25.347670
9	1158.0 - 1217.0	1217.0	0.174650	2.134208e-70	100.000000	2.765412
10	1158.0 - 1218.0	1158.0	1.143317	9.220474e-01	11.528315	25.347670
11	1158.0 - 1218.0	1218.0	1.061146	1.027439e-201	10.800000	23.055197
12	1158.0 - 1316.0	1158.0	1.143317	9.220474e-01	11.528315	25.347670
13	1158.0 - 1316.0	1316.0	0.872238	1.173989e-231	19.000000	19.322939
14	1217.0 - 1218.0	1217.0	0.174650	2.134208e-70	100.000000	2.765412

15	1217.0 - 1218.0	1218.0	1.061146	1.027439e-201	10.800000	23.055197
16	1217.0 - 1316.0	1217.0	0.174650	2.134208e-70	100.000000	2.765412
17	1217.0 - 1316.0	1316.0	0.872238	1.173989e-231	19.000000	19.322939
18	1218.0 - 1316.0	1218.0	1.061146	1.027439e-201	10.800000	23.055197
19	1218.0 - 1316.0	1316.0	0.872238	1.173989e-231	19.000000	19.322939

In [289…

```
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in results_df
for index, row in results_df.iterrows():
    inventory_code = row['Product']
    mean_price = row['Mean Price']
    mean_quantity = row['Mean Quantity']
    elasticity = row['Coefficient (Log Price)']

    # Set bounds for optimization
    bounds = [(0, None)]  # Optimize only price

    # Perform optimization
    result = minimize(calculate_revenue, x0=[mean_price], args=(elasticity, mean_quantit

    # Extract optimized price
    optimized_price = result.x[0]

    optimized_prices.append(optimized_price)

# Add optimized prices to the results DataFrame
results_df['Optimized Price'] = optimized_prices

# Display the updated results_df
print(results_df[['Pair', 'Product', 'Optimized Price']])
```

```
                Pair Product  Optimized Price
0   1155.0 - 1158.0  1155.0              0.0
1   1155.0 - 1158.0  1158.0              0.0
2   1155.0 - 1217.0  1155.0              0.0
3   1155.0 - 1217.0  1217.0              0.0
4   1155.0 - 1218.0  1155.0              0.0
5   1155.0 - 1218.0  1218.0              0.0
6   1155.0 - 1316.0  1155.0              0.0
7   1155.0 - 1316.0  1316.0              0.0
8   1158.0 - 1217.0  1158.0              0.0
9   1158.0 - 1217.0  1217.0              0.0
10  1158.0 - 1218.0  1158.0              0.0
11  1158.0 - 1218.0  1218.0              0.0
12  1158.0 - 1316.0  1158.0              0.0
13  1158.0 - 1316.0  1316.0              0.0
14  1217.0 - 1218.0  1217.0              0.0
15  1217.0 - 1218.0  1218.0              0.0
16  1217.0 - 1316.0  1217.0              0.0
17  1217.0 - 1316.0  1316.0              0.0
18  1218.0 - 1316.0  1218.0              0.0
19  1218.0 - 1316.0  1316.0              0.0
```

Ready to Cook(RTC) Retail Self PED and Price Optimization

In [290…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = rtc_rt_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_rtc_rt_df = rtc_rt_df[rtc_rt_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
rtc_rt_df = filtered_rtc_rt_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [291…
```
rtc_rt_df3 = rtc_rt_df
```

In [292…
```
## Creating Log Price and Log Quantity in columns

rtc_rt_df.loc[:, 'Log Price'] = np.log(rtc_rt_df['Price Per Unit'])
rtc_rt_df.loc[:, 'Log Qty'] = np.log(rtc_rt_df['Qty'])
```

In [293…
```
rtc_rt_df3
```

Out[293]:

	Transaction Date	Inventory Code	Qty	Price Per Unit	Log Price	Log Qty
0	2022-07-01	1155.0	46.8	10.9	2.388763	3.845883
1	2022-07-01	1158.0	2.6	11.5	2.442347	0.955511
2	2022-07-01	1217.0	2.6	100.0	4.605170	0.955511
3	2022-07-01	1218.0	32.5	10.8	2.379546	3.481240
4	2022-07-01	1316.0	26.0	19.0	2.944439	3.258097
…	…	…	…	…	…	…
1390	2023-05-31	1155.0	23.4	11.3	2.424803	3.152736
1391	2023-05-31	1158.0	1.3	11.5	2.442347	0.262364
1392	2023-05-31	1217.0	2.6	100.0	4.605170	0.955511
1393	2023-05-31	1218.0	19.5	10.8	2.379546	2.970414
1394	2023-05-31	1316.0	14.3	19.0	2.944439	2.660260

1395 rows × 6 columns

In [294…
```
## Seasonality Values


rtc_rt_df3.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = rtc_rt_df3['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)
# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')
```

```
# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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

In [295…

```
#Copying Seasonal Value to Data Frame and resetting the index

rtc_rt_df3 = rtc_rt_df.copy()  # Create a copy of the DataFrame
rtc_rt_df3['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

rtc_rt_df3.reset_index(inplace=True)
```

In [296…

```
## Setting Weekend and Holiday Binary Values

rtc_rt_df3['Transaction Date'] = pd.to_datetime(rtc_rt_df3['Transaction Date'])

# Convert 'Transaction Date' to date format
rtc_rt_df3['Transaction Date'] = rtc_rt_df3['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
```

```
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
rtc_rt_df3['IsWeekend'] = (rtc_rt_df3['Transaction Date'].apply(lambda x: x.weekday()) >
rtc_rt_df3['IsHoliday'] = rtc_rt_df3['Transaction Date'].isin(holiday_dates).astype(int)
```

In [297…

```
# Linear Regression for Self Price PED RTC Retail

# Get unique product codes from the 'Inventory Code' column
unique_inventory = rtc_rt_df3['Inventory Code'].unique()

# Initialize lists to store results
inventory_codes = []
coefficients_price = []
p_values_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []

# Loop through the unique product codes
for inventory_code in unique_inventory:
    inventory_data = rtc_rt_df3[rtc_rt_df3['Inventory Code'] == inventory_code]

    # Create features and target variables
    X = sm.add_constant(inventory_data[['Log Price', 'IsWeekend', 'IsHoliday', 'Seasonal

    model = sm.OLS(inventory_data['Log Qty'], X).fit()

    # Append results to lists
    inventory_codes.append(inventory_code)
    coefficients_price.append(model.params['Log Price'])
    p_values_price.append(model.pvalues['Log Price'])
    coefficients_seasonal.append(model.params['Seasonal'])
    p_values_seasonal.append(model.pvalues['Seasonal'])
    coefficients_is_weekend.append(model.params['IsWeekend'])
    p_values_is_weekend.append(model.pvalues['IsWeekend'])
    coefficients_is_holiday.append(model.params['IsHoliday'])
    p_values_is_holiday.append(model.pvalues['IsHoliday'])

# Create a DataFrame with the aggregated coefficients and p-values
table_data = {
    'Inventory Code': inventory_codes,
    'Coefficient (Log Price)': coefficients_price,
    'P-Value (Log Price)': p_values_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

own_price_rtc_rt_df = pd.DataFrame(table_data)

# Apply the significance labels
```

```
for column in own_price_rtc_rt_df.columns:
    if 'P-Value' in column:
        significance_column = 'Significance' + column.split('P-Value', 1)[1]
        own_price_rtc_rt_df[significance_column] = own_price_rtc_rt_df[column].apply(lam

# Display the table in the Jupyter Notebook
print("Ready to Cook Retail Aggregated Coefficients and Significance Table:")
display(own_price_rtc_rt_df)
```

```
Ready to Cook Retail Aggregated Coefficients and Significance Table:
```

	Inventory Code	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	1155.0	53.403724	2.619056e-33	0.006571	0.210108	-0.159358	1.139135e-01	-0.346526	0.177698
1	1158.0	2.003653	8.643239e-01	0.027941	0.027736	-0.220958	3.713466e-01	-0.602873	0.340302
2	1217.0	0.153055	2.371748e-61	0.003182	0.440258	0.786794	7.953768e-20	0.288468	0.157588
3	1218.0	1.194183	1.398109e-205	0.002139	0.587532	0.820402	2.033862e-22	0.106391	0.590190
4	1316.0	0.945764	5.249763e-232	-0.000998	0.742726	0.537966	3.721748e-17	-0.034932	0.820999

In [298…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = rtc_rt_df3.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
rtc_rt_min_max_data = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
rtc_rt_min_max_data.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', 'M

# Display the result
print(rtc_rt_min_max_data)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1155.0       10.9       12.0      7.8    780.0
1          1158.0       11.4       11.8      1.3    715.0
2          1217.0      100.0      100.0      1.3     10.4
3          1218.0       10.8       10.8      6.5     71.5
4          1316.0       19.0       19.0      3.9     57.2
```

In [299…

```
## Price Optimization Using Min & Max Qty & Price as bounds

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rtc_rt_df, rtc_rt_min_max_data, on='Inventory Code')

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, qty, coefficient_price):
    return -price * qty * coefficient_price  # Negative sign because scipy.optimize mini

# Function to be minimized (negative revenue for maximization)
def objective_function(prices):
    total_revenue = 0
    for index, row in merged_data.iterrows():
```

```
        min_price, max_price = row['Min Price'], row['Max Price']
        min_qty, max_qty = row['Min Qty'], row['Max Qty']
        coefficient_price = row['Coefficient (Log Price)']

        # Ensure prices are within bounds
        price = max(min(max_price, prices[index]), min_price)

        # Calculate revenue
        qty = max(min(max_qty, prices[len(merged_data) + index]), min_qty)
        total_revenue += calculate_revenue(price, qty, coefficient_price)

    return total_revenue

# Initial guess for prices
initial_prices = merged_data['Min Price'].values.tolist() + merged_data['Min Qty'].value

# Constraints
price_constraints = [(min_price, max_price) for min_price, max_price in zip(merged_data[
qty_constraints = [(min_qty, max_qty) for min_qty, max_qty in zip(merged_data['Min Qty']
constraints = price_constraints + qty_constraints

# Optimization
result = minimize(objective_function, initial_prices, constraints={'type': 'ineq', 'fun'
                  bounds=constraints)

# Extract optimized prices and quantities
optimized_prices = result.x[:len(merged_data)]
optimized_quantities = result.x[len(merged_data):]

# Display optimized prices and quantities
optimized_data = merged_data.copy()
optimized_data['Optimized Price'] = optimized_prices
optimized_data['Optimized Quantity'] = optimized_quantities

print("Optimized Prices and Quantities:")
print(optimized_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Optimized Prices and Quantities:
   Inventory Code  Optimized Price  Optimized Quantity
0          1155.0             12.0               780.0
1          1158.0             11.8               715.0
2          1217.0            100.0                10.4
3          1218.0             10.8                71.5
4          1316.0             19.0                57.2
```

In [300…

```
## Price Optimization Using Min Price as bounds

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_rtc_rt_df, rtc_rt_min_max_data, on='Inventory Code')

# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in merged_data
for index, row in merged_data.iterrows():
```

```
    inventory_code = row['Inventory Code']
    min_price = row['Min Price']

    # Get elasticity from own_price_rte_sp_df based on the inventory code
    elasticity = own_price_rtc_rt_df.loc[own_price_rtc_rt_df['Inventory Code'] == invent

    # Set bounds for optimization
    bounds = [(min_price, None), (None, None)]  # Remove lower bound for quantity

    # Perform optimization
    result = minimize(objective_function, x0=[min_price, row['Min Qty']], args=(elastici

    # Extract optimized price and quantity
    optimized_price, optimized_quantity = result.x

    # Append results to lists
    optimized_prices.append(optimized_price)
    optimized_quantities.append(optimized_quantity)

# Add optimized prices and quantities to the DataFrame
merged_data['Optimized Price'] = optimized_prices
merged_data['Optimized Quantity'] = optimized_quantities

# Display the DataFrame
print(merged_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

	Inventory Code	Optimized Price	Optimized Quantity
0	1155.0	2.926032e+143	4.038986e+143
1	1158.0	7.446008e+147	1.729159e+148
2	1217.0	9.873500e+152	8.594363e+152
3	1218.0	1.502582e+146	1.810446e+146
4	1316.0	2.964528e+145	4.556819e+145

Raw Retail Cross PED and Price Optimization

In [301…

```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = raw_rt_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_raw_rt_df = raw_rt_df[raw_rt_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
raw_rt_df = filtered_raw_rt_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [302…

```
#Using mode of each inventory code to replace missing data in dataframe


# Step 1: Identify unique inventory codes for each transaction date
unique_inventory_codes = raw_rt_df.groupby('Transaction Date')['Inventory Code'].unique(

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = raw_rt_df.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(cal

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(raw_rt_df['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))
```

```
    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
raw_rt_df = pd.concat([raw_rt_df, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
raw_rt_df = raw_rt_df.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
raw_rt_df.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(raw_rt_df)
```

```
    Transaction Date  Inventory Code    Qty  Price Per Unit
0         2022-07-06          1030.0  312.0             4.9
1         2022-07-06          1031.0  312.0             4.7
2         2022-07-06          1032.0   31.2             5.2
3         2022-07-06          1036.0  124.8             5.6
4         2022-07-13          1030.0  312.0             4.9
..               ...             ...    ...             ...
315       2023-05-26          1036.0  140.4             6.2
316       2023-05-31          1030.0  312.0             5.6
317       2023-05-31          1031.0   46.8             5.6
318       2023-05-31          1032.0   93.6             5.6
319       2023-05-31          1036.0   62.4             6.2

[320 rows x 4 columns]
```

In [303…

```
# Create a list of unique inventory codes
unique_inventory_codes = raw_rt_df['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1030.0, 1031.0)
Inventory Code Pair: (1030.0, 1032.0)
Inventory Code Pair: (1030.0, 1036.0)
Inventory Code Pair: (1031.0, 1032.0)
Inventory Code Pair: (1031.0, 1036.0)
Inventory Code Pair: (1032.0, 1036.0)
```

In [304…

```
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = raw_rt_df[raw_rt_df['Inventory Code'] == code1]
    data_code2 = raw_rt_df[raw_rt_df['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:     Transaction Date  Inventory Code    Qty  Price Per Unit
2         2022-07-06          1032.0   31.2             5.2
6         2022-07-13          1032.0   46.8             5.2
10        2022-07-20          1032.0   15.6             5.2
```

```
14          2022-07-27           1032.0   93.6              5.6
18          2022-08-10           1032.0   46.8              5.2
..                 ...              ...    ...              ...
302         2023-05-17           1032.0   46.8              5.6
306         2023-05-19           1032.0  124.8              5.6
310         2023-05-24           1032.0   78.0              5.6
314         2023-05-26           1032.0  187.2              5.6
318         2023-05-31           1032.0   93.6              5.6

[80 rows x 4 columns]
Data for Code Pair 2:     Transaction Date  Inventory Code    Qty  Price Per Unit
3           2022-07-06           1036.0  124.8              5.6
7           2022-07-13           1036.0  124.8              5.6
11          2022-07-20           1036.0   31.2              5.6
15          2022-07-27           1036.0   15.6              5.6
19          2022-08-10           1036.0  156.0              5.6
..                 ...              ...    ...              ...
303         2023-05-17           1036.0   78.0              6.2
307         2023-05-19           1036.0   78.0              6.2
311         2023-05-24           1036.0   93.6              6.2
315         2023-05-26           1036.0  140.4              6.2
319         2023-05-31           1036.0   62.4              6.2

[80 rows x 4 columns]
```

In [305…

```python
#Creating the cross reference for each pair on a daily basis

# Create an empty DataFrame to store the results
raw_rt_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1',

# Create a list of unique inventory codes
unique_inventory_codes = raw_rt_df['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = raw_rt_df[raw_rt_df['Inventory Code'] == code1]
    data_code2 = raw_rt_df[raw_rt_df['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    raw_rt_elasticity_df = pd.concat([raw_rt_elasticity_df, results])

# Print or inspect the elasticity results
print(raw_rt_elasticity_df)
```

```
   Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0        2022-07-06                1030.0                1031.0
1        2022-07-13                1030.0                1031.0
2        2022-07-20                1030.0                1031.0
3        2022-07-27                1030.0                1031.0
4        2022-08-10                1030.0                1031.0
..              ...                   ...                   ...
75       2023-05-17                1032.0                1036.0
76       2023-05-19                1032.0                1036.0
77       2023-05-24                1032.0                1036.0
78       2023-05-26                1032.0                1036.0
79       2023-05-31                1032.0                1036.0
```

```
    Relative_Price  Relative_Volume
0         1.042553         1.000000
1         1.042553         1.000000
2         1.042553         7.000000
3         1.042553         4.000000
4         1.042553         2.272727
..             ...              ...
75        0.903226         0.600000
76        0.903226         1.600000
77        0.903226         0.833333
78        0.903226         1.333333
79        0.903226         1.500000

[480 rows x 5 columns]
```

In [306…

```
## Creating Log Relative Price and Log Relative Volume in columns

raw_rt_elasticity_df.loc[:, 'Log Relative Price'] = np.log(raw_rt_elasticity_df['Relativ
raw_rt_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(raw_rt_elasticity_df['Relati
```

In [307…

```
#Seasonality
raw_rt_elasticity_df.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = raw_rt_elasticity_df['Relative_Volume']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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

In [308…

```
#Copying Seasonal Value to Data Frame and resetting the index

raw_rt_elasticity_df = raw_rt_elasticity_df.copy()  # Create a copy of the DataFrame
raw_rt_elasticity_df['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the c

raw_rt_elasticity_df.reset_index(inplace=True)
```

In [309…

```
## Setting Weekend and Holiday Binary Values

raw_rt_elasticity_df['Transaction Date'] = pd.to_datetime(raw_rt_elasticity_df['Transact

# Convert 'Transaction Date' to date format
raw_rt_elasticity_df['Transaction Date'] = raw_rt_elasticity_df['Transaction Date'].dt.d

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
raw_rt_elasticity_df['IsWeekend'] = (raw_rt_elasticity_df['Transaction Date'].apply(lamb
raw_rt_elasticity_df['IsHoliday'] = raw_rt_elasticity_df['Transaction Date'].isin(holida
```

In [310…

```
# Group the data by unique product pairs
unique_product_pairs = raw_rt_elasticity_df[['Inventory Code_code1', 'Inventory Code_cod

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = raw_rt_elasticity_df[(raw_rt_elasticity_df['Inventory Code_code1'] == inven

    X = sm.add_constant(subset[['Log Relative Price', 'Seasonal', 'IsWeekend', 'IsHolida
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        coefficients_seasonal.append(model.params['Seasonal'])
        p_values_seasonal.append(model.pvalues['Seasonal'])
        coefficients_is_weekend.append(model.params['IsWeekend'])
        p_values_is_weekend.append(model.pvalues['IsWeekend'])
        coefficients_is_holiday.append(model.params['IsHoliday'])
        p_values_is_holiday.append(model.pvalues['IsHoliday'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

raw_rt_table_df = pd.DataFrame(table_data)

raw_rt_table_df['Type'] = raw_rt_table_df['Coefficient (Log Relative Price)'].apply(lamb
raw_rt_table_df['Significance (Log Relative Price)'] = raw_rt_table_df['P-Value (Log Rel
raw_rt_table_df['Significance (Seasonal)'] = raw_rt_table_df['P-Value (Seasonal)'].apply
raw_rt_table_df['Significance (IsWeekend)'] = raw_rt_table_df['P-Value (IsWeekend)'].app
raw_rt_table_df['Significance (IsHoliday)'] = raw_rt_table_df['P-Value (IsHoliday)'].app

# Create a DataFrame to store the OLS results
raw_rt_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
```

```
})

# Display both tables in the Jupyter Notebook
display(raw_rt_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	(1030.0, 1031.0)	-30.439935	1.310961e-08	0.455702	0.000449	0.0	NaN	0.0	NaN
1	(1030.0, 1032.0)	-8.314821	1.416621e-02	0.063475	0.649028	0.0	NaN	0.0	NaN
2	(1030.0, 1036.0)	-2.634959	6.313861e-01	0.318760	0.003973	0.0	NaN	0.0	NaN
3	(1031.0, 1032.0)	-17.688081	1.591836e-12	-0.213644	0.140433	0.0	NaN	0.0	NaN
4	(1031.0, 1036.0)	-18.581196	3.770329e-13	-0.059324	0.532015	0.0	NaN	0.0	NaN
5	(1032.0, 1036.0)	-5.466785	2.369962e-01	0.039874	0.729259	0.0	NaN	0.0	NaN

In [311…

```
# Initialize lists to store the optimization results
optimized_prices_list = []
maximized_revenue_list = []

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Iterate over rows in the DataFrame
for _, row in raw_rt_table_df.iterrows():
    # Extract the cross-price elasticity coefficient from the table
    cross_price_elasticity_coefficient = row['Coefficient (Log Relative Price)']

    # Initial guess for prices (you may need to adjust this based on your specific conte
    initial_prices = [1.0, 1.0]  # Assuming you are optimizing prices for a pair of prod

    # Define bounds on prices
    bounds = [(0, None), (0, None)]  # Separate bounds for each item in the pair

    # Run the optimization
    result = minimize(objective_function, initial_prices, args=(cross_price_elasticity_c

    # Extract the optimized prices and maximized revenue
    optimized_prices = result.x.tolist()
    maximized_revenue = -result.fun  # Negate to get the actual maximum revenue

    # Append the results to the lists
    optimized_prices_list.append(optimized_prices)
    maximized_revenue_list.append(maximized_revenue)

# Add new columns to the DataFrame
raw_rt_table_df['Optimized Prices'] = optimized_prices_list
```

```
raw_rt_table_df['Maximized Revenue'] = maximized_revenue_list

# Display the updated DataFrame
display(raw_rt_table_df[['Inventory Code Pair', 'Optimized Prices', 'Maximized Revenue']])
```

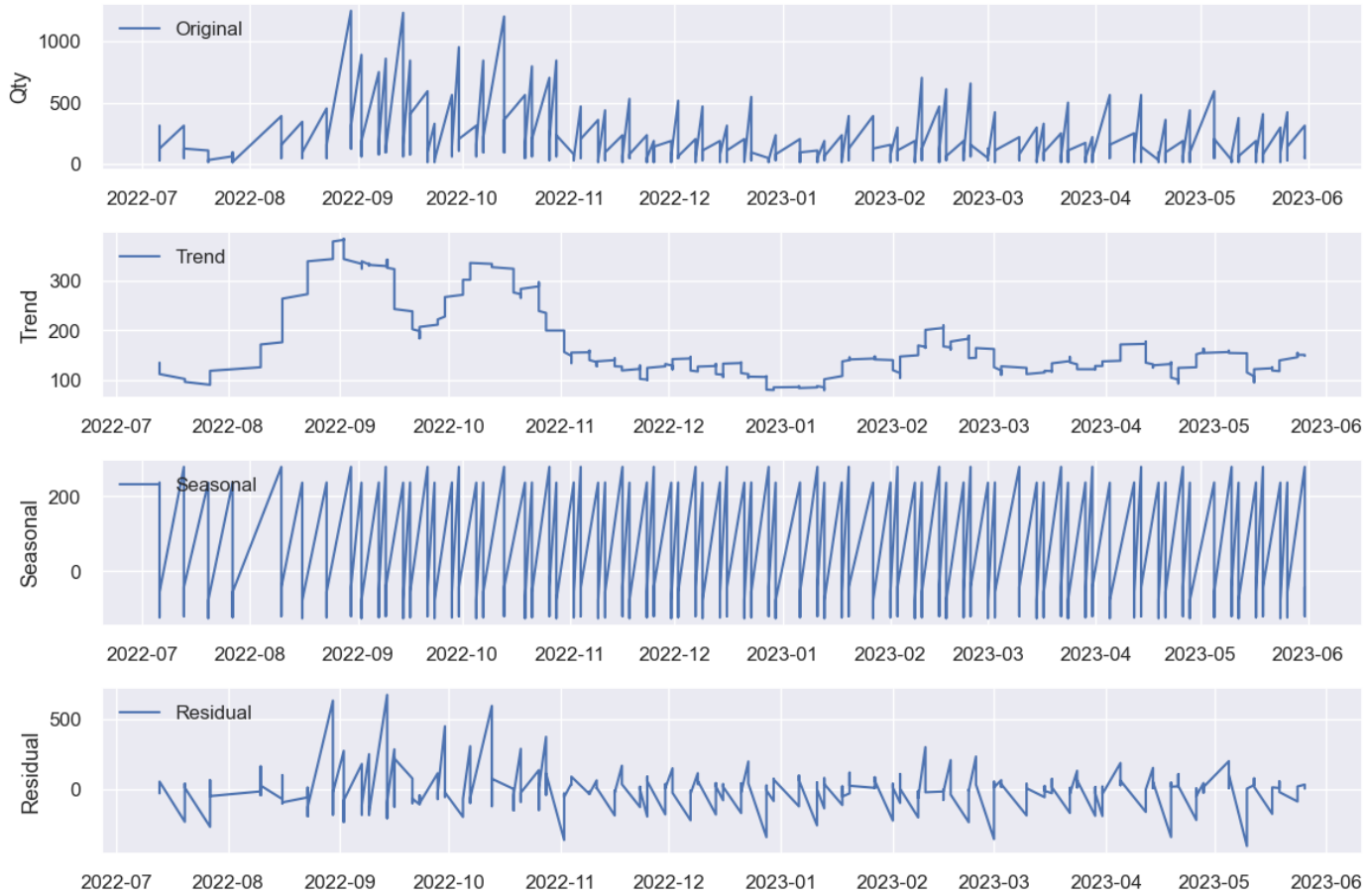
	Inventory Code Pair	Optimized Prices	Maximized Revenue
0	(1030.0, 1031.0)	[0.0, 0.0]	-0.000000e+00
1	(1030.0, 1032.0)	[1.1228869153455905e-09, 1.1228869153455905e-09]	-9.223075e-18
2	(1030.0, 1036.0)	[4.463385332080185e-09, 4.463385332080185e-09]	-3.257134e-17
3	(1031.0, 1032.0)	[0.0, 0.0]	-0.000000e+00
4	(1031.0, 1036.0)	[1.030105506439441e-09, 1.030105506439441e-09]	-1.865571e-17
5	(1032.0, 1036.0)	[2.720220306562983e-09, 2.720220306562983e-09]	-3.305242e-17

