SIG742 – MODERN DATA SCIENCE
END TERM ASSESSMENT

BY

Reshma Joseph (Student ID: 224208476)

B Priyankaa (Student ID: 224207694)

Deepak Arun S (Student ID: 224207809)

Import the packages

Code:

```
#import the packages
from pyspark.sql import SparkSession
from pyspark.sql.functions import regexp_replace, col, when
from mlxtend.frequent_patterns import apriori, association_rules
from sklearn.exceptions import ConvergenceWarning
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from math import sqrt
import matplotlib.pyplot as pyplot
import warnings
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform
from statsmodels.tsa.seasonal import seasonal_decompose
warnings.filterwarnings("ignore", category=Warning)  # Ignore all warnings
```

Creating Spark session

Code:

```
#  initializes a Spark session named "End_term_assignment," to perform Spark
operations in the application
spark=SparkSession.builder.appName('End_term_assignment').getOrCreate()
```

Loading the Data

Code:

```
# Path to your .gz file
file_path = '/content/transactionrecord.gz'
# Read the .gz file (assuming it contains a CSV)
df = spark.read.csv(file_path, header=True, inferSchema=True)
# Show the DataFrame
df.show()
```

Output:

```
+-------------+---------+---------+--------------------+----------------+-----
+--------+----------+--------------+
|TransactionNo|     Date|ProductNo|
ProductName|Product_category|Price|Quantity|CustomerNo|       Country|
+-------------+---------+---------+--------------------+----------------+-----
+--------+----------+--------------+
|       581482|12/9/2019|    22485|Set Of 2 Wooden M...|             0ca|21.47|
12|     17490|United Kingdom|
|       581475|12/9/2019|    22596|Christmas Star Wi...|             0ca|10.65|
36|     13069|United Kingdom|
|       581475|12/9/2019|    23235|Storage Tin Vinta...|             0ca|11.53|
12|     13069|United Kingdom|
|       581475|12/9/2019|    23272|Tree T-Light Hold...|             0ca|10.65|
12|     13069|United Kingdom|
|       581475|12/9/2019|     23239|Set Of 4 Knick Kn...|             0ca|11.94|
6|     13069|United Kingdom|
|       581475|12/9/2019|     21705|Bag 500g Swirly M...|             0ca|10.65|
24|     13069|United Kingdom|
|       581475|12/9/2019|     22118|Joy Wooden Block ...|             0ca|11.53|
18|     13069|United Kingdom|
|       581475|12/9/2019|     22119|Peace Wooden Bloc...|             0ca|12.25|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22217|T-Light Holder Ha...|             0ca|10.65|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22216|T-Light Holder Wh...|             0ca|10.55|
24|     13069|United Kingdom|
|       581475|12/9/2019|     22380|   Toy Tidy Spaceboy|             0ca|11.06|
20|     13069|United Kingdom|
|       581475|12/9/2019|     22442|Grow Your Own Flo...|             0ca|12.25|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22664|Toy Tidy Dolly Gi...|             0ca|11.06|
20|     13069|United Kingdom|
|       581475|12/9/2019|     22721|Set Of 3 Cake Tin...|             0ca|12.25|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22723|Set Of 6 Herb Tin...|             0ca|11.53|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22785|Squarecushion Cov...|             0ca|11.53|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22955|36 Foil Star Cake...|             0ca|11.06|
24|     13069|United Kingdom|
|       581475|12/9/2019|     23141|Triple Wire Hook ...|             0ca|11.06|
12|     13069|United Kingdom|
|       581475|12/9/2019|     22956|36 Foil Heart Cak...|             0ca|11.06|
24|     13069|United Kingdom|
|       581475|12/9/2019|     22581|Wood Stocking Chr...|             0ca|10.55|
48|     13069|United Kingdom|
+-------------+---------+---------+--------------------+----------------+-----
+--------+----------+--------------+
only showing top 20 rows
```

Question: 1

Question: 1.1

Using PySpark to do some of the data wrangling process, so that:

Question 1.1.1

For the 'NA' in CustomerNo columns, change it to '-1'.

Code:

```
# 1.1.1. Replace 'NA' in CustomerNo with '-1'
# Here we treat None as NA and replace it with -1
df = df.withColumn("CustomerNo", when(col("CustomerNo").isNull(), -
1).otherwise(col("CustomerNo")))
df.show()
```

Output:

```
+-------------+---------+---------+--------------------+---------------+-----
+--------+---------+--------------+
|TransactionNo|     Date|ProductNo|
ProductName|Product_category|Price|Quantity|CustomerNo|       Country|
+-------------+---------+---------+--------------------+---------------+-----
+--------+---------+--------------+
|       581482|12/9/2019|    22485|Set Of 2 Wooden M...|            0ca|21.47|
12|     17490|United Kingdom|
|       581475|12/9/2019|    22596|Christmas Star Wi...|            0ca|10.65|
36|     13069|United Kingdom|
|       581475|12/9/2019|    23235|Storage Tin Vinta...|            0ca|11.53|
12|     13069|United Kingdom|
|       581475|12/9/2019|    23272|Tree T-Light Hold...|            0ca|10.65|
12|     13069|United Kingdom|
|       581475|12/9/2019|    23239|Set Of 4 Knick Kn...|            0ca|11.94|
6|     13069|United Kingdom|
|       581475|12/9/2019|    21705|Bag 500g Swirly M...|            0ca|10.65|
24|     13069|United Kingdom|
|       581475|12/9/2019|    22118|Joy Wooden Block ...|            0ca|11.53|
18|     13069|United Kingdom|
|       581475|12/9/2019|    22119|Peace Wooden Bloc...|            0ca|12.25|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22217|T-Light Holder Ha...|            0ca|10.65|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22216|T-Light Holder Wh...|            0ca|10.55|
24|     13069|United Kingdom|
|       581475|12/9/2019|    22380|   Toy Tidy Spaceboy|            0ca|11.06|
20|     13069|United Kingdom|
|       581475|12/9/2019|    22442|Grow Your Own Flo...|            0ca|12.25|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22664|Toy Tidy Dolly Gi...|            0ca|11.06|
20|     13069|United Kingdom|
|       581475|12/9/2019|    22721|Set Of 3 Cake Tin...|            0ca|12.25|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22723|Set Of 6 Herb Tin...|            0ca|11.53|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22785|Squarecushion Cov...|            0ca|11.53|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22955|36 Foil Star Cake...|            0ca|11.06|
24|     13069|United Kingdom|
|       581475|12/9/2019|    23141|Triple Wire Hook ...|            0ca|11.06|
12|     13069|United Kingdom|
|       581475|12/9/2019|    22956|36 Foil Heart Cak...|            0ca|11.06|
24|     13069|United Kingdom|
|       581475|12/9/2019|    22581|Wood Stocking Chr...|            0ca|10.55|
48|     13069|United Kingdom|
+-------------+---------+---------+--------------------+---------------+-----
+--------+---------+--------------+
only showing top 20 rows
```

Answer:

Explanation of the Code and Logic Used:

- The above code modifies the 'CustomerNo' column in the DataFrame 'df'. It checks if the 'CustomerNo' is 'None' (treated as NA). If true, it replaces that value with '-1'; otherwise, it retains the original 'CustomerNo'.
- Replacing 'None' values with '-1' ensures that every entry has a valid identifier, which helps maintain data integrity and avoids errors in future analyses. This approach also allows for easy identification of records with missing customer data.

Reason for using this solution: This solution was chosen because it effectively handles missing values in the `CustomerNo` column by replacing them with a value of '-1' to maintain data integrity and avoid errors in further processing or analysis. This approach ensures that subsequent operations on the DataFrame can proceed without encountering null values, which could lead to errors or complications. Using '-1' as a placeholder allows for easy identification of originally missing entries during data analysis. The solution also utilizes the PySpark `when` function, making it efficient for large datasets and maintaining performance in distributed computing environments.

Alternative Solutions: An alternative solution could involve using the fillna method in PySpark, which simplifies the replacement of null values by filling them with a specified value, such as '-1'. Another option is to impute the missing CustomerNo values with the mean, mode, or a placeholder string like "Unknown". While this maintains data integrity, it could introduce bias in analyses if the values are not carefully chosen. Dropping rows with missing values is another approach, but it could lead to significant data loss, which may impact the quality of the analysis.

Optimality: This solution is optimal for large datasets because it preserves data while handling missing values effectively. By replacing missing values with a clear placeholder like '-1', it ensures data completeness without deleting rows or introducing complexities in the analysis. This approach balances efficiency and clarity, making it ideal for distributed computing environments like PySpark. The ability to easily identify and filter records with missing CustomerNo values later on adds flexibility, making it a practical solution for both simple and complex data processing tasks.

Question 1.1.2

Process the text in productName column, only alphabet characters left, and save the processed result to a new column productName_process and show the first 5 rows.

Code:

```
# 1.1.2. Process the ProductName to keep only alphabetic characters
df = df.withColumn('productName_process', regexp_replace(col('ProductName'),
'[^a-zA-Z]', ''))

# Show the first 5 rows
df.select("TransactionNo", "Date", "ProductNo", "ProductName", "CustomerNo",
"productName_process").show(5)
```

Output:

```
+-------------+---------+---------+-------------------+----------+------------
--------+
|TransactionNo|     Date|ProductNo|        ProductName|CustomerNo|
productName_process|
+-------------+---------+---------+-------------------+----------+------------
--------+
|       581482|12/9/2019|    22485|Set Of 2 Wooden M...|
17490|SetOfWoodenMarket...|
|       581475|12/9/2019|    22596|Christmas Star Wi...|
13069|ChristmasStarWish...|
|       581475|12/9/2019|    23235|Storage Tin Vinta...|
13069|StorageTinVintage...|
|       581475|12/9/2019|    23272|Tree T-Light Hold...|
13069|TreeTLightHolderW...|
|       581475|12/9/2019|    23239|Set Of 4 Knick Kn...|
13069|SetOfKnickKnackTi...|
+-------------+---------+---------+-------------------+----------+------------
--------+
only showing top 5 rows
```

Answer:

Explanation of the Code and Logic Used:

- The above code creates new column 'productName_process' by applying regular expression replacement on 'ProductName' column. It removes any characters that are not alphabetic (it replaces non-alphabetic characters with empty string).
- Cleaning 'ProductName' ensures consistency and prepares data for analysis, such as text mining or matching. By retaining only alphabetic characters, dataset becomes cleaner and more suitable for further processing, improving overall quality of data.
- Specific columns from DataFrame and displayed the first five rows of the resulting DataFrame.
- Displaying sample of data allows for quick verification of previous transformations, ensuring that the replacements and cleaning were executed correctly.

Reason for using this solution: This solution was chosen because it effectively cleans the 'ProductName' column by using a regular expression to retain only alphabetic characters. Utilizing the 'regexp_replace' function in PySpark is a powerful and efficient way to clean product names, ensuring consistency and uniformity in the data. Removing non-alphabetic characters helps eliminate noise, making subsequent analyses more reliable and improving model performance in tasks like text classification or clustering. This approach ensures that any irrelevant symbols or numbers do not interfere with data processing or analytics. It enhances the data quality.

Alternative Solutions: Another option would be to use the translate function in PySpark, which allows for selectively removing specific characters or sets of characters by translating them to an empty string. Additionally, Python's built-in string methods such as str.isalpha combined with user-defined functions (UDFs) can also achieve similar cleaning by iterating over each product name. While these approaches work, they are less efficient than PySpark's regexp_replace for large datasets. One could also opt for a more nuanced cleaning by retaining certain non-alphabetic characters if they hold value for the analysis.

Optimality: This solution is optimal because it strikes a balance between simplicity and efficiency, particularly when dealing with large datasets in distributed environments like PySpark. The use of regexp_replace is highly efficient and directly addresses the problem of unwanted characters without introducing unnecessary complexity. However, if the task requires retaining some non-alphabetic characters (like punctuation), a more flexible regular expression can be applied. Given its scalability and precision, this approach is well-suited for general data-cleaning tasks, ensuring that data remains clean and useful for subsequent analysis.

Question 1.2:

Find out the revenue on each transaction date. In order to achieve the above, some wrangling work is required to be done:

Question 1.2.1

Using pyspark to calculate the revenue (price * Quantity) and save as float format in pyspark dataframe to show the top 5 rows.

Code:

```
#1.2.1. Calculate the total revenue for each transaction
df = df.withColumn("Revenue", (col("Price") * col("Quantity")).cast("float"))

# Show the top 5 rows
df.select("TransactionNo", "Date", "Revenue").show(5)
```

Output:

```
+-------------+---------+-------+
|TransactionNo|     Date|Revenue|
+-------------+---------+-------+
|       581482|12/9/2019| 257.64|
|       581475|12/9/2019|  383.4|
|       581475|12/9/2019| 138.36|
|       581475|12/9/2019|  127.8|
|       581475|12/9/2019|  71.64|
+-------------+---------+-------+
only showing top 5 rows
```

Answer:

Explanation of the Code and Logic Used:

- The code creates a new column named 'Revenue' in the DataFrame 'df' by multiplying the 'Price' and 'Quantity' columns, casting the result to float for accurate financial representation.
- Displays the columns 'TransactionNo', 'Date', and the newly created 'Revenue' from the DataFrame and displays the first five rows.
- Displaying a sample of the revenue data allows for quick verification of the revenue calculations. This step ensures that the computed values are accurate and correctly

reflect the intended revenue for each transaction. It also helps in assessing the overall structure of the data post-calculation.

Reason for using this solution: This is a crucial step in sales analysis as revenue is a key performance indicator. Calculating total revenue per transaction is crucial for financial analysis and reporting. This step provides insight into the sales performance by quantifying how much money was generated for each transaction. By storing this information in a new column, it becomes easy to aggregate, filter, or analyze revenue data in subsequent steps.Displaying the first few rows allows for quick verification of correct calculations and DataFrame structure. This method is used as it is straightforward and efficient.

Alternative Solutions: An alternative approach could involve performing an aggregation operation at the transactional level, summing up all item-level revenues using groupBy before calculating the total. This could be useful in cases where each transaction involves multiple products or services. However, this might add unnecessary complexity if applied prematurely, especially for simple datasets where direct column manipulation is more efficient. Other options could include the use of SQL-style queries, but that would require additional overhead and may not yield better performance compared to the straightforward column-wise calculation.

Optimality: This solution is optimal because it calculates the total revenue in a single, direct operation, making it both efficient and scalable for large datasets. The simplicity of the method minimizes the risk of errors while preserving the DataFrame's structure, allowing for easy manipulation in future steps. Other methods, such as grouping or aggregation, might add complexity and processing time without providing significant benefits. For datasets where revenue is calculated at the line item level, this approach remains the most efficient, clear, and effective in handling performance and readability.

Question 1.2.2

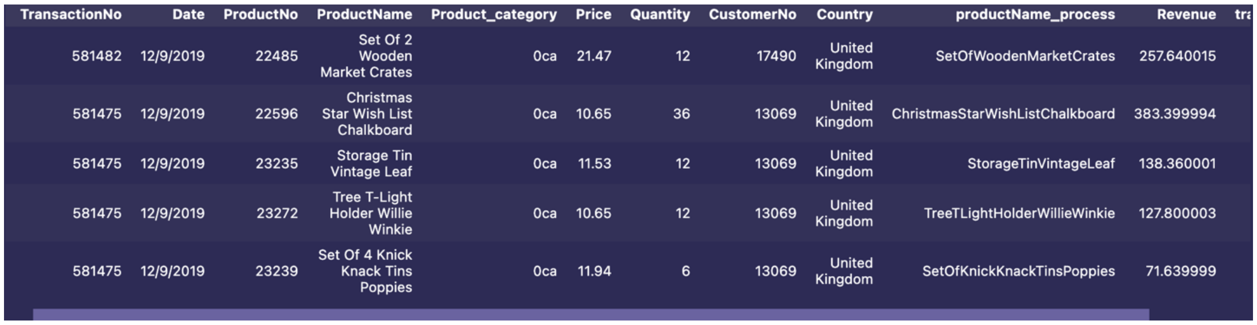
Transform the pyspark dataframe to pandas dataframe (named as df) and create the column transaction_date with date format according to Date. Print your df pandas dataframe with top 5 rows after creating the column transaction_date.

Code:

```
# 1.2.2. Transform to Pandas DataFrame and Create transaction_date Column
# Convert to Pandas DataFrame
df = df.toPandas()

# Convert 'Date' to datetime and create 'transaction_date' column
df['transaction_date'] = pd.to_datetime(df['Date'], format='%m/%d/%Y')

# Print the top 5 rows of the Pandas DataFrame
df.head()
```

{"type":"dataframe","variable_name":"df"}

Output:



TransactionNo	Date	ProductNo	ProductName	Product_category	Price	Quantity	CustomerNo	Country	productName_process	Revenue	tra
581482	12/9/2019	22485	Set Of 2 Wooden Market Crates	0ca	21.47	12	17490	United Kingdom	SetOfWoodenMarketCrates	257.640015	
581475	12/9/2019	22596	Christmas Star Wish List Chalkboard	0ca	10.65	36	13069	United Kingdom	ChristmasStarWishListChalkboard	383.399994	
581475	12/9/2019	23235	Storage Tin Vintage Leaf	0ca	11.53	12	13069	United Kingdom	StorageTinVintageLeaf	138.360001	
581475	12/9/2019	23272	Tree T-Light Holder Willie Winkie	0ca	10.65	12	13069	United Kingdom	TreeTLightHolderWillieWinkie	127.800003	
581475	12/9/2019	23239	Set Of 4 Knick Knack Tins Poppies	0ca	11.94	6	13069	United Kingdom	SetOfKnickKnackTinsPoppies	71.639999	

Answer:

Explanation of the Code and Logic Used:

- The code converts a Spark DataFrame ('df') into a Pandas DataFrame to utilize Pandas' intuitive data manipulation capabilities, particularly beneficial for smaller datasets.
- Converts the 'Date' column to a datetime format, creating a new 'transaction_date' column, which enhances date-based operations like filtering and plotting. This facilitates time-based analyses and improves handling of time-series data.
- Outputs the first five rows of the Pandas DataFrame, providing a quick view of the transformed data.