Method 2 - Seperate Cross Price in 2 Entries

In [312…
```
raw_rt_df2 = raw_rt_df
```

In [313…
```
## Creating Log Relative Price and Log Relative Volume in columns

raw_rt_df2.loc[:, 'Log Price'] = np.log(raw_rt_df2['Price Per Unit'])
raw_rt_df2.loc[:, 'Log Qty'] = np.log(raw_rt_df2['Qty'])
```

In [314…
```
raw_rt_df2.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = raw_rt_df2['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))

# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [315…
```
#Copying Seasonal Value to Data Frame and resetting the index

raw_rt_df2 = raw_rt_df2.copy()  # Create a copy of the DataFrame
raw_rt_df2['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

raw_rt_df2.reset_index(inplace=True)
```

In [316…
```
raw_rt_df2['Transaction Date'] = pd.to_datetime(raw_rt_df2['Transaction Date'])

# Convert 'Transaction Date' to date format
raw_rt_df2['Transaction Date'] = raw_rt_df2['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
raw_rt_df2['IsWeekend'] = (raw_rt_df2['Transaction Date'].apply(lambda x: x.weekday()) >
raw_rt_df2['IsHoliday'] = raw_rt_df2['Transaction Date'].isin(holiday_dates).astype(int)
```

In [317…
```
# Assuming you have separate demand functions for each product
products = raw_rt_df2['Inventory Code'].unique()
```

```
# Initialize a list to store results
results_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = raw_rt_df2[(raw_rt_df2['Inventory Code'] == products[i]) | (raw_rt_d

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_1.params['Seasonal'],
            'P-Value (Seasonality)': model_1.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_1.params['IsWeekend'],
            'P-Value (IsWeekend)': model_1.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_1.params['IsHoliday'],
            'P-Value (IsHoliday)': model_1.pvalues['IsHoliday'],
        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Coefficient (Seasonality)': model_2.params['Seasonal'],
            'P-Value (Seasonality)': model_2.pvalues['Seasonal'],
            'Coefficient (IsWeekend)': model_2.params['IsWeekend'],
            'P-Value (IsWeekend)': model_2.pvalues['IsWeekend'],
            'Coefficient (IsHoliday)': model_2.params['IsHoliday'],
            'P-Value (IsHoliday)': model_2.pvalues['IsHoliday'],
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[317]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonality)	P-Value (Seasonality)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)
0	1030.0 -	1030.0	-2.447199	1.902839e-01	0.000435	0.922686	0.0	NaN	0.0

	1031.0								
1	1030.0 - 1031.0	1031.0	-9.157443	5.850303e-11	0.020157	0.569412	0.0	NaN	0.0
2	1030.0 - 1032.0	1030.0	-2.447199	1.902839e-01	0.000435	0.922686	0.0	NaN	0.0
3	1030.0 - 1032.0	1032.0	8.200710	1.962906e-04	0.011428	0.160996	0.0	NaN	0.0
4	1030.0 - 1036.0	1030.0	-2.447199	1.902839e-01	0.000435	0.922686	0.0	NaN	0.0
5	1030.0 - 1036.0	1036.0	-2.384596	2.289576e-01	0.010008	0.047147	0.0	NaN	0.0
6	1031.0 - 1032.0	1031.0	-9.157443	5.850303e-11	0.020157	0.569412	0.0	NaN	0.0
7	1031.0 - 1032.0	1032.0	8.200710	1.962906e-04	0.011428	0.160996	0.0	NaN	0.0
8	1031.0 - 1036.0	1031.0	-9.157443	5.850303e-11	0.020157	0.569412	0.0	NaN	0.0
9	1031.0 - 1036.0	1036.0	-2.384596	2.289576e-01	0.010008	0.047147	0.0	NaN	0.0
10	1032.0 - 1036.0	1032.0	8.200710	1.962906e-04	0.011428	0.160996	0.0	NaN	0.0
11	1032.0 - 1036.0	1036.0	-2.384596	2.289576e-01	0.010008	0.047147	0.0	NaN	0.0

In [318…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = raw_rt_df2.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
raw_rt_min_max_data2 = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
raw_rt_min_max_data2.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', '

# Display the result
print(raw_rt_min_max_data2)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1030.0        4.9        5.6     31.2   1248.0
1          1031.0        4.7        5.6     15.6    312.0
```

2	1032.0	5.2	5.6	15.6	218.4
3	1036.0	5.6	6.2	15.6	405.6

In [319…

```
# Assuming you have separate demand functions for each product
products = raw_rt_df2['Inventory Code'].unique()

# Initialize a list to store results
results_list = []

# Initialize lists to store mean price and quantity
mean_price_list = []
mean_quantity_list = []

# Iterate over unique pairs
for i in range(len(products)):
    for j in range(i + 1, len(products)):
        # Subset the DataFrame for the current pair
        pair_data = raw_rt_df2[(raw_rt_df2['Inventory Code'] == products[i]) | (raw_rt_d

        # Calculate mean price and quantity for each product in the pair
        mean_price_i = pair_data[pair_data['Inventory Code'] == products[i]]['Price Per
        mean_quantity_i = pair_data[pair_data['Inventory Code'] == products[i]]['Qty'].m

        mean_price_j = pair_data[pair_data['Inventory Code'] == products[j]]['Price Per
        mean_quantity_j = pair_data[pair_data['Inventory Code'] == products[j]]['Qty'].m

        # Store the mean price and quantity in the lists
        mean_price_list.extend([mean_price_i, mean_price_j])
        mean_quantity_list.extend([mean_quantity_i, mean_quantity_j])

        # Define the dependent variable (Log Quantity) and independent variables
        log_qty = pair_data['Log Qty']
        log_price = pair_data['Log Price']
        seasonal = pair_data['Seasonal']
        is_weekend = pair_data['IsWeekend']
        is_holiday = pair_data['IsHoliday']

        # Add a constant term to the independent variables
        X = sm.add_constant(pd.concat([log_price, seasonal, is_weekend, is_holiday], axi

        # Fit the regression model for Log(Q1)
        model_1 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[i]], X[pair_dat

        # Fit the regression model for Log(Q2)
        model_2 = sm.OLS(log_qty[pair_data['Inventory Code'] == products[j]], X[pair_dat

        # Store the results for Product 1
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[i]}',
            'Coefficient (Log Price)': model_1.params['Log Price'],
            'P-Value (Log Price)': model_1.pvalues['Log Price'],
            'Mean Price': mean_price_i,
            'Mean Quantity': mean_quantity_i,

        })

        # Store the results for Product 2
        results_list.append({
            'Pair': f'{products[i]} - {products[j]}',
            'Product': f'{products[j]}',
            'Coefficient (Log Price)': model_2.params['Log Price'],
            'P-Value (Log Price)': model_2.pvalues['Log Price'],
            'Mean Price': mean_price_j,
            'Mean Quantity': mean_quantity_j,
```

```
        })

# Create a DataFrame from the results
results_df = pd.DataFrame(results_list)

# Display the results_df
results_df
```

Out[319]:

	Pair	Product	Coefficient (Log Price)	P-Value (Log Price)	Mean Price	Mean Quantity
0	1030.0 - 1031.0	1030.0	-2.447199	1.902839e-01	5.495	414.375
1	1030.0 - 1031.0	1031.0	-9.157443	5.850303e-11	5.465	53.040
2	1030.0 - 1032.0	1030.0	-2.447199	1.902839e-01	5.495	414.375
3	1030.0 - 1032.0	1032.0	8.200710	1.962906e-04	5.545	99.450
4	1030.0 - 1036.0	1030.0	-2.447199	1.902839e-01	5.495	414.375
5	1030.0 - 1036.0	1036.0	-2.384596	2.289576e-01	6.110	109.980
6	1031.0 - 1032.0	1031.0	-9.157443	5.850303e-11	5.465	53.040
7	1031.0 - 1032.0	1032.0	8.200710	1.962906e-04	5.545	99.450
8	1031.0 - 1036.0	1031.0	-9.157443	5.850303e-11	5.465	53.040
9	1031.0 - 1036.0	1036.0	-2.384596	2.289576e-01	6.110	109.980
10	1032.0 - 1036.0	1032.0	8.200710	1.962906e-04	5.545	99.450
11	1032.0 - 1036.0	1036.0	-2.384596	2.289576e-01	6.110	109.980

In [320…

```
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in results_df
for index, row in results_df.iterrows():
    inventory_code = row['Product']
    mean_price = row['Mean Price']
    mean_quantity = row['Mean Quantity']
    elasticity = row['Coefficient (Log Price)']

    # Set bounds for optimization
    bounds = [(0, None)]  # Optimize only price

    # Perform optimization
    result = minimize(calculate_revenue, x0=[mean_price], args=(elasticity, mean_quantit

    # Extract optimized price
    optimized_price = result.x[0]

    optimized_prices.append(optimized_price)

# Add optimized prices to the results DataFrame
results_df['Optimized Price'] = optimized_prices

# Display the updated results_df
print(results_df[['Pair', 'Product', 'Optimized Price']])
```

```
               Pair Product  Optimized Price
0   1030.0 - 1031.0  1030.0     5.240159e+07
1   1030.0 - 1031.0  1031.0     5.712044e+37
2   1030.0 - 1032.0  1030.0     5.240159e+07
3   1030.0 - 1032.0  1032.0     0.000000e+00
4   1030.0 - 1036.0  1030.0     5.240159e+07
```

```
5   1030.0 - 1036.0  1036.0     3.310430e+19
6   1031.0 - 1032.0  1031.0     5.712044e+37
7   1031.0 - 1032.0  1032.0     0.000000e+00
8   1031.0 - 1036.0  1031.0     5.712044e+37
9   1031.0 - 1036.0  1036.0     3.310430e+19
10  1032.0 - 1036.0  1032.0     0.000000e+00
11  1032.0 - 1036.0  1036.0     3.310430e+19
```

Raw Retail Self PED and Price Optimization

In [321…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = raw_rt_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_raw_rt_df = raw_rt_df[raw_rt_df['Inventory Code'].isin(unique_inventory_codes)]

# Group and aggregate the filtered DataFrame as you intended
raw_rt_df = filtered_raw_rt_df.groupby(['Transaction Date', 'Inventory Code']).agg({'Qty
```

In [322…
```
raw_rt_df3 = raw_rt_df
```

In [323…
```
## Creating Log Price and Log Quantity in columns

raw_rt_df3.loc[:, 'Log Price'] = np.log(raw_rt_df3['Price Per Unit'])
raw_rt_df3.loc[:, 'Log Qty'] = np.log(raw_rt_df3['Qty'])
```

In [324…
```
raw_rt_df3
```

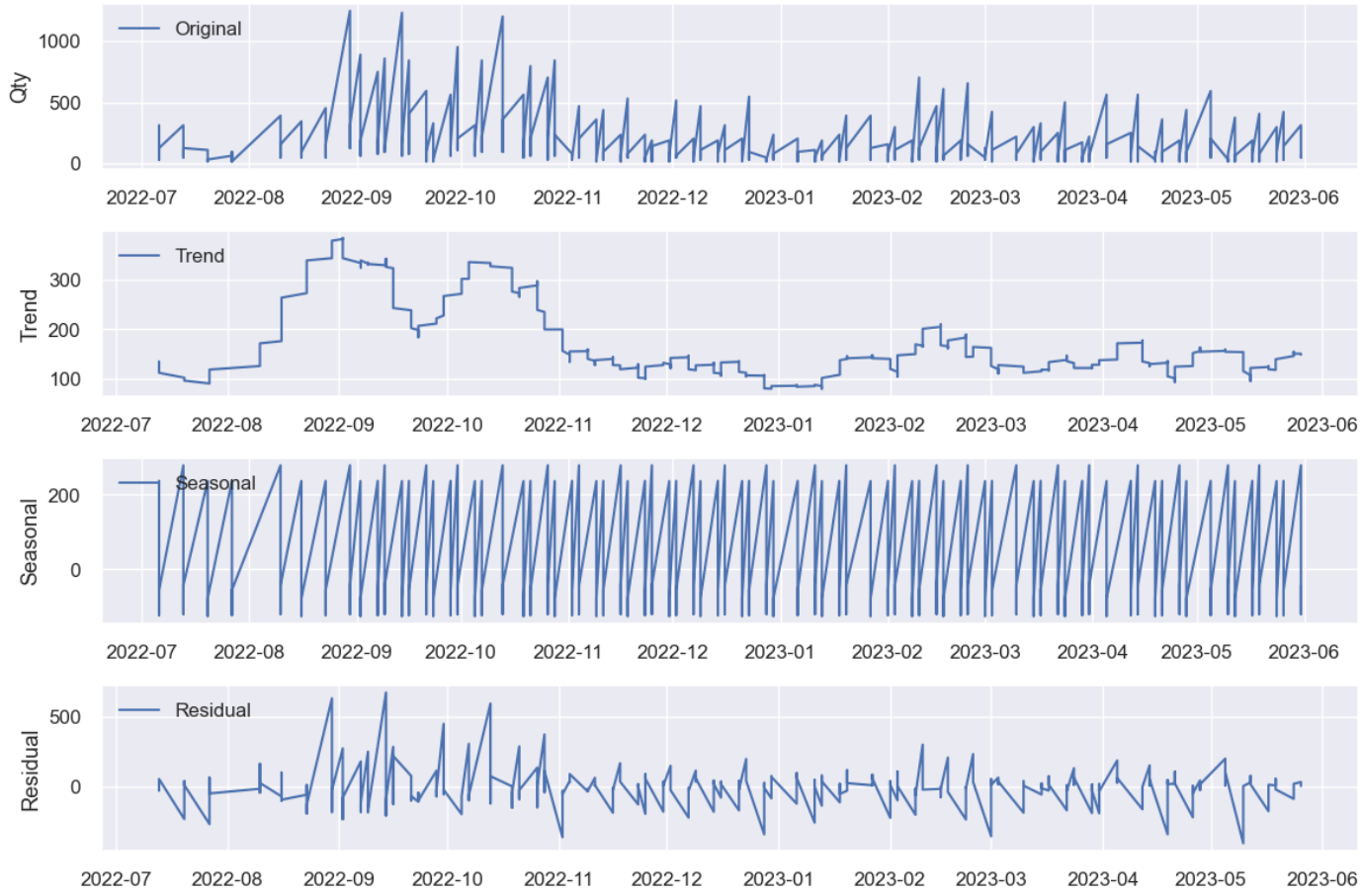
Out[324]:

	Transaction Date	Inventory Code	Qty	Price Per Unit	Log Price	Log Qty
0	2022-07-06	1030.0	312.0	4.9	1.589235	5.743003
1	2022-07-06	1031.0	312.0	4.7	1.547563	5.743003
2	2022-07-06	1032.0	31.2	5.2	1.648659	3.440418
3	2022-07-06	1036.0	124.8	5.6	1.722767	4.826712
4	2022-07-13	1030.0	312.0	4.9	1.589235	5.743003
…	…	…	…	…	…	…
315	2023-05-26	1036.0	140.4	6.2	1.824549	4.944495
316	2023-05-31	1030.0	312.0	5.6	1.722767	5.743003
317	2023-05-31	1031.0	46.8	5.6	1.722767	3.845883
318	2023-05-31	1032.0	93.6	5.6	1.722767	4.539030
319	2023-05-31	1036.0	62.4	6.2	1.824549	4.133565

320 rows × 6 columns

In [325…
```
## Seasonality Values
raw_rt_df3.set_index("Transaction Date", inplace=True)

# Perform seasonal decomposition
time_series = raw_rt_df3['Qty']
result = sm.tsa.seasonal_decompose(time_series, model='additive', period=12)  # Ask abou

# Plot the components
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))
```

```
# Original time series
ax1.plot(time_series.index, time_series, label='Original')
ax1.set_ylabel('Qty')
ax1.legend(loc='upper left')

# Trend component
ax2.plot(result.trend.index, result.trend, label='Trend')
ax2.set_ylabel('Trend')
ax2.legend(loc='upper left')

# Seasonal component
ax3.plot(result.seasonal.index, result.seasonal, label='Seasonal')
ax3.set_ylabel('Seasonal')
ax3.legend(loc='upper left')

# Residual component
ax4.plot(result.resid.index, result.resid, label='Residual')
ax4.set_ylabel('Residual')
ax4.legend(loc='upper left')

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



In [326…

```
#Copying Seasonal Value to Data Frame and resetting the index

raw_rt_df3 = raw_rt_df3.copy()  # Create a copy of the DataFrame
raw_rt_df3['Seasonal'] = result.seasonal  # Add the 'Seasonal' column to the copied Data

raw_rt_df3.reset_index(inplace=True)
```

In [327…

```
## Setting Weekend and Holiday Binary Values

raw_rt_df3['Transaction Date'] = pd.to_datetime(rtc_rt_df3['Transaction Date'])
```

```
# Convert 'Transaction Date' to date format
raw_rt_df3['Transaction Date'] = raw_rt_df3['Transaction Date'].dt.date

# Define holiday dates
holiday_dates = [
    "2022-07-11",
    "2022-08-09",
    "2022-10-24",
    "2022-12-26",
    "2023-01-02",
    "2023-01-22",
    "2023-01-24",
    "2023-04-07",
    "2023-04-22",
    "2023-05-01",
]

# Convert holiday_dates to datetime64 for proper matching
holiday_dates = pd.to_datetime(holiday_dates).date

# Assign IsWeekend and IsHoliday
raw_rt_df3['IsWeekend'] = (raw_rt_df3['Transaction Date'].apply(lambda x: x.weekday()) >
raw_rt_df3['IsHoliday'] = raw_rt_df3['Transaction Date'].isin(holiday_dates).astype(int)
```

In [328…

```
# Linear Regression for Self Price PED RAW Retail

# Get unique product codes from the 'Inventory Code' column
unique_inventory = raw_rt_df3['Inventory Code'].unique()

# Initialize lists to store results
inventory_codes = []
coefficients_price = []
p_values_price = []
coefficients_seasonal = []
p_values_seasonal = []
coefficients_is_weekend = []
p_values_is_weekend = []
coefficients_is_holiday = []
p_values_is_holiday = []

# Loop through the unique product codes
for inventory_code in unique_inventory:
    inventory_data = raw_rt_df3[raw_rt_df3['Inventory Code'] == inventory_code]

    # Create features and target variables
    X = sm.add_constant(inventory_data[['Log Price', 'IsWeekend', 'IsHoliday', 'Seasonal

    model = sm.OLS(inventory_data['Log Qty'], X).fit()

    # Append results to lists
    inventory_codes.append(inventory_code)
    coefficients_price.append(model.params['Log Price'])
    p_values_price.append(model.pvalues['Log Price'])
    coefficients_seasonal.append(model.params['Seasonal'])
    p_values_seasonal.append(model.pvalues['Seasonal'])
    coefficients_is_weekend.append(model.params['IsWeekend'])
    p_values_is_weekend.append(model.pvalues['IsWeekend'])
    coefficients_is_holiday.append(model.params['IsHoliday'])
    p_values_is_holiday.append(model.pvalues['IsHoliday'])

# Create a DataFrame with the aggregated coefficients and p-values
table_data = {
    'Inventory Code': inventory_codes,
    'Coefficient (Log Price)': coefficients_price,
    'P-Value (Log Price)': p_values_price,
```

```
    'Coefficient (Seasonal)': coefficients_seasonal,
    'P-Value (Seasonal)': p_values_seasonal,
    'Coefficient (IsWeekend)': coefficients_is_weekend,
    'P-Value (IsWeekend)': p_values_is_weekend,
    'Coefficient (IsHoliday)': coefficients_is_holiday,
    'P-Value (IsHoliday)': p_values_is_holiday
}

own_price_raw_rt_df = pd.DataFrame(table_data)

# Apply the significance labels
for column in own_price_raw_rt_df.columns:
    if 'P-Value' in column:
        significance_column = 'Significance' + column.split('P-Value', 1)[1]
        own_price_raw_rt_df[significance_column] = own_price_raw_rt_df[column].apply(lam

# Display the table in the Jupyter Notebook
print("Raw Retail Aggregated Coefficients and Significance Table:")
display(own_price_raw_rt_df)
```

```
Raw Retail Aggregated Coefficients and Significance Table:
```

	Inventory Code	Coefficient (Log Price)	P-Value (Log Price)	Coefficient (Seasonal)	P-Value (Seasonal)	Coefficient (IsWeekend)	P-Value (IsWeekend)	Coefficient (IsHoliday)	P-Value (IsHoliday)
0	1030.0	-2.442502	1.938956e-01	0.000218	0.962211	-0.059351	0.803904	0.0	NaN
1	1031.0	-9.285811	3.811997e-11	0.018078	0.608664	-0.255796	0.200991	0.0	NaN
2	1032.0	8.206441	2.143935e-04	0.011381	0.165847	0.019811	0.892503	0.0	NaN
3	1036.0	-2.290421	2.514034e-01	0.010193	0.044687	0.109221	0.566987	0.0	NaN

In [329…

```
#Finding Min and Max Price & quantity for optimization

# Group by 'Inventory Code'
grouped_data = raw_rt_df3.groupby('Inventory Code')

# Calculate minimum and maximum values for each group
raw_rt_min_max_data = grouped_data.agg({
    'Price Per Unit': ['min', 'max'],
    'Qty': ['min', 'max']
}).reset_index()

# Rename the columns for clarity
raw_rt_min_max_data.columns = ['Inventory Code', 'Min Price', 'Max Price', 'Min Qty', 'M

# Display the result
print(raw_rt_min_max_data)
```

```
   Inventory Code  Min Price  Max Price  Min Qty  Max Qty
0          1030.0        4.9        5.6     31.2   1248.0
1          1031.0        4.7        5.6     15.6    312.0
2          1032.0        5.2        5.6     15.6    218.4
3          1036.0        5.6        6.2     15.6    405.6
```

In [330…

```
## Price Optimization Using Min & Max Qty & Price as bounds

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_raw_rt_df, raw_rt_min_max_data, on='Inventory Code')

# Function to calculate revenue based on price and quantity
```

```
def calculate_revenue(price, qty, coefficient_price):
    return -price * qty * coefficient_price  # Negative sign because scipy.optimize mini

# Function to be minimized (negative revenue for maximization)
def objective_function(prices):
    total_revenue = 0
    for index, row in merged_data.iterrows():
        min_price, max_price = row['Min Price'], row['Max Price']
        min_qty, max_qty = row['Min Qty'], row['Max Qty']
        coefficient_price = row['Coefficient (Log Price)']

        # Ensure prices are within bounds
        price = max(min(max_price, prices[index]), min_price)

        # Calculate revenue
        qty = max(min(max_qty, prices[len(merged_data) + index]), min_qty)
        total_revenue += calculate_revenue(price, qty, coefficient_price)

    return total_revenue

# Initial guess for prices
initial_prices = merged_data['Min Price'].values.tolist() + merged_data['Min Qty'].value

# Constraints
price_constraints = [(min_price, max_price) for min_price, max_price in zip(merged_data[
qty_constraints = [(min_qty, max_qty) for min_qty, max_qty in zip(merged_data['Min Qty']
constraints = price_constraints + qty_constraints

# Optimization
result = minimize(objective_function, initial_prices, constraints={'type': 'ineq', 'fun'
                  bounds=constraints)

# Extract optimized prices and quantities
optimized_prices = result.x[:len(merged_data)]
optimized_quantities = result.x[len(merged_data):]

# Display optimized prices and quantities
optimized_data = merged_data.copy()
optimized_data['Optimized Price'] = optimized_prices
optimized_data['Optimized Quantity'] = optimized_quantities

print("Optimized Prices and Quantities:")
print(optimized_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
Optimized Prices and Quantities:
   Inventory Code  Optimized Price  Optimized Quantity
0          1030.0         4.900000                31.2
1          1031.0         4.700000                15.6
2          1032.0         5.599999               218.4
3          1036.0         5.600000                15.6
```

In [331…

```
## Price Optimization Using Min Price as bounds

# Function to calculate revenue based on price and quantity
def calculate_revenue(price, elasticity, quantity):
    return price * quantity * (1 + elasticity)

# Function to be minimized (negative revenue to maximize)
def objective_function(params, *args):
    price, quantity = params
    elasticity = args[0]  # Extract elasticity from the tuple
    return -calculate_revenue(price, elasticity, quantity)

# Merge the two DataFrames on 'Inventory Code'
merged_data = pd.merge(own_price_raw_rt_df, raw_rt_min_max_data, on='Inventory Code')
```

```
# Initialize empty lists to store results
optimized_prices = []
optimized_quantities = []

# Loop through each row in merged_data
for index, row in merged_data.iterrows():
    inventory_code = row['Inventory Code']
    min_price = row['Min Price']

    # Get elasticity from own_price_rte_sp_df based on the inventory code
    elasticity = own_price_raw_rt_df.loc[own_price_raw_rt_df['Inventory Code'] == invent

    # Set bounds for optimization
    bounds = [(min_price, None), (None, None)]  # Remove lower bound for quantity

    # Perform optimization
    result = minimize(objective_function, x0=[min_price, row['Min Qty']], args=(elastici

    # Extract optimized price and quantity
    optimized_price, optimized_quantity = result.x

    # Append results to lists
    optimized_prices.append(optimized_price)
    optimized_quantities.append(optimized_quantity)

# Add optimized prices and quantities to the DataFrame
merged_data['Optimized Price'] = optimized_prices
merged_data['Optimized Quantity'] = optimized_quantities

# Display the DataFrame
print(merged_data[['Inventory Code', 'Optimized Price', 'Optimized Quantity']])
```

```
   Inventory Code  Optimized Price  Optimized Quantity
0          1030.0    1.636095e+145      -8.298601e+151
1          1031.0    2.025432e+143      -4.078056e+142
2          1032.0    2.166628e+144       5.079904e+144
3          1036.0    5.101833e+136      -4.499506e+154
```

Market Basket Analysis

In [332…

```
rfm
```

Out[332]:

	Customer Code	Recency	Frequency	Monetary	R	F	M	RFM_Segment	RFM_Score	Segment	Score	RFM
0	A001	1	3082	316950.01	5	5	5	555	15	Champions	Platinum	
1	A002	14	19	2536.04	3	3	2	332	8	Need attention	Silver	
2	A003	15	120	7482.41	3	5	4	354	12	Need attention	Gold	
3	A004	61	113	4414.15	2	4	3	243	9	Hibernating	Silver	
4	A005	3	2803	220783.16	5	5	5	555	15	Champions	Platinum	
...	...	...	...	...	...	...	...	...	...	...	...	
593	Z004	2	102	6887.79	5	4	4	544	13	Loyal	Gold	
594	Z005	121	15	1501.76	1	2	2	122	5	Hibernating	Green	
595	Z006	1	67	10282.74	5	4	4	544	13	Loyal	Gold	
596	Z008	27	27	3095.56	2	3	3	233	8	Hibernating	Silver	

597	Z009	28	10	1135.68	2	2	2	222	6	Hibernating	Bronze

598 rows × 12 columns

In [333…
```
# Extract the 'Segment' column from 'rfm_df'
segment_column = rfm[['Customer Code', 'Segment']]

# Merge 'clean_df' with the 'segment_column' based on the 'Customer Code' column
df = pd.merge(df_sales, segment_column, on='Customer Code', how='left')
```

In [334…
```
df
```

Out[334]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	Q
0	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	Ready to Cook (RTC)	156
1	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	156
2	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1160.0	(9) SKINLESS CHIX BREAST 345G	Ready to Cook (RTC)	156
3	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1162.0	(3) CRISPY CHIX S/WEED 400G	Ready to Cook (RTC)	312
4	2022-07-01	SO00000002	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	15
...	...	...	...	...	...	...	...	...	
74930	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1173.0	YAKITORI WITH SAUCE	Ready to Cook (RTC)	1
74931	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1190.0	N1 CHICKEN NUGGET (10PKT)	Ready to Cook (RTC)	2
74932	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1191.0	TEMPURA CHICKEN NUGGET	Ready to Cook (RTC)	3
74933	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1184.0	CHEESE CHICKEN BALL	Ready to Cook (RTC)	5
74934	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1360.0	COCONUT WATER WITH MEAT 250ML	Ready to Cook (RTC)	26

74935 rows × 14 columns

In [335…
```
df = df.drop(df[df['Inventory Desc'] == 'Containers or Stocks'].index)
```

In [336…
```
# Create a binary-encoded DataFrame
encoded_data = pd.get_dummies(df['Inventory Code']).groupby(df['Sales Order No.']).max()
encoded_data
```

Out[336]:

Sales Order No.	1010.0	1011.0	1013.0	1014.0	1015.0	1030.0	1031.0	1032.0	1033.0	1034.0	…	1602.0	1603.
SO00000001	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00000002	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00000003	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00000004	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00000005	False	False	False	False	False	False	False	False	False	False	…	False	Fals
…	…	…	…	…	…	…	…	…	…	…	…	…	
SO00030679	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00030681	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00030683	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00030684	False	False	False	False	False	False	False	False	False	False	…	False	Fals
SO00030685	False	False	False	False	False	False	False	False	False	False	…	False	Fals

28261 rows × 125 columns

In [337…
```
# Calculate support for all item combinations
support = encoded_data.mean()

# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data.astype(bool), min_support=0.03, use_colnames=Tr

# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres

# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1)

# Concatenate the two DataFrames
c_rules = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)
```

In [338…
```
# For the mapping of antecedents and consequents
sku_to_category = df.set_index('Inventory Code')['Inventory Category'].to_dict()

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules['Antecedent Product Category'] = c_rules['antecedents'].apply(lambda x: [sku_to_
c_rules['Consequent Product Category'] = c_rules['consequents'].apply(lambda x: [sku_to_
```

In [339…
```
# Display frequent itemsets and association rules
c_rules.sort_values('support', ascending = False)
c_rules.head()
```

Out[339]:

	antecedents	consequents	antecedent	consequent	support	confidence	lift	leverage	conviction	zhan

			support	support					
0	(1142.0)	(1126.0)	0.171332	0.126747	0.036340	0.212102	1.673430	0.014624	1.108333
1	(1126.0)	(1142.0)	0.126747	0.171332	0.036340	0.286711	1.673430	0.014624	1.161758
2	(1145.0)	(1126.0)	0.077492	0.126747	0.035632	0.459817	3.627833	0.025810	1.616588
3	(1126.0)	(1145.0)	0.126747	0.077492	0.035632	0.281128	3.627833	0.025810	1.283271
4	(1145.0)	(1142.0)	0.077492	0.171332	0.040338	0.520548	3.038250	0.027061	1.728366

In [340…

```
### Remove duplicates based on 'antecedents' and 'consequents'
c_rules = c_rules.drop_duplicates(subset=['antecedents', 'consequents'])

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules = c_rules.sort_values(by='lift', ascending=False)

# Display the top 20 cross-sell/up-sell opportunities
top_20_opportunities = sorted_rules.head(20)

# Display the top 5 cross-sell/up-sell opportunities
print("Top 20 Cross-Sell/Up-Sell Associations for Overall Dataset")
top_20_opportunities
```

```
Top 20 Cross-Sell/Up-Sell Associations for Overall Dataset
```

Out[340]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
36	(1707.0)	(1709.0)	0.061887	0.061003	0.048371	0.781589	12.812355	0.044595	4.299231	
37	(1709.0)	(1707.0)	0.061003	0.061887	0.048371	0.792923	12.812355	0.044595	4.530269	
40	(1177.0, 1142.0)	(1176.0)	0.047274	0.067160	0.030714	0.649701	9.673967	0.027539	2.662980	
41	(1176.0)	(1177.0, 1142.0)	0.067160	0.047274	0.030714	0.457323	9.673967	0.027539	1.755606	
42	(1177.0)	(1176.0, 1142.0)	0.088744	0.037224	0.030714	0.346093	9.297453	0.027410	1.472342	
39	(1176.0, 1142.0)	(1177.0)	0.037224	0.088744	0.030714	0.825095	9.297453	0.027410	5.210006	
33	(1177.0)	(1176.0)	0.088744	0.067160	0.045504	0.512759	7.634925	0.039544	1.914536	
32	(1176.0)	(1177.0)	0.067160	0.088744	0.045504	0.677555	7.634925	0.039544	2.826084	

19	(1154.0)	(1145.0)	0.082198	0.077492	0.031563	0.383986	4.955176	0.025193	1.497545
18	(1145.0)	(1154.0)	0.077492	0.082198	0.031563	0.407306	4.955176	0.025193	1.548526
38	(1176.0, 1177.0)	(1142.0)	0.045504	0.171332	0.030714	0.674961	3.939504	0.022917	2.549444
43	(1142.0)	(1176.0, 1177.0)	0.171332	0.045504	0.030714	0.179265	3.939504	0.022917	1.162976
2	(1145.0)	(1126.0)	0.077492	0.126747	0.035632	0.459817	3.627833	0.025810	1.616588
3	(1126.0)	(1145.0)	0.126747	0.077492	0.035632	0.281128	3.627833	0.025810	1.283271
34	(1177.0)	(1191.0)	0.088744	0.104349	0.031138	0.350877	3.362543	0.021878	1.379787
35	(1191.0)	(1177.0)	0.104349	0.088744	0.031138	0.298406	3.362543	0.021878	1.298837
12	(1176.0)	(1142.0)	0.067160	0.171332	0.037224	0.554268	3.235059	0.025718	1.859117
13	(1142.0)	(1176.0)	0.171332	0.067160	0.037224	0.217266	3.235059	0.025718	1.191771
14	(1177.0)	(1142.0)	0.088744	0.171332	0.047274	0.532695	3.109150	0.032069	1.773294
15	(1142.0)	(1177.0)	0.171332	0.088744	0.047274	0.275919	3.109150	0.032069	1.258500

In [341…
```
supermarket_df = df[df['Customer Category Desc'] == 'SUPERMARKET']
```

In [342…
```
supermarket_df
```

Out[342]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	
0	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	Ready to Cook (RTC)	15
1	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	15
2	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1160.0	(9) SKINLESS CHIX BREAST 345G	Ready to Cook (RTC)	15
3	2022-07-01	SO00000001	C033	CMM MARKETING	SUPERMARKET	1162.0	(3) CRISPY CHIX	Ready to Cook	3

				MANAGEMENT PTE LTD			S/WEED 400G	(RTC)	
4	2022-07-01	SO00000002	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	1
...	...	...	...	...	...	...	...	...	
74917	2023-05-31	SO00030681	S017	7 ELEVEN SINGAPORE PTE LTD	SUPERMARKET	1334.0	BETAGRO BREAST BLACK PEPPER CK CUT	Ready to Eat (RTE)	284
74918	2023-05-31	SO00030681	S017	7 ELEVEN SINGAPORE PTE LTD	SUPERMARKET	1330.0	BETAGRO BREAST HOT & SPICY	Ready to Eat (RTE)	295
74919	2023-05-31	SO00030681	S017	7 ELEVEN SINGAPORE PTE LTD	SUPERMARKET	1332.0	BETAGRO BREAST HERBS	Ready to Eat (RTE)	305
74920	2023-05-31	SO00030681	S017	7 ELEVEN SINGAPORE PTE LTD	SUPERMARKET	1331.0	BETAGRO BREAST GARLIC PEPPER	Ready to Eat (RTE)	315
74925	2023-05-31	SO00030684	S017	7 ELEVEN SINGAPORE PTE LTD	SUPERMARKET	1141.0	FS07 FRIED CRISPY DRUM	Ready to Cook (RTC)	

16204 rows × 14 columns

In [343…
```
unique_sku_count_supermarket = supermarket_df['Inventory Code'].nunique()
print("Number of unique SKUs for Supermarket Customers:", unique_sku_count_supermarket)
```

```
Number of unique SKUs for Supermarket Customers: 63
```

In [344…
```
# Create a binary-encoded DataFrame
encoded_data_supermarket = pd.get_dummies(supermarket_df['Inventory Code']).groupby(supe
encoded_data_supermarket
```

Out[344]:

Sales Order No.	1010.0	1011.0	1013.0	1014.0	1015.0	1030.0	1031.0	1032.0	1033.0	1034.0	...	1334.0	1345.
SO00000001	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000002	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000003	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000004	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000005	False	False	False	False	False	False	False	False	False	False	...	False	Fals
...	...	...	...	...	...	...	...	...	...	...	...	...	
SO00030608	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030610	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030677	False	False	False	False	False	True	True	True	True	True	...	False	Fals
SO00030681	False	False	False	False	False	False	False	False	False	False	...	True	Tru

SO00030684	False	False	False	False	False	False	False	False	False	False	...	False	Fals

7150 rows × 63 columns

In [345…

```
# Calculate support for all item combinations
support = encoded_data_supermarket.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_supermarket.astype(bool), min_support=0.04, use
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_supermarket = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_supermarket.sort_values('support', ascending = False)
c_rules_supermarket.head(20)
```

Out[345]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zł
0	(1030.0)	(1031.0)	0.049091	0.041958	0.041818	0.851852	20.302469	0.039758	6.466783	
1	(1031.0)	(1030.0)	0.041958	0.049091	0.041818	0.996667	20.302469	0.039758	285.272727	
2	(1032.0)	(1030.0)	0.049091	0.049091	0.048252	0.982906	20.022159	0.045842	55.628182	
3	(1030.0)	(1032.0)	0.049091	0.049091	0.048252	0.982906	20.022159	0.045842	55.628182	
4	(1033.0)	(1030.0)	0.049650	0.049091	0.048392	0.974648	19.853938	0.045954	37.508081	
5	(1030.0)	(1033.0)	0.049091	0.049650	0.048392	0.985755	19.853938	0.045954	66.714545	
6	(1034.0)	(1030.0)	0.044196	0.049091	0.043357	0.981013	19.983591	0.041187	50.081212	
7	(1030.0)	(1034.0)	0.049091	0.044196	0.043357	0.883191	19.983591	0.041187	8.182616	
8	(1035.0)	(1030.0)	0.041678	0.049091	0.040839	0.979866	19.960229	0.038793	47.228485	
9	(1030.0)	(1035.0)	0.049091	0.041678	0.040839	0.831909	19.960229	0.038793	5.701202	
10	(1036.0)	(1030.0)	0.043217	0.049091	0.041399	0.957929	19.513364	0.039277	22.602378	
11	(1030.0)	(1036.0)	0.049091	0.043217	0.041399	0.843305	19.513364	0.039277	6.106017	
12	(1037.0)	(1030.0)	0.043636	0.049091	0.041958	0.961538	19.586895	0.039816	24.723636	
13	(1030.0)	(1037.0)	0.049091	0.043636	0.041958	0.854701	19.586895	0.039816	6.582032	
14	(1032.0)	(1031.0)	0.049091	0.041958	0.041678	0.849003	20.234568	0.039619	6.344768	
15	(1031.0)	(1032.0)	0.041958	0.049091	0.041678	0.993333	20.234568	0.039619	142.636364	
16	(1033.0)	(1031.0)	0.049650	0.041958	0.041678	0.839437	20.006573	0.039595	5.966753	
17	(1031.0)	(1033.0)	0.041958	0.049650	0.041678	0.993333	20.006573	0.039595	142.552448	
18	(1034.0)	(1031.0)	0.044196	0.041958	0.041119	0.930380	22.174051	0.039265	13.760966	
19	(1031.0)	(1034.0)	0.041958	0.044196	0.041119	0.980000	22.174051	0.039265	47.790210	

In [346…

```
# For the mapping of antecedents and consequents
sku_to_category = supermarket_df.set_index('Inventory Code')['Inventory Category'].to_di

# Map the antecedents and consequents in your market basket analysis DataFrame
```

```
c_rules_supermarket['Antecedent Product Category'] = c_rules_supermarket['antecedents'].
c_rules_supermarket['Consequent Product Category'] = c_rules_supermarket['consequents'].
```

In [347…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_supermarket = c_rules_supermarket.drop_duplicates(subset=['antecedents', 'conseq

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_supermarket = c_rules_supermarket.sort_values(by='support', ascending=False

# Select the top 10 cross-sell/up-sell opportunities
top_opportunities_supermarket = sorted_rules_supermarket.head(20)

# Display the top 10 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for Supermarket Customers:")
top_opportunities_supermarket
```

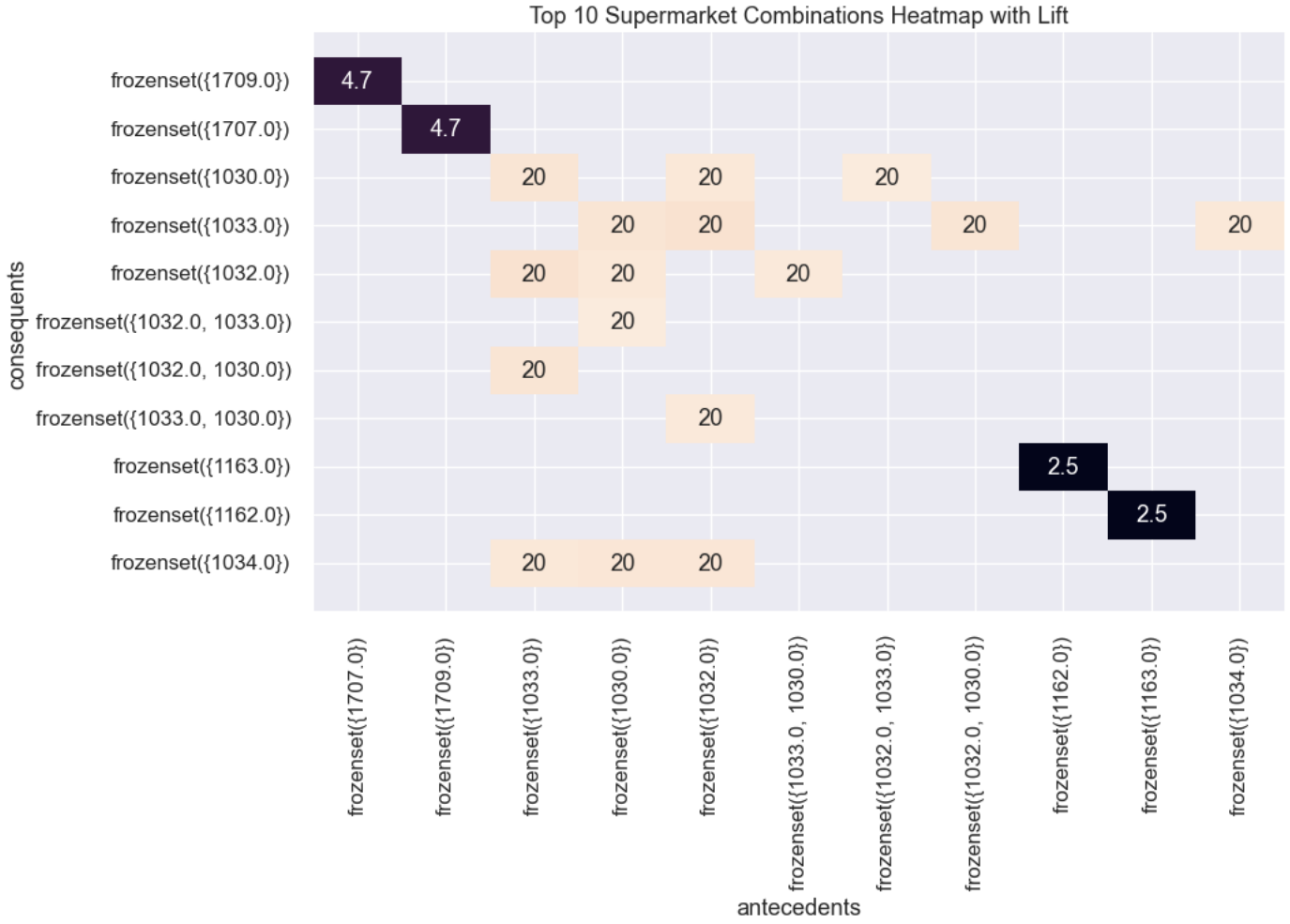
```
Top Cross-Sell/Up-Sell Opportunities for Supermarket Customers:
```

Out[347]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	z
58	(1707.0)	(1709.0)	0.189930	0.168951	0.149650	0.787923	4.663620	0.117561	3.918627	
59	(1709.0)	(1707.0)	0.168951	0.189930	0.149650	0.885762	4.663620	0.117561	7.091047	
4	(1033.0)	(1030.0)	0.049650	0.049091	0.048392	0.974648	19.853938	0.045954	37.508081	
5	(1030.0)	(1033.0)	0.049091	0.049650	0.048392	0.985755	19.853938	0.045954	66.714545	
2	(1032.0)	(1030.0)	0.049091	0.049091	0.048252	0.982906	20.022159	0.045842	55.628182	
3	(1030.0)	(1032.0)	0.049091	0.049091	0.048252	0.982906	20.022159	0.045842	55.628182	
27	(1033.0)	(1032.0)	0.049650	0.049091	0.048112	0.969014	19.739176	0.045675	30.688430	
26	(1032.0)	(1033.0)	0.049091	0.049650	0.048112	0.980057	19.739176	0.045675	47.653247	
98	(1033.0, 1030.0)	(1032.0)	0.048392	0.049091	0.047692	0.985549	20.076001	0.045317	65.802909	
101	(1030.0)	(1032.0, 1033.0)	0.049091	0.048112	0.047692	0.971510	20.192722	0.045330	33.411273	
100	(1033.0)	(1032.0, 1030.0)	0.049650	0.048252	0.047692	0.960563	19.907328	0.045297	24.133616	
99	(1032.0)	(1033.0, 1030.0)	0.049091	0.048392	0.047692	0.971510	20.076001	0.045317	33.401455	
96	(1032.0, 1033.0)	(1030.0)	0.048112	0.049091	0.047692	0.991279	20.192722	0.045330	109.037576	
97	(1032.0, 1030.0)	(1033.0)	0.048252	0.049650	0.047692	0.988406	19.907328	0.045297	81.967657	
56	(1162.0)	(1163.0)	0.166993	0.110070	0.046014	0.275544	2.503358	0.027633	1.228412	
57	(1163.0)	(1162.0)	0.110070	0.166993	0.046014	0.418043	2.503358	0.027633	1.431390	
37	(1034.0)	(1033.0)	0.044196	0.049650	0.043916	0.993671	20.013371	0.041722	150.155245	
36	(1033.0)	(1034.0)	0.049650	0.044196	0.043916	0.884507	20.013371	0.041722	8.275866	
28	(1032.0)	(1034.0)	0.049091	0.044196	0.043357	0.883191	19.983591	0.041187	8.182616	
7	(1030.0)	(1034.0)	0.049091	0.044196	0.043357	0.883191	19.983591	0.041187	8.182616	

In [348…

```
# Transform the DataFrame of rules into a matrix using the lift metric
pivot2 = top_opportunities_supermarket.pivot(index = 'consequents',
                    columns = 'antecedents', values= 'lift')

# Generate a heatmap with annotations on and the colorbar off
plt.figure(figsize=(10,6))
sns.heatmap(pivot2, annot = True, cbar = False)
b, t = plt.ylim()
b += 0.5
t -= 0.5
plt.ylim(b, t)
plt.yticks(rotation=0)
plt.xticks(rotation=90)
plt.title('Top 10 Supermarket Combinations Heatmap with Lift')
plt.show()
```