Reason for using this solution: This solution is used as after the conversion to pandas, specific functions and methods, which can be more convenient for certain types of data manipulation and analysis. Using Pandas can simplify certain operations, like data exploration or visualization. It also allows for easier use of libraries and functions specific to Pandas that may not be directly available in Spark.Created a transaction_date column as it enables easier filtering, sorting, and date-related calculations in subsequent steps.

Alternative Solutions: An alternative solution could be to continue working with the data in Spark by utilizing Spark's date manipulation functions, such as to_date or date_format. Spark can handle larger datasets more efficiently and is better suited for distributed computing. This would be particularly beneficial if the dataset is too large for Pandas or if you intend to scale the operations across multiple machines. Another alternative would be to use Dask, which combines the ease of Pandas with the scalability of Spark, allowing for parallel processing while still offering Pandas-like syntax.

Optimality: This solution is optimal when working with smaller datasets or when specific Pandas functionalities are required for more complex operations, such as data exploration, plotting, or detailed statistical analysis. Pandas is known for its ease of use and rich set of tools for manipulating data, making it a great choice for this transformation. However, if the dataset were larger or needed to be processed in a distributed manner, keeping it in Spark would be more efficient, as it is optimized for big data tasks and can handle distributed computation across clusters.

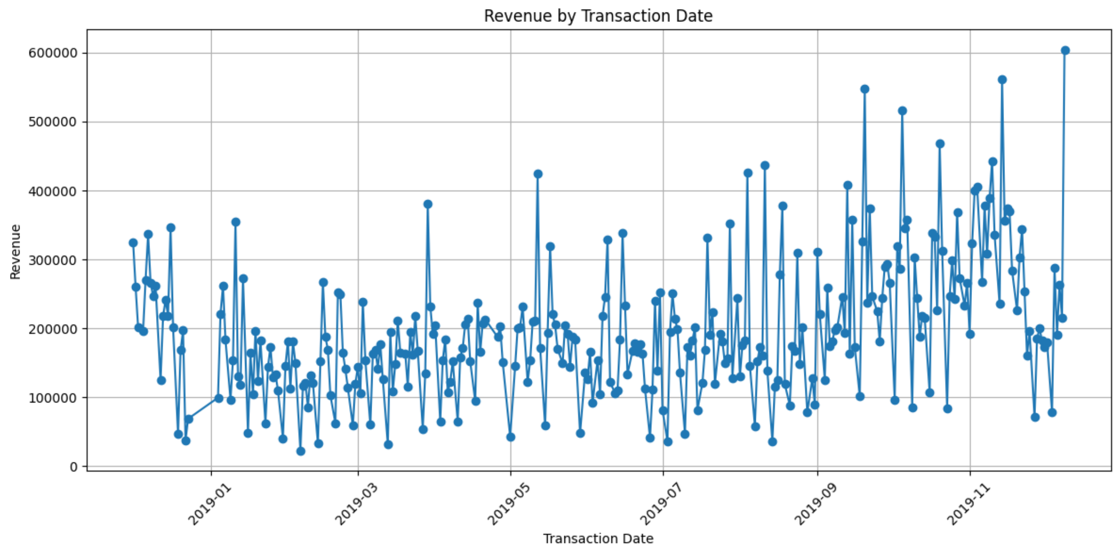
Question 1.2.3

Plot the sum of revenue on transaction_date in a line plot and find out any immediate pattern / insight?

Code:

```
#1.2.3. Plot the Sum of Revenue by transaction_date
# Group by transaction_date and sum the revenue
revenue_by_date = df.groupby('transaction_date')['Revenue'].sum().reset_index()

# Plotting
plt.figure(figsize=(12, 6))
plt.plot(revenue_by_date['transaction_date'], revenue_by_date['Revenue'],
marker='o')
plt.title('Revenue by Transaction Date')
plt.xlabel('Transaction Date')
plt.ylabel('Revenue')
plt.xticks(rotation=45)
plt.grid()
plt.tight_layout()
plt.show()
```

Output:



Answer:

Explanation of the Code and Logic Used

- groups DataFrame by `transaction_date` column and calculates total revenue for each date using `sum()` function. The `reset_index()` method is used to convert resulting Series back into DataFrame format.
- Summing the revenue by date provides a clear view of sales performance over time, allowing for trend analysis and identification of peak sales periods.
- sets up a line plot to visualize total revenue against transaction dates. The marker='o' adds points to the line for better visibility. Visualizing revenue trends over time helps identify patterns and fluctuations in sales, facilitating strategic decision-making and operational adjustments.

Reason for using this solution: The code generates a visual representation of daily revenue trends based on transaction dates. By aggregating and plotting revenue, it enables the identification of patterns, such as seasonal variations or spikes, which aids in informed decision-making for marketing strategies and inventory management.

Alternative Solutions: Alternative visualizations could include bar charts, which provide a clear comparison of revenue across dates, or area plots that show the cumulative revenue over time. A scatter plot with a trendline could also be used to highlight specific outliers or peaks in sales. Additionally, using interactive plots (such as Plotly) might offer more engagement, allowing users to zoom in on particular date ranges. However, these methods may introduce unnecessary complexity when the goal is to simply identify trends. Each option offers different benefits depending on the analysis focus.

Optimality: This line plot is optimal for the task at hand as it effectively highlights trends and fluctuations in revenue over time. It provides a clean and concise visual that is easy to interpret, making it ideal for identifying seasonal trends, spikes, or dips in sales. The addition of markers ('o') ensures key data points are visible, which enhances the clarity of the graph. Alternative visualizations might add complexity without offering additional insights, making this approach both efficient and suitable for most trend analyses.

Question 1.3:

Let's continue to analyse on the transaction_date vs revenue.

Question 1.3.1

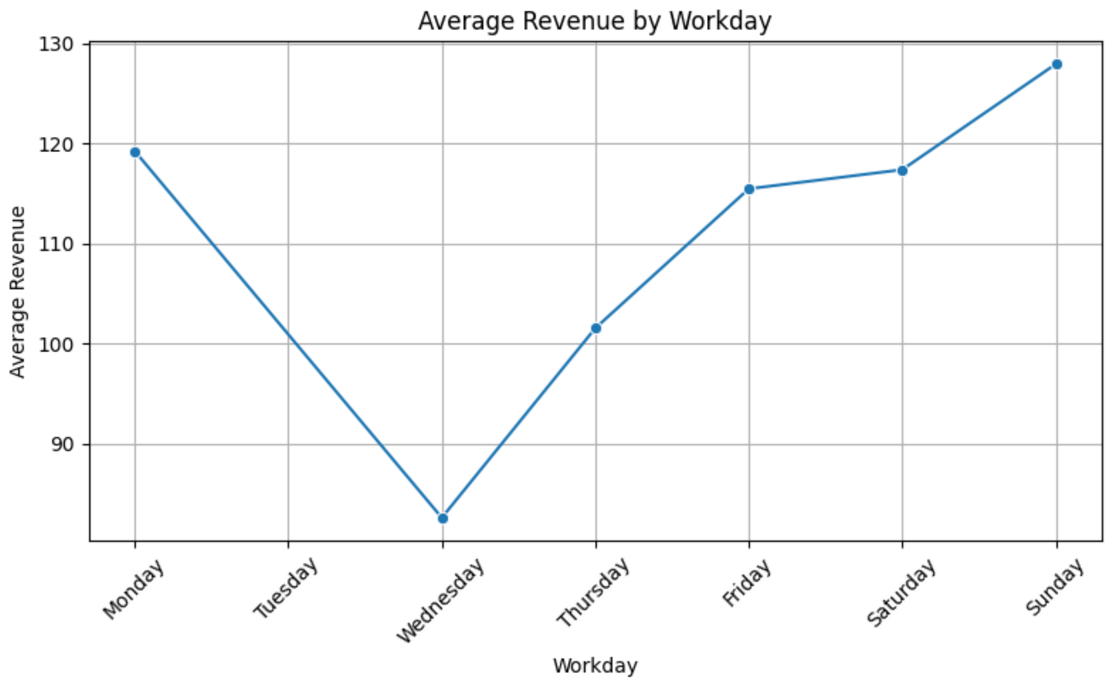
Determine which workday (day of the week), generates the most sales (plotting the results in a line chart with workday on averaged revenues).

Code:

```
# Add a column for the day of the week
df['workday'] = df['transaction_date'].dt.day_name()

# Group by workday and calculate the average revenue
average_revenue_by_workday =
df.groupby('workday')['Revenue'].mean().reset_index()

# sort by workday to follow the order (mon-sun)
order=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday',
'Sunday']
average_revenue_by_workday['workday'] =
pd.Categorical(average_revenue_by_workday['workday'], categories=order,
ordered=True)
average_revenue_by_workday=average_revenue_by_workday.sort_values('workday')

# Plotting the results
plt.figure(figsize=(8, 5))
sns.lineplot(data=average_revenue_by_workday, x='workday', y='Revenue',
marker='o')
plt.title('Average Revenue by Workday')
plt.xlabel('Workday')
plt.ylabel('Average Revenue')
plt.grid()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Identify the workday with the highest average revenue
```

```
max_revenue_workday =
average_revenue_by_workday.loc[average_revenue_by_workday['Revenue'].idxmax(),
'workday']
print(f"The workday that generates the most sales is: {max_revenue_workday}")
```

Output:



```
The workday that generates the most sales is: Sunday
```

Answer:

Explanation and the logic used:

- The code adds a workday column to the DataFrame, capturing the day names from transaction_date. It then groups the data by workday, calculating the average revenue for each day and ensuring the order is Monday to Sunday.
- A line plot visualizes this average revenue, making trends easy to identify. Finally, it determines the workday with the highest average revenue using idxmax(). This analysis helps businesses understand sales patterns, enabling them to optimize operations and marketing strategies based on peak revenue days.

Reason for using this solution: This solution is used as by visualizing average revenue by workday, businesses can better allocate resources and optimize operations based on customer purchasing behavior. This analysis helps a business capitalize on peak days, adjust marketing strategies for lower-performing days, and ultimately enhance overall profitability.

Alternative Solutions: Alternative approaches could include applying time series analysis or decomposition to identify revenue trends over weekdays, taking into account seasonal or promotional effects. Another option could involve using pivot tables with different aggregation methods, such as calculating the median revenue or total revenue per workday, which might provide additional insights into sales performance on specific days. Additionally, incorporating advanced statistical techniques like regression could help uncover more subtle patterns. These alternatives offer more detailed insights but may introduce unnecessary complexity for a straightforward comparison like average revenue by day.

Optimality: This solution is optimal for the goal of identifying revenue patterns by workday as it offers a simple and effective visualization of the average revenue trends. By maintaining a clear and intuitive format, the line plot facilitates easy interpretation, helping businesses identify peak revenue days without the need for excessive computational effort. While more complex analyses could uncover deeper insights, this approach strikes a balance between simplicity and actionable data, making it ideal for quick decision-making and trend analysis based on customer behavior.

Question 1.3.2

Identify the name of product (column productName_process) that contributes the highest revenue on 'that workday' (you need to find out from 1.3.1) and the name of product (column productName_process) that has the highest sales volume (sum of the Quantity), no need to remove negative quantity transactions.) on 'that workday' (you need to find out from 1.3.1).

Code:

```
# Filter the DataFrame for the max revenue workday
max_revenue_df = df[df['workday'] == max_revenue_workday]

# Product with the highest revenue
highest_revenue_product =
max_revenue_df.groupby('productName_process')['Revenue'].sum().idxmax()
highest_revenue_value =
max_revenue_df.groupby('productName_process')['Revenue'].sum().max()

# Product with the highest sales volume
highest_volume_product =
max_revenue_df.groupby('productName_process')['Quantity'].sum().idxmax()
highest_volume_value =
max_revenue_df.groupby('productName_process')['Quantity'].sum().max()

print(f"Product with the highest revenue on {max_revenue_workday}:
{highest_revenue_product} (${highest_revenue_value:.2f})")
print(f"Product with the highest sales volume on {max_revenue_workday}:
{highest_volume_product} (Quantity: {highest_volume_value})")
```

Output:

```
Product with the highest revenue on Sunday: WorldWarGlidersAsstdDesigns
($187081.34)
Product with the highest sales volume on Sunday: WorldWarGlidersAsstdDesigns
(Quantity: 18051)
```

Answer:

Explanation and the logic used:

- The code filters DataFrame 'df' to create 'max_revenue_df', containing records for day with highest average revenue. This allows analysis of which products generated most revenue and sales volume on that day.

- Groups by 'productName_process', calculating total revenue to identify highest-revenue product, and identifies product with highest revenue using idxmax(). The corresponding maximum revenue value is retrieved with max().
- Then it is grouped by productName_process, calculates total quantity sold for each product, and identifies product with highest sales volume.
- Then the results are displayed along with respective revenue and sales volume for the day with highest average revenue.

Reason for using this solution: Identify which product generates the most revenue and which product sells the most units on the day that sees the highest average revenue. Byidentifying key products on peak sales days, businesses can tailor promotions or ensure adequate stock levels to maximize sales opportunities.

Alternative Solutions: An alternative approach could be to rank all products based on both revenue and sales volume, offering a more comprehensive view by analyzing the top N products instead of just the highest performer. This could be extended to analyze patterns over multiple peak days, not just the highest revenue day, to provide a broader understanding of product performance over time. One could also apply advanced techniques such as clustering similar products based on sales patterns, allowing businesses to group products for promotional strategies. These methods provide richer insights but may introduce complexity for businesses looking for immediate results.

Optimality: This solution is optimal for the specific goal of identifying top-performing products on the highest revenue day. By focusing on the product with the highest revenue and sales volume, it allows businesses to quickly pinpoint which items are driving success on peak days. The analysis is computationally efficient, easily interpretable, and provides actionable data for businesses looking to tailor their inventory or marketing strategies based on performance. While more comprehensive approaches could yield deeper insights, this straightforward method is well-suited for quick decision-making without unnecessary complexity.

Question 1.3.3

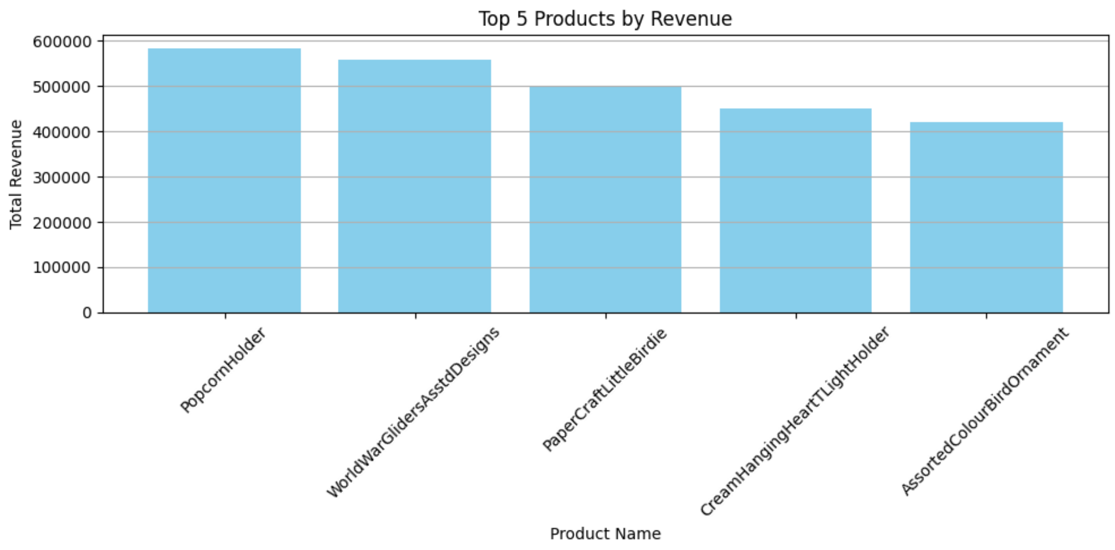
Please provide two plots showing the top 5 products that contribute the highest revenues in general and top 5 products that have the highest sales volumes in general.