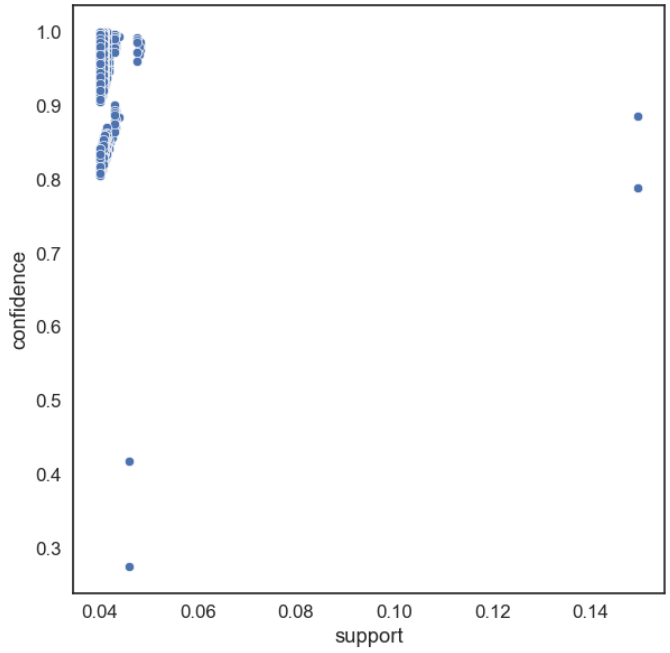


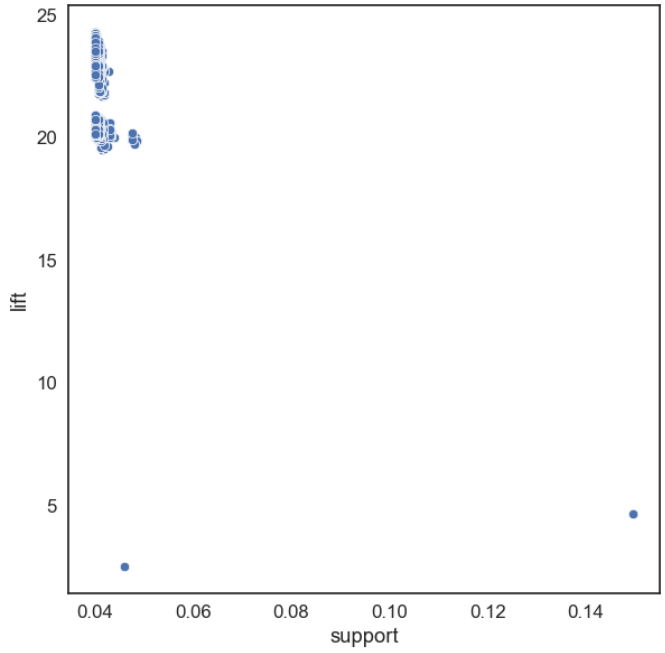
In [349…

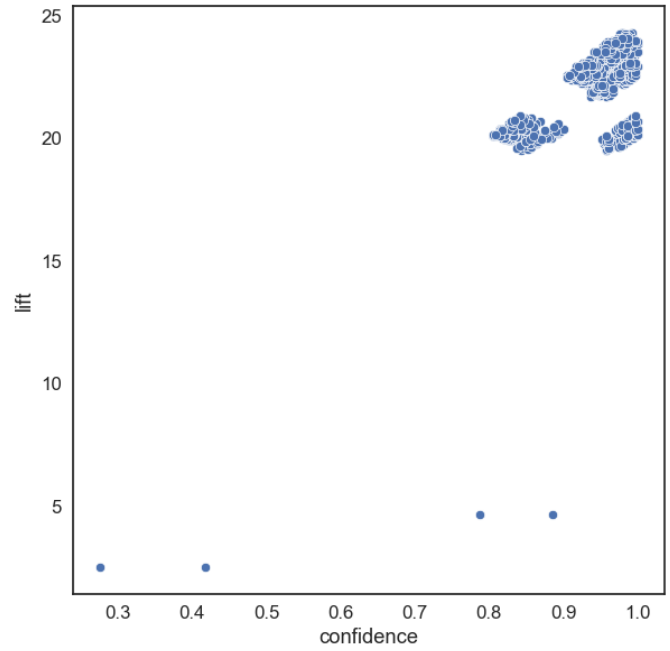
```
#Setting up the style
plt.figure(figsize = (15, 15))
plt.style.use('seaborn-white')
#Plotting the relationship between the metrics
plt.subplot(221)
sns.scatterplot(x="support", y="confidence",data=c_rules_supermarket)
plt.subplot(222)
sns.scatterplot(x="support", y="lift",data=c_rules_supermarket)
plt.subplot(223)
sns.scatterplot(x="confidence", y="lift",data=c_rules_supermarket)
plt.subplot(224)
sns.scatterplot(x="antecedent support", y="consequent support",data=c_rules_supermarket)
```

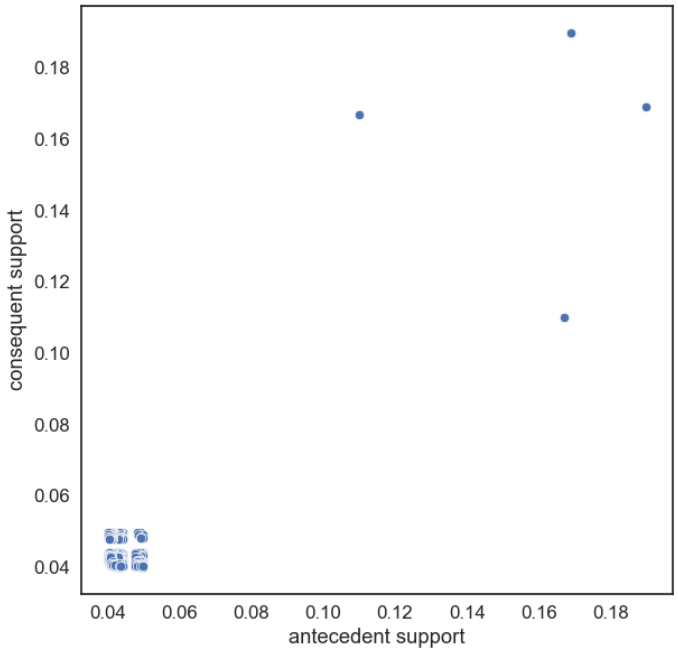
Out[349]:

```
<Axes: xlabel='antecedent support', ylabel='consequent support'>
```





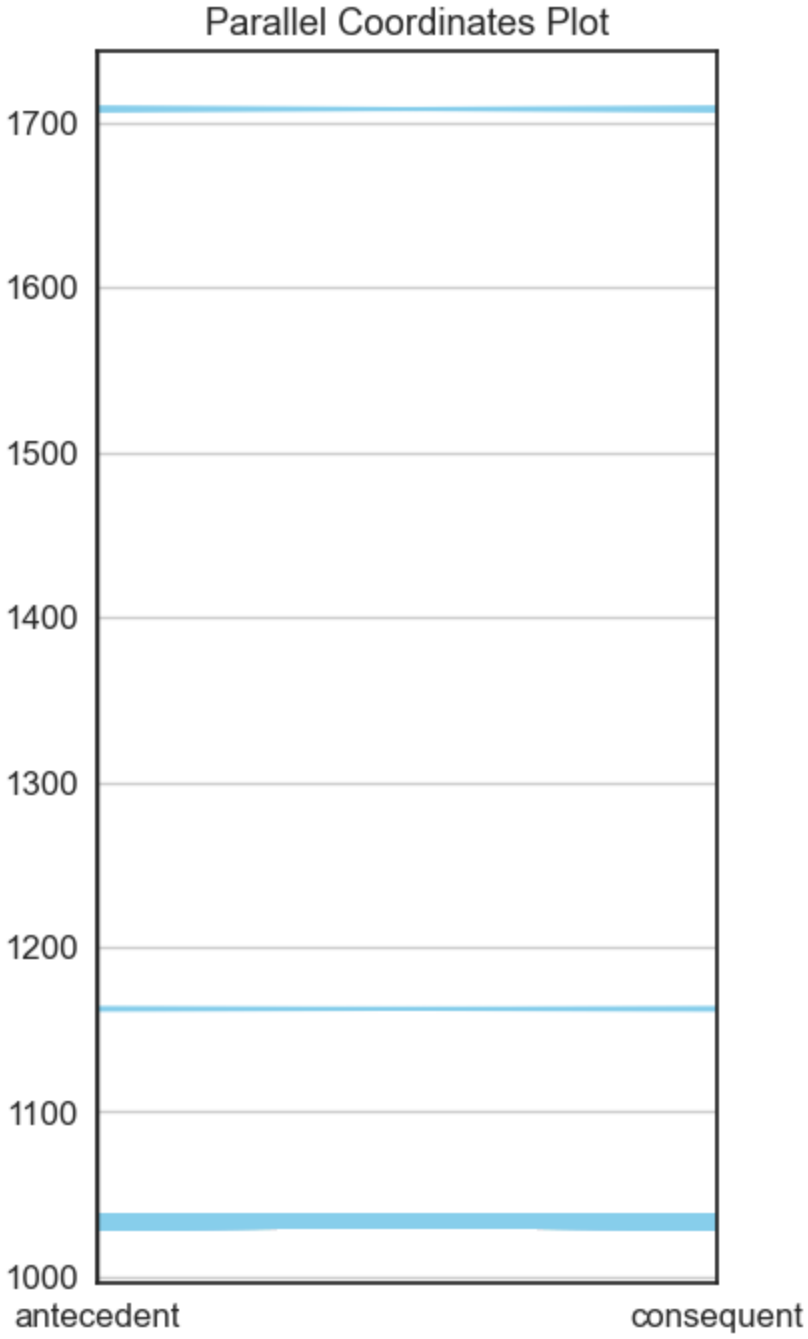




In [350…

```
### Parallel coordinates plot
#The parallel coordinates plot will allow us to visualize whether a relationship exist b

# Function to convert rules to coordinates.
def rules_to_coordinates(c_rules):
    c_rules_supermarket['antecedent'] = c_rules_supermarket['antecedents'].apply(lambda
    c_rules_supermarket['consequent'] = c_rules_supermarket['consequents'].apply(lambda
    c_rules_supermarket['rule'] = c_rules.index
    return c_rules_supermarket[['antecedent','consequent','rule']]

from pandas.plotting import parallel_coordinates

# Convert rules into coordinates suitable for use in a parallel coordinates plot
coords = rules_to_coordinates(c_rules_supermarket)

# Generate parallel coordinates plot
plt.figure(figsize=(4,8))
parallel_coordinates(coords, 'rule', color='skyblue')
plt.legend([])
plt.grid(True)
```

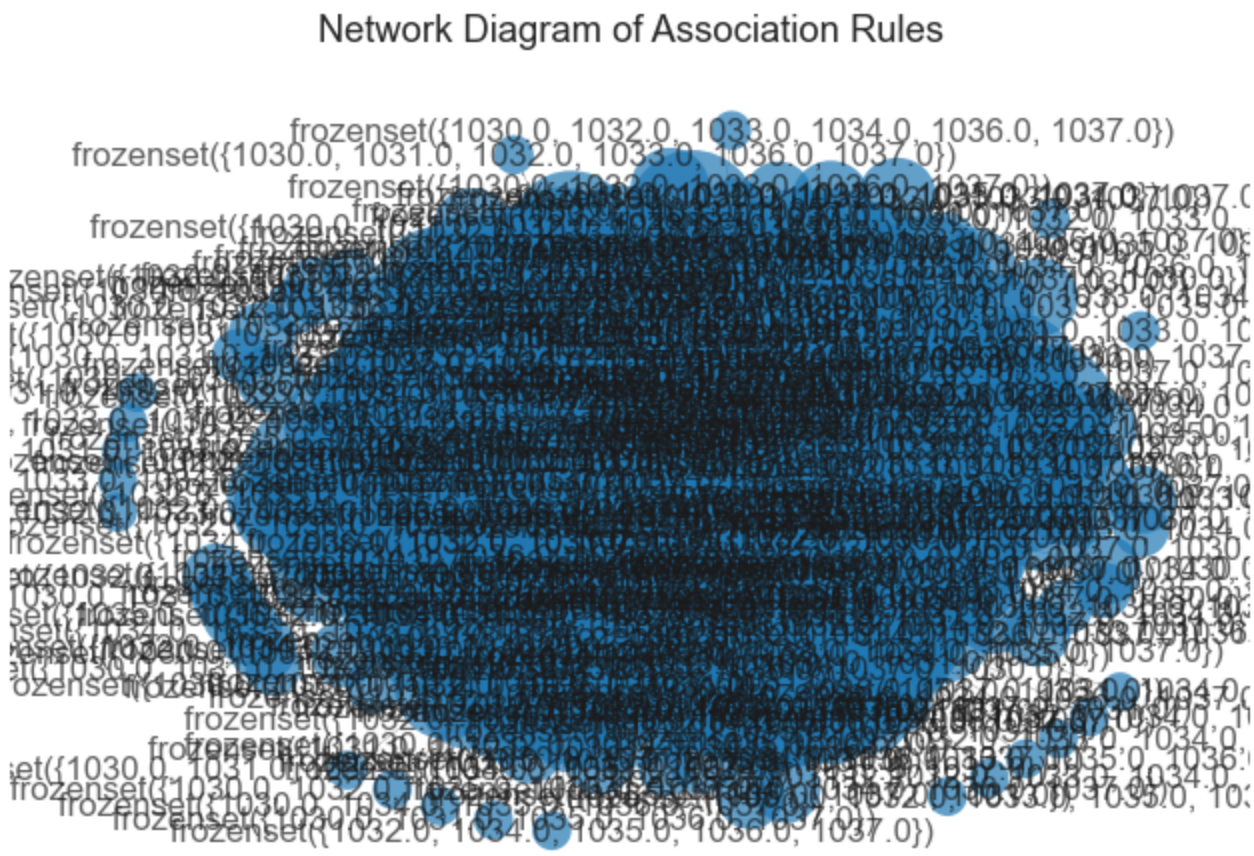
```
plt.title('Parallel Coordinates Plot')
plt.show()
```



In [351…

```
import plotly.graph_objects as go
from plotly.offline import download_plotlyjs, init_notebook_mode, iplot
import networkx as nx

bk_network = c_rules_supermarket[['antecedents', 'consequents']]

bk_network_G = nx.from_pandas_edgelist(
    bk_network,
    source = 'antecedents',
    target = 'consequents',
    create_using = nx.DiGraph())

network_bt = nx.in_degree_centrality(bk_network_G)
bk_network_bt = pd.DataFrame(list(network_bt.items()), columns = ['item', 'centrality'])
bk_network_bt = bk_network_bt.dropna(axis=0)

pos = nx.kamada_kawai_layout(bk_network_G)

sizes = [x[1]*100 for x in bk_network_G.degree()]
```

```
nx.draw_networkx(bk_network_G, pos,
    with_labels = True,
    node_size = sizes,
    width = 0.1, alpha = 0.7,
    arrowsize = 2, linewidths = 0)

plt.axis('off')
plt.title('Network Diagram of Association Rules')
plt.show()
```



Market Basket Analysis for Retail

In [352…
```
retail_df = df[df['Customer Category Desc'] == 'RETAIL']
```

In [353…
```
retail_df
```

Out[353]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	Qty	Tota Bas Am
52	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1341.0	SEA SALT YAKITORI	Ready to Cook (RTC)	20.8	37.4
53	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1331.0	BETAGRO BREAST GARLIC PEPPER	Ready to Eat (RTE)	20.8	37.4
54	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1333.0	BETAGRO BREAST PINK SALT	Ready to Eat (RTE)	20.8	37.4

55	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1340.0	CURRY YAKITORI	Ready to Cook (RTC)	20.8	37.4
56	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1332.0	BETAGRO BREAST HERBS	Ready to Eat (RTE)	20.8	37.4
...	...	...	...	...	...	...	...	...	...	.
74818	2023-05-31	SO00030635	A001	ADVANCE FOOD SYSTEMS	RETAIL	1103.0	JPS03 PORK SAUSAGE ARABIKI	Ready to Cook (RTC)	15.6	53.0
74819	2023-05-31	SO00030635	A001	ADVANCE FOOD SYSTEMS	RETAIL	1217.0	HONEY PORK RIBS	Ready to Cook (RTC)	1.3	130.0
74820	2023-05-31	SO00030635	A001	ADVANCE FOOD SYSTEMS	RETAIL	1218.0	HONEY CHAR SIEW 5KG	Ready to Cook (RTC)	13.0	140.4
74821	2023-05-31	SO00030635	A001	ADVANCE FOOD SYSTEMS	RETAIL	1316.0	GRILLED CHIX STEAK (5PKT)	Ready to Cook (RTC)	10.4	197.6
74910	2023-05-31	SO00030679	A001	ADVANCE FOOD SYSTEMS	RETAIL	1217.0	HONEY PORK RIBS	Ready to Cook (RTC)	1.3	130.0

6802 rows × 14 columns

In [354…
```
unique_sku_count_retail = retail_df['Inventory Code'].nunique()
print("Number of unique SKUs for Retail Customers:", unique_sku_count_retail)
```

```
Number of unique SKUs for Retail Customers: 56
```

In [355…
```
# Create a binary-encoded DataFrame
encoded_data_retail = pd.get_dummies(retail_df['Inventory Code']).groupby(retail_df['Sal
encoded_data_retail
```

Out[355]:

Sales Order No.	1010.0	1030.0	1031.0	1032.0	1036.0	1065.0	1090.0	1091.0	1092.0	1100.0	...	1341.0	1342.
SO00000024	False	False	False	False	False	False	False	False	False	False	...	True	Tru
SO00000026	False	False	False	False	False	False	False	False	False	False	...	True	Tru
SO00000027	False	False	False	False	False	False	False	False	False	False	...	True	Tru
SO00000030	False	False	False	False	False	False	False	False	False	False	...	True	Tru
SO00000032	False	False	False	False	False	False	False	False	False	False	...	False	Fals
...	...	...	...	...	...	...	...	...	...	...	...	...	.
SO00030597	False	True	True	True	True	False	False	False	False	False	...	False	Fals
SO00030603	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030634	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030635	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030679	False	False	False	False	False	False	False	False	False	False	...	False	Fals

1675 rows × 56 columns

In [356…
```
# Calculate support for all item combinations
support = encoded_data_retail.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_retail.astype(bool), min_support=0.04, use_coln
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_retail = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_retail.sort_values('support', ascending = False)
c_rules_retail.head(20)
```

Out[356]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
0	(1030.0)	(1031.0)	0.047164	0.045373	0.044776	0.949367	20.923551	0.042636	18.853881	
1	(1031.0)	(1030.0)	0.045373	0.047164	0.044776	0.986842	20.923551	0.042636	72.415522	
2	(1032.0)	(1030.0)	0.047761	0.047164	0.046567	0.975000	20.672468	0.044315	38.113433	
3	(1030.0)	(1032.0)	0.047164	0.047761	0.046567	0.987342	20.672468	0.044315	75.226866	
4	(1036.0)	(1030.0)	0.048955	0.047164	0.047164	0.963415	20.426829	0.044855	26.044179	
5	(1030.0)	(1036.0)	0.047164	0.048955	0.047164	1.000000	20.426829	0.044855	inf	
6	(1032.0)	(1031.0)	0.047761	0.045373	0.044776	0.937500	20.662007	0.042609	15.274030	
7	(1031.0)	(1032.0)	0.045373	0.047761	0.044776	0.986842	20.662007	0.042609	72.370149	
8	(1036.0)	(1031.0)	0.048955	0.045373	0.045373	0.926829	20.426829	0.043152	13.046567	
9	(1031.0)	(1036.0)	0.045373	0.048955	0.045373	1.000000	20.426829	0.043152	inf	
10	(1032.0)	(1036.0)	0.047761	0.048955	0.047761	1.000000	20.426829	0.045423	inf	
11	(1036.0)	(1032.0)	0.048955	0.047761	0.047761	0.975610	20.426829	0.045423	39.041791	
12	(1217.0)	(1103.0)	0.248955	0.071642	0.047761	0.191847	2.677858	0.029926	1.148740	
13	(1103.0)	(1217.0)	0.071642	0.248955	0.047761	0.666667	2.677858	0.029926	2.253134	
14	(1218.0)	(1103.0)	0.296119	0.071642	0.048955	0.165323	2.307628	0.027741	1.112236	
15	(1103.0)	(1218.0)	0.071642	0.296119	0.048955	0.683333	2.307628	0.027741	2.222781	
16	(1300.0)	(1103.0)	0.293134	0.071642	0.044776	0.152749	2.132128	0.023775	1.095730	
17	(1103.0)	(1300.0)	0.071642	0.293134	0.044776	0.625000	2.132128	0.023775	1.884975	
18	(1316.0)	(1103.0)	0.334328	0.071642	0.047164	0.141071	1.969122	0.023212	1.080833	
19	(1103.0)	(1316.0)	0.071642	0.334328	0.047164	0.658333	1.969122	0.023212	1.948307	

In [357…
```
# For the mapping of antecedents and consequents
sku_to_category = retail_df.set_index('Inventory Code')['Inventory Category'].to_dict()

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_retail['Antecedent Product Category'] = c_rules_retail['antecedents'].apply(lamb
c_rules_retail['Consequent Product Category'] = c_rules_retail['consequents'].apply(lamb
```

In [358…
```
# Remove duplicates based on 'antecedents' and 'consequents'
```

```
c_rules_retail = c_rules_retail.drop_duplicates(subset=['antecedents', 'consequents'])

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_retail = c_rules_retail.sort_values(by='support', ascending=False)

# Select the top 10 cross-sell/up-sell opportunities
top_opportunities_retail = sorted_rules_retail.head(20)

# Display the top 10 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for Retail Customers:")
top_opportunities_retail
```

```
Top Cross-Sell/Up-Sell Opportunities for Retail Customers:
```

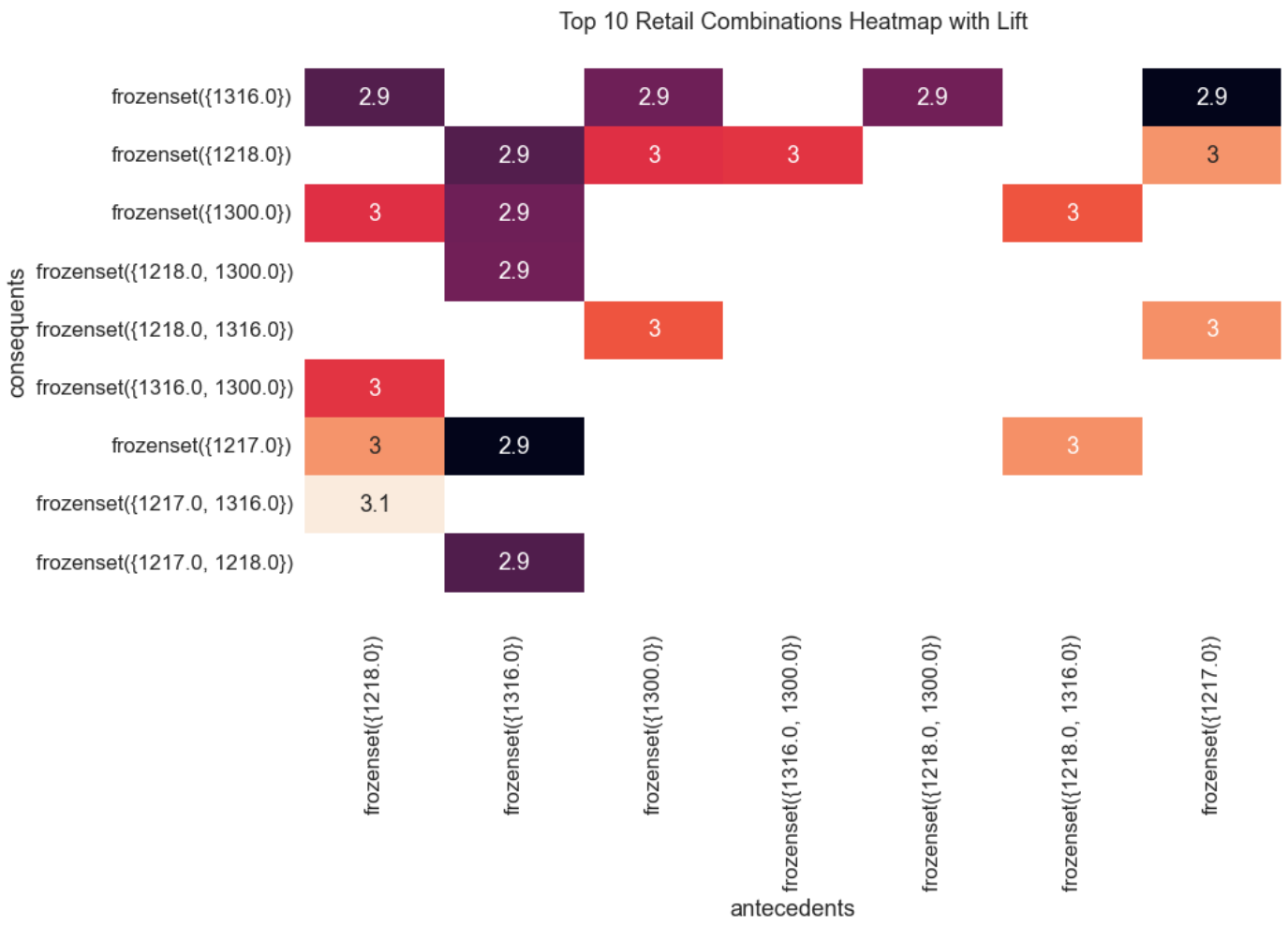
Out[358]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
102	(1218.0)	(1316.0)	0.296119	0.334328	0.290746	0.981855	2.936798	0.191745	36.685904	
103	(1316.0)	(1218.0)	0.334328	0.296119	0.290746	0.869643	2.936798	0.191745	5.399632	
113	(1300.0)	(1316.0)	0.293134	0.334328	0.288955	0.985743	2.948429	0.190952	46.692111	
112	(1316.0)	(1300.0)	0.334328	0.293134	0.288955	0.864286	2.948429	0.190952	5.208484	
98	(1218.0)	(1300.0)	0.296119	0.293134	0.260299	0.879032	2.998735	0.173496	5.843423	
99	(1300.0)	(1218.0)	0.293134	0.296119	0.260299	0.887984	2.998735	0.173496	6.283734	
720	(1316.0, 1300.0)	(1218.0)	0.288955	0.296119	0.256716	0.888430	3.000242	0.171151	6.308856	
722	(1316.0)	(1218.0, 1300.0)	0.334328	0.260299	0.256716	0.767857	2.949910	0.169691	3.186406	
719	(1218.0, 1300.0)	(1316.0)	0.260299	0.334328	0.256716	0.986239	2.949910	0.169691	48.372139	
718	(1218.0, 1316.0)	(1300.0)	0.290746	0.293134	0.256716	0.882957	3.012124	0.171489	6.039361	
723	(1300.0)	(1218.0, 1316.0)	0.293134	0.290746	0.256716	0.875764	3.012124	0.171489	5.708911	
721	(1218.0)	(1316.0, 1300.0)	0.296119	0.288955	0.256716	0.866935	3.000242	0.171151	5.343609	
90	(1217.0)	(1316.0)	0.248955	0.334328	0.241194	0.968825	2.897825	0.157961	21.352698	
91	(1316.0)	(1217.0)	0.334328	0.248955	0.241194	0.721429	2.897825	0.157961	2.696058	

84	(1217.0)	(1218.0)	0.248955	0.296119	0.223881	0.899281	3.036885	0.150160	6.988529
85	(1218.0)	(1217.0)	0.296119	0.248955	0.223881	0.756048	3.036885	0.150160	3.078663
644	(1218.0)	(1217.0, 1316.0)	0.296119	0.241194	0.219701	0.741935	3.076094	0.148279	2.940373
643	(1217.0)	(1218.0, 1316.0)	0.248955	0.290746	0.219701	0.882494	3.035272	0.147319	6.035894
645	(1316.0)	(1217.0, 1218.0)	0.334328	0.223881	0.219701	0.657143	2.935238	0.144852	2.263682
642	(1218.0, 1316.0)	(1217.0)	0.290746	0.248955	0.219701	0.755647	3.035272	0.147319	3.073603

In [359…

```
# Transform the DataFrame of rules into a matrix using the lift metric
pivot2 = top_opportunities_retail.pivot(index = 'consequents',
                    columns = 'antecedents', values= 'lift')

# Generate a heatmap with annotations on and the colorbar off
plt.figure(figsize=(10,6))
sns.heatmap(pivot2, annot = True, cbar = False)
b, t = plt.ylim()
b += 0.5
t -= 0.5
plt.ylim(b, t)
plt.yticks(rotation=0)
plt.xticks(rotation=90)
plt.title('Top 10 Retail Combinations Heatmap with Lift')
plt.show()
```

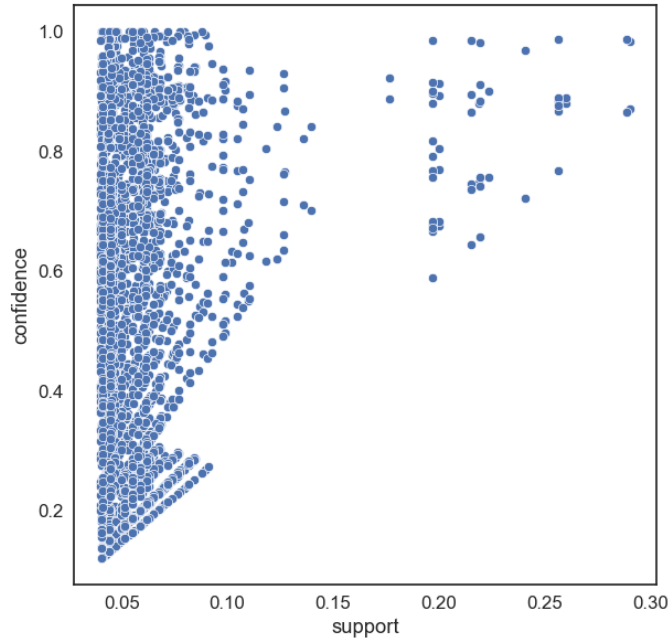


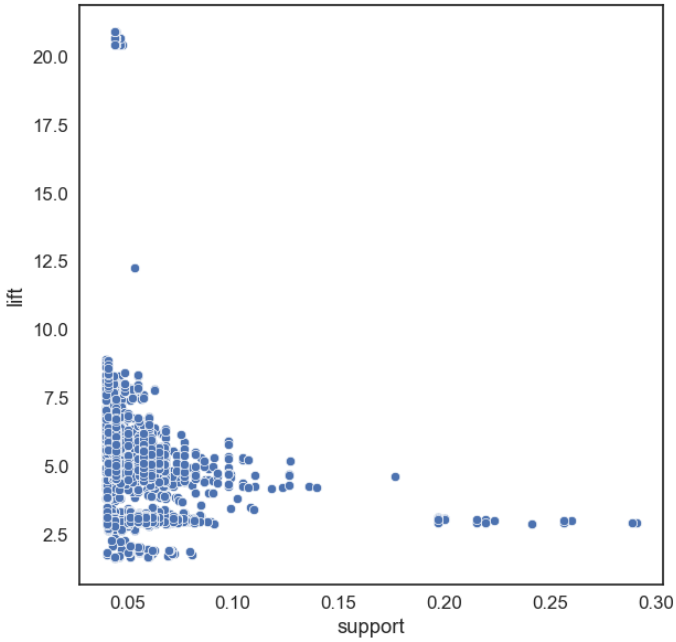
In [360…

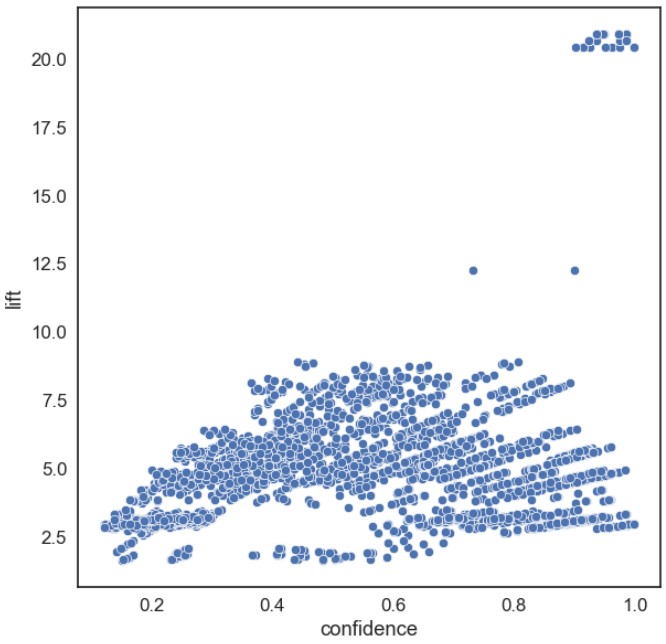
```
#Setting up the style
plt.figure(figsize = (15, 15))
plt.style.use('seaborn-white')
#Plotting the relationship between the metrics
plt.subplot(221)
sns.scatterplot(x="support", y="confidence",data=c_rules_retail)
plt.subplot(222)
sns.scatterplot(x="support", y="lift",data=c_rules_retail)
plt.subplot(223)
sns.scatterplot(x="confidence", y="lift",data=c_rules_retail)
plt.subplot(224)
sns.scatterplot(x="antecedent support", y="consequent support",data=c_rules_retail)
```

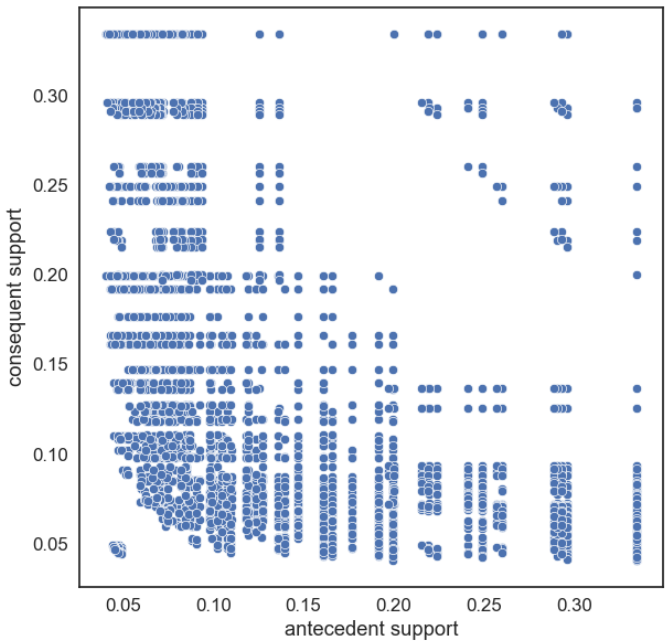
Out[360]:

```
<Axes: xlabel='antecedent support', ylabel='consequent support'>
```



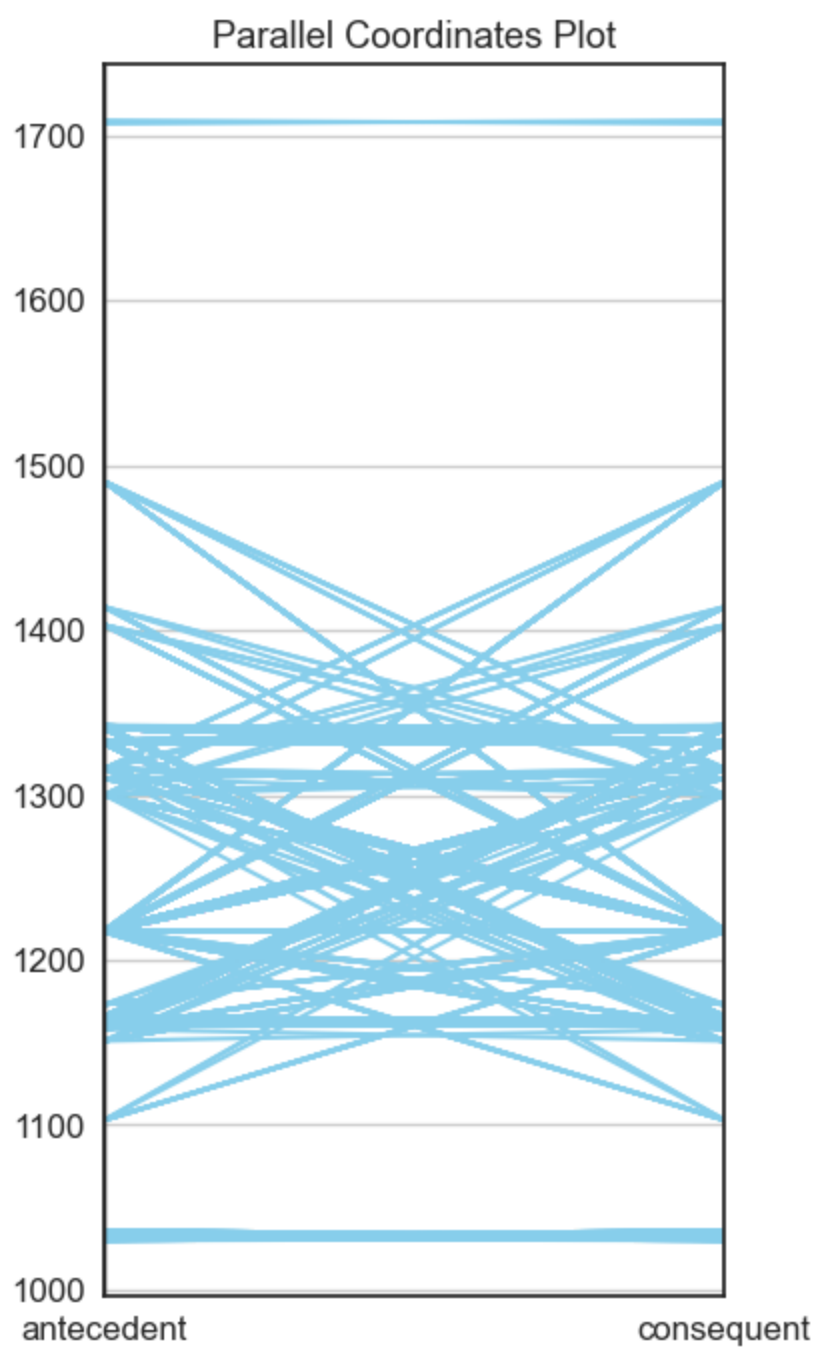






In [361…

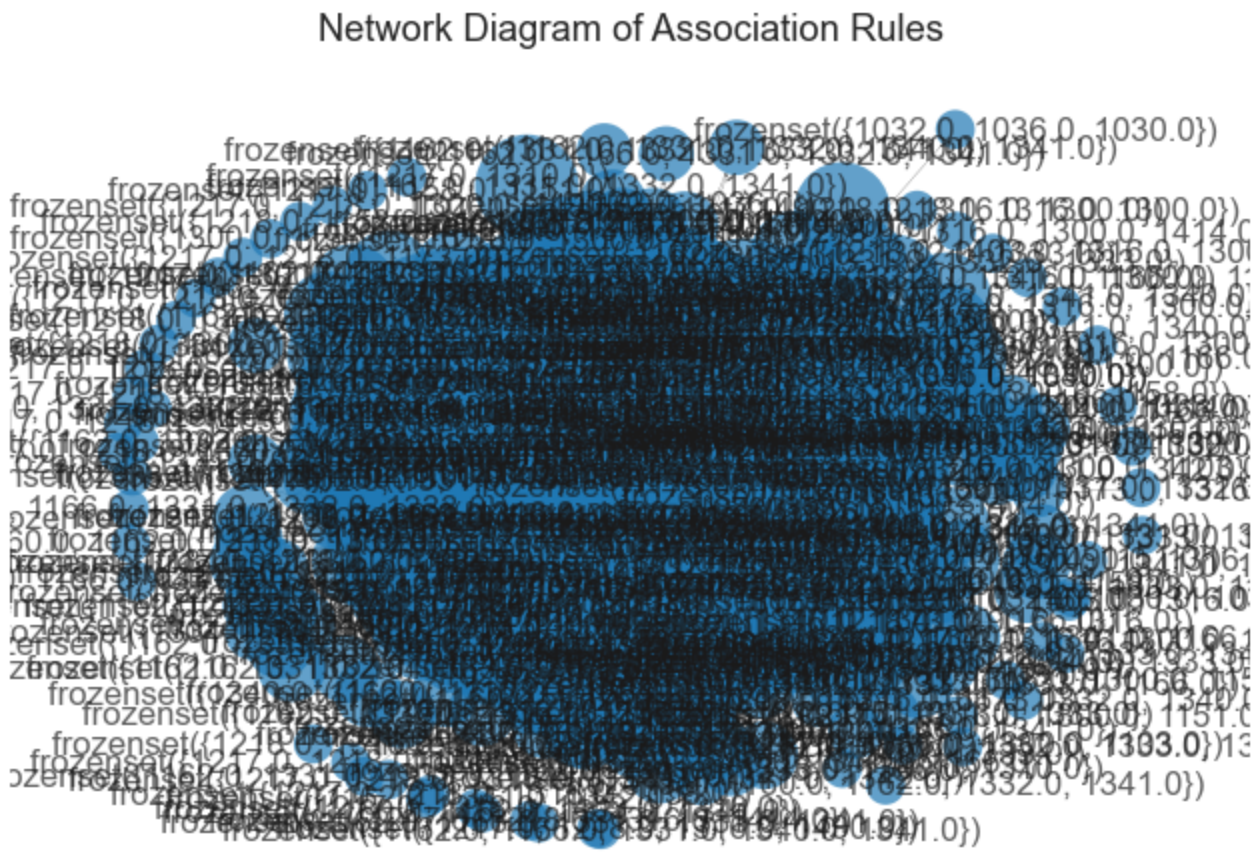
```
### Parallel coordinates plot
#The parallel coordinates plot will allow us to visualize whether a relationship exist b

# Function to convert rules to coordinates.
def rules_to_coordinates(c_rules):
    c_rules_retail['antecedent'] = c_rules_retail['antecedents'].apply(lambda antecedent
    c_rules_retail['consequent'] = c_rules_retail['consequents'].apply(lambda consequent
    c_rules_retail['rule'] = c_rules.index
    return c_rules_retail[['antecedent','consequent','rule']]

from pandas.plotting import parallel_coordinates

# Convert rules into coordinates suitable for use in a parallel coordinates plot
coords = rules_to_coordinates(c_rules_retail)

# Generate parallel coordinates plot
plt.figure(figsize=(4,8))
parallel_coordinates(coords, 'rule', color='skyblue')
plt.legend([])
plt.grid(True)
plt.title('Parallel Coordinates Plot')
plt.show()
```



In [362…

```
import plotly.graph_objects as go
from plotly.offline import download_plotlyjs, init_notebook_mode, iplot
import networkx as nx

bk_network = c_rules_retail[['antecedents', 'consequents']]

bk_network_G = nx.from_pandas_edgelist(
    bk_network,
    source = 'antecedents',
    target = 'consequents',
    create_using = nx.DiGraph())

network_bt = nx.in_degree_centrality(bk_network_G)
bk_network_bt = pd.DataFrame(list(network_bt.items()), columns = ['item', 'centrality'])
bk_network_bt = bk_network_bt.dropna(axis=0)

pos = nx.kamada_kawai_layout(bk_network_G)

sizes = [x[1]*100 for x in bk_network_G.degree()]

nx.draw_networkx(bk_network_G, pos,
    with_labels = True,
```

```
    node_size = sizes,
    width = 0.1, alpha = 0.7,
    arrowsize = 2, linewidths = 0)

plt.axis('off')
plt.title('Network Diagram of Association Rules')
plt.show()
```



Market Basket Analysis for Ready to Cook (RTC) Category

Market Basket Analysis

Market Basket Analysis uses measures like support, confidence, and lift to identify association rules. The threshold values for these measures are typically set based on the specific goals of the analysis, the nature of the data, and the domain of the problem. There's no one-size-fits-all answer, but we decided the following combination of threshold of support, lift and confidence based on our desktop research.

Support Threshold: This represents the frequency of items or itemsets in the dataset.

A low support threshold will generate a large number of rules, but many might not be practically significant. A high support threshold may miss out on potentially interesting item combinations that don't occur frequently. For large datasets, a typical starting threshold might be around 1% to 5%. For smaller datasets, you might consider higher values, such as 10% or more. Confidence Threshold: This measure indicates the likelihood that an item Y is purchased when item X is purchased.

Confidence thresholds are often set in the range of 50% to 70% or higher, depending on the specific application. A higher confidence threshold means that only strong rules, where the consequent is highly likely given the antecedent, will be considered. This can help filter out weaker associations. Lift Threshold: Lift indicates the strength of a rule over the random occurrence of X and Y together.

A lift threshold of 1 is often used to consider associations that are neutral (not more or less likely to occur than if the items were independent). Values greater than 1 indicate that the antecedent and consequent are positively associated, while values less than 1 indicate a negative association.

In [363…
```
rtc_df = df[df['Inventory Category'] == 'Ready to Cook (RTC)']
```

In [364…
```
unique_sku_count_rtc = rtc_df['Inventory Code'].nunique()
print("Number of unique SKUs in RTC Category:", unique_sku_count_rtc)
```

```
Number of unique SKUs in RTC Category: 111
```

In [365…
```
# Create a binary-encoded DataFrame
encoded_data_rtc = pd.get_dummies(rtc_df['Inventory Code']).groupby(rtc_df['Sales Order
encoded_data_rtc
```

Out[365]:

Sales Order No.	1010.0	1011.0	1013.0	1014.0	1015.0	1060.0	1061.0	1062.0	1063.0	1064.0	...	1602.0	1603.
SO00000001	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000002	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000003	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000004	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00000005	False	False	False	False	False	False	False	False	False	False	...	False	Fals
...	...	...	...	...	...	...	...	...	...	...	...	...	
SO00030679	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030681	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030683	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030684	False	False	False	False	False	False	False	False	False	False	...	False	Fals
SO00030685	False	False	False	False	False	False	False	False	False	False	...	False	Fals

27175 rows × 111 columns

In [366…
```
# Calculate support for all item combinations
support = encoded_data_rtc.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_rtc.astype(bool), min_support=0.04, use_colname
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_rtc = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_rtc.sort_values('support', ascending = False)
c_rules_rtc.head(20)
```

Out[366]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
0	(1145.0)	(1142.0)	0.080589	0.178178	0.041950	0.520548	2.921497	0.027591	1.714085	
1	(1142.0)	(1145.0)	0.178178	0.080589	0.041950	0.235440	2.921497	0.027591	1.202536	

2	(1142.0)	(1151.0)	0.178178	0.181417	0.054167	0.304007	1.675736	0.021843	1.176137
3	(1151.0)	(1142.0)	0.181417	0.178178	0.054167	0.298580	1.675736	0.021843	1.171654
4	(1154.0)	(1142.0)	0.085483	0.178178	0.040037	0.468360	2.628600	0.024806	1.545823
5	(1142.0)	(1154.0)	0.178178	0.085483	0.040037	0.224701	2.628600	0.024806	1.179566
6	(1142.0)	(1158.0)	0.178178	0.177774	0.048022	0.269517	1.516066	0.016347	1.125592
7	(1158.0)	(1142.0)	0.177774	0.178178	0.048022	0.270130	1.516066	0.016347	1.125984
8	(1177.0)	(1142.0)	0.092291	0.178178	0.049163	0.532695	2.989673	0.032719	1.758642
9	(1142.0)	(1177.0)	0.178178	0.092291	0.049163	0.275919	2.989673	0.032719	1.253602
10	(1158.0)	(1151.0)	0.177774	0.181417	0.047580	0.267646	1.475313	0.015329	1.117743
11	(1151.0)	(1158.0)	0.181417	0.177774	0.047580	0.262272	1.475313	0.015329	1.114538
12	(1151.0)	(1191.0)	0.181417	0.108519	0.042907	0.236511	2.179447	0.023220	1.167641
13	(1191.0)	(1151.0)	0.108519	0.181417	0.042907	0.395388	2.179447	0.023220	1.353899
14	(1176.0)	(1177.0)	0.069844	0.092291	0.047323	0.677555	7.341533	0.040877	2.815085
15	(1177.0)	(1176.0)	0.092291	0.069844	0.047323	0.512759	7.341533	0.040877	1.909028
16	(1707.0)	(1709.0)	0.064361	0.063441	0.050304	0.781589	12.320008	0.046221	4.288069
17	(1709.0)	(1707.0)	0.063441	0.064361	0.050304	0.792923	12.320008	0.046221	4.518326
18	(1145.0)	(1142.0)	0.080589	0.178178	0.041950	0.520548	2.921497	0.027591	1.714085
19	(1177.0)	(1142.0)	0.092291	0.178178	0.049163	0.532695	2.989673	0.032719	1.758642

In [367…

```
# For the mapping of antecedents and consequents
sku_to_category = rtc_df.set_index('Inventory Code')['Inventory Category'].to_dict()

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_rtc['Antecedent Product Category'] = c_rules_rtc['antecedents'].apply(lambda x:
c_rules_rtc['Consequent Product Category'] = c_rules_rtc['consequents'].apply(lambda x:
```

In [368…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_rtc = c_rules_rtc.drop_duplicates(subset=['antecedents', 'consequents'])

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_rtc = c_rules_rtc.sort_values(by='support', ascending=False)