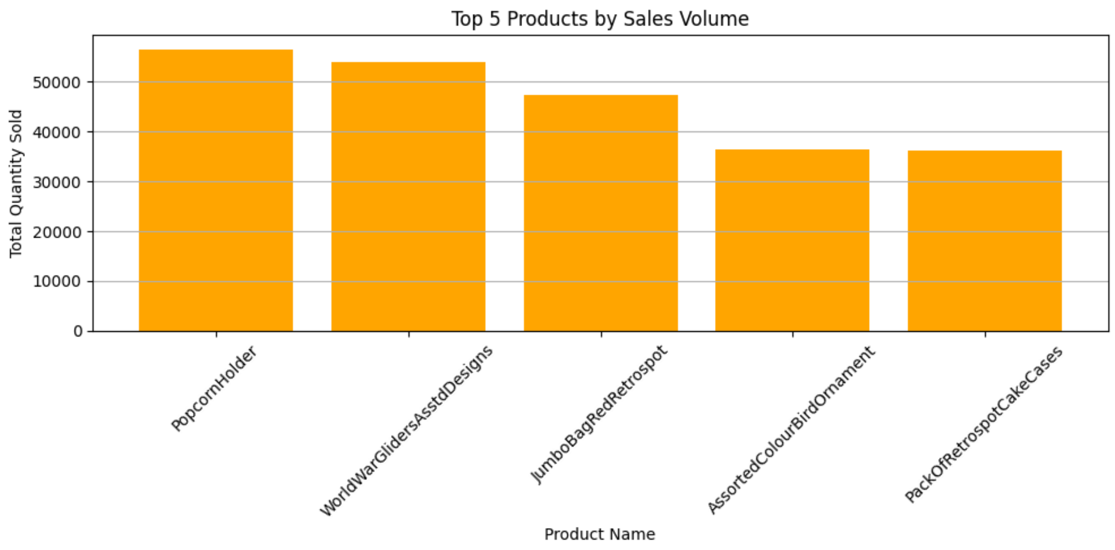
Code:

```
# Top 5 products by revenue
top_revenue_products =
df.groupby('productName_process')['Revenue'].sum().nlargest(5).reset_index()

# Plotting top 5 products by revenue
plt.figure(figsize=(10, 5))
plt.bar(top_revenue_products['productName_process'],
top_revenue_products['Revenue'], color='skyblue')
plt.title('Top 5 Products by Revenue')
plt.xlabel('Product Name')
plt.ylabel('Total Revenue')
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

```
# Top 5 products by sales volume
top_volume_products =
df.groupby('productName_process')['Quantity'].sum().nlargest(5).reset_index()

# Plotting top 5 products by sales volume
plt.figure(figsize=(10, 5))
plt.bar(top_volume_products['productName_process'],
top_volume_products['Quantity'], color='orange')
plt.title('Top 5 Products by Sales Volume')
plt.xlabel('Product Name')
plt.ylabel('Total Quantity Sold')
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

Output:





Answer:

Explanation and the logic used:

- The code groups the DataFrame 'df' by 'productName_process' to calculate total revenue, selecting the top five products with the highest revenue using 'nlargest(5)' and converting the results to a DataFrame. This analysis identifies key revenue-generating products, guiding inventory and marketing strategies
- A bar plot visualizes this data, helping stakeholders quickly recognize high performers.
- Similarly, it groups by 'productName_process' to determine the top five products by sales volume, providing insights into customer preferences.
- Then it creates a bar plot for top five products by sales volume. Visualizing sales volume helps businesses make informed decisions regarding inventory and product offerings.

Reason for using this solution: The logic used is grouping by product and summing revenue or quantity that helps in quickly identify which products are most successful, both in terms of revenue and units sold. This approach allows a clear comparison of products based on total revenue and sales volume, providing insights into which products are performing best in market.

Alternative Solutions: A more detailed approach could involve calculating and visualizing profit margins for the top products, which might provide deeper insight into profitability rather than just sales figures. Additionally, analyzing the performance of products over multiple time frames (e.g., monthly or seasonal) could help identify trends in product sales, providing a dynamic view of how products perform over time. One could also cluster similar products based on their sales and revenue patterns to identify groups of products that perform similarly, allowing for more strategic group-level marketing or stocking decisions.

Optimality: This solution is optimal for providing quick, actionable insights into product performance. It leverages simple, efficient calculations to rank products by revenue and sales volume, producing easily interpretable visualizations. This makes it ideal for businesses needing immediate results for decision-making without the overhead of more complex analyses. While alternative methods could offer richer insights, this approach strikes a good balance between simplicity, speed, and clarity, making it a strong choice for initial exploratory analysis or high-level reporting.

Question 1.4:

Which country generates the highest revenue? Additionally, identify the month in that country that has the highest revenue.

Code:

```
# Extract month and year
df['month'] = df['transaction_date'].dt.month_name()
df['year'] = df['transaction_date'].dt.year


#Find the Country with Highest Revenue
#Calculate the total revenue for each country.
```

```
# Group by country and sum the revenue
country_revenue = df.groupby('Country')['Revenue'].sum().reset_index()

# Identify the country with the highest revenue
max_revenue_country = country_revenue.loc[country_revenue['Revenue'].idxmax()]

print(f"The country that generates the highest revenue is:
{max_revenue_country['Country']} (${max_revenue_country['Revenue']:.2f})")


#Identify the Month with Highest Revenue in that Country
#Filter the DataFrame for the identified country and group by month to find the
month with the highest revenue.

# Filter for the country with the highest revenue
highest_revenue_country_df = df[df['Country'] ==
max_revenue_country['Country']]

# Group by month and sum the revenue
monthly_revenue =
highest_revenue_country_df.groupby('month')['Revenue'].sum().reset_index()

# Identify the month with the highest revenue
max_revenue_month = monthly_revenue.loc[monthly_revenue['Revenue'].idxmax()]

print(f"The month with the highest revenue in {max_revenue_country['Country']}
is: {max_revenue_month['month']} (${max_revenue_month['Revenue']:.2f})")
```

Output:

```
The country that generates the highest revenue is: United Kingdom
($49994032.00)
The month with the highest revenue in United Kingdom is: November ($6737640.00)
```

Answer:

Explanation and the logic used:

- The code adds two columns, 'month' and 'year', to facilitate time-based revenue analysis, aiding seasonal strategies.
- It groups the DataFrame by 'Country', summing total revenue to identify the most profitable markets, guiding resource allocation. The country with the highest revenue is found using `idxmax()`, providing insights for strategic expansion.
- The code outputs this country's name and revenue, enhancing decision-making. It then filters transactions for that country and groups by 'month' to summarize monthly revenue, helping to identify seasonal trends.
- Finally, it identifies the peak revenue month, enabling optimized marketing strategies during high-demand periods.

Reason for using this solution: In this we have extracted month and year from transaction dates. Then we aggregated the revenue by country and month. Identified the country and the month with the highest revenue. This provides an overview of revenue generation by country and highlights the peak revenue periods, useful for strategic decision-making in marketing or sales efforts.

Alternative Solutions: An alternative solution could involve conducting a more detailed analysis over multiple years to assess trends in revenue generation for the identified country, such as determining whether revenue is increasing or decreasing over time. Additionally, segmenting the data by product categories or customer demographics within the top revenue country could yield insights into specific market segments that are driving revenue. Moreover, employing visualization techniques, such as heat maps or time series plots, could help illustrate seasonal trends or monthly revenue fluctuations more clearly, providing a richer understanding of performance dynamics.

Optimality: This solution is optimal for quickly identifying key insights related to revenue generation across countries and specific months. By aggregating revenue and using functions like idxmax(), it efficiently highlights the most profitable markets and periods for focused strategic initiatives. This method maintains a straightforward approach while providing actionable data that can drive business decisions. The analysis supports targeted marketing efforts during peak periods and informs resource allocation to maximize revenue opportunities, making it effective for immediate operational insights.

Question 1.5:

Let's do some analysis on the CustomerNo and their transactions. Determine the shopping frequency of customers to identify who shops most frequently (find out the highest distinct count of transactionNo on customer level, be careful with those transactions that is not for shopping – filter those transaction quantity <= 0). Also, find out what products (column productName_process) 'this customer' typically buys based on the Quantity of products purchased.

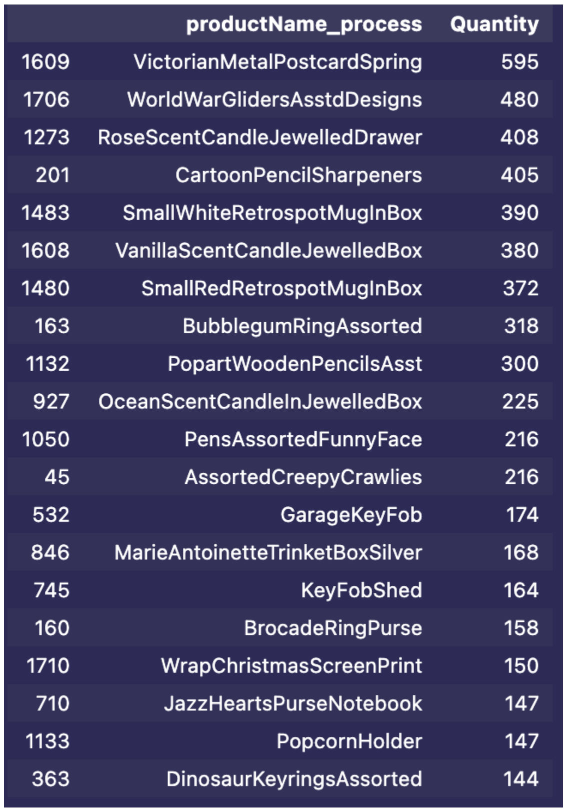
Code:

```
#Filter for valid transactions (Quantity > 0)
valid_transactions_df = df[df['Quantity'] > 0]

#Determine the Shopping Frequency of Each Customer
# Count distinct TransactionNo per CustomerNo
customer_frequency =
valid_transactions_df.groupby('CustomerNo')['TransactionNo'].nunique().reset_in
dex()

# Identify the customer with the highest frequency
most_frequent_customer =
customer_frequency.loc[customer_frequency['TransactionNo'].idxmax()]

print(f"The customer who shops most frequently is:
{most_frequent_customer['CustomerNo']} with
{most_frequent_customer['TransactionNo']} transactions.")

#Find Products Typically Bought by the Most Frequent Customer
# Filter the DataFrame for the most frequent customer
frequent_customer_df =
valid_transactions_df[valid_transactions_df['CustomerNo'] ==
most_frequent_customer['CustomerNo']]
```

```
# Group by productName_process and sum the Quantity
products_purchased =
frequent_customer_df.groupby('productName_process')['Quantity'].sum().reset_ind
ex()

# Sort by Quantity to identify the most purchased products
top_products = products_purchased.sort_values(by='Quantity', ascending=False)

print("Products typically bought by the most frequent customer:")
top_products.head(20)
```

Output:

```
The customer who shops most frequently is: 12748 with 207 transactions.
Products typically bought by the most frequent customer:
```



	productName_process	Quantity
1609	VictorianMetalPostcardSpring	595
1706	WorldWarGlidersAsstdDesigns	480
1273	RoseScentCandleJewelledDrawer	408
201	CartoonPencilSharpeners	405
1483	SmallWhiteRetrospotMugInBox	390
1608	VanillaScentCandleJewelledBox	380
1480	SmallRedRetrospotMugInBox	372
163	BubblegumRingAssorted	318
1132	PopartWoodenPencilsAsst	300
927	OceanScentCandleInJewelledBox	225
1050	PensAssortedFunnyFace	216
45	AssortedCreepyCrawlies	216
532	GarageKeyFob	174
846	MarieAntoinetteTrinketBoxSilver	168
745	KeyFobShed	164
160	BrocadeRingPurse	158
1710	WrapChristmasScreenPrint	150
710	JazzHeartsPurseNotebook	147
1133	PopcornHolder	147
363	DinosaurKeyringsAssorted	144

Answer:

Explanation of the Code and Logic Used:

- The code creates a DataFrame, 'valid_transactions_df', filtering out transactions with zero quantity to ensure meaningful data for analysis.
- It groups this data by 'CustomerNo', counting unique transactions to identify shopping frequency, crucial for understanding customer loyalty.
- The most frequent shopper is found using idxmax(), enabling targeted marketing efforts.
- The customer ID and transaction count are printed for key insights.
- The code then filters for this customer's transactions, allowing detailed analysis of their purchasing habits.

- Products are grouped and summed by quantity, sorted to reveal top items, and the top 20 products are displayed to aid in personalized marketing strategies.

Reason for using this solution: The code filters out any invalid transactions (quantity ≤ 0). It calculates the frequency of transactions for each customer and identifies the one with highest frequency. For the most frequent customer, it identifies and ranks the products they buy most often. Then we have displayed the most frequent customer and their top products.

Alternative Solutions: An alternative approach could involve segmenting customers based on additional demographics, such as age, gender, or location, to understand shopping patterns and preferences across different customer profiles. This could yield insights into whether certain products perform better with specific demographics, allowing for more targeted marketing strategies. Another option is to perform cohort analysis, tracking customers over time to see how shopping behavior changes with seasonality or promotional activities. Implementing machine learning algorithms for clustering customers based on purchasing patterns could also provide valuable insights for personalized marketing strategies.

Optimality: This analysis is optimal for businesses aiming to deepen their understanding of customer loyalty and preferences. By focusing on the most frequent customers and their purchasing habits, companies can tailor marketing campaigns and product offerings to enhance customer satisfaction and retention. The solution efficiently filters out irrelevant data, enabling targeted analysis that supports strategic decision-making. Identifying top products for frequent shoppers allows businesses to leverage customer loyalty effectively, enhancing profitability while minimizing resource allocation inefficiencies. Overall, this method provides actionable insights that can drive customer engagement and maximize sales opportunities.

Question 1.6:

As the data scientist, you would like to build a basket-level analysis on the product customer buying (filter the 'df' dataframe with df['Quantity']>0). In this task, you need to:

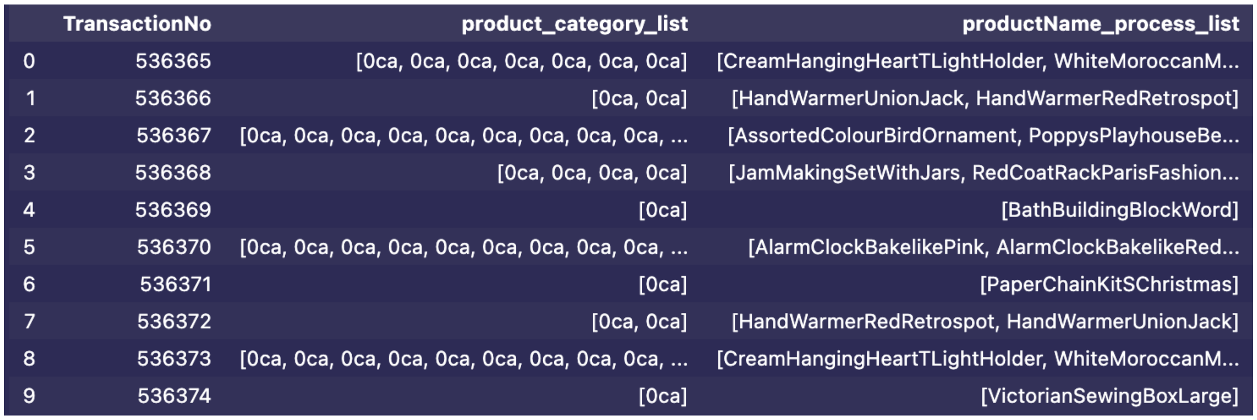
Question 1.6.1

Group by the transactionNo and aggregate the category of product (column product_category) into list on transactionNo level. Similarly, group and aggregate name of product (column productName_process) into list on transactionNo level.

Code:

```
#Filter for valid transactions (Quantity > 0)
valid_transactions_df = df[df['Quantity'] > 0]

#Group by TransactionNo and Aggregate Categories and Product Names
# Group by TransactionNo and aggregate product_category and productName
basket_analysis = valid_transactions_df.groupby('TransactionNo').agg(
    product_category_list=('Product_category', lambda x: list(x)),
    productName_process_list=('productName_process', lambda x: list(x))
).reset_index()

# Display the result
basket_analysis.head(10)
```

Output:



	TransactionNo	product_category_list	productName_process_list
0	536365	[0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca]	[CreamHangingHeartTLightHolder, WhiteMoroccanM...
1	536366	[0ca, 0ca]	[HandWarmerUnionJack, HandWarmerRedRetrospot]
2	536367	[0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, ...	[AssortedColourBirdOrnament, PoppysPlayhouseBe...
3	536368	[0ca, 0ca, 0ca, 0ca]	[JamMakingSetWithJars, RedCoatRackParisFashion...
4	536369	[0ca]	[BathBuildingBlockWord]
5	536370	[0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, ...	[AlarmClockBakelikePink, AlarmClockBakelikeRed...
6	536371	[0ca]	[PaperChainKitSChristmas]
7	536372	[0ca, 0ca]	[HandWarmerRedRetrospot, HandWarmerUnionJack]
8	536373	[0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, 0ca, ...	[CreamHangingHeartTLightHolder, WhiteMoroccanM...
9	536374	[0ca]	[VictorianSewingBoxLarge]

Answer:

Explanation of the Code and Logic Used:

- The code creates a DataFrame, 'valid_transactions_df', filtering for transactions with a quantity greater than zero, ensuring only meaningful purchases are analyzed.
- It groups this DataFrame by 'TransactionNo' and aggregates product categories and processed product names into lists using the 'agg' function. This results in two new columns: 'product_category_list' and 'productName_process_list', providing insights into products purchased together within each transaction.
- Finally, it displays the first ten rows of the 'basket_analysis' DataFrame, allowing for quick verification of the aggregation process and enhancing understanding of purchasing patterns, which can inform marketing strategies and inventory management.

Reason for using this solution: The code filters out invalid transactions to focus on actual sales, then groups valid transactions by their unique transaction numbers, aggregating associated product categories and processed product names into lists. This enables insights into frequently bought products and categories, aiding in understanding customer preferences and potential cross-selling opportunities.

Alternative Solutions: An alternative approach could involve using association rule mining, such as the Apriori algorithm, to derive more detailed insights into product affinities. This technique identifies sets of products that are frequently purchased together across all transactions, providing actionable data on how to bundle products or run promotions. Additionally, clustering algorithms could be applied to categorize transactions into groups based on purchasing patterns, allowing for a deeper understanding of customer segments and more targeted marketing efforts. Incorporating visualization techniques, such as heatmaps or network graphs, could also provide intuitive insights into customer purchasing behaviors.

Optimality: This method is optimal for businesses aiming to gain insights into transaction compositions & customer preferences. By filtering out invalid transactions and aggregating relevant product information, the analysis establishes a clear picture of purchasing behaviors, which can inform marketing strategies and inventory management. The resulting lists of product categories and names reveal opportunities for cross-selling & promotional campaigns, directly enhancing customer engagement and sales performance. This approach serves as a foundational step for more complex analyses, enabling businesses to leverage transaction data effectively for strategic decision-making. Overall, it balances clarity and depth, making it a valuable tool for understanding consumer behavior.