# Select the top 10 cross-sell/up-sell opportunities
top_opportunities_rtc = sorted_rules_rtc.head(20)

# Display the top 10 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for RTC:")
top_opportunities_rtc
```

```
Top Cross-Sell/Up-Sell Opportunities for RTC:
```

Out[368]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
2	(1142.0)	(1151.0)	0.178178	0.181417	0.054167	0.304007	1.675736	0.021843	1.176137	
3	(1151.0)	(1142.0)	0.181417	0.178178	0.054167	0.298580	1.675736	0.021843	1.171654	
17	(1709.0)	(1707.0)	0.063441	0.064361	0.050304	0.792923	12.320008	0.046221	4.518326	

16	(1707.0)	(1709.0)	0.064361	0.063441	0.050304	0.781589	12.320008	0.046221	4.288069
8	(1177.0)	(1142.0)	0.092291	0.178178	0.049163	0.532695	2.989673	0.032719	1.758642
9	(1142.0)	(1177.0)	0.178178	0.092291	0.049163	0.275919	2.989673	0.032719	1.253602
6	(1142.0)	(1158.0)	0.178178	0.177774	0.048022	0.269517	1.516066	0.016347	1.125592
7	(1158.0)	(1142.0)	0.177774	0.178178	0.048022	0.270130	1.516066	0.016347	1.125984
10	(1158.0)	(1151.0)	0.177774	0.181417	0.047580	0.267646	1.475313	0.015329	1.117743
11	(1151.0)	(1158.0)	0.181417	0.177774	0.047580	0.262272	1.475313	0.015329	1.114538
14	(1176.0)	(1177.0)	0.069844	0.092291	0.047323	0.677555	7.341533	0.040877	2.815085
15	(1177.0)	(1176.0)	0.092291	0.069844	0.047323	0.512759	7.341533	0.040877	1.909028
12	(1151.0)	(1191.0)	0.181417	0.108519	0.042907	0.236511	2.179447	0.023220	1.167641
13	(1191.0)	(1151.0)	0.108519	0.181417	0.042907	0.395388	2.179447	0.023220	1.353899
1	(1142.0)	(1145.0)	0.178178	0.080589	0.041950	0.235440	2.921497	0.027591	1.202536
0	(1145.0)	(1142.0)	0.080589	0.178178	0.041950	0.520548	2.921497	0.027591	1.714085
5	(1142.0)	(1154.0)	0.178178	0.085483	0.040037	0.224701	2.628600	0.024806	1.179566
4	(1154.0)	(1142.0)	0.085483	0.178178	0.040037	0.468360	2.628600	0.024806	1.545823

Market Basket Analysis for Raw Category

In [369…
```
raw_df = df[df['Inventory Category'] == 'Raw']
```

In [370…
```
unique_sku_count_raw = raw_df['Inventory Code'].nunique()
print("Number of unique SKUs in Raw Category:", unique_sku_count_raw)
```

```
Number of unique SKUs in Raw Category: 9
```

In [371…
```
# Create a binary-encoded DataFrame
encoded_data_raw = pd.get_dummies(raw_df['Inventory Code']).groupby(raw_df['Sales Order
encoded_data_raw
```

Out[371]:

Sales Order No.	1030.0	1031.0	1032.0	1033.0	1034.0	1035.0	1036.0	1037.0	1152.0
SO00000089	False	False	False	True	False	False	False	False	False
SO00000108	True	True	True	True	True	True	True	True	False
SO00000118	True	False	False	False	True	False	False	True	False

SO00000120	False	False	False	True	False	False	False	False	False
SO00000123	False	False	False	True	False	False	True	False	False
...	...	...	...	...	...	...	...	...	...
SO00030581	True	True	True	True	True	True	False	False	False
SO00030597	True	True	True	False	False	False	True	False	False
SO00030604	False	False	False	False	False	False	False	False	True
SO00030659	True	False	False	True	False	False	True	True	False
SO00030677	True	True	True	True	True	True	True	True	False

1138 rows × 9 columns

In [372…

```
# Calculate support for all item combinations
support = encoded_data_raw.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_raw.astype(bool), min_support=0.03, use_colname
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_raw = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_raw.sort_values('support', ascending = False)
c_rules_raw.head(20)
```

Out[372]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zha
0	(1030.0)	(1031.0)	0.783831	0.385764	0.376977	0.480942	1.246724	0.074603	1.183365	
1	(1031.0)	(1030.0)	0.385764	0.783831	0.376977	0.977221	1.246724	0.074603	9.489807	
2	(1032.0)	(1030.0)	0.485940	0.783831	0.449033	0.924051	1.178890	0.068138	2.846221	
3	(1030.0)	(1032.0)	0.783831	0.485940	0.449033	0.572870	1.178890	0.068138	1.203520	
4	(1033.0)	(1030.0)	0.500879	0.783831	0.404218	0.807018	1.029581	0.011614	1.120147	
5	(1030.0)	(1033.0)	0.783831	0.500879	0.404218	0.515695	1.029581	0.011614	1.030593	
6	(1034.0)	(1030.0)	0.388401	0.783831	0.368190	0.947964	1.209398	0.063749	4.154199	
7	(1030.0)	(1034.0)	0.783831	0.388401	0.368190	0.469731	1.209398	0.063749	1.153375	
8	(1035.0)	(1030.0)	0.338313	0.783831	0.328647	0.971429	1.239334	0.063467	7.565905	
9	(1030.0)	(1035.0)	0.783831	0.338313	0.328647	0.419283	1.239334	0.063467	1.139430	
10	(1037.0)	(1030.0)	0.411248	0.783831	0.356766	0.867521	1.106771	0.034417	1.631725	
11	(1030.0)	(1037.0)	0.783831	0.411248	0.356766	0.455157	1.106771	0.034417	1.080590	
12	(1032.0)	(1031.0)	0.485940	0.385764	0.376098	0.773960	2.006302	0.188640	2.717378	
13	(1031.0)	(1032.0)	0.385764	0.485940	0.376098	0.974943	2.006302	0.188640	20.515657	
14	(1033.0)	(1031.0)	0.500879	0.385764	0.306678	0.612281	1.587188	0.113457	1.584227	
15	(1031.0)	(1033.0)	0.385764	0.500879	0.306678	0.794989	1.587188	0.113457	2.434603	
16	(1034.0)	(1031.0)	0.388401	0.385764	0.306678	0.789593	2.046826	0.156847	2.919270	

17	(1031.0)	(1034.0)	0.385764	0.388401	0.306678	0.794989	2.046826	0.156847	2.983245
18	(1035.0)	(1031.0)	0.338313	0.385764	0.301406	0.890909	2.309464	0.170897	5.630492
19	(1031.0)	(1035.0)	0.385764	0.338313	0.301406	0.781321	2.309464	0.170897	3.025840

In [373…

```
# For the mapping of antecedents and consequents
sku_to_category = df.set_index('Inventory Code')['Inventory Category'].to_dict()

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_raw['Antecedent Product Category'] = c_rules_raw['antecedents'].apply(lambda x:
c_rules_raw['Consequent Product Category'] = c_rules_raw['consequents'].apply(lambda x:
```

In [374…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_raw = c_rules_raw.drop_duplicates(subset=['antecedents', 'consequents'])

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_raw = c_rules_raw.sort_values(by='lift', ascending=False)

# Select the top 10 cross-sell/up-sell opportunities
top_opportunities_raw = sorted_rules_raw.head(20)

# Display the top 10 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for Raw Category:")
top_opportunities_raw
```

```
Top Cross-Sell/Up-Sell Opportunities for Raw Category:
```

Out[374]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	z
5854	(1030.0, 1032.0, 1033.0, 1035.0, 1037.0)	(1034.0, 1036.0, 1031.0)	0.266257	0.257469	0.253076	0.950495	3.691684	0.184523	14.999121	
5987	(1034.0, 1036.0, 1031.0)	(1030.0, 1032.0, 1033.0, 1035.0, 1037.0)	0.257469	0.266257	0.253076	0.982935	3.691684	0.184523	42.997364	
5720	(1032.0, 1033.0, 1035.0, 1037.0)	(1034.0, 1036.0, 1031.0)	0.268014	0.257469	0.253076	0.944262	3.667476	0.184070	13.321875	
5945	(1032.0, 1033.0, 1035.0, 1037.0)	(1034.0, 1036.0, 1030.0, 1031.0)	0.268014	0.257469	0.253076	0.944262	3.667476	0.184070	13.321875	
5741	(1034.0, 1036.0, 1031.0)	(1032.0, 1033.0, 1035.0, 1037.0)	0.257469	0.268014	0.253076	0.982935	3.667476	0.184070	42.894376	
5896	(1034.0, 1036.0, 1030.0, 1031.0)	(1032.0, 1033.0, 1035.0, 1037.0)	0.257469	0.268014	0.253076	0.982935	3.667476	0.184070	42.894376	
5926	(1032.0, 1034.0,	(1033.0, 1035.0,	0.255712	0.270650	0.253076	0.989691	3.656714	0.183867	70.746924	

	1036.0, 1031.0)	1037.0, 1030.0)							
5915	(1033.0, 1035.0, 1037.0, 1030.0)	(1032.0, 1034.0, 1036.0, 1031.0)	0.270650	0.255712	0.253076	0.935065	3.656714	0.183867	11.462039
5908	(1032.0, 1035.0, 1036.0, 1030.0)	(1033.0, 1034.0, 1037.0, 1031.0)	0.268014	0.258348	0.253076	0.944262	3.655002	0.183835	13.306110
5933	(1033.0, 1034.0, 1037.0, 1031.0)	(1032.0, 1035.0, 1036.0, 1030.0)	0.258348	0.268014	0.253076	0.979592	3.655002	0.183835	35.867311
5985	(1033.0, 1037.0, 1031.0)	(1030.0, 1032.0, 1034.0, 1035.0, 1036.0)	0.260105	0.266257	0.253076	0.972973	3.654268	0.183821	27.148506
5856	(1030.0, 1032.0, 1034.0, 1035.0, 1036.0)	(1033.0, 1037.0, 1031.0)	0.266257	0.260105	0.253076	0.950495	3.654268	0.183821	14.945870
5499	(1034.0, 1036.0, 1031.0)	(1033.0, 1035.0, 1037.0, 1030.0)	0.257469	0.270650	0.253954	0.986348	3.644364	0.184270	53.424868
5458	(1033.0, 1035.0, 1037.0, 1030.0)	(1034.0, 1036.0, 1031.0)	0.270650	0.257469	0.253954	0.938312	3.644364	0.184270	12.036814
5989	(1035.0, 1036.0, 1031.0)	(1030.0, 1032.0, 1033.0, 1034.0, 1037.0)	0.256591	0.270650	0.253076	0.986301	3.644191	0.183629	53.242531
5852	(1030.0, 1032.0, 1033.0, 1034.0, 1037.0)	(1035.0, 1036.0, 1031.0)	0.270650	0.256591	0.253076	0.935065	3.644191	0.183629	11.448506
5205	(1032.0, 1035.0, 1036.0, 1030.0)	(1033.0, 1037.0, 1031.0)	0.268014	0.260105	0.253954	0.947541	3.642911	0.184242	14.104240
5248	(1033.0, 1037.0, 1031.0)	(1032.0, 1035.0, 1036.0, 1030.0)	0.260105	0.268014	0.253954	0.976351	3.642911	0.184242	30.952548
5947	(1032.0, 1034.0, 1035.0, 1036.0)	(1033.0, 1037.0, 1030.0, 1031.0)	0.268014	0.259227	0.253076	0.944262	3.642612	0.183599	13.290344
5894	(1033.0, 1037.0,	(1032.0, 1034.0,	0.259227	0.268014	0.253076	0.976271	3.642612	0.183599	30.847979

1030.0, 1031.0)

1035.0, 1036.0)

Market Basket Analysis for Ready to Eat (RTE) Category

In [375…
```
rte_df = df[df['Inventory Category'] == 'Ready to Eat (RTE)']
```

In [376…
```
unique_sku_count_rte = rte_df['Inventory Code'].nunique()
print("Number of unique SKUs in RTE Category:", unique_sku_count_rte)
```

```
Number of unique SKUs in RTE Category: 5
```

In [377…
```
# Create a binary-encoded DataFrame
encoded_data_rte = pd.get_dummies(rte_df['Inventory Code']).groupby(rte_df['Sales Order

# Calculate support for all item combinations
support = encoded_data_rte.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_rte.astype(bool), min_support=0.04, use_colname
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_rte = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_rte.sort_values('support', ascending = False)
c_rules_rte.head(20)
```

Out[377]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zha
0	(1330.0)	(1331.0)	0.260459	0.869096	0.256410	0.984456	1.132736	0.030047	8.421502	
1	(1331.0)	(1330.0)	0.869096	0.260459	0.256410	0.295031	1.132736	0.030047	1.049041	
2	(1330.0)	(1332.0)	0.260459	0.843455	0.251012	0.963731	1.142599	0.031327	4.316175	
3	(1332.0)	(1330.0)	0.843455	0.260459	0.251012	0.297600	1.142599	0.031327	1.052878	
4	(1330.0)	(1333.0)	0.260459	0.570850	0.232119	0.891192	1.561166	0.083436	3.944091	
5	(1333.0)	(1330.0)	0.570850	0.260459	0.232119	0.406619	1.561166	0.083436	1.246318	
6	(1331.0)	(1332.0)	0.869096	0.843455	0.759784	0.874224	1.036480	0.026741	1.244631	
7	(1332.0)	(1331.0)	0.843455	0.869096	0.759784	0.900800	1.036480	0.026741	1.319599	
8	(1332.0)	(1333.0)	0.843455	0.570850	0.510121	0.604800	1.059472	0.028635	1.085905	
9	(1333.0)	(1332.0)	0.570850	0.843455	0.510121	0.893617	1.059472	0.028635	1.471525	
10	(1330.0, 1331.0)	(1332.0)	0.256410	0.843455	0.251012	0.978947	1.160640	0.034742	7.435897	
11	(1330.0, 1332.0)	(1331.0)	0.251012	0.869096	0.251012	1.000000	1.150621	0.032859	inf	
12	(1331.0, 1332.0)	(1330.0)	0.759784	0.260459	0.251012	0.330373	1.268427	0.053120	1.104408	
13	(1330.0)	(1331.0, 1332.0)	0.260459	0.759784	0.251012	0.963731	1.268427	0.053120	6.623096	
14	(1331.0)	(1330.0,	0.869096	0.251012	0.251012	0.288820	1.150621	0.032859	1.053162	

		1332.0)							
15	(1332.0)	(1330.0, 1331.0)	0.843455	0.256410	0.251012	0.297600	1.160640	0.034742	1.058641
16	(1330.0, 1331.0)	(1333.0)	0.256410	0.570850	0.230769	0.900000	1.576596	0.084397	4.291498
17	(1330.0, 1333.0)	(1331.0)	0.232119	0.869096	0.230769	0.994186	1.143931	0.029036	22.515520
18	(1331.0, 1333.0)	(1330.0)	0.481781	0.260459	0.230769	0.478992	1.839030	0.105285	1.419442
19	(1330.0)	(1331.0, 1333.0)	0.260459	0.481781	0.230769	0.886010	1.839030	0.105285	4.546191

In [378…

```
# For the mapping of antecedents and consequents
sku_to_category = rte_df.set_index('Inventory Code')['Inventory Category'].to_dict()

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_rte['Antecedent Product Category'] = c_rules_rte['antecedents'].apply(lambda x:
c_rules_rte['Consequent Product Category'] = c_rules_rte['consequents'].apply(lambda x:
```

In [379…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_rte = c_rules_rte.drop_duplicates(subset=['antecedents', 'consequents'])

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_rte = c_rules_rte.sort_values(by='support', ascending=False)

# Select the top 10 cross-sell/up-sell opportunities
top_opportunities_rte = sorted_rules_rte.head(20)

# Display the top 10 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for RTE Category:")
top_opportunities_rte
```

```
Top Cross-Sell/Up-Sell Opportunities for RTE Category:
```

Out[379]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
6	(1331.0)	(1332.0)	0.869096	0.843455	0.759784	0.874224	1.036480	0.026741	1.244631	
7	(1332.0)	(1331.0)	0.843455	0.869096	0.759784	0.900800	1.036480	0.026741	1.319599	
8	(1332.0)	(1333.0)	0.843455	0.570850	0.510121	0.604800	1.059472	0.028635	1.085905	
9	(1333.0)	(1332.0)	0.570850	0.843455	0.510121	0.893617	1.059472	0.028635	1.471525	
54	(1333.0)	(1331.0)	0.570850	0.869096	0.481781	0.843972	0.971092	-0.014342	0.838977	
53	(1331.0)	(1333.0)	0.869096	0.570850	0.481781	0.554348	0.971092	-0.014342	0.962970	
33	(1333.0)	(1331.0, 1332.0)	0.570850	0.759784	0.465587	0.815603	1.073467	0.031864	1.302709	
32	(1332.0)	(1331.0, 1333.0)	0.843455	0.481781	0.465587	0.552000	1.145748	0.059226	1.156738	

31	(1331.0)	(1332.0, 1333.0)	0.869096	0.510121	0.465587	0.535714	1.050170	0.022243	1.055123
30	(1332.0, 1333.0)	(1331.0)	0.510121	0.869096	0.465587	0.912698	1.050170	0.022243	1.499448
29	(1331.0, 1333.0)	(1332.0)	0.481781	0.843455	0.465587	0.966387	1.145748	0.059226	4.657220
28	(1331.0, 1332.0)	(1333.0)	0.759784	0.570850	0.465587	0.612789	1.073467	0.031864	1.108309
1	(1331.0)	(1330.0)	0.869096	0.260459	0.256410	0.295031	1.132736	0.030047	1.049041
0	(1330.0)	(1331.0)	0.260459	0.869096	0.256410	0.984456	1.132736	0.030047	8.421502
14	(1331.0)	(1330.0, 1332.0)	0.869096	0.251012	0.251012	0.288820	1.150621	0.032859	1.053162
13	(1330.0)	(1331.0, 1332.0)	0.260459	0.759784	0.251012	0.963731	1.268427	0.053120	6.623096
2	(1330.0)	(1332.0)	0.260459	0.843455	0.251012	0.963731	1.142599	0.031327	4.316175
3	(1332.0)	(1330.0)	0.843455	0.260459	0.251012	0.297600	1.142599	0.031327	1.052878
15	(1332.0)	(1330.0, 1331.0)	0.843455	0.256410	0.251012	0.297600	1.160640	0.034742	1.058641
10	(1330.0, 1331.0)	(1332.0)	0.256410	0.843455	0.251012	0.978947	1.160640	0.034742	7.435897

Market Basket Analysis for Champions Customer Segment

In [380…
```
champ_df = df[df['Segment'] == 'Champions']
```

In [381…
```
unique_sku_count_champ = champ_df['Inventory Code'].nunique()
print("Number of unique SKUs for Champions Cystomer Segment:", unique_sku_count_champ)
```

```
Number of unique SKUs for Champions Cystomer Segment: 119
```

In [382…

```
# Create a binary-encoded DataFrame
encoded_data_champ = pd.get_dummies(champ_df['Inventory Code']).groupby(champ_df['Sales

# Calculate support for all item combinations
support = encoded_data_champ.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_champ.astype(bool), min_support=0.05, use_colna
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_champ = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_champ.sort_values('support', ascending = False)
c_rules_champ.head(20)
```

Out[382]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
0	(1142.0)	(1151.0)	0.147243	0.159962	0.061761	0.419446	2.622155	0.038207	1.446958	
1	(1151.0)	(1142.0)	0.159962	0.147243	0.061761	0.386095	2.622155	0.038207	1.389069	
2	(1142.0)	(1158.0)	0.147243	0.218883	0.059453	0.403777	1.844713	0.027224	1.310108	
3	(1158.0)	(1142.0)	0.218883	0.147243	0.059453	0.271622	1.844713	0.027224	1.170761	
4	(1176.0)	(1142.0)	0.080868	0.147243	0.053656	0.663497	4.506126	0.041749	2.534171	
5	(1142.0)	(1176.0)	0.147243	0.080868	0.053656	0.364403	4.506126	0.041749	1.446093	
6	(1177.0)	(1142.0)	0.099444	0.147243	0.060637	0.609756	4.141148	0.045994	2.185189	
7	(1142.0)	(1177.0)	0.147243	0.099444	0.060637	0.411812	4.141148	0.045994	1.531068	
8	(1158.0)	(1151.0)	0.218883	0.159962	0.057205	0.261351	1.633833	0.022192	1.137263	
9	(1151.0)	(1158.0)	0.159962	0.218883	0.057205	0.357618	1.633833	0.022192	1.215970	
10	(1151.0)	(1191.0)	0.159962	0.125473	0.055017	0.343935	2.741101	0.034946	1.332988	
11	(1191.0)	(1151.0)	0.125473	0.159962	0.055017	0.438472	2.741101	0.034946	1.495987	
12	(1176.0)	(1177.0)	0.080868	0.099444	0.062234	0.769568	7.738718	0.054192	3.908128	
13	(1177.0)	(1176.0)	0.099444	0.080868	0.062234	0.625818	7.738718	0.054192	2.456375	
14	(1707.0)	(1709.0)	0.083885	0.088855	0.075071	0.894922	10.071750	0.067617	8.671168	
15	(1709.0)	(1707.0)	0.088855	0.083885	0.075071	0.844874	10.071750	0.067617	5.905597	
16	(1176.0)	(1177.0)	0.080868	0.099444	0.062234	0.769568	7.738718	0.054192	3.908128	
17	(1707.0)	(1709.0)	0.083885	0.088855	0.075071	0.894922	10.071750	0.067617	8.671168	
18	(1709.0)	(1707.0)	0.088855	0.083885	0.075071	0.844874	10.071750	0.067617	5.905597	

In [383…

```
# For the mapping of antecedents and consequents
sku_to_category = champ_df.set_index('Inventory Code')['Inventory Category'].to_dict()

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_champ['Antecedent Product Category'] = c_rules_champ['antecedents'].apply(lambda
c_rules_champ['Consequent Product Category'] = c_rules_champ['consequents'].apply(lambda
```

In [384…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_champ = c_rules_champ.drop_duplicates(subset=['antecedents', 'consequents'])
```

```
# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_champ = c_rules_champ.sort_values(by='lift', ascending=False)

# Select the top 20 cross-sell/up-sell opportunities
top_opportunities_champ = sorted_rules_champ.head(20)

# Display the top 20 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for Champions Customer Segment:")
top_opportunities_champ
```

```
Top Cross-Sell/Up-Sell Opportunities for Champions Customer Segment:
```

Out[384]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
14	(1707.0)	(1709.0)	0.083885	0.088855	0.075071	0.894922	10.071750	0.067617	8.671168	
15	(1709.0)	(1707.0)	0.088855	0.083885	0.075071	0.844874	10.071750	0.067617	5.905597	
12	(1176.0)	(1177.0)	0.080868	0.099444	0.062234	0.769568	7.738718	0.054192	3.908128	
13	(1177.0)	(1176.0)	0.099444	0.080868	0.062234	0.625818	7.738718	0.054192	2.456375	
4	(1176.0)	(1142.0)	0.080868	0.147243	0.053656	0.663497	4.506126	0.041749	2.534171	
5	(1142.0)	(1176.0)	0.147243	0.080868	0.053656	0.364403	4.506126	0.041749	1.446093	
6	(1177.0)	(1142.0)	0.099444	0.147243	0.060637	0.609756	4.141148	0.045994	2.185189	
7	(1142.0)	(1177.0)	0.147243	0.099444	0.060637	0.411812	4.141148	0.045994	1.531068	
10	(1151.0)	(1191.0)	0.159962	0.125473	0.055017	0.343935	2.741101	0.034946	1.332988	
11	(1191.0)	(1151.0)	0.125473	0.159962	0.055017	0.438472	2.741101	0.034946	1.495987	
1	(1151.0)	(1142.0)	0.159962	0.147243	0.061761	0.386095	2.622155	0.038207	1.389069	
0	(1142.0)	(1151.0)	0.147243	0.159962	0.061761	0.419446	2.622155	0.038207	1.446958	
2	(1142.0)	(1158.0)	0.147243	0.218883	0.059453	0.403777	1.844713	0.027224	1.310108	
3	(1158.0)	(1142.0)	0.218883	0.147243	0.059453	0.271622	1.844713	0.027224	1.170761	
9	(1151.0)	(1158.0)	0.159962	0.218883	0.057205	0.357618	1.633833	0.022192	1.215970	
8	(1158.0)	(1151.0)	0.218883	0.159962	0.057205	0.261351	1.633833	0.022192	1.137263	

Market Basket Analysis for Loyal Customers

In [385…

```
loyal_df = df[df['Segment'] == 'Loyal']
```

In [386…
```
unique_sku_count_loyal = loyal_df['Inventory Code'].nunique()
print("Number of unique SKUs for Loyal Customers Cystomer Segment:", unique_sku_count_lo
```

```
Number of unique SKUs for Loyal Customers Cystomer Segment: 86
```

In [387…
```
# Create a binary-encoded DataFrame
encoded_data_loyal = pd.get_dummies(loyal_df['Inventory Code']).groupby(loyal_df['Sales

# Calculate support for all item combinations
support = encoded_data_loyal.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_loyal.astype(bool), min_support=0.05, use_colna
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_loyal = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_loyal.sort_values('support', ascending = False)
c_rules_loyal.head(20)
```

Out[387]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zha
0	(1142.0)	(1126.0)	0.303810	0.173016	0.077965	0.256626	1.483244	0.025401	1.112473	
1	(1126.0)	(1142.0)	0.173016	0.303810	0.077965	0.450624	1.483244	0.025401	1.267239	
2	(1144.0)	(1126.0)	0.105067	0.173016	0.050275	0.478505	2.765659	0.032097	1.585793	
3	(1126.0)	(1144.0)	0.173016	0.105067	0.050275	0.290579	2.765659	0.032097	1.261498	
4	(1145.0)	(1126.0)	0.148861	0.173016	0.077376	0.519789	3.004274	0.051621	1.722125	
5	(1126.0)	(1145.0)	0.173016	0.148861	0.077376	0.447219	3.004274	0.051621	1.539740	
6	(1154.0)	(1126.0)	0.178319	0.173016	0.080715	0.452643	2.616185	0.049863	1.510867	
7	(1126.0)	(1154.0)	0.173016	0.178319	0.080715	0.466515	2.616185	0.049863	1.540215	
8	(1141.0)	(1142.0)	0.093873	0.303810	0.060880	0.648536	2.134676	0.032360	1.980827	
9	(1142.0)	(1141.0)	0.303810	0.093873	0.060880	0.200388	2.134676	0.032360	1.133208	
10	(1144.0)	(1141.0)	0.105067	0.093873	0.073056	0.695327	7.407125	0.063193	2.974099	
11	(1141.0)	(1144.0)	0.093873	0.105067	0.073056	0.778243	7.407125	0.063193	4.035642	
12	(1145.0)	(1141.0)	0.148861	0.093873	0.056559	0.379947	4.047471	0.042585	1.461371	
13	(1141.0)	(1145.0)	0.093873	0.148861	0.056559	0.602510	4.047471	0.042585	2.141287	
14	(1154.0)	(1141.0)	0.178319	0.093873	0.074234	0.416300	4.434723	0.057495	1.552384	
15	(1141.0)	(1154.0)	0.093873	0.178319	0.074234	0.790795	4.434723	0.057495	3.927636	
16	(1144.0)	(1142.0)	0.105067	0.303810	0.068539	0.652336	2.147186	0.036619	2.002482	
17	(1142.0)	(1144.0)	0.303810	0.105067	0.068539	0.225598	2.147186	0.036619	1.155644	
18	(1145.0)	(1142.0)	0.148861	0.303810	0.098586	0.662269	2.179880	0.053361	2.061375	
19	(1142.0)	(1145.0)	0.303810	0.148861	0.098586	0.324499	2.179880	0.053361	1.260012	

In [388…
```
# For the mapping of antecedents and consequents
sku_to_category = loyal_df.set_index('Inventory Code')['Inventory Category'].to_dict()
```

```
# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_loyal['Antecedent Product Category'] = c_rules_loyal['antecedents'].apply(lambda
c_rules_loyal['Consequent Product Category'] = c_rules_loyal['consequents'].apply(lambda
```

In [389…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_loyal = c_rules_loyal.drop_duplicates(subset=['antecedents', 'consequents'])

# Sort the rules by a relevant metric, e.g., lift, in descending order
sorted_rules_loyal = c_rules_loyal.sort_values(by='lift', ascending=False)

# Select the top 20 cross-sell/up-sell opportunities
top_opportunities_loyal = sorted_rules_loyal.head(20)

# Display the top 20 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for Loyal Customers Customer Segment:")
top_opportunities_loyal
```

```
Top Cross-Sell/Up-Sell Opportunities for Loyal Customers Customer Segment:
```

Out[389]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
63	(1141.0)	(1144.0, 1145.0)	0.093873	0.057738	0.050668	0.539749	9.348305	0.045248	2.047279	
58	(1144.0, 1145.0)	(1141.0)	0.057738	0.093873	0.050668	0.877551	9.348305	0.045248	7.400039	
64	(1144.0, 1154.0)	(1141.0)	0.066575	0.093873	0.058130	0.873156	9.301490	0.051881	7.143654	
69	(1141.0)	(1144.0, 1154.0)	0.093873	0.066575	0.058130	0.619247	9.301490	0.051881	2.451523	
49	(1144.0)	(1141.0, 1142.0)	0.105067	0.060880	0.054792	0.521495	8.565981	0.048395	1.962614	
48	(1141.0, 1142.0)	(1144.0)	0.060880	0.105067	0.054792	0.900000	8.565981	0.048395	8.949332	
60	(1145.0, 1141.0)	(1144.0)	0.056559	0.105067	0.050668	0.895833	8.526324	0.044725	8.591359	
61	(1144.0)	(1145.0, 1141.0)	0.105067	0.056559	0.050668	0.482243	8.526324	0.044725	1.822169	
47	(1144.0, 1142.0)	(1141.0)	0.068539	0.093873	0.054792	0.799427	8.516071	0.048358	4.517692	

50	(1141.0)	(1144.0, 1142.0)	0.093873	0.068539	0.054792	0.583682	8.516071	0.048358	2.237379
66	(1154.0, 1141.0)	(1144.0)	0.074234	0.105067	0.058130	0.783069	7.453058	0.050331	4.125424
67	(1144.0)	(1154.0, 1141.0)	0.105067	0.074234	0.058130	0.553271	7.453058	0.050331	2.072321
10	(1144.0)	(1141.0)	0.105067	0.093873	0.073056	0.695327	7.407125	0.063193	2.974099
11	(1141.0)	(1144.0)	0.093873	0.105067	0.073056	0.778243	7.407125	0.063193	4.035642
98	(1145.0, 1154.0)	(1126.0, 1142.0)	0.099764	0.077965	0.056756	0.568898	7.296793	0.048978	2.138783
103	(1126.0, 1142.0)	(1145.0, 1154.0)	0.077965	0.099764	0.056756	0.727960	7.296793	0.048978	3.309200
99	(1145.0, 1142.0)	(1154.0, 1126.0)	0.098586	0.080715	0.056756	0.575697	7.132482	0.048798	2.166578
102	(1154.0, 1126.0)	(1145.0, 1142.0)	0.080715	0.098586	0.056756	0.703163	7.132482	0.048798	3.036731
101	(1154.0, 1142.0)	(1145.0, 1126.0)	0.110369	0.077376	0.056756	0.514235	6.645898	0.048216	1.899321
100	(1145.0, 1126.0)	(1154.0, 1142.0)	0.077376	0.110369	0.056756	0.733503	6.645898	0.048216	3.338234

Market Basket Analysis for Hibernating Customer Segment

In [390…
```
hibernating_df = df[df['Segment'] == 'Hibernating']
```

In [391…
```
unique_sku_count_hibernating = hibernating_df['Inventory Code'].nunique()
print("Number of unique SKUs for Hibernating Cystomer Segment:", unique_sku_count_hibern
```

```
Number of unique SKUs for Hibernating Cystomer Segment: 97
```

In [392…
```
# Create a binary-encoded DataFrame
encoded_data_hibernating = pd.get_dummies(hibernating_df['Inventory Code']).groupby(hibe
```

```
# Calculate support for all item combinations
support = encoded_data_hibernating.mean()
# Calculate confidence and lift for item pairs
frequent_itemsets = apriori(encoded_data_hibernating.astype(bool), min_support=0.01, use
# Run association rules for confidence metric
a_rules_confidence = association_rules(frequent_itemsets, metric="confidence", min_thres
# Run association rules for lift metric
a_rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
# Concatenate the two DataFrames
c_rules_hibernating = pd.concat([a_rules_lift, a_rules_confidence], ignore_index=True)

# Display frequent itemsets and association rules
c_rules_hibernating.sort_values('support', ascending = False)
c_rules_hibernating.head(20)
```

Out[392]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
0	(1033.0)	(1036.0)	0.016559	0.021002	0.013732	0.829268	39.485929	0.013384	5.734133	
1	(1036.0)	(1033.0)	0.021002	0.016559	0.013732	0.653846	39.485929	0.013384	2.841052	
2	(1126.0)	(1142.0)	0.084814	0.114297	0.025444	0.300000	2.624735	0.015750	1.265290	
3	(1142.0)	(1126.0)	0.114297	0.084814	0.025444	0.222615	2.624735	0.015750	1.177262	
4	(1145.0)	(1126.0)	0.060178	0.084814	0.026656	0.442953	5.222627	0.021552	1.642924	
5	(1126.0)	(1145.0)	0.084814	0.060178	0.026656	0.314286	5.222627	0.021552	1.370574	
6	(1154.0)	(1126.0)	0.048465	0.084814	0.018982	0.391667	4.617937	0.014872	1.504415	
7	(1126.0)	(1154.0)	0.084814	0.048465	0.018982	0.223810	4.617937	0.014872	1.225904	
8	(1171.0)	(1126.0)	0.041195	0.084814	0.015751	0.382353	4.508123	0.012257	1.481729	
9	(1126.0)	(1171.0)	0.084814	0.041195	0.015751	0.185714	4.508123	0.012257	1.177479	
10	(1145.0)	(1142.0)	0.060178	0.114297	0.014943	0.248322	2.172599	0.008065	1.178301	
11	(1142.0)	(1145.0)	0.114297	0.060178	0.014943	0.130742	2.172599	0.008065	1.081178	
12	(1154.0)	(1142.0)	0.048465	0.114297	0.016963	0.350000	3.062191	0.011423	1.362620	
13	(1142.0)	(1154.0)	0.114297	0.048465	0.016963	0.148410	3.062191	0.011423	1.117362	
14	(1177.0)	(1142.0)	0.090065	0.114297	0.016963	0.188341	1.647816	0.006669	1.091225	
15	(1142.0)	(1177.0)	0.114297	0.090065	0.016963	0.148410	1.647816	0.006669	1.068513	
16	(1145.0)	(1154.0)	0.060178	0.048465	0.014540	0.241611	4.985235	0.011623	1.254679	
17	(1154.0)	(1145.0)	0.048465	0.060178	0.014540	0.300000	4.985235	0.011623	1.342603	
18	(1145.0)	(1171.0)	0.060178	0.041195	0.014540	0.241611	5.864982	0.012061	1.264264	
19	(1171.0)	(1145.0)	0.041195	0.060178	0.014540	0.352941	5.864982	0.012061	1.452453	

In [393…

```
# For the mapping of antecedents and consequents
sku_to_category = hibernating_df.set_index('Inventory Code')['Inventory Category'].to_di

# Map the antecedents and consequents in your market basket analysis DataFrame
c_rules_hibernating['Antecedent Product Category'] = c_rules_hibernating['antecedents'].
c_rules_hibernating['Consequent Product Category'] = c_rules_hibernating['consequents'].
```

In [394…

```
# Remove duplicates based on 'antecedents' and 'consequents'
c_rules_hibernating = c_rules_hibernating.drop_duplicates(subset=['antecedents', 'conseq

# Sort the rules by a relevant metric, e.g., lift, in descending order
```

```
sorted_rules_hibernating = c_rules_hibernating.sort_values(by='lift', ascending=False)

# Select the top 20 cross-sell/up-sell opportunities
top_opportunities_hibernating = sorted_rules_hibernating.head(20)

# Display the top 20 cross-sell/up-sell opportunities
print("Top Cross-Sell/Up-Sell Opportunities for Hibernating Customer Segment:")
top_opportunities_hibernating
```

```
Top Cross-Sell/Up-Sell Opportunities for Hibernating Customer Segment:
```

Out[394]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zh
0	(1033.0)	(1036.0)	0.016559	0.021002	0.013732	0.829268	39.485929	0.013384	5.734133	
1	(1036.0)	(1033.0)	0.021002	0.016559	0.013732	0.653846	39.485929	0.013384	2.841052	
60	(1331.0)	(1332.0)	0.024637	0.027060	0.022617	0.918033	33.926107	0.021950	11.869871	
61	(1332.0)	(1331.0)	0.027060	0.024637	0.022617	0.835821	33.926107	0.021950	5.940850	
95	(1176.0)	(1177.0, 1158.0)	0.040388	0.017367	0.012520	0.310000	17.850233	0.011819	1.424106	
94	(1177.0, 1158.0)	(1176.0)	0.017367	0.040388	0.012520	0.720930	17.850233	0.011819	3.438611	
87	(1145.0, 1142.0)	(1154.0)	0.014943	0.048465	0.010905	0.729730	15.056757	0.010180	3.520679	
90	(1154.0)	(1145.0, 1142.0)	0.048465	0.014943	0.010905	0.225000	15.056757	0.010180	1.271041	
38	(1162.0)	(1166.0)	0.054927	0.026656	0.018982	0.345588	12.964795	0.017518	1.487357	
39	(1166.0)	(1162.0)	0.026656	0.054927	0.018982	0.712121	12.964795	0.017518	3.282884	
82	(1171.0, 1126.0)	(1145.0)	0.015751	0.060178	0.012116	0.769231	12.782654	0.011168	4.072563	
83	(1145.0)	(1171.0, 1126.0)	0.060178	0.015751	0.012116	0.201342	12.782654	0.011168	1.232379	
84	(1171.0)	(1145.0, 1126.0)	0.041195	0.026656	0.012116	0.294118	11.033868	0.011018	1.378904	
81	(1145.0, 1126.0)	(1171.0)	0.026656	0.041195	0.012116	0.454545	11.033868	0.011018	1.757808	

37	(1163.0)	(1162.0)	0.029483	0.054927	0.017771	0.602740	10.973409	0.016151	2.378976
36	(1162.0)	(1163.0)	0.054927	0.029483	0.017771	0.323529	10.973409	0.016151	1.434677
88	(1154.0, 1142.0)	(1145.0)	0.016963	0.060178	0.010905	0.642857	10.682646	0.009884	2.631502
89	(1145.0)	(1154.0, 1142.0)	0.060178	0.016963	0.010905	0.181208	10.682646	0.009884	1.200595
70	(1142.0, 1126.0)	(1154.0)	0.025444	0.048465	0.012924	0.507937	10.480423	0.011691	1.933764
71	(1154.0)	(1142.0, 1126.0)	0.048465	0.025444	0.012924	0.266667	10.480423	0.011691	1.328940

Champion Category Price Elasticity

In [395…
```
champ_df = df[df['Segment'] == 'Champions']
```

In [396…
```
champ_df
```

Out[396]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	Qt
4	2022-07-01	SO00000002	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	15.
5	2022-07-01	SO00000003	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1709.0	FS SWEET CORN ON COB	Ready to Cook (RTC)	26.
6	2022-07-01	SO00000003	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1707.0	BBQ SWEET CORN ON COB	Ready to Cook (RTC)	26.
7	2022-07-01	SO00000004	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1166.0	(2) SUPER CRISPY CHIX 400G	Ready to Cook (RTC)	15.
8	2022-07-01	SO00000005	C001	COLD STORAGE SUPERMARKETS	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	Ready to Cook (RTC)	15.
…	…	…	…	…	…	…	…	…	
74930	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1173.0	YAKITORI WITH SAUCE	Ready to Cook (RTC)	1.

74931	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1190.0	N1 CHICKEN NUGGET (10PKT)	Ready to Cook (RTC)	2.
74932	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1191.0	TEMPURA CHICKEN NUGGET	Ready to Cook (RTC)	3.
74933	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1184.0	CHEESE CHICKEN BALL	Ready to Cook (RTC)	5.
74934	2023-05-31	SO00030685	C021	CASH - STAFF PURCHASE	HAWKER	1360.0	COCONUT WATER WITH MEAT 250ML	Ready to Cook (RTC)	26.

46734 rows × 14 columns

In [397…
```
champ_top5 = champ_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Inventory

champ_top5_df = champ_top5.sort_values(by='Total Base Amt', ascending=False).head(5)

print(champ_top5_df)
```

```
                Total Base Amt                    Inventory Desc
Inventory Code
1030.0                2150519.23         SP01 SKINLESS CHIX BREAST
1158.0                2126522.32  (33) CRISPY CHICKEN SEAWEED 1KG
1032.0                1292526.89               SP03 CHICKEN FILLET
1332.0                1130700.22              BETAGRO BREAST HERBS
1331.0                1072250.92      BETAGRO BREAST GARLIC PEPPER
```

In [398…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = champ_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_champ_top5_df = champ_df[champ_df['Inventory Code'].isin(unique_inventory_codes

# Group and aggregate the filtered DataFrame as you intended
champ_df_pe = filtered_champ_top5_df.groupby(['Transaction Date', 'Inventory Code']).agg
```

In [399…
```
champ_df_pe
```

Out[399]:

	Transaction Date	Inventory Code	Qty	Price Per Unit
0	2022-07-01	1030.0	2964.0	4.400000
1	2022-07-01	1032.0	1731.6	4.600000
2	2022-07-01	1158.0	145.6	11.354286
3	2022-07-01	1331.0	2880.8	1.750000
4	2022-07-01	1332.0	2652.0	1.750000
...	...	...	...	...
1245	2023-05-31	1030.0	1560.0	5.250000
1246	2023-05-31	1032.0	920.4	5.200000
1247	2023-05-31	1158.0	163.8	11.580000
1248	2023-05-31	1331.0	3151.2	1.700000

1249	2023-05-31	1332.0	3057.6	1.700000

1250 rows × 4 columns

In [400…

```
unique_inventory_codes = champ_df_pe.groupby('Transaction Date')['Inventory Code'].uniqu

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = champ_df_pe.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(c

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(champ_df_pe['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
champ_df_pe = pd.concat([champ_df_pe, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
champ_df_pe = champ_df_pe.sort_values(by=['Transaction Date', 'Inventory Code'])

# Reset the index
champ_df_pe.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(champ_df_pe)
```

```
     Transaction Date  Inventory Code     Qty  Price Per Unit
0          2022-07-01          1030.0  2964.0        4.400000
1          2022-07-01          1032.0  1731.6        4.600000
2          2022-07-01          1158.0   145.6       11.354286
3          2022-07-01          1331.0  2880.8        1.750000
4          2022-07-01          1332.0  2652.0        1.750000
...               ...             ...     ...             ...
1425       2023-05-31          1030.0  1560.0        5.250000
1426       2023-05-31          1032.0   920.4        5.200000
1427       2023-05-31          1158.0   163.8       11.580000
1428       2023-05-31          1331.0  3151.2        1.700000
1429       2023-05-31          1332.0  3057.6        1.700000

[1430 rows x 4 columns]
```

In [401…

```
# Create a list of unique inventory codes
unique_inventory_codes = champ_df_pe['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1030.0, 1032.0)
```

```
Inventory Code Pair: (1030.0, 1158.0)
Inventory Code Pair: (1030.0, 1331.0)
Inventory Code Pair: (1030.0, 1332.0)
Inventory Code Pair: (1032.0, 1158.0)
Inventory Code Pair: (1032.0, 1331.0)
Inventory Code Pair: (1032.0, 1332.0)
Inventory Code Pair: (1158.0, 1331.0)
Inventory Code Pair: (1158.0, 1332.0)
Inventory Code Pair: (1331.0, 1332.0)
```

In [402…

```python
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = champ_df_pe[champ_df_pe['Inventory Code'] == code1]
    data_code2 = champ_df_pe[champ_df_pe['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:       Transaction Date  Inventory Code     Qty  Price Per Unit
3           2022-07-01          1331.0  2880.8            1.75
8           2022-07-02          1331.0  3151.2            1.70
13          2022-07-04          1331.0  3764.8            1.70
18          2022-07-05          1331.0  3099.2            1.75
23          2022-07-06          1331.0  2069.6            1.70
...                ...             ...     ...             ...
1408        2023-05-26          1331.0  3588.0            1.70
1413        2023-05-27          1331.0  3151.2            1.70
1418        2023-05-29          1331.0  4482.4            1.70
1423        2023-05-30          1331.0  4191.2            1.70
1428        2023-05-31          1331.0  3151.2            1.70

[286 rows x 4 columns]
Data for Code Pair 2:       Transaction Date  Inventory Code     Qty  Price Per Unit
4           2022-07-01          1332.0  2652.0            1.75
9           2022-07-02          1332.0  2652.0            1.70
14          2022-07-04          1332.0  3504.8            1.70
19          2022-07-05          1332.0  3120.0            1.75
24          2022-07-06          1332.0  2111.2            1.70
...                ...             ...     ...             ...
1409        2023-05-26          1332.0  3452.8            1.70
1414        2023-05-27          1332.0  2652.0            1.70
1419        2023-05-29          1332.0  4607.2            1.70
1424        2023-05-30          1332.0  4264.0            1.70
1429        2023-05-31          1332.0  3057.6            1.70

[286 rows x 4 columns]
```

In [403…

```python
# Create an empty DataFrame to store the results
champ_pe_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1

# Create a list of unique inventory codes
unique_inventory_codes = champ_df_pe['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = champ_df_pe[champ_df_pe['Inventory Code'] == code1]
    data_code2 = champ_df_pe[champ_df_pe['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
```

```
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    champ_pe_elasticity_df = pd.concat([champ_pe_elasticity_df, results])

# Print or inspect the elasticity results
print(champ_pe_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1030.0                1032.0
1         2022-07-02                1030.0                1032.0
2         2022-07-04                1030.0                1032.0
3         2022-07-05                1030.0                1032.0
4         2022-07-06                1030.0                1032.0
..               ...                   ...                   ...
281       2023-05-26                1331.0                1332.0
282       2023-05-27                1331.0                1332.0
283       2023-05-29                1331.0                1332.0
284       2023-05-30                1331.0                1332.0
285       2023-05-31                1331.0                1332.0

     Relative_Price  Relative_Volume
0          0.956522         1.711712
1          1.000000         1.145833
2          0.956522         1.617978
3          1.047619         1.366667
4          0.951389         1.682243
..              ...              ...
281        1.000000         1.039157
282        1.000000         1.188235
283        1.000000         0.972912
284        1.000000         0.982927
285        1.000000         1.030612

[2860 rows x 5 columns]
```

In [404…

```
## Creating Log Relative Price and Log Relative Volume in columns

champ_pe_elasticity_df.loc[:, 'Log Relative Price'] = np.log(champ_pe_elasticity_df['Rel
champ_pe_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(champ_pe_elasticity_df['Re
```

In [405…

```
import pandas as pd
import statsmodels.api as sm

# Group the data by unique product pairs
unique_product_pairs = champ_pe_elasticity_df[['Inventory Code_code1', 'Inventory Code_c

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = champ_pe_elasticity_df[(champ_pe_elasticity_df['Inventory Code_code1'] == i

    X = sm.add_constant(subset[['Log Relative Price']])
    y = subset['Log Relative Volume']

    try:
```

```
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
}

champ_pe_table_df = pd.DataFrame(table_data)

champ_pe_table_df['Type'] = champ_pe_table_df['Coefficient (Log Relative Price)'].apply(

champ_pe_table_df['Significance (Log Relative Price)'] = champ_pe_table_df['P-Value (Log

# Create a DataFrame to store the OLS results
champ_pe_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(champ_pe_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Type	Significance (Log Relative Price)
0	(1030.0, 1032.0)	5.945908	4.002882e-06	Substitute	Significant
1	(1030.0, 1158.0)	0.703646	5.358258e-01	Substitute	Insignificant
2	(1030.0, 1331.0)	-0.082703	9.324136e-01	Complementary	Insignificant
3	(1030.0, 1332.0)	0.042594	9.655717e-01	Substitute	Insignificant
4	(1032.0, 1158.0)	-0.393675	7.436527e-01	Complementary	Insignificant
5	(1032.0, 1331.0)	-0.852654	3.694985e-01	Complementary	Insignificant
6	(1032.0, 1332.0)	-0.648806	5.056749e-01	Complementary	Insignificant
7	(1158.0, 1331.0)	10.258937	2.219343e-04	Substitute	Significant
8	(1158.0, 1332.0)	11.992330	1.528137e-05	Substitute	Significant
9	(1331.0, 1332.0)	12.195004	2.433044e-08	Substitute	Significant

Loyal Category Price Elasticity

In [406…
```
loyal_df = df[df['Segment'] == 'Loyal']
```

In [407…
```
loyal_df
```

Out[407]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	Qty

52	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1341.0	SEA SALT YAKITORI	Ready to Cook (RTC)	20.8
53	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1331.0	BETAGRO BREAST GARLIC PEPPER	Ready to Eat (RTE)	20.8
54	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1333.0	BETAGRO BREAST PINK SALT	Ready to Eat (RTE)	20.8
55	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1340.0	CURRY YAKITORI	Ready to Cook (RTC)	20.8
56	2022-07-01	SO00000024	S007	STORES (SINGAPORE) PTE LTD	RETAIL	1332.0	BETAGRO BREAST HERBS	Ready to Eat (RTE)	20.8
...	...	...	...	...	...	...	...	...	...
74867	2023-05-31	SO00030663	Z006	ZHUFRAN TRADING	WET MARKET	1142.0	FS01 FRIED 2 JOINT WING	Ready to Cook (RTC)	23.4
74868	2023-05-31	SO00030664	J031	JOKIA PTE LTD	WHOLESALER	1157.0	KT01 CHICKEN KATSU 70G	Ready to Cook (RTC)	78.0
74871	2023-05-31	SO00030667	C028	CHI YONG WAH ENTERPRISE	FOOD MANUFACTURER	1142.0	FS01 FRIED 2 JOINT WING	Ready to Cook (RTC)	31.2
74889	2023-05-31	SO00030673	M023	MFC FOOD EXPRESS PTE LTD	CAFE	1151.0	(22) SUPER CRISPY CHICKEN 1KG	Ready to Cook (RTC)	7.8
74890	2023-05-31	SO00030673	M023	MFC FOOD EXPRESS PTE LTD	CAFE	1158.0	(33) CRISPY CHICKEN SEAWEED 1KG	Ready to Cook (RTC)	13.0

14451 rows × 14 columns

In [408…