Question 1.6.2

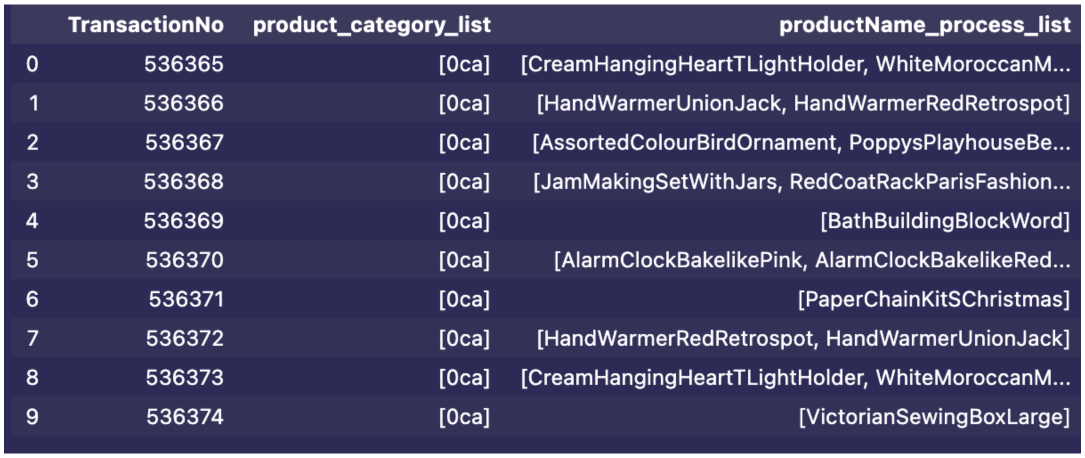
Removing duplicates on adjacent elements in the list from product_category you obtained from 1.6.1, such as [product category 1, product category 1, product category 2, ...] will be processed as [product category 1, product category 2,....]. After this processing, there will be no duplicates on on adjacent elements in the list. Please save your processed dataframe as 'df_1' and print the top 10 rows.

Code:

```
#Remove Duplicates on Adjacent Elements
def remove_adjacent_duplicates(lst):
    if not lst:
        return lst
    new_lst = [lst[0]]  # Start with the first element
    for item in lst[1:]:
        if item != new_lst[-1]:  # Only add if different from last added
            new_lst.append(item)
    return new_lst

# Apply the function to remove adjacent duplicates
basket_analysis['product_category_list'] =
basket_analysis['product_category_list'].apply(remove_adjacent_duplicates)

# Rename the processed DataFrame as 'df_1'
df_1 = basket_analysis

# Display the top 10 rows of df_1
print("Processed DataFrame 'df_1' with top 10 rows:")
df_1.head(10)
```

Output:

```
Processed DataFrame 'df_1' with top 10 rows:
```



	TransactionNo	product_category_list	productName_process_list
0	536365	[0ca]	[CreamHangingHeartTLightHolder, WhiteMoroccanM...
1	536366	[0ca]	[HandWarmerUnionJack, HandWarmerRedRetrospot]
2	536367	[0ca]	[AssortedColourBirdOrnament, PoppysPlayhouseBe...
3	536368	[0ca]	[JamMakingSetWithJars, RedCoatRackParisFashion...
4	536369	[0ca]	[BathBuildingBlockWord]
5	536370	[0ca]	[AlarmClockBakelikePink, AlarmClockBakelikeRed...
6	536371	[0ca]	[PaperChainKitSChristmas]
7	536372	[0ca]	[HandWarmerRedRetrospot, HandWarmerUnionJack]
8	536373	[0ca]	[CreamHangingHeartTLightHolder, WhiteMoroccanM...
9	536374	[0ca]	[VictorianSewingBoxLarge]

Answer:

Explanation of the Code and Logic Used:

- The code defines a function, `remove_adjacent_duplicates`, which takes a list and removes adjacent duplicate elements. It initializes a new list with the first item and iterates through the rest, adding only unique items.
- This function is applied to the `product_category_list` column of the `basket_analysis` DataFrame, cleaning the data for better analysis of unique product categories in transactions.
- The processed DataFrame is renamed to `df_1`, indicating it's ready for further analysis.
- Finally, the first ten rows of `df_1` are displayed for quick verification of successful duplicate removal and to provide an overview of the cleaned dataset.

Reason for using this solution: The code implements a function to remove adjacent duplicates from a list, streamlining the product categories and enhancing data cleanliness for further analysis. This function is applied to the product_category_list in the DataFrame, and the processed DataFrame is renamed and displayed for verification.

Alternative Solutions: An alternative method could involve using built-in Python libraries, such as pandas, to leverage its vectorized operations for more efficient duplicate removal. For example, using the shift() function could help compare adjacent elements directly in a more concise way. Another approach could be to utilize sets or dictionaries to track the occurrences of products, although this might alter the order of items. If the goal is to remove all duplicates regardless of adjacency, the unique() method can be employed to filter out duplicates from the list entirely. This provides a broader scope for data cleanliness but may not retain the order of the original entries.

Optimality: This method is optimal for maintaining the integrity of the order of product categories while eliminating redundant adjacent entries, making it particularly useful for analyzing sequences in purchasing behavior. By ensuring that only unique, consecutive items are kept, this approach enhances the dataset's clarity, enabling better insights into purchasing trends without the noise of repeated categories. It strikes a balance between preserving the original data structure and providing actionable insights, making it effective for understanding customer behavior in transactions. Overall, this method supports clearer visualizations & analyses in subsequent steps, ultimately leading to more informed decision-making in marketing and inventory management.

Question 1.7:

Continue work on the results of question 1.6, now for each of the transaction, you will have a list of product categories. To further conduct the analysis, you need to finish below by using dataframe 'df_1':

Question 1.7.1

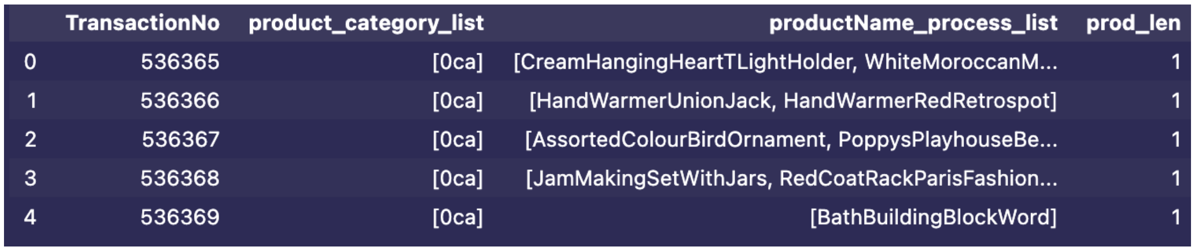
Create new column prod_len to find out the length of the list from product_category on each transaction. Print the first five rows of dataframe 'df_1'.

Code:

```
#Calculate the Length of the Product Category List
# Calculate the length of the product category list and create a new column
'prod_len'
df_1['prod_len'] = df_1['product_category_list'].apply(len)

# Print the first five rows of the dataframe
print("First five rows of 'df_1' with 'prod_len':")
df_1.head(5)
```

Output:

```
First five rows of 'df_1' with 'prod_len':
```



	TransactionNo	product_category_list	productName_process_list	prod_len
0	536365	[0ca]	[CreamHangingHeartTLightHolder, WhiteMoroccanM...	1
1	536366	[0ca]	[HandWarmerUnionJack, HandWarmerRedRetrospot]	1
2	536367	[0ca]	[AssortedColourBirdOrnament, PoppysPlayhouseBe...	1
3	536368	[0ca]	[JamMakingSetWithJars, RedCoatRackParisFashion...	1
4	536369	[0ca]	[BathBuildingBlockWord]	1

Answer:

Explanation of the Code and Logic Used:

- The code creates a new column, 'prod_len', in the 'df_1' DataFrame, calculating the length of the 'product_category_list' for each row using the built-in 'len' function. This quantifies the number of different product categories in each transaction, offering insights into transaction complexity and customer purchasing behavior.
- Additionally, it prints a message and displays the first five rows of the modified 'df_1', including the new 'prod_len' column. This sample allows for quick verification of the length calculations and provides an overview of typical product category counts in transactions, aiding in understanding customer preferences and potential bundling strategies.

Reason for using this solution: This approach is used as length of the product category list provides insights into customer behavior; longer lists may indicate more complex purchasing decisions, while shorter lists suggest simpler choices. This metric inform further analyses, like customer segmentation/behavior modeling.

Alternative Solutions: An alternative solution could involve using statistical measures to analyze the distribution of the product category list lengths. For instance, one could calculate descriptive statistics (mean, median, mode) to understand the central tendencies and variability in transaction complexity. Additionally, visualizing the distribution with histograms or box plots could offer deeper insights into how often customers make more complex versus simpler purchases. Another option might be to categorize transactions into bins (e.g., low, medium, high complexity) based on the length of the product category list, which can facilitate targeted marketing efforts based on complexity levels.

Optimality: This method is optimal for enhancing the understanding of customer purchasing behavior by quantifying transaction complexity. The addition of the ‘prod_len’ column allows for more nuanced analyses, such as identifying trends related to transaction size and frequency of complex purchases. It effectively serves as a key metric for further analyses, enabling businesses to tailor their strategies based on customer preferences. By quantifying the diversity in product categories purchased per transaction, it provides actionable insights that can inform marketing campaigns, inventory management, and overall customer engagement strategies. This level of detail empowers businesses to better understand their customers and respond to their needs more effectively.

Question 1.7.2

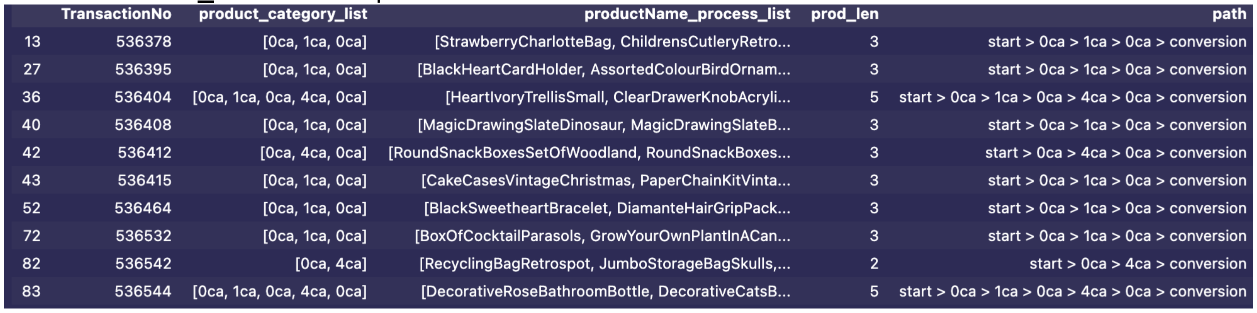
Transform the list in product_category from [productcategory1, productcategory2...] to 'start > productcategory1 > productcategory2 > ... > conversion' with new column path. You need to add 'start' as the first element, and 'conversion' as the last. Also you need to use ' > ' to connect each of the transition on products (there is a space between the elements and the transition symbol >). The final format after the transition is given in example as below fig. 2. Define the function data_processing to achieve above with three arguments: df which is the dataframe name, maxlength with default value of 3 for filtering the dataframe with prod_len" <=maxlength and minlength with default value of 1 for filtering the dataframe with prod_len >=minlength. The function data_processing will return the new dataframe'df_2'. Run your defined function with dataframe 'df_1', maxlength = 5 and minlength = 2, print the dataframe 'df_2' with top 10 rows.

Code:

```
#Transform the Product Category List to a Path Format
#Define a Function to Create the Path Format

def data_processing(df, maxlength=3, minlength=1):
    # Filter the DataFrame based on the prod_len
    filtered_df = df[(df['prod_len'] <= maxlength) & (df['prod_len'] >=
minlength)]

    # Create the 'path' column
    filtered_df['path'] = 'start > ' +
filtered_df['product_category_list'].apply(lambda x: ' > '.join(x)) + ' >
conversion'

    return filtered_df

#Apply the Function to Get df_2
#Run the Function with the Required Parameters

# Create df_2 using the data_processing function
df_2 = data_processing(df_1, maxlength=5, minlength=2)

# Print the top 10 rows of df_2
print("DataFrame 'df_2' with top 10 rows:")
df_2.head(10)
```

Output:

```
DataFrame 'df_2' with top 10 rows:
```



	TransactionNo	product_category_list	productName_process_list	prod_len	path
13	536378	[0ca, 1ca, 0ca]	[StrawberryCharlotteBag, ChildrensCutleryRetro...	3	start > 0ca > 1ca > 0ca > conversion
27	536395	[0ca, 1ca, 0ca]	[BlackHeartCardHolder, AssortedColourBirdOrnam...	3	start > 0ca > 1ca > 0ca > conversion
36	536404	[0ca, 1ca, 0ca, 4ca, 0ca]	[HeartIvoryTrellisSmall, ClearDrawerKnobAcryli...	5	start > 0ca > 1ca > 0ca > 4ca > 0ca > conversion
40	536408	[0ca, 1ca, 0ca]	[MagicDrawingSlateDinosaur, MagicDrawingSlateB...	3	start > 0ca > 1ca > 0ca > conversion
42	536412	[0ca, 4ca, 0ca]	[RoundSnackBoxesSetOfWoodland, RoundSnackBoxes...	3	start > 0ca > 4ca > 0ca > conversion
43	536415	[0ca, 1ca, 0ca]	[CakeCasesVintageChristmas, PaperChainKitVinta...	3	start > 0ca > 1ca > 0ca > conversion
52	536464	[0ca, 1ca, 0ca]	[BlackSweetheartBracelet, DiamanteHairGripPack...	3	start > 0ca > 1ca > 0ca > conversion
72	536532	[0ca, 1ca, 0ca]	[BoxOfCocktailParasols, GrowYourOwnPlantInACan...	3	start > 0ca > 1ca > 0ca > conversion
82	536542	[0ca, 4ca]	[RecyclingBagRetrospot, JumboStorageBagSkulls,...	2	start > 0ca > 4ca > conversion
83	536544	[0ca, 1ca, 0ca, 4ca, 0ca]	[DecorativeRoseBathroomBottle, DecorativeCatsB...	5	start > 0ca > 1ca > 0ca > 4ca > 0ca > conversion

Answer:

Explanation of the Code and Logic Used:

- The code defines a function, 'data_processing', which processes a DataFrame to create a "path" format from the 'product_category_list'.
- It filters the DataFrame to include only rows where the product category list length ('prod_len') is between specified limits (2 to 5). This ensures meaningful transactions are analyzed.
- The function then constructs a new column, 'path', formatted as "start > category1 > ... > conversion," visually representing the customer journey.
- After applying this function to create 'df_2', the top 10 rows are printed to verify the transformations, providing insights into customer behavior and potential conversion pathways.
- The code filters the DataFrame by product category count, creating a new column that formats these categories into a path representing the customer's journey from start to conversion, named df_2.

Reason for using this solution: This approach effectively transforms the product category list into a clear path format, facilitating the visualization of customer journeys. By focusing on relevant transactions, businesses can better analyze customer navigation through product categories, optimize sales strategies, and identify potential conversion pathways, ultimately improving marketing efforts.

Alternative Solutions: An alternative approach could involve visualizing the customer journey paths using flowcharts or Sankey diagrams. These visual tools can provide an intuitive understanding of how customers move through product categories, enabling businesses to identify drop-off points or popular paths leading to conversion. This would complement the path format by offering a more engaging and interactive way to analyze customer behavior and enhance marketing strategies.

Optimality: This method is optimal for creating a structured view of customer interactions with product categories, allowing businesses to better understand purchasing behaviors and conversion pathways. By converting lists into a path format, the analysis becomes more insightful, enabling the identification of trends and strategies to enhance marketing effectiveness. This structured representation aids in refining targeted marketing campaigns and optimizing product placements, ultimately leading to improved sales and customer satisfaction.

Question 1.8:

Continue to work on the results of question 1.7, the dataframe 'df_2', we would like to build the transition matrix together, but before we actually conduct the programming, we will need to finish few questions for exploration:

Question 1.8.1

Check on your transaction level basket with results from question 1.7, could you please find out respectively how many transactions ended with pattern '... > 0ca > conversion' / '...> 1ca > conversion' / '... > 2ca > conversion' / '... > 3ca > conversion' / '... > 4ca > conversion' (1 result for each pattern, total 5 results are expected).

Code:

```
# Task 1.8.1: Count transactions ending with specific patterns
patterns_ending = ['0ca > conversion', '1ca > conversion', '2ca > conversion',
'3ca > conversion', '4ca > conversion']

# Check how many transactions end with these patterns
for pattern in patterns_ending:
    count = df_2['path'].apply(lambda x: x.endswith(pattern)).sum()
    print(f"Number of transactions ending with '{pattern}': {count}")
```

Output:

```
Number of transactions ending with '0ca > conversion': 3056
Number of transactions ending with '1ca > conversion': 26
Number of transactions ending with '2ca > conversion': 144
Number of transactions ending with '3ca > conversion': 68
Number of transactions ending with '4ca > conversion': 198
```

Answer:

Explanation of the Code and Logic Used:

- The code counts transactions in the DataFrame 'df_2' that end with specific conversion patterns.
- First, it defines a list of patterns, 'patterns_ending', which represent different paths leading to conversion.
- The code then iterates over each pattern, applying a lambda function to the 'path' column of 'df_2' to check if each path ends with the current pattern.
- The 'sum()' function aggregates the results, counting how many transactions match that criterion.
- Finally, it prints the count for each pattern, providing insights into the frequency of these specific conversion paths, which can inform marketing and customer journey analysis.

Reason for using this solution: The approach counts the transaction paths in df_2 that end with specific patterns, indicating various conversion stages. This is significant for evaluating customer journeys, helping businesses understand which paths successfully lead to conversions. By focusing on predefined patterns, the analysis offers a clear picture of the most effective transaction routes, aiding in refining marketing strategies and improving customer engagement. Understanding these pathways allows businesses to better align their offerings with customer behaviors and needs, ultimately enhancing sales outcomes.

Alternative Solutions: An alternative method could involve a more in-depth analysis that correlates the identified conversion patterns with additional metrics such as conversion rates, customer satisfaction, or retention. This could provide a richer understanding of customer behavior and preferences. Additionally, conducting a qualitative analysis through customer feedback or surveys could yield insights into why certain paths are more successful, allowing businesses to tailor their approaches accordingly. Combining quantitative and qualitative data may enhance strategic decision-making processes.

Optimality: This method is optimal as it allows businesses to quickly quantify the effectiveness of specific transaction paths leading to conversions. By providing actionable insights into customer behavior, the analysis can directly inform marketing strategies and optimization efforts. Understanding the frequency of these paths aids in resource allocation, ensuring that businesses can focus on the most successful routes for driving conversions. Ultimately, this method enhances decision-making, empowering organizations to implement targeted initiatives that align with customer preferences and boost sales performance.

Question 1.8.2

Check on your transaction level basket with results from question 1.7, could you please find out respectively how many times the transactions contains '0ca > 0ca' / '0ca > 1ca' / '0ca > 2ca' / '0ca > 3ca' / '0ca > 4ca' / '0ca > conversion' in the whole data (1 result for each pattern, total 6 results are expected and each transaction could contain those patterns multiple times, such as 'start > 0ca > 1ca > 0ca > 1ca > conversion' will count 'two' times with pattern '0ca > 1ca', if there is not any, then return 0, you need to sum the counts from each transaction to return the final value).

Code:

```
# Task 1.8.2: Count occurrences of specific patterns within the entire path
(can occur multiple times in a transaction)
patterns_within = ['0ca > 0ca', '0ca > 1ca', '0ca > 2ca', '0ca > 3ca', '0ca >
4ca', '0ca > conversion']

# Count occurrences of each pattern within the path and sum them up
for pattern in patterns_within:
    count = df_2['path'].apply(lambda x: x.count(pattern)).sum()
    print(f"Total occurrences of '{pattern}' in all transactions: {count}")
```

Output:

```
Total occurrences of '0ca > 0ca' in all transactions: 0
Total occurrences of '0ca > 1ca' in all transactions: 1222
Total occurrences of '0ca > 2ca' in all transactions: 1137
Total occurrences of '0ca > 3ca' in all transactions: 343
Total occurrences of '0ca > 4ca' in all transactions: 1198
Total occurrences of '0ca > conversion' in all transactions: 3056
```

Answer:

Explanation of the Code and Logic Used:

- The code counts occurrences of specific patterns within 'path' column of 'df_2' DataFrame, allowing for multiple occurrences within a single transaction.
- It first defines a list of patterns, 'patterns_within', which includes various sequences that may appear in paths.
- The code then iterates over each pattern, using lambda function to apply count() method on each 'path', which counts how many times the pattern appears.
- The sum() function aggregates these counts across all transactions, providing a total for each pattern.
- Finally, it prints total occurrences, which can help analyze customer behavior and product interactions in the conversion paths.

Reason for using this solution: This code counts occurrences of specific patterns within the transaction paths in the df_2 DataFrame, capturing multiple instances within a single path. This granularity allows for a more nuanced understanding of customer journeys, highlighting which sequences of interactions are most common. By analyzing these patterns, businesses can identify prevalent customer behaviors and preferences, providing insight into how often particular conversion paths are navigated. This understanding can inform marketing strategies and product placements, ultimately aiding in optimizing customer experiences and increasing conversion rates through tailored approaches.

Alternative Solutions: An alternative approach could involve using advanced visualization techniques, such as heatmaps or network graphs, to illustrate the frequency and relationships of these patterns within customer journeys. Visual representations can make it easier to identify trends and anomalies at a glance, facilitating discussions among stakeholders regarding strategic direction. Additionally, combining quantitative analysis with qualitative feedback from customer surveys can provide deeper insights into why certain patterns are more common. This mixed-method approach could enhance the overall understanding of customer behavior and preferences.

Optimality: This method is optimal for providing a detailed understanding of how frequently specific patterns appear across transactions. By quantifying these occurrences, businesses can pinpoint successful conversion strategies and identify potential bottlenecks in the customer journey. This data-driven approach supports informed decision-making, enabling marketing teams to refine their tactics based on observed patterns. Moreover, by focusing on how often these paths are traversed, organizations can better allocate resources and efforts toward enhancing customer engagement strategies, ultimately improving conversion rates and customer satisfaction.

Question 1.8.3

Check on your transaction level basket with results from task question 1.7, could you please find out how many times the transactions contains '...> 0ca > ...' in the whole data (1 result is expected and each transaction could contain the pattern multiple times, such as 'start > 0ca > 1ca > 0ca > 1ca > conversion' will count 'two' times, you need to sum the counts from each transaction to return the final value).

Code:

```
# 1.8.3 Count occurrences of the pattern '... > 0ca > ...'
total_0ca_pattern_count = df_2['path'].str.count('> 0ca >').sum()
print(f"Total occurrences of the pattern '... > 0ca > ...' in all transactions:
{total_0ca_pattern_count}")
```

Output:

```
Total occurrences of the pattern '... > 0ca > ...' in all transactions: 6956
```

Answer:

Explanation of the Code and Logic Used:

- The code counts occurrences of the pattern '> 0ca >' within the 'path' column of the 'df_2' DataFrame. The str.count() method is used to count how many times this specific pattern appears in each entry of the 'path'.
- The sum() function then aggregates these counts across all transactions. This analysis helps to identify how frequently the product category 0ca appears as part of a broader sequence in conversion paths, offering insights into customer behavior and product relationships during transactions.
- Finally, the total count is printed, providing a clear summary of this specific interaction.

Reason for using this solution: This code provides a focused analysis of how often the pattern '> 0ca >' appears within transaction paths. By counting this specific pattern, we gain valuable insights into a particular stage in the customer journey. Understanding how often customers navigate through this category can help us identify trends and potential bottlenecks in the purchasing process. This knowledge is crucial for refining our marketing strategies and improving customer experience, ensuring that we effectively guide customers towards conversion paths that include the 0ca category.

Alternative Solutions: Instead of concentrating solely on the 0ca pattern, we could expand our analysis to include other product categories using similar counting methods. By doing this, we could compare how different categories influence customer behavior and their impact on conversion rates. Moreover, we could incorporate qualitative feedback from customers to gain deeper insights into their preferences and experiences. This combined approach would provide a more comprehensive understanding of customer journeys and help identify key areas for improvement.

Optimality: This method effectively targets a specific pathway in transaction analysis, offering a granular view of how often the 0ca category is involved in customer interactions. By quantifying its occurrence in conversion paths, we can assess its significance in influencing purchasing decisions. This focused approach enables marketing teams to tailor strategies that leverage the strengths of the 0ca category, optimizing customer engagement and increasing the likelihood of conversions. Overall, this analysis is a key step in enhancing our understanding of customer behaviors and improving our marketing efforts.

Question 1.8.4

Use the 6 results from 1.8.2 to divide the result from 1.8.3 and then sum all of them and return the value.

Code:

```
# 1.8.4 Calculate the final value based on the previous counts
patterns_within = ['0ca > 0ca', '0ca > 1ca', '0ca > 2ca', '0ca > 3ca', '0ca >
4ca', '0ca > conversion']

# Calculate and print the counts for each pattern
pattern_counts = {}
for pattern in patterns_within:
    count = df_2['path'].apply(lambda x: x.count(pattern)).sum()
    pattern_counts[pattern] = count
    print(f"Pattern '{pattern}' occurs {count} times.")

print("\n")

# Task 1.8.4: Compute ratios and final sum
ratios = {}
for pattern, count in pattern_counts.items():
    ratio = count / total_0ca_pattern_count if total_0ca_pattern_count > 0 else
0
    ratios[pattern] = ratio
    print(f"Ratio for pattern '{pattern}' is: {ratio}")

# Sum the ratios to get the final value
final_value = sum(ratios.values())
print(f"Final value (sum of ratios): {final_value}")
```

Output:

```
Pattern '0ca > 0ca' occurs 0 times.
Pattern '0ca > 1ca' occurs 1222 times.
Pattern '0ca > 2ca' occurs 1137 times.
Pattern '0ca > 3ca' occurs 343 times.
Pattern '0ca > 4ca' occurs 1198 times.
Pattern '0ca > conversion' occurs 3056 times.


Ratio for pattern '0ca > 0ca' is: 0.0
Ratio for pattern '0ca > 1ca' is: 0.17567567567567569
Ratio for pattern '0ca > 2ca' is: 0.16345600920069006
Ratio for pattern '0ca > 3ca' is: 0.04930994824611846
Ratio for pattern '0ca > 4ca' is: 0.17222541690626797
Ratio for pattern '0ca > conversion' is: 0.43933294997124783
Final value (sum of ratios): 1.0
```

Answer:

Explanation of the Code and Logic Used:

- The segment calculates the frequency of specific patterns in transaction paths within the df_2 DataFrame, particularly focusing on '0ca'.
- It iterates over predefined patterns ('patterns_within'), counting occurrences using 'str.count()' and storing results in the 'pattern_counts' dictionary. Each count is printed for reference.

- Next, the code computes ratios of these counts to the total occurrences of the pattern '> 0ca >'. This ratio provides insights into each pattern's relative frequency.
- Finally, it sums these ratios to derive a final value, reflecting the significance of the patterns in relation to '0ca', enhancing understanding of customer behavior during transactions.

Reason for using this solution: This approach provides a clear quantitative overview of how specific patterns relate to the broader 0ca pathway in customer journeys. By calculating the occurrence counts and their ratios, we can discern which paths are most common and effective in guiding customers towards conversion. This data-driven insight is invaluable for optimizing marketing strategies and enhancing customer experiences. Ultimately, understanding these relationships allows us to refine our tactics and better meet customer needs throughout their purchasing journey.

Alternative Solutions: Instead of focusing solely on counts and ratios, we could explore a weighted scoring system that takes into account additional factors, such as conversion rates or customer feedback. This scoring could prioritize patterns based on their potential impact on sales, allowing for a more nuanced understanding of their effectiveness. Furthermore, incorporating visualization techniques like graphs could make the data more accessible and easier to interpret, leading to more informed strategic decisions.

Optimality: This method is optimal for distilling complex data into actionable insights, highlighting the significance of various patterns within the context of customer behavior. By summarizing these relationships through counts and ratios, we can effectively assess how different paths contribute to overall conversion rates. This focused analysis is essential for guiding marketing efforts and ensuring that we effectively engage customers at critical stages in their purchasing process, ultimately driving business growth.

Question 1.9:

Let’s now look at the question 1.6 again, you have the list of product and list of product category for each transaction. We will use the transactionNo and productName_process to conduct the Association rule learning.

Question 1.9.1

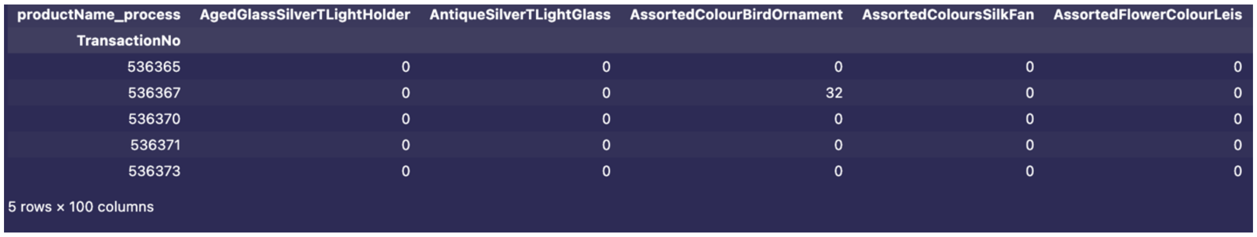
Work on the dataframe df from question 1.2 (filter out the transaction with negative quantity value and also only keep those top 100 products by ranking the sum of quantity) and build the transaction level product dataframe (each row represents transactionNo and productName_process become the columns, the value in the column is the Quantity). Hint: you might consider to use pivot function in pandas.

Code:

```
# 1.9.1 Filter out transactions with negative quantity values
df_filtered = df[df['Quantity'] > 0]

# Rank products based on sum of quantity, select top 100
top_products =
df_filtered.groupby('productName_process')['Quantity'].sum().nlargest(100).inde
x

# Filter the DataFrame to keep only the top 100 products
```

```
df_top_products =
df_filtered[df_filtered['productName_process'].isin(top_products)]

# Pivot the DataFrame to create a transaction-level view
df_transaction_level = df_top_products.pivot_table(index='TransactionNo',

                                             columns='productName_process',
                                                         values='Quantity',
                                                         aggfunc='sum',
                                                         fill_value=0)

# Show the transaction-level DataFrame
df_transaction_level.head()
```

Output:



productName_process	AgedGlassSilverTLightHolder	AntiqueSilverTLightGlass	AssortedColourBirdOrnament	AssortedColoursSilkFan	AssortedFlowerColourLeis
TransactionNo					
536365	0	0	0	0	0
536367	0	0	32	0	0
536370	0	0	0	0	0
536371	0	0	0	0	0
536373	0	0	0	0	0

5 rows × 100 columns

Answer:

Explanation of the Code and Logic Used:

- The code processes the DataFrame 'df' to analyze product transactions based on quantity.
- It filters out transactions with negative quantities, ensuring only valid sales data is included. This is crucial for accurate analysis, as negative values may indicate returns or errors.
- Next, it ranks products by summing their quantities to identify top 100 products with highest sales volume using 'nlargest(100)'.
- Finally, it pivots DataFrame to create a transaction-level view, where each row represents a transaction and each column represents a product, summing quantities and filling missing values with zero.

Reason for using this solution: This approach is essential for ensuring data integrity by filtering out negative quantities, which could skew insights. Ranking the products by quantity sold gives a clear indication of which items are performing best, allowing for strategic decision-making. The pivot table then organizes this data into a user-friendly format, making it easier to analyze transactions at a glance. This comprehensive overview is particularly beneficial for identifying trends, informing inventory management, and enhancing marketing strategies based on top-selling products.

Alternative Solutions: Instead of solely focusing on quantity, we could explore analyzing sales based on revenue generated, which might provide a different perspective on product performance. Additionally, incorporating customer demographics or transaction time could enrich the analysis, allowing us to identify patterns related to specific customer segments or seasonal trends. Leveraging visualization techniques could also enhance understanding, making it easier to communicate insights to stakeholders.

Optimality: This method is optimal for delivering a clear, concise view of product performance across transactions. By emphasizing valid data and structuring it into an easily interpretable format, we can quickly derive actionable insights. This streamlined analysis not only aids in tracking sales effectiveness but also informs future strategies, ensuring that business decisions are data-driven and aligned with customer preferences.

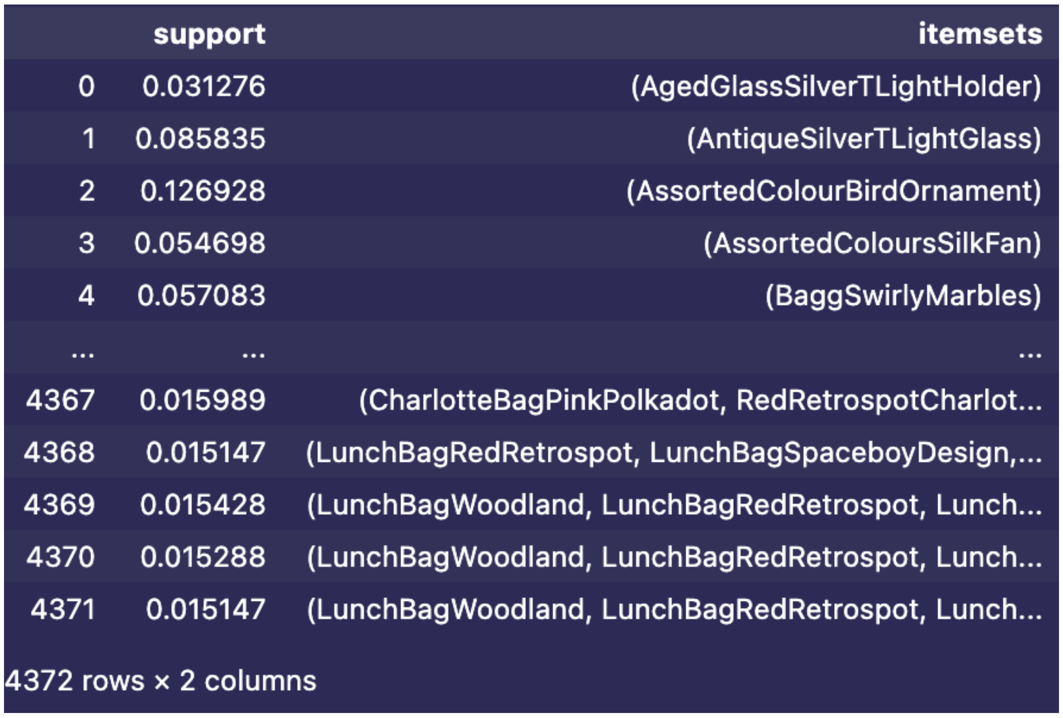
Question 1.9.2

Run the apriori algorithm to identify items with minimum support of 1.5% (only looking at baskets with 4 or more items). Hint: you might consider to use mlxtend.frequent_patterns to run apriori rules.

Code:

```
#1.9.2
# Binarize the DataFrame (Convert quantities to binary (1 if present, else 0))
df_transaction_level = df_transaction_level.map(lambda x: 1 if x > 0 else 0)

# Filter transactions with 4 or more items
df_transaction_level['num_items'] = (df_transaction_level > 0).sum(axis=1)
df_filtered_baskets = df_transaction_level[df_transaction_level['num_items'] >=
4].drop(columns='num_items')
df_filtered_baskets_boolean = df_filtered_baskets.astype(bool)

# Run Apriori algorithm with minimum support of 1.5%
frequent_itemsets = apriori(df_filtered_baskets_boolean, min_support=0.015,
use_colnames=True)

# Show frequent itemsets
frequent_itemsets
```

Output:



	support	itemsets
0	0.031276	(AgedGlassSilverTLightHolder)
1	0.085835	(AntiqueSilverTLightGlass)
2	0.126928	(AssortedColourBirdOrnament)
3	0.054698	(AssortedColoursSilkFan)
4	0.057083	(BaggSwirlyMarbles)
...	...	...
4367	0.015989	(CharlotteBagPinkPolkadot, RedRetrospotCharlot...
4368	0.015147	(LunchBagRedRetrospot, LunchBagSpaceboyDesign,...
4369	0.015428	(LunchBagWoodland, LunchBagRedRetrospot, Lunch...
4370	0.015288	(LunchBagWoodland, LunchBagRedRetrospot, Lunch...
4371	0.015147	(LunchBagWoodland, LunchBagRedRetrospot, Lunch...

4372 rows × 2 columns

Answer:

Explanation of the Code and Logic Used:

- The code analyzes transaction data using the Apriori algorithm to identify frequent itemsets.
- It starts by binarizing the DataFrame 'df_transaction_level', converting quantities greater than zero to 1 and others to 0, preparing the data for association rule mining.
- Next, it filters transactions to retain only those with four or more items by counting products in each transaction with a new column, 'num_items'.
- After filtering, it drops this column. Finally, the Apriori algorithm is applied with a minimum support of 1.5%, identifying frequent itemsets from the filtered transaction data. The results are stored in 'frequent_itemsets'.

Reason for using this solution: This approach is particularly valuable for businesses as it enables them to uncover crucial product affinities and gain a deeper understanding of customer purchasing behavior. By binarizing the DataFrame and focusing on transactions containing at least four items, the analysis targets significant purchase patterns that can reveal how different products are related. This insight is essential for inventory management, allowing businesses to stock items that are frequently purchased together, and it informs targeted marketing strategies that can boost sales performance. Overall, this data-driven approach aids companies in making informed decisions that enhance customer satisfaction and loyalty.

Alternative Solutions: An alternative approach could involve adjusting the minimum support threshold to examine how varying levels of support impact the results, potentially revealing less common but still significant associations. Additionally, utilizing algorithms like FP-Growth can be more efficient, especially with larger datasets, as it can handle high-dimensional data more effectively. Another option is to explore clustering techniques, which could segment transactions based on purchasing patterns, providing further insights into customer preferences and behaviors. These alternatives can complement the Apriori algorithm by offering different perspectives on the same data.

Optimality: This method is optimal as it efficiently uncovers valuable associations between products, significantly aiding in market basket analysis and informing strategic business decisions. By identifying co-purchased items, businesses can develop targeted promotional strategies that enhance customer engagement and drive cross-selling opportunities. Furthermore, these insights inform inventory decisions, ensuring that popular product combinations are always available. The thoughtful combination of filtering criteria and the implementation of the Apriori algorithm guarantees that the insights generated are not only actionable but also relevant, ultimately contributing to improved sales performance and customer satisfaction.

Question 1.9.3

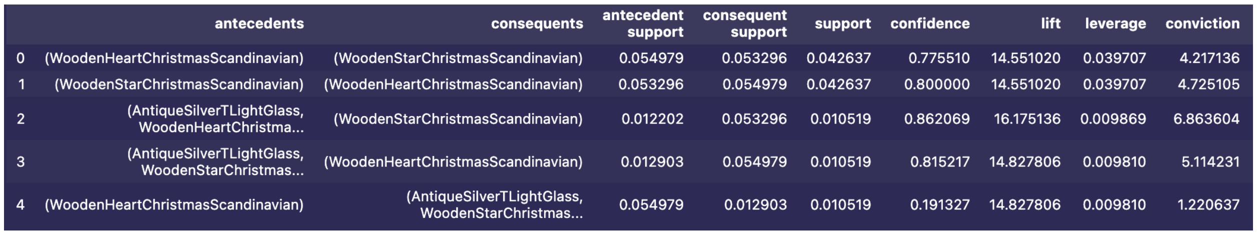
Run the apriori algorithm to find the items with support >= 1.0% and lift > 10.

Code:

```
#1.9.3
# Run Apriori algorithm with minimum support of 1.0%
frequent_itemsets_support_1 = apriori(df_filtered_baskets_boolean,
min_support=0.01, use_colnames=True)
```

```
# Generate association rules with lift greater than 10
rules = association_rules(frequent_itemsets_support_1, metric="lift",
min_threshold=10)

# Show rules with lift greater than 10
rules[rules['lift'] > 10].head()
```

Output:



	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(WoodenHeartChristmasScandinavian)	(WoodenStarChristmasScandinavian)	0.054979	0.053296	0.042637	0.775510	14.551020	0.039707	4.217136
1	(WoodenStarChristmasScandinavian)	(WoodenHeartChristmasScandinavian)	0.053296	0.054979	0.042637	0.800000	14.551020	0.039707	4.725105
2	(AntiqueSilverTLightGlass, WoodenHeartChristma...	(WoodenStarChristmasScandinavian)	0.012202	0.053296	0.010519	0.862069	16.175136	0.009869	6.863604
3	(AntiqueSilverTLightGlass, WoodenStarChristmas...	(WoodenHeartChristmasScandinavian)	0.012903	0.054979	0.010519	0.815217	14.827806	0.009810	5.114231
4	(WoodenHeartChristmasScandinavian)	(AntiqueSilverTLightGlass, WoodenStarChristmas...	0.054979	0.012903	0.010519	0.191327	14.827806	0.009810	1.220637

Answer:

Explanation of the Code and Logic Used:

- The code continues the analysis of transaction data by applying the Apriori algorithm and generating association rules.
- It first runs the Apriori algorithm on the binarized DataFrame 'df_filtered_baskets_boolean' with a minimum support threshold of 1.0%, identifying frequent itemsets.
- Next, it generates association rules from these itemsets using the 'association_rules' function, focusing on rules with a lift greater than 10, which indicates strong associations between items.
- Finally, the code displays the top rules that meet this lift criterion, providing insights into significant product combinations that could inform marketing strategies and inventory management.

Reason for using this solution: This approach is beneficial as it uncovers valuable insights into customer purchasing behavior. By applying the Apriori algorithm with a lower minimum support threshold of 1.0%, a wider range of itemsets can be identified. Focusing on rules with a lift greater than 10 ensures that the identified relationships are not just frequent but also meaningful. This understanding can inform targeted marketing strategies, promotional activities, and inventory management, leading to more effective sales performance and enhanced customer satisfaction.

Alternative Solutions: An alternative approach could involve experimenting with different metrics for rule generation, such as confidence or conviction, to identify other meaningful relationships between items. Adjusting the minimum support threshold could also reveal more itemsets, facilitating a more nuanced analysis. Additionally, one might consider integrating clustering techniques to group similar transactions, enabling deeper insights into customer segments and preferences.

Optimality: This method is optimal for uncovering strong product associations that can drive business strategies. By identifying combinations of products that frequently co-occur in transactions, businesses can create effective product recommendations and promotions, enhancing customer experience. The focus on high-lift associations ensures that the insights gained are actionable and relevant, making it easier to optimize inventory decisions and marketing campaigns for improved sales performance.

Question 1.9.4

Please explore three more examples with different support/confidence/ lift measurements (you could leverage your rule mining with one of the three measurements or all of them) to find out any of the interesting patterns from the Association rule learning. Save your code and results in a clean and tidy format and writing down your insights.

Code:

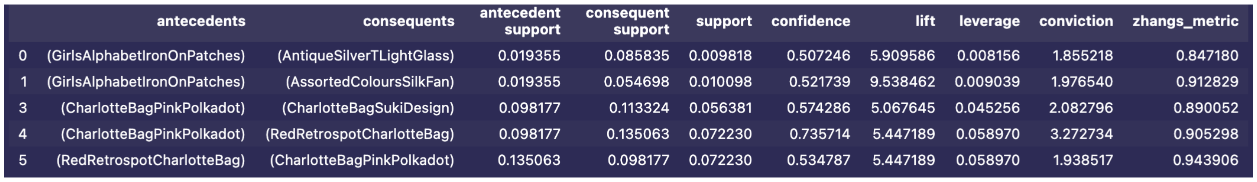
```
#Example 1 - Support >= 0.5%, Confidence >= 50%, Lift > 5

# Apply the Apriori algorithm with a minimum support of 0.5%
frequent_itemsets_support_05 = apriori(df_filtered_baskets_boolean,
min_support=0.005, use_colnames=True)



# Generate association rules based on the frequent itemsets with confidence >=
50%
rules_support_05_conf_50_lift_5 =
association_rules(frequent_itemsets_support_05, metric="confidence",
min_threshold=0.5)

# Filter rules to retain only those with lift greater than 5
rules_support_05_conf_50_lift_5 =
rules_support_05_conf_50_lift_5[rules_support_05_conf_50_lift_5['lift'] > 5]


#Example 2 - Support >= 2%, Confidence >= 60%, Lift > 7

# Apply the Apriori algorithm with a minimum support of 2%
frequent_itemsets_support_2 = apriori(df_filtered_baskets_boolean,
min_support=0.02, use_colnames=True)

# Generate association rules based on the frequent itemsets with confidence >=
60%
rules_support_2_conf_60_lift_7 = association_rules(frequent_itemsets_support_2,
metric="confidence", min_threshold=0.6)

# Filter rules to retain only those with lift greater than 7
rules_support_2_conf_60_lift_7 =
rules_support_2_conf_60_lift_7[rules_support_2_conf_60_lift_7['lift'] > 7]


#Example 3 - Support >= 1%, Confidence >= 70%, Lift > 8

# Apply the Apriori algorithm with a minimum support of 1%
frequent_itemsets_support_1_example3 = apriori(df_filtered_baskets_boolean,
min_support=0.01, use_colnames=True)

# Generate association rules based on the frequent itemsets with confidence >=
70%
rules_support_1_conf_70_lift_8 =
association_rules(frequent_itemsets_support_1_example3, metric="confidence",
min_threshold=0.7)

# Filter rules to retain only those with lift greater than 8
rules_support_1_conf_70_lift_8 =
rules_support_1_conf_70_lift_8[rules_support_1_conf_70_lift_8['lift'] > 8]
```

Code:

```
# Print the first few rows of the association rules generated in Example 1 -
Support >= 0.5%, Confidence >= 50%, Lift > 5
print("Example 1:")
rules_support_05_conf_50_lift_5.head()
```

Output:

```
Example 1:
```



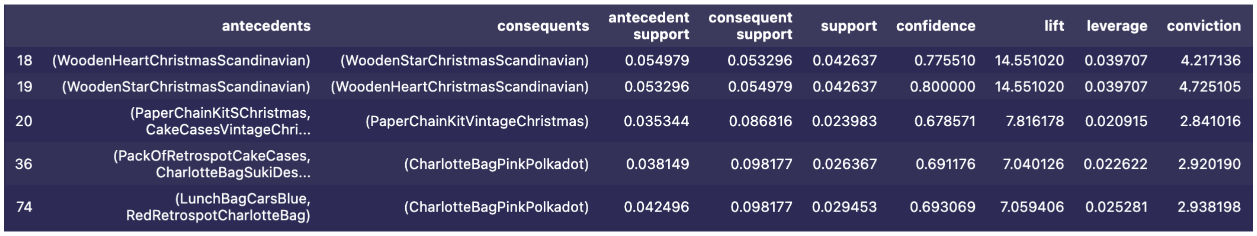
	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zhangs_metric
0	(GirlsAlphabetIronOnPatches)	(AntiqueSilverTLightGlass)	0.019355	0.085835	0.009818	0.507246	5.909586	0.008156	1.855218	0.847180
1	(GirlsAlphabetIronOnPatches)	(AssortedColoursSilkFan)	0.019355	0.054698	0.010098	0.521739	9.538462	0.009039	1.976540	0.912829
3	(CharlotteBagPinkPolkadot)	(CharlotteBagSukiDesign)	0.098177	0.113324	0.056381	0.574286	5.067645	0.045256	2.082796	0.890052
4	(CharlotteBagPinkPolkadot)	(RedRetrospotCharlotteBag)	0.098177	0.135063	0.072230	0.735714	5.447189	0.058970	3.272734	0.905298
5	(RedRetrospotCharlotteBag)	(CharlotteBagPinkPolkadot)	0.135063	0.098177	0.072230	0.534787	5.447189	0.058970	1.938517	0.943906

Code:

```
# Print the first few rows of the association rules generated in Example 2 -
Support >= 2%, Confidence >= 60%, Lift > 7
print("Example 2:")
rules_support_2_conf_60_lift_7.head()
```

Output:

```
Example 2:
```



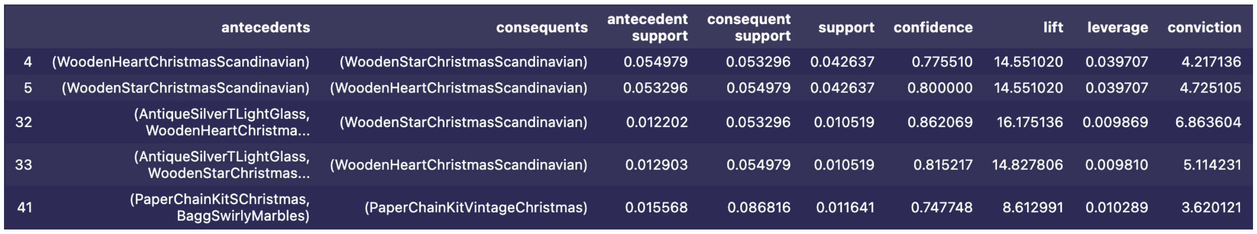
	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
18	(WoodenHeartChristmasScandinavian)	(WoodenStarChristmasScandinavian)	0.054979	0.053296	0.042637	0.775510	14.551020	0.039707	4.217136
19	(WoodenStarChristmasScandinavian)	(WoodenHeartChristmasScandinavian)	0.053296	0.054979	0.042637	0.800000	14.551020	0.039707	4.725105
20	(PaperChainKitSChristmas, CakeCasesVintageChri...	(PaperChainKitVintageChristmas)	0.035344	0.086816	0.023983	0.678571	7.816178	0.020915	2.841016
36	(PackOfRetrospotCakeCases, CharlotteBagSukiDes...	(CharlotteBagPinkPolkadot)	0.038149	0.098177	0.026367	0.691176	7.040126	0.022622	2.920190
74	(LunchBagCarsBlue, RedRetrospotCharlotteBag)	(CharlotteBagPinkPolkadot)	0.042496	0.098177	0.029453	0.693069	7.059406	0.025281	2.938198

Code:

```
# Print the first few rows of the association rules generated in Example 3 -
Support >= 1%, Confidence >= 70%, Lift > 8
print("Example 3:")
rules_support_1_conf_70_lift_8.head()
```

Output:

```
Example 3:
```



	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
4	(WoodenHeartChristmasScandinavian)	(WoodenStarChristmasScandinavian)	0.054979	0.053296	0.042637	0.775510	14.551020	0.039707	4.217136
5	(WoodenStarChristmasScandinavian)	(WoodenHeartChristmasScandinavian)	0.053296	0.054979	0.042637	0.800000	14.551020	0.039707	4.725105
32	(AntiqueSilverTLightGlass, WoodenHeartChristma...	(WoodenStarChristmasScandinavian)	0.012202	0.053296	0.010519	0.862069	16.175136	0.009869	6.863604
33	(AntiqueSilverTLightGlass, WoodenStarChristmas...	(WoodenHeartChristmasScandinavian)	0.012903	0.054979	0.010519	0.815217	14.827806	0.009810	5.114231
41	(PaperChainKitSChristmas, BaggSwirlyMarbles)	(PaperChainKitVintageChristmas)	0.015568	0.086816	0.011641	0.747748	8.612991	0.010289	3.620121

Answer:

Explanation of the Code and Logic Used:

- The code applies the Apriori algorithm to uncover meaningful product associations, enabling businesses to improve strategies like cross-selling and targeted marketing. It performs three separate analyses with varying thresholds for support, confidence, and lift, refining the identification of significant product relationships.

- **Example 1** examines product rules with a minimum support of 0.5%, confidence of 50%, and lift greater than 5. This allows for the discovery of moderately strong associations between products.

- **Example 2** raises the thresholds to 2% support, 60% confidence, and lift greater than 7. This focuses on identifying stronger and more frequent associations between products.

- **Example 3** sets more stringent criteria, requiring at least 1% support, 70% confidence, and a lift higher than 8. This analysis seeks highly reliable product combinations that are frequently co-purchased with high confidence.

- For each example, the top association rules are printed to reveal the strongest product relationships, providing insights into consumer purchasing patterns.

Reason for using this solution: This solution is ideal for uncovering different levels of product associations by adjusting support, confidence, and lift criteria. By varying these thresholds, the code uncovers a broad spectrum of relationships, from general product trends to highly confident co-purchases. This flexibility provides a more detailed and segmented view of customer behavior, helping businesses to optimize promotions, enhance product bundling, and develop more targeted marketing strategies.

Alternative Solutions: Alternative approaches could involve using other metrics like leverage or conviction, which offer additional perspectives on the strength and significance of product relationships. Additionally, further adjusting the support and confidence thresholds could help focus on different levels of product interaction, such as rare but highly significant combinations or more frequent, lower-confidence associations. Combining these techniques with clustering methods could offer a more holistic understanding of purchasing behaviors across various customer segments.

Optimality: This method is optimal because it effectively balances depth and breadth in analyzing product associations. The use of varying support, confidence, and lift thresholds allows for a more comprehensive exploration of product relationships, making it easier to identify key trends and insights. This, in turn, enables more precise business decisions, such as targeted marketing campaigns, better product recommendations, and improved inventory management. The flexibility of this approach ensures that it can be adapted to different business needs and goals, making it highly valuable for businesses looking to leverage customer purchasing data.

Question 1.10:

After we finished the Association rule learning, it is a time for us to consider to do customer analysis based on their shopping behaviours.

Question 1.10.1

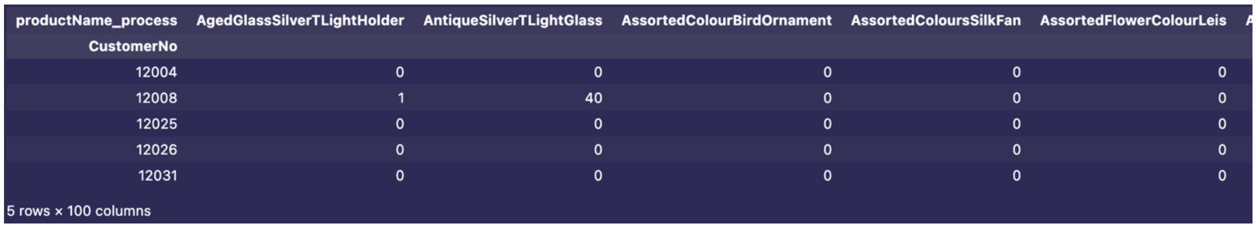
Work on the dataframe df from question 1.2 and build the customer product dataframe (each row represents single customerNo and productName_process become as the columns, the value in the columns is the aggregated Quantity value from all transactions and the result is a N by M matrix where N is the number of distinct customerNo and M is the number of distinct productName_process. Please filter out the transaction with negative quantity value and also only keep those top 100 product by ranking the sum of quantity).

Code:

```
# Assuming df from Question 1.2 is already available
# Filter out negative quantities
df_filtered = df[df['Quantity'] > 0]

# Get the top 100 products by aggregated quantity
top_products =
df_filtered.groupby('productName_process')['Quantity'].sum().nlargest(100).inde
x

# Filter to only keep top 100 products
df_top_products =
df_filtered[df_filtered['productName_process'].isin(top_products)]

# Create Customer-Product DataFrame
customer_product_df = df_top_products.pivot_table(
    index='CustomerNo',
    columns='productName_process',
    values='Quantity',
    aggfunc='sum',
    fill_value=0
)

print("Customer-Product DataFrame:")
customer_product_df.head()
```

Output:

```
Customer-Product DataFrame:
```



productName_process	AgedGlassSilverTLightHolder	AntiqueSilverTLightGlass	AssortedColourBirdOrnament	AssortedColoursSilkFan	AssortedFlowerColourLeis
CustomerNo					
12004	0	0	0	0	0
12008	1	40	0	0	0
12025	0	0	0	0	0
12026	0	0	0	0	0
12031	0	0	0	0	0

5 rows × 100 columns

Answer:

Explanation of the Code and Logic Used:

- This processes a DataFrame to analyze customer-product interactions.
- It removes transactions with negative quantities, ensuring only valid purchases are considered.
- It groups the filtered data by product name, summing the quantities, and selects the top 100 products based on total quantity sold. This step helps focus on the most popular items.
- It creates a pivot table where each row represents a customer, and each column corresponds to a product. The values indicate the total quantity purchased by each customer for each product, filling missing values with zeros.
- Finally, prints first few rows of resulting customer-product DataFrame for inspection.

Reason for using this solution: This solution is particularly valuable as it focuses on analyzing only positive quantities, ensuring that the dataset is free of erroneous or misleading data such as returns or mistakes. By ranking products based on their quantity sold and narrowing down to the top 100, the analysis becomes more targeted, concentrating on the most significant products that drive the majority of sales. This selective approach helps businesses to understand which products have the highest demand, allowing for more strategic decisions in inventory management, product placement, and promotional activities, all of which are critical for enhancing sales performance and customer satisfaction.

Alternative Solutions: Alternative approaches could involve analyzing the entire range of products without limiting it to the top 100. This might reveal niche items or long-tail products that contribute to profitability in less obvious ways. Another option is to rank products based on different metrics, such as revenue or profit margins, which would highlight high-value or high-profit items rather than just those sold in large quantities. Additionally, segmenting the analysis by customer demographics, seasonality, or geographic region could uncover more tailored insights, helping businesses refine their marketing efforts and stock management for specific segments or time frames.

Optimality: This method is optimal for businesses seeking a concise and actionable analysis of their top-performing products. By filtering out irrelevant or negative quantities and focusing on high-demand items, the analysis remains manageable yet powerful in identifying key drivers of sales. The insights gleaned from such an approach are directly applicable to refining inventory strategies, enhancing marketing campaigns, and ensuring that high-demand products are adequately stocked and promoted. This approach balances simplicity with effectiveness, ensuring that the results are both relevant and immediately useful for operational and strategic decision-making.

Question 1.10.2

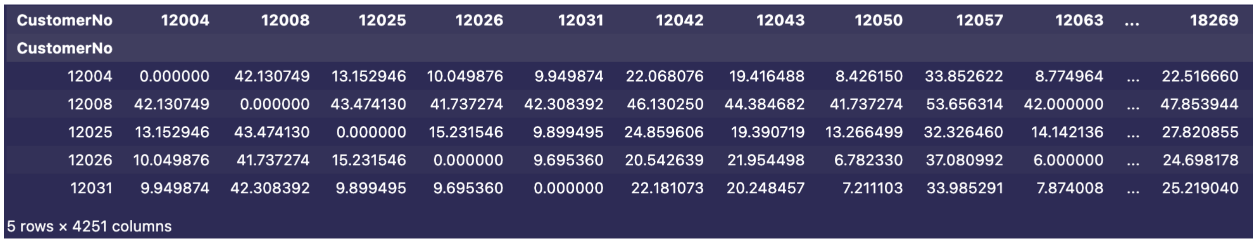
Use the customer-product dataframe, let’s calculate the Pairwise Euclidean distance on customer level (you will need to use the product Quantity information on each customer to calculate the Euclidean distance for all other customers and the result is a N by N matrix where N is the number of distinct customerNo).

Code:

```
# Calculate pairwise Euclidean distances
distances = pdist(customer_product_df, metric='euclidean')
distance_matrix = squareform(distances)

# Create a DataFrame for better visualization
distance_df = pd.DataFrame(distance_matrix, index=customer_product_df.index,
columns=customer_product_df.index)
print("Pairwise Euclidean Distance Matrix:")
distance_df.head()
```

Output:

```
Pairwise Euclidean Distance Matrix:
```



CustomerNo	12004	12008	12025	12026	12031	12042	12043	12050	12057	12063	...	18269
CustomerNo												
12004	0.000000	42.130749	13.152946	10.049876	9.949874	22.068076	19.416488	8.426150	33.852622	8.774964	...	22.516660
12008	42.130749	0.000000	43.474130	41.737274	42.308392	46.130250	44.384682	41.737274	53.656314	42.000000	...	47.853944
12025	13.152946	43.474130	0.000000	15.231546	9.899495	24.859606	19.390719	13.266499	32.326460	14.142136	...	27.820855
12026	10.049876	41.737274	15.231546	0.000000	9.695360	20.542639	21.954498	6.782330	37.080992	6.000000	...	24.698178
12031	9.949874	42.308392	9.899495	9.695360	0.000000	22.181073	20.248457	7.211103	33.985291	7.874008	...	25.219040

5 rows × 4251 columns

Answer:

Explanation of the Code and Logic Used:

- The code calculates pairwise Euclidean distances between customers based on their product purchase quantities. It uses the 'pdist' function from 'scipy.spatial.distance' to compute these distances, specifying `metric='euclidean'`.
- The resulting condensed distance vector is transformed into a square distance matrix using the 'squareform' function, which enables easy comparison between every pair of customers.
- A new DataFrame, 'distance_df', is created from this matrix, with customer identifiers as both rows and columns for better readability.
- Finally, the first few rows of the 'distance_df' are printed, providing insights into customer similarities based on their purchasing behavior.

Reason for using this solution: This solution is valuable because it computes pairwise Euclidean distances between customers, allowing for an effective understanding of customer behavior based on their purchase quantities. By leveraging this distance metric, businesses can identify customers with similar purchasing patterns, which can help in segmentation, creating targeted marketing campaigns, or tailoring product recommendations. The Euclidean distance provides a straightforward measure of similarity or difference in behavior, making it a practical tool for clustering and understanding relationships between different customers in a clear, interpretable manner.

Alternative Solutions: Other distance metrics, such as Manhattan or Cosine, could be used depending on the nature of the data and the specific goals of the analysis. For example, Manhattan distance might be better suited when you need to account for absolute differences in behavior, while Cosine similarity might be useful if the focus is on the directionality of the customer's purchase preferences rather than their magnitude. Additionally, using other methods like hierarchical clustering or K-Means after calculating distances could provide further insights into groupings and customer segmentation.

Optimality: This method is optimal because the Euclidean distance metric is widely recognized for its effectiveness in measuring similarity between customers when the focus is on purchase quantities. It allows for easy visualization of customer relationships, which can be applied to enhance customer experience, optimize marketing strategies, and improve recommendation systems. Furthermore, the approach is computationally efficient and interpretable, making it a suitable choice for most customer similarity analyses, especially when seeking a balance between simplicity and effectiveness.

Question 1.10.3

Use the customer Pairwise Euclidean distance to find out the top 3 most similar customer to CustomerNo == 13069 and CustomerNo == 17490.

Code:

```
# Find top 3 most similar customers to CustomerNo == 13069
customer_13069_distances = distance_df.loc[13069]
top_3_similar_13069 = customer_13069_distances.nsmallest(4).index[1:]  # 
Exclude self

# Find top 3 most similar customers to CustomerNo == 17490
customer_17490_distances = distance_df.loc[17490]
top_3_similar_17490 = customer_17490_distances.nsmallest(4).index[1:]  # 
Exclude self

print("Top 3 most similar customers to 13069:", top_3_similar_13069.tolist())
print("Top 3 most similar customers to 17490:", top_3_similar_17490.tolist())
```

Output:

```
Top 3 most similar customers to 13069: [15118, 17523, 18179]
Top 3 most similar customers to 17490: [12519, 12582, 12652]
```

Answer:

Explanation of the Code and Logic Used:

- It identifies the top three most similar customers to two specified customers based on their purchase behaviors.
- It first retrieves the distance values for CustomerNo 13069 from the distance matrix and finds the three closest customers using the 'nsmallest' method, excluding the customer itself by slicing the index.
- The same process is repeated for CustomerNo 17490.

- Finally, it prints the results as lists, showcasing which customers exhibit the most similar purchasing patterns to the specified ones. This analysis can inform targeted marketing strategies or recommendations by highlighting closely aligned customer behaviors.

Reason for using this solution: This solution is valuable because it identifies the most similar customers to two specified customers, enabling businesses to personalize recommendations and marketing efforts. By focusing on the top three closest customers, it narrows down potential leads for targeted strategies, such as personalized promotions, cross-selling, or loyalty programs. The use of pairwise Euclidean distances ensures that similarities in purchase behavior are measured effectively, helping to build tailored customer experiences and improve retention rates.

Alternative Solutions: An alternative approach could involve clustering algorithms like K-Means or hierarchical clustering, which would group customers based on their purchasing behaviors rather than focusing on pairwise distances. This would provide broader insights into customer segments, allowing for a more scalable strategy when handling larger datasets. Additionally, implementing more advanced similarity metrics like Cosine similarity could offer better performance in cases where the direction of customer preferences (rather than quantity) is important.

Optimality: This method is optimal because it offers a straightforward way to identify customers with similar purchase behaviors, making it easier to apply personalized strategies on an individual basis. By leveraging specific similarities, businesses can enhance the effectiveness of targeted marketing campaigns, improve recommendation systems, and create loyalty incentives. Additionally, the focus on individual customer pairs is computationally efficient, making it well-suited for smaller datasets or highly personalized customer interactions.

Question 1.10.4

For the customer CustomerNo == 13069, you could see there are some products that this customer has never shopped before, could you please give some suggestions on how to recommend these product to this customer? please write down your suggestions and provide a coding logic (steps on how to achieve, not actual code).

Answer:

Recommendations for CustomerNo 13069:

To provide personalized product recommendations for CustomerNo 13069, we can implement the following detailed steps:

1. Identify Purchased Products: Compile a comprehensive list of products that CustomerNo 13069 has previously purchased. This serves as the foundation for determining potential gaps in their shopping behavior.

2. Analyze Similar Customers: Review the purchase histories of the top similar customers identified earlier. Compile a list of all the products these similar customers have bought. This comparison will highlight any products that CustomerNo 13069 has not yet explored.

3. Comparison of Purchase Histories: Cross-reference the list of products purchased by CustomerNo 13069 with the complete list from the similar customers. This will allow

us to pinpoint specific items that have been overlooked by CustomerNo 13069 but are popular among similar shoppers.

4. Examine Product Attributes: Investigate the characteristics of both purchased and unpurchased products (e.g., categories, features, price ranges). Look for trends that might indicate potential interest, such as a preference for electronic items. If CustomerNo 13069 typically buys gadgets, suggest similar electronic products they haven't yet tried.

5. Segment Customer Behavior: Categorize CustomerNo 13069 based on shopping patterns (e.g., frequency of purchases, types of products bought). This segmentation helps tailor recommendations more effectively and provides insights into what other customers with similar profiles are purchasing.

6. Leverage Collaborative Filtering: Utilize collaborative filtering techniques to recommend products that similar customers have bought. Focus on items that resonate with CustomerNo 13069's interests and past behaviors, enhancing the likelihood of engagement.

7. Implement Recommendation Algorithms: Apply advanced recommendation algorithms such as Content-Based Filtering (which suggests similar items based on product attributes) and Collaborative Filtering (which recommends items based on the behaviors of similar users). These algorithms will aid in uncovering unpurchased products that align with CustomerNo 13069's preferences.

8. Promote Bundled Products: If there are ongoing promotions or bundled offers, recommend these items to CustomerNo 13069, particularly those that complement items they have already purchased. Bundling can encourage additional purchases and enhance customer satisfaction.

9. Solicit Customer Feedback: Encourage CustomerNo 13069 to provide feedback on the recommendations. Analyzing this feedback can inform future suggestions and allow for continuous improvement of the recommendation engine, ensuring it remains aligned with customer preferences.

Coding Logic Steps (Pseudocode):

Step 1. Load transaction data and product data.

Step 2. Filter transactions to get products purchased by CustomerNo 13069.

Step 3. Create a list of all unique products available.

Step 4. Unpurchased Products = All Products bought by other similar customers - Purchased Products by CustomerNo 13069

Step 5. For each unpurchased product, check if its attributes match those of previously purchased products.

Step 6. Determine the customer segment for CustomerNo 13069 based on their purchase patterns.

Step 7. Identify similar customers and their unpurchased products. Recommend these products to CustomerNo 13069.

Step 8. Use the filtered list of unpurchased products to suggest them to CustomerNo 13069 based on the analysis and collaborative filtering results.

Step 9. Generate a final list of recommended products and display it to the customer.

Step 10. Record customer feedback on recommended products to improve future recommendations.

By following these steps, we can effectively recommend new products to CustomerNo 13069 that they haven't yet explored, while also aligning with their shopping behavior and preferences.

Question 2:

Question 2.1

You are required to explore the revenue time series. There are some days not available in the revenue time series such as 2019-01-01. Please add those days into the revenue time series with default revenue value with the mean value of the revenue in the whole data (without any filtering on transactions). After that, decompose the revenue time series with addictive mode and analyses on the results to find if there is any seasonality pattern (you could leverage the M05A material from lab session with default setting in seasonal_decompose function).

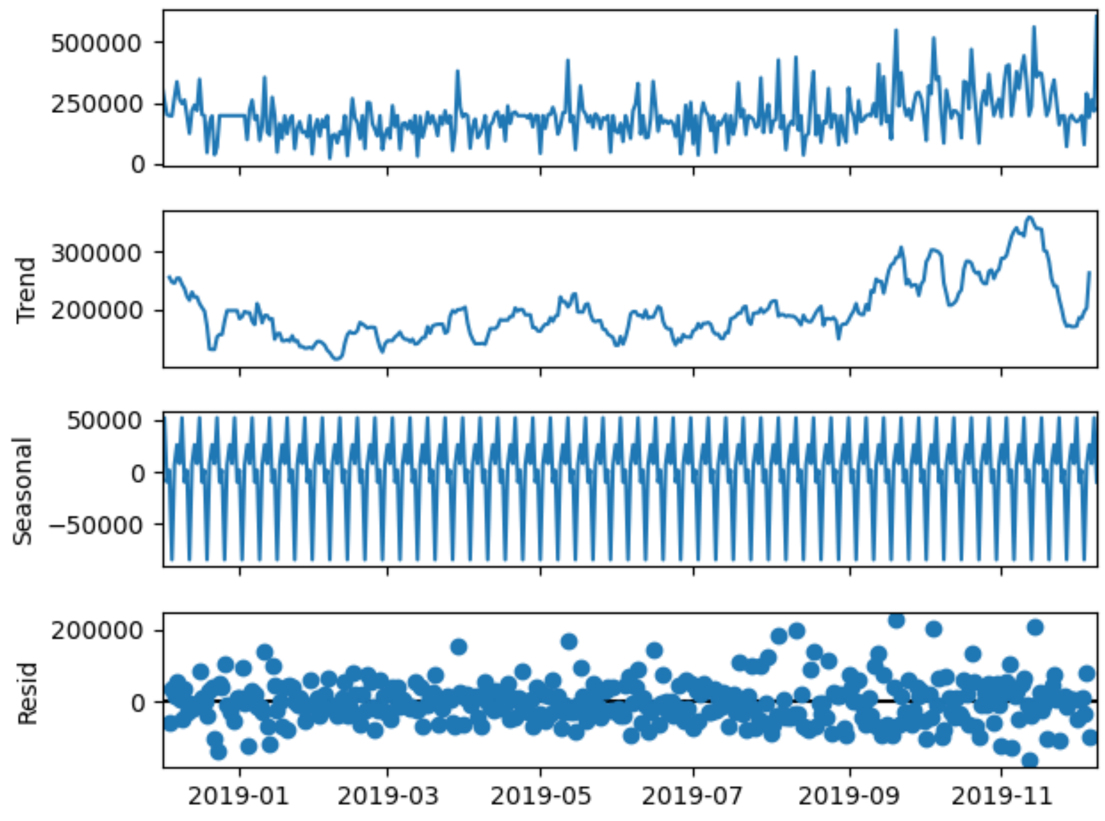
Code:

```
# converted Date column to datetime data type
df['Date'] = pd.to_datetime(df['Date'])

# Group by transaction date and sum revenue
# Create a Time Series of Revenue by Transaction Date
# aggregate the revenue by transaction date
revenue_time_series = df.groupby('Date')['Revenue'].sum().reset_index()
revenue_time_series.set_index('Date', inplace=True)

# find the mean value of the revenue
mean_revenue = revenue_time_series['Revenue'].mean()
print("Mean Revenue: ",mean_revenue)

 # Add Missing Dates
 # you'll want to reindex the DataFrame to include all dates in the range,
filling in any gaps with the mean revenue.

 # Create a date range from the start to end of the time series
date_range = pd.date_range(start=revenue_time_series.index.min(),
end=revenue_time_series.index.max())

# Reindex the revenue DataFrame to include all dates and Fill missing dates
with the mean revenue
revenue_time_series = revenue_time_series.reindex(date_range,
fill_value=mean_revenue)

print(revenue_time_series.Revenue.head(10))
```

```
# Decompose the Time Series
# Finally, decompose the time series using the seasonal_decompose function from
statsmodels.

# Decompose the time series
decomposed = seasonal_decompose(revenue_time_series, model='additive')

# Plot the decomposition
decomposed.plot()
plt.show()
```

Output:

```
Mean Revenue:  197639.42
2018-12-01    324649.218750
2018-12-02    260301.656250
2018-12-03    201975.187500
2018-12-04    197639.421875
2018-12-05    196630.796875
2018-12-06    270656.875000
2018-12-07    336505.875000
2018-12-08    265961.593750
2018-12-09    247149.875000
2018-12-10    262008.187500
Freq: D, Name: Revenue, dtype: float32
```



Answer:

Explanation of the Code and Logic Used:

- The 'Date' column in the DataFrame is converted to a datetime type for accurate time series operations.
- Revenue is aggregated by transaction date, forming a time series with the date as the index.
- The mean revenue is calculated to address any missing dates in the time series.
- A complete date range is created, reindexing the revenue DataFrame to include all dates and filling missing entries with the mean revenue.
- The 'seasonal_decompose' function from 'statsmodels' decomposes time series into trend, seasonal, and residual components, which are visualized with Matplotlib, aiding in understanding revenue patterns for forecasting and decision-making.

Reason for using this solution: Decomposing the time series into its components—trend, seasonality, and residuals—provides a comprehensive view of the underlying patterns in the revenue data. This approach helps businesses identify long-term growth or decline trends, regular seasonal fluctuations, and any irregularities that may indicate anomalies. By filling in missing dates with the mean revenue, the analysis remains consistent and avoids gaps that could distort the results. Understanding these patterns is critical for more accurate forecasting, better resource allocation, and informed strategic decision-making related to inventory, staffing, and promotions.

Alternative Solutions: Alternative approaches for filling missing values could include interpolation methods, such as linear or spline interpolation, which estimate missing data points based on surrounding values and may better reflect the nature of the data. Forward filling could also be used, which would assume that the most recent available revenue value carries forward to the next missing date. For more complex patterns, advanced models like ARIMA (AutoRegressive Integrated Moving Average), SARIMA (Seasonal ARIMA), or Prophet (developed by Facebook) could be used to predict future revenue, as they handle seasonality, trends, and cyclic behavior in a more sophisticated way.

Optimality: This method is optimal for time series analysis because it effectively dissects the data into meaningful components, allowing businesses to understand both short-term fluctuations and long-term trends. The seasonal decomposition approach is particularly useful for industries with predictable seasonal patterns, such as retail, where demand may peak during specific times like holidays. This method not only aids in understanding past behavior but also enhances forecasting capabilities, helping businesses make data-driven decisions about marketing, promotions, and capacity planning. By preserving the integrity of the time structure, it ensures insights are both actionable and relevant.

Question 2.2:

We will try to use time series model ARIMA for forecasting the future. you need to find the best model with different parameters on ARIMA model. The parameter range for p,d,q are all from [0, 1, 2]. In total, you need to find out the best model with lowest Mean Absolute Error from 27 choices based on the time from ”Jan-01-2019” to ”Nov-01-2019” (you might need to split the time series to train and test with grid search according to the M05B material).

Code:

```
# Assuming revenue_time_series is already created from previous steps
# Filter the time series to the specified date range
start_date = '2019-01-01'
end_date = '2019-11-01'
revenue_time_series = revenue_time_series[start_date:end_date]

# split into train and test sets
X = revenue_time_series.values
X = X.astype('float32')
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:]
history = [x for x in train]
predictions = list()
MAE = []
p = d = q = range(0, 3)

# Initialize variables to store the best model and its MAE
best_aic = float('inf')
best_pdq = None

# walk-forward validation
for i1 in p:
  for i2 in q:
    for i3 in d:
      for t in range(len(test)):
        model = ARIMA(history, order=(i1,i3,i2))
        model_fit = model.fit()
        output = model_fit.forecast()
        yhat = output[0]
        predictions.append(yhat)
        obs = test[t]
        history.append(obs)
  #print('predicted=%f, expected=%f' % (yhat, obs))
      # Calculate MAE
      mae = mean_absolute_error(test, predictions)
      history = [x for x in train]
      predictions = list()
      MAE.append(mae)
      # Update the best model if this one is better
      if mae < best_aic:
          best_aic = mae
          best_pdq = (i1,i2,i3)
      print('Test MAE: %.3f' % mae,'(',i1,i2,i3,')')

print(f'Best ARIMA{best_pdq} - MAE: {best_aic}')
```

Output:

```
Test MAE: 76228.944 ( 0 0 0 )
Test MAE: 100594.486 ( 0 0 1 )
Test MAE: 176442.474 ( 0 0 2 )
Test MAE: 74302.392 ( 0 1 0 )
Test MAE: 67607.163 ( 0 1 1 )
Test MAE: 100866.357 ( 0 1 2 )
Test MAE: 73021.923 ( 0 2 0 )
Test MAE: 67545.565 ( 0 2 1 )
Test MAE: 68954.259 ( 0 2 2 )
Test MAE: 73865.251 ( 1 0 0 )
Test MAE: 87989.344 ( 1 0 1 )
Test MAE: 119117.744 ( 1 0 2 )
Test MAE: 69481.847 ( 1 1 0 )
Test MAE: 67505.387 ( 1 1 1 )
Test MAE: 88408.755 ( 1 1 2 )
Test MAE: 73314.430 ( 1 2 0 )
Test MAE: 68608.457 ( 1 2 1 )
Test MAE: 68598.622 ( 1 2 2 )
Test MAE: 72901.632 ( 2 0 0 )
Test MAE: 84480.643 ( 2 0 1 )
Test MAE: 112862.847 ( 2 0 2 )
Test MAE: 73013.377 ( 2 1 0 )
Test MAE: 67896.989 ( 2 1 1 )
Test MAE: 85005.497 ( 2 1 2 )
Test MAE: 73594.450 ( 2 2 0 )
Test MAE: 68588.516 ( 2 2 1 )
Test MAE: 88297.400 ( 2 2 2 )
Best ARIMA(1, 1, 1) - MAE: 67505.38691743014
```

Code:

```
# Forecasting the next n steps
n_steps = 12  # Number of steps to forecast
forecast = best_model.forecast(steps=n_steps)

# Print or plot the forecast
print(f'Forecast for the next {n_steps} periods:')
print(forecast)
```

Output:

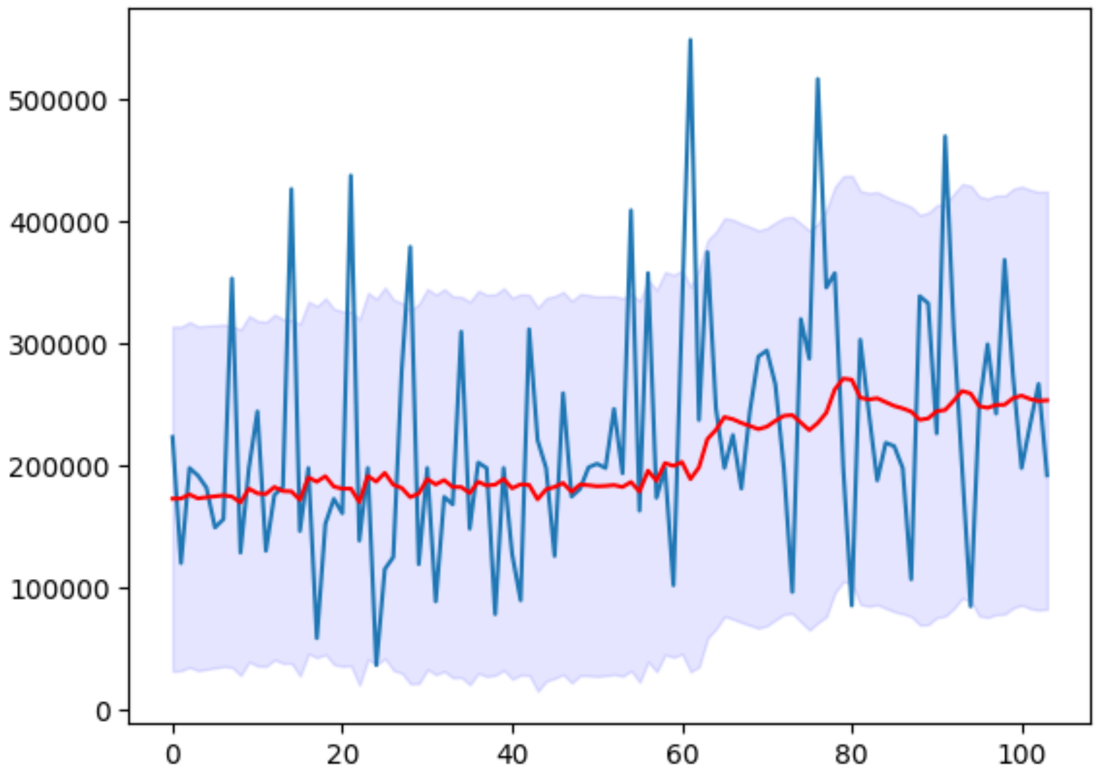
```
Forecast for the next 12 periods:
2019-09-02    175355.343430
2019-09-03    210219.695497
2019-09-04    222950.367553
2019-09-05    206979.667655
2019-09-06    215619.262422
2019-09-07    215263.473042
2019-09-08    214610.363048
2019-09-09    216438.087951
2019-09-10    216699.878844
2019-09-11    217367.825408
```

```
2019-09-12    218164.664548
2019-09-13    218773.091329
Freq: D, Name: predicted_mean, dtype: float64
```

Code:

```
# split into train and test sets
X = revenue_time_series.values
X = X.astype('float32')
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:]
history = [x for x in train]
predictions = list()
best_parameter = [1,1,1]
confidence_interval = []

# walk-forward validation
for t in range(len(test)):
    model = ARIMA(history,
order=(best_parameter[0],best_parameter[1],best_parameter[2]))
    model_fit = model.fit()
    output = model_fit.get_forecast()
    yhat = output.predicted_mean
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    ci = output.conf_int(0.05)
    confidence_interval.append(ci[0])

# plot forecasts against actual outcomes and also the confidence int at 95%
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.fill_between(list(range(len(test))),
                 np.array(confidence_interval)[:,0],
np.array(confidence_interval)[:,1],
                alpha=0.1, color='b')
pyplot.show()
```

Output:



Answer:

Explanation of the Code and Logic Used:

- The time series data is first filtered to include only records from January 1, 2019, to November 1, 2019. This targeted timeframe is essential for analyzing the relevant trends and patterns in revenue during a specific period, ensuring the model captures seasonal effects appropriately.

- The filtered dataset is then split into training (66% of the data) and testing (34% of the data) datasets. This division is crucial for evaluating model performance, as it allows us to train the model on historical data and test its predictive capability on unseen data.

- A grid search is employed to identify the optimal parameters for the ARIMA model, specifically the values of p, d, and q. Each parameter is tested in a range from 0 to 2, generating all possible combinations of these parameters. For each combination, the ARIMA model is fitted to the training data, assessing how well it captures the underlying patterns.

- After fitting the models, predictions are generated for the testing dataset for each combination of parameters. The performance of these predictions is evaluated using the Mean Absolute Error (MAE), a metric that quantifies the average magnitude of errors in a set of predictions, without considering their direction.

- The model that achieves the lowest MAE during this evaluation is selected as the best-performing ARIMA model. This model is then utilized to forecast revenue for the next

12 periods, allowing for insights into future trends. The forecasts can be printed or visualized using plots, making it easier to interpret and communicate findings.

Reason for using this solution: The ARIMA model is a well-established and widely used approach for time series forecasting due to its flexibility and capability to effectively capture various underlying trends and seasonality patterns within the data. By identifying optimal parameters through grid search, the model ensures a more tailored fit to the data, which increases the reliability of the forecasts. This approach is particularly valuable for businesses needing to make informed decisions based on projected future revenues, as it helps anticipate changes in demand, optimize inventory levels, and enhance overall strategic planning.

Alternative Solutions: There are several alternative approaches for forecasting time series data. One notable alternative is the Seasonal ARIMA (SARIMA) model, which extends ARIMA by adding seasonal components to account for repeating patterns within seasonal data. This is especially useful for datasets exhibiting strong seasonal fluctuations. Additionally, machine learning techniques such as Long Short-Term Memory (LSTM) networks could be considered, as they can capture complex patterns and dependencies in time series data. These methods can be particularly beneficial when working with large datasets that display intricate nonlinear behaviors, offering potentially higher accuracy than traditional models.

Optimality: The grid search method is optimal for this task as it systematically evaluates a range of ARIMA parameters to minimize the Mean Absolute Error (MAE). By focusing on minimizing prediction errors, this approach ensures that the selected model provides reliable forecasts. The systematic nature of grid search allows for a comprehensive exploration of parameter combinations, which is critical when aiming for the best possible fit. This reliability is essential for businesses relying on accurate revenue predictions to guide strategic planning and resource allocation, ultimately leading to better decision-making and enhanced operational efficiency.

Question 2.3:

There are many deep learning time series forecasting methods, could you please explore those methods and write down the necessary data wrangling and modeling steps (steps on how to achieve, not actual code). Also please give the reference of the deep learning time series forecasting models you are using.

Common Deep Learning Methods for Time Series Forecasting

1.Recurrent Neural Networks (RNNs):

RNNs are specifically designed for sequence prediction tasks, maintaining a hidden state that captures information from previous inputs. This makes them particularly suitable for time series data, as they can effectively model temporal dependencies. However, traditional RNNs can struggle with long sequences due to issues like the vanishing gradient problem, limiting their performance on tasks requiring long-term context.

2.Long Short-Term Memory Networks (LSTMs):

LSTMs are an advanced type of RNN that are designed to learn long-term dependencies while mitigating the vanishing gradient problem. They use a complex gating mechanism to control the flow of information, allowing them to retain information over extended sequences. This makes LSTMs particularly effective for tasks in which the prediction of future values depends on long historical sequences, such as stock prices or climate data.

3.Gated Recurrent Units (GRUs):

GRUs are similar to LSTMs but with a simpler architecture, combining the forget and input gates into a single update gate. This simplification allows GRUs to capture data dependencies effectively while being less computationally intensive than LSTMs. As a result, GRUs are often favored when resources are limited or when the training time is a significant concern.

4.Convolutional Neural Networks (CNNs):

CNNs, while traditionally used for image data, can be adapted for time series forecasting by treating the data as a 1D signal. They are particularly effective at extracting local patterns and features from the input data. By applying convolutional layers, CNNs can identify important features at various temporal resolutions, making them suitable for capturing short-term patterns in time series data.

5.Temporal Convolutional Networks (TCNs):

TCNs extend the concept of CNNs by using causal convolutions, ensuring that the model does not access future data when predicting the current value. TCNs are designed to capture long-range dependencies more effectively than traditional RNNs, often outperforming them in various sequence tasks. Their architecture allows them to model temporal sequences while maintaining a linear complexity concerning the sequence length.

Data Wrangling and Modeling Steps

1.Data Collection:

The first step is to gather time series data relevant to the forecasting task. This can include historical sales data, financial metrics, weather data, or any other time-dependent metrics. Data should be comprehensive and cover a sufficient time frame to ensure that the model can learn meaningful patterns.

2.Data Preprocessing:

Preprocessing is crucial for ensuring data quality. This may involve imputing missing values using techniques such as forward filling or interpolation. Additionally, normalization methods like Min-Max scaling or Z-score normalization should be applied to standardize the data, making it more suitable for modeling. It's also essential to ensure that the data is correctly indexed by date and time for accurate time series analysis.

3.Feature Engineering:

Feature engineering involves creating new input variables that can enhance the model's predictive capabilities. This can include generating lagged versions of the target variable, calculating moving averages, or extracting temporal features such as the day of the week, month, or season. These features can help the model understand cyclical patterns and improve its forecasting accuracy.

4.Splitting Data:

The dataset should be split into training, validation, and test sets, typically allocating 70-80% for training. The training set is used to fit the model, while the validation set helps tune hyperparameters. The test set, which remains unseen during the training