```
loyal_top5 = loyal_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum', 'Inventory

loyal_top5_df = loyal_top5.sort_values(by='Total Base Amt', ascending=False).head(5)


print(loyal_top5_df)
```

```
               Total Base Amt                    Inventory Desc
Inventory Code
1158.0              1043523.78  (33) CRISPY CHICKEN SEAWEED 1KG
1157.0               391282.84           KT01 CHICKEN KATSU 70G
1142.0               333225.10         FS01 FRIED 2 JOINT WING
1151.0               270797.54    (22) SUPER CRISPY CHICKEN 1KG
1126.0               181771.46         A11 POP CORN CHICKEN 1KG
```

In [409…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = loyal_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_loyal_top5_df = loyal_df[loyal_df['Inventory Code'].isin(unique_inventory_codes

# Group and aggregate the filtered DataFrame as you intended
loyal_df_pe = filtered_loyal_top5_df.groupby(['Transaction Date', 'Inventory Code']).agg
```

In [410…
```
loyal_df_pe
```

Out[410]:

	Transaction Date	Inventory Code	Qty	Price Per Unit
0	2022-07-01	1142.0	96.2	9.466667
1	2022-07-01	1151.0	20.8	8.500000
2	2022-07-01	1157.0	7.8	7.700000
3	2022-07-02	1142.0	22.1	9.380000
4	2022-07-02	1151.0	15.6	8.500000
...	...	...	...	...
1199	2023-05-31	1126.0	18.2	7.750000
1200	2023-05-31	1142.0	137.8	9.500000
1201	2023-05-31	1151.0	72.8	8.500000
1202	2023-05-31	1157.0	132.6	7.533333
1203	2023-05-31	1158.0	23.4	11.500000

1204 rows × 4 columns

In [411…
```
unique_inventory_codes = loyal_df_pe.groupby('Transaction Date')['Inventory Code'].uniqu

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = loyal_df_pe.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].apply(c

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(loyal_df_pe['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
loyal_df_pe = pd.concat([loyal_df_pe, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
loyal_df_pe = loyal_df_pe.sort_values(by=['Transaction Date', 'Inventory Code'])
```

```
# Reset the index
loyal_df_pe.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(loyal_df_pe)
```

```
     Transaction Date  Inventory Code    Qty  Price Per Unit
0          2022-07-01          1126.0   10.4        7.600000
1          2022-07-01          1142.0   96.2        9.466667
2          2022-07-01          1151.0   20.8        8.500000
3          2022-07-01          1157.0    7.8        7.700000
4          2022-07-01          1158.0   13.0       11.500000
...               ...             ...    ...             ...
1370       2023-05-31          1126.0   18.2        7.750000
1371       2023-05-31          1142.0  137.8        9.500000
1372       2023-05-31          1151.0   72.8        8.500000
1373       2023-05-31          1157.0  132.6        7.533333
1374       2023-05-31          1158.0   23.4       11.500000

[1375 rows x 4 columns]
```

In [412…

```
# Create a list of unique inventory codes
unique_inventory_codes = loyal_df_pe['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1126.0, 1142.0)
Inventory Code Pair: (1126.0, 1151.0)
Inventory Code Pair: (1126.0, 1157.0)
Inventory Code Pair: (1126.0, 1158.0)
Inventory Code Pair: (1142.0, 1151.0)
Inventory Code Pair: (1142.0, 1157.0)
Inventory Code Pair: (1142.0, 1158.0)
Inventory Code Pair: (1151.0, 1157.0)
Inventory Code Pair: (1151.0, 1158.0)
Inventory Code Pair: (1157.0, 1158.0)
```

In [413…

```
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = loyal_df_pe[loyal_df_pe['Inventory Code'] == code1]
    data_code2 = loyal_df_pe[loyal_df_pe['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:       Transaction Date  Inventory Code    Qty  Price Per Unit
3          2022-07-01          1157.0    7.8        7.700000
8          2022-07-02          1157.0   15.6        7.700000
13         2022-07-04          1157.0   15.6        7.700000
18         2022-07-05          1157.0   15.6        7.700000
23         2022-07-06          1157.0    7.8        7.700000
...               ...             ...    ...             ...
1353       2023-05-26          1157.0   85.8        7.700000
1358       2023-05-27          1157.0   31.2        7.700000
1363       2023-05-29          1157.0  163.8        7.700000
1368       2023-05-30          1157.0  163.8        7.450000
1373       2023-05-31          1157.0  132.6        7.533333

[275 rows x 4 columns]
```

```
Data for Code Pair 2:      Transaction Date  Inventory Code     Qty  Price Per Unit
4                2022-07-01           1158.0    13.0            11.5
9                2022-07-02           1158.0    74.1            11.5
14               2022-07-04           1158.0     2.6            11.8
19               2022-07-05           1158.0    26.0            11.5
24               2022-07-06           1158.0    13.0            11.5
...                     ...              ...     ...             ...
1354             2023-05-26           1158.0    42.9            11.8
1359             2023-05-27           1158.0    52.0            11.5
1364             2023-05-29           1158.0  1300.0            10.8
1369             2023-05-30           1158.0    62.4            11.5
1374             2023-05-31           1158.0    23.4            11.5

[275 rows x 4 columns]
```

In [414…

```python
# Create an empty DataFrame to store the results
loyal_pe_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code_code1

# Create a list of unique inventory codes
unique_inventory_codes = loyal_df_pe['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = loyal_df_pe[loyal_df_pe['Inventory Code'] == code1]
    data_code2 = loyal_df_pe[loyal_df_pe['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    loyal_pe_elasticity_df = pd.concat([loyal_pe_elasticity_df, results])

# Print or inspect the elasticity results
print(loyal_pe_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1126.0                1142.0
1         2022-07-02                1126.0                1142.0
2         2022-07-04                1126.0                1142.0
3         2022-07-05                1126.0                1142.0
4         2022-07-06                1126.0                1142.0
..               ...                   ...                   ...
270       2023-05-26                1157.0                1158.0
271       2023-05-27                1157.0                1158.0
272       2023-05-29                1157.0                1158.0
273       2023-05-30                1157.0                1158.0
274       2023-05-31                1157.0                1158.0

     Relative_Price  Relative_Volume
0          0.802817         0.108108
1          0.810235         0.470588
2          0.800000         0.666667
3          0.823495         0.208955
4          0.802817         0.023256
..              ...              ...
270        0.652542         2.000000
271        0.669565         0.600000
272        0.712963         0.126000
273        0.647826         2.625000
274        0.655072         5.666667
```

[2750 rows x 5 columns]

In [415…
```
## Creating Log Relative Price and Log Relative Volume in columns

loyal_pe_elasticity_df.loc[:, 'Log Relative Price'] = np.log(loyal_pe_elasticity_df['Rel
loyal_pe_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(loyal_pe_elasticity_df['Re
```

In [416…
```
unique_product_pairs = loyal_pe_elasticity_df[['Inventory Code_code1', 'Inventory Code_c

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = loyal_pe_elasticity_df[(loyal_pe_elasticity_df['Inventory Code_code1'] == i

    X = sm.add_constant(subset[['Log Relative Price']])
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")

# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
}

loyal_pe_table_df = pd.DataFrame(table_data)

loyal_pe_table_df['Type'] = loyal_pe_table_df['Coefficient (Log Relative Price)'].apply(

loyal_pe_table_df['Significance (Log Relative Price)'] = loyal_pe_table_df['P-Value (Log

# Create a DataFrame to store the OLS results
loyal_pe_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(loyal_pe_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Type	Significance (Log Relative Price)
0	(1126.0, 1142.0)	29.485367	1.553659e-02	Substitute	Significant
1	(1126.0, 1151.0)	2.768520	5.065161e-01	Substitute	Insignificant
2	(1126.0, 1157.0)	-16.625335	5.190603e-04	Complementary	Significant

3	(1126.0, 1158.0)	-28.597415	1.231575e-07	Complementary	Significant
4	(1142.0, 1151.0)	-28.717761	4.489440e-16	Complementary	Significant
5	(1142.0, 1157.0)	-35.392947	4.496765e-11	Complementary	Significant
6	(1142.0, 1158.0)	-52.396642	1.804076e-31	Complementary	Significant
7	(1151.0, 1157.0)	-17.139570	1.396358e-06	Complementary	Significant
8	(1151.0, 1158.0)	-42.195761	6.757131e-31	Complementary	Significant
9	(1157.0, 1158.0)	-34.641885	6.325481e-22	Complementary	Significant

Hibernating Category Price Elasticity

In [417…
```
hibernating_df = df[df['Segment'] == 'Hibernating']
```

In [418…
```
hibernating_df
```

Out[418]:

	Transaction Date	Sales Order No.	Customer Code	Customer Name	Customer Category Desc	Inventory Code	Inventory Desc	Inventory Category	
0	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1161.0	(4) HAINANESE CHIX THIGH 380G	Ready to Cook (RTC)	1
1	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1163.0	(1) CRISPY CHIX ORIGINAL 400G	Ready to Cook (RTC)	1
2	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1160.0	(9) SKINLESS CHIX BREAST 345G	Ready to Cook (RTC)	1
3	2022-07-01	SO00000001	C033	CMM MARKETING MANAGEMENT PTE LTD	SUPERMARKET	1162.0	(3) CRISPY CHIX S/WEED 400G	Ready to Cook (RTC)	3
117	2022-07-01	SO00000040	M021	MYEONGDONG DISTRICT	CAFE	1145.0	FS03 CRISPY CHICKEN STRIP	Ready to Cook (RTC)	
...	...	...	...	...	...	...	...	...	...
74036	2023-05-26	SO00030285	O020	OVER THE COUNTER CAFE (OTC CAFE)	CAFE	1173.0	YAKITORI WITH SAUCE	Ready to Cook (RTC)	
74061	2023-05-26	SO00030295	E030	ER MARKERTING (S) PTE LTD	FOOD MANUFACTURER	1126.0	A11 POP CORN CHICKEN 1KG	Ready to Cook (RTC)	1
74161	2023-05-27	SO00030338	S058	SIONG HONG SEAFOOD	FOOD MANUFACTURER	1120.0	A01 SPICY WINGSTICK	Ready to Cook	

				ENTERPRISE PTE LTD				(RTC)
74162	2023-05-27	SO00030338	S058	SIONG HONG SEAFOOD ENTERPRISE PTE LTD	FOOD MANUFACTURER	1157.0	KT01 CHICKEN KATSU 70G	Ready to Cook (RTC)
74199	2023-05-27	SO00030357	N022	NAGOMI DELI PTE LTD	RESTAURANT	1171.0	BAKED CHICKEN BITES	Ready to Cook (RTC)

5012 rows × 14 columns

In [419…
```
hibernating_top5 = hibernating_df.groupby('Inventory Code').agg({'Total Base Amt': 'sum'

hibernating_top5_df = hibernating_top5.sort_values(by='Total Base Amt', ascending=False)


print(hibernating_top5_df)
```

```
                Total Base Amt                    Inventory Desc
Inventory Code
1173.0                290650.62                YAKITORI WITH SAUCE
1172.0                120338.40                 BBQ CHICKEN BREAST
1151.0                109606.25     (22) SUPER CRISPY CHICKEN 1KG
1070.0                106080.00  CHICKEN BONELESS LEG (2KGX6PACK)
1150.0                 86668.40      33C CHICKEN WITH SEAWEED 1KG
```

In [420…
```
# Get the unique Inventory Codes from the top 5
unique_inventory_codes = hibernating_top5_df.index

# Filter the original DataFrame using the top 5 Inventory Codes
filtered_hibernating_top5_df = hibernating_df[hibernating_df['Inventory Code'].isin(uniq

# Group and aggregate the filtered DataFrame as you intended
hibernating_df_pe = filtered_hibernating_top5_df.groupby(['Transaction Date', 'Inventory
```

In [421…
```
hibernating_df_pe
```

Out[421]:

	Transaction Date	Inventory Code	Qty	Price Per Unit
0	2022-07-01	1151.0	7.8	8.500000
1	2022-07-01	1173.0	62.4	13.500000
2	2022-07-02	1173.0	2.6	13.500000
3	2022-07-04	1151.0	23.4	8.500000
4	2022-07-05	1151.0	7.8	8.500000
...	...	...	...	...
397	2023-05-22	1151.0	187.2	8.333333
398	2023-05-22	1173.0	3.9	13.500000
399	2023-05-24	1173.0	390.0	12.500000
400	2023-05-26	1151.0	31.2	8.500000
401	2023-05-26	1173.0	5.2	13.500000

402 rows × 4 columns

In [422…
```
unique_inventory_codes = hibernating_df_pe.groupby('Transaction Date')['Inventory Code']

# Step 2: Calculate the mode Quantity (Qty) and Price Per Unit for each unique code
def calculate_mode(group):
    return group.mode().iloc[0]

mode_values = hibernating_df_pe.groupby(['Inventory Code'])[['Qty', 'Price Per Unit']].a

# Step 3 and 4: Check and add missing rows
new_rows = []

for index, row in unique_inventory_codes.iterrows():
    date, codes = row['Transaction Date'], row['Inventory Code']
    all_codes = set(hibernating_df_pe['Inventory Code'].unique())
    missing_codes = list(all_codes - set(codes))

    for code in missing_codes:
        mean_qty = mode_values[mode_values['Inventory Code'] == code]['Qty'].values[0]
        mean_price = mode_values[mode_values['Inventory Code'] == code]['Price Per Unit'
        new_row = {'Transaction Date': date, 'Inventory Code': code, 'Qty': mean_qty, 'P
        new_rows.append(new_row)

# Append the missing rows to the original DataFrame
missing_rows_df = pd.DataFrame(new_rows)
hibernating_df_pe = pd.concat([hibernating_df_pe, missing_rows_df], ignore_index=True)

# Sort the DataFrame by Transaction Date and Inventory Code
hibernating_df_pe = hibernating_df_pe.sort_values(by=['Transaction Date', 'Inventory Cod

# Reset the index
hibernating_df_pe.reset_index(drop=True, inplace=True)

# Display the updated DataFrame
print(hibernating_df_pe)
```

```
     Transaction Date  Inventory Code      Qty  Price Per Unit
0          2022-07-01          1070.0  31200.0             3.4
1          2022-07-01          1150.0    130.0             8.9
2          2022-07-01          1151.0      7.8             8.5
3          2022-07-01          1172.0    704.6            84.0
4          2022-07-01          1173.0     62.4            13.5
...               ...             ...      ...             ...
1195       2023-05-26          1070.0  31200.0             3.4
1196       2023-05-26          1150.0    130.0             8.9
1197       2023-05-26          1151.0     31.2             8.5
1198       2023-05-26          1172.0    704.6            84.0
1199       2023-05-26          1173.0      5.2            13.5

[1200 rows x 4 columns]
```

In [423…
```
# Create a list of unique inventory codes
unique_inventory_codes = hibernating_df_pe['Inventory Code'].unique()

# Generate a list of unique code pairs using itertools.combinations
code_pairs = list(itertools.combinations(unique_inventory_codes, 2))

# Print the generated pairs
for pair in code_pairs:
    print("Inventory Code Pair:", pair)
```

```
Inventory Code Pair: (1070.0, 1150.0)
Inventory Code Pair: (1070.0, 1151.0)
Inventory Code Pair: (1070.0, 1172.0)
Inventory Code Pair: (1070.0, 1173.0)
Inventory Code Pair: (1150.0, 1151.0)
Inventory Code Pair: (1150.0, 1172.0)
```

```
Inventory Code Pair: (1150.0, 1173.0)
Inventory Code Pair: (1151.0, 1172.0)
Inventory Code Pair: (1151.0, 1173.0)
Inventory Code Pair: (1172.0, 1173.0)
```

In [424…

```python
#Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = hibernating_df_pe[hibernating_df_pe['Inventory Code'] == code1]
    data_code2 = hibernating_df_pe[hibernating_df_pe['Inventory Code'] == code2]

# Print or inspect the filtered data for each inventory code pair
print("Data for Code Pair 1:", data_code1)
print("Data for Code Pair 2:", data_code2)
```

```
Data for Code Pair 1:      Transaction Date  Inventory Code    Qty  Price Per Unit
3           2022-07-01          1172.0  704.6            84.0
8           2022-07-02          1172.0  704.6            84.0
13          2022-07-04          1172.0  704.6            84.0
18          2022-07-05          1172.0  704.6            84.0
23          2022-07-06          1172.0  704.6            84.0
...                ...             ...    ...             ...
1178        2023-05-19          1172.0  704.6            84.0
1183        2023-05-20          1172.0  704.6            84.0
1188        2023-05-22          1172.0  704.6            84.0
1193        2023-05-24          1172.0  704.6            84.0
1198        2023-05-26          1172.0  704.6            84.0

[240 rows x 4 columns]
Data for Code Pair 2:      Transaction Date  Inventory Code    Qty  Price Per Unit
4           2022-07-01          1173.0   62.4       13.500000
9           2022-07-02          1173.0    2.6       13.500000
14          2022-07-04          1173.0   15.6       13.500000
19          2022-07-05          1173.0   39.0       13.500000
24          2022-07-06          1173.0  795.6       13.433333
...                ...             ...    ...             ...
1179        2023-05-19          1173.0  183.3       13.500000
1184        2023-05-20          1173.0    7.8       13.500000
1189        2023-05-22          1173.0    3.9       13.500000
1194        2023-05-24          1173.0  390.0       12.500000
1199        2023-05-26          1173.0    5.2       13.500000

[240 rows x 4 columns]
```

In [425…

```python
# Create an empty DataFrame to store the results
hibernating_pe_elasticity_df = pd.DataFrame(columns=['Transaction Date', 'Inventory Code

# Create a list of unique inventory codes
unique_inventory_codes = hibernating_df_pe['Inventory Code'].unique()

# Iterate through each unique inventory code pair
for code1, code2 in itertools.combinations(unique_inventory_codes, 2):
    # Filter data for the two inventory codes
    data_code1 = hibernating_df_pe[hibernating_df_pe['Inventory Code'] == code1]
    data_code2 = hibernating_df_pe[hibernating_df_pe['Inventory Code'] == code2]

    # Merge data based on the "Transaction Date"
    merged_data = data_code1.merge(data_code2, on='Transaction Date', suffixes=('_code1'

    # Calculate relative volume and relative price
    merged_data['Relative_Volume'] = merged_data['Qty_code1'] / merged_data['Qty_code2']
    merged_data['Relative_Price'] = merged_data['Price Per Unit_code1'] / merged_data['P

    # Store the results in the elasticity_df
    results = merged_data[['Transaction Date', 'Inventory Code_code1', 'Inventory Code_c
    hibernating_pe_elasticity_df = pd.concat([hibernating_pe_elasticity_df, results])
```

```
# Print or inspect the elasticity results
print(hibernating_pe_elasticity_df)
```

```
    Transaction Date  Inventory Code_code1  Inventory Code_code2  \
0         2022-07-01                1070.0                1150.0
1         2022-07-02                1070.0                1150.0
2         2022-07-04                1070.0                1150.0
3         2022-07-05                1070.0                1150.0
4         2022-07-06                1070.0                1150.0
..               ...                   ...                   ...
235       2023-05-19                1172.0                1173.0
236       2023-05-20                1172.0                1173.0
237       2023-05-22                1172.0                1173.0
238       2023-05-24                1172.0                1173.0
239       2023-05-26                1172.0                1173.0

     Relative_Price  Relative_Volume
0          0.382022       240.000000
1          0.382022       240.000000
2          0.382022       240.000000
3          0.382022       240.000000
4          0.382022       240.000000
..              ...              ...
235        6.222222         3.843972
236        6.222222        90.333333
237        6.222222       180.666667
238        6.720000         1.806667
239        6.222222       135.500000

[2400 rows x 5 columns]
```

In [426…
```
## Creating Log Relative Price and Log Relative Volume in columns

hibernating_pe_elasticity_df.loc[:, 'Log Relative Price'] = np.log(hibernating_pe_elasti
hibernating_pe_elasticity_df.loc[:, 'Log Relative Volume'] = np.log(hibernating_pe_elast
```

In [427…
```
unique_product_pairs = hibernating_pe_elasticity_df[['Inventory Code_code1', 'Inventory

# Initialize lists to store the results
inventory_pairs = []
coefficients_log_price = []
p_values_log_price = []
ols_results = []  # To store OLS results

# Loop through each unique product pair and perform the regression
for _, pair in unique_product_pairs.iterrows():
    inventory1 = pair['Inventory Code_code1']
    inventory2 = pair['Inventory Code_code2']

    subset = hibernating_pe_elasticity_df[(hibernating_pe_elasticity_df['Inventory Code_

    X = sm.add_constant(subset[['Log Relative Price']])
    y = subset['Log Relative Volume']

    try:
        model = sm.OLS(y, X).fit()
        # Append the results and product pair information
        inventory_pairs.append((inventory1, inventory2))
        coefficients_log_price.append(model.params['Log Relative Price'])
        p_values_log_price.append(model.pvalues['Log Relative Price'])
        ols_results.append(model)  # Store the OLS result object
    except Exception as e:
        print(f"Skipped pair ({inventory1}, {inventory2}) due to error: {e}")
```

```
# Create a DataFrame with the extracted data
table_data = {
    'Inventory Code Pair': inventory_pairs,
    'Coefficient (Log Relative Price)': coefficients_log_price,
    'P-Value (Log Relative Price)': p_values_log_price,
}

hibernating_pe_table_df = pd.DataFrame(table_data)

hibernating_pe_table_df['Type'] = hibernating_pe_table_df['Coefficient (Log Relative Pri

hibernating_pe_table_df['Significance (Log Relative Price)'] = hibernating_pe_table_df['

# Create a DataFrame to store the OLS results
hibernating_pe_results_df = pd.DataFrame({
    'Inventory Code Pair': inventory_pairs,
    'OLS Results': ols_results
})

# Display both tables in the Jupyter Notebook
display(hibernating_pe_table_df)
```

	Inventory Code Pair	Coefficient (Log Relative Price)	P-Value (Log Relative Price)	Type	Significance (Log Relative Price)
0	(1070.0, 1150.0)	-41.953859	4.148901e-17	Complementary	Significant
1	(1070.0, 1151.0)	-63.747845	1.788482e-26	Complementary	Significant
2	(1070.0, 1172.0)	-1.181901	0.000000e+00	Complementary	Significant
3	(1070.0, 1173.0)	-66.971569	1.749150e-14	Complementary	Significant
4	(1150.0, 1151.0)	-64.214728	3.550520e-27	Complementary	Significant
5	(1150.0, 1172.0)	-41.956844	4.147581e-17	Complementary	Significant
6	(1150.0, 1173.0)	-63.229009	1.911902e-15	Complementary	Significant
7	(1151.0, 1172.0)	-63.751635	1.766758e-26	Complementary	Significant
8	(1151.0, 1173.0)	-70.191821	1.472535e-24	Complementary	Significant
9	(1172.0, 1173.0)	-66.976043	1.741086e-14	Complementary	Significant

In [ ]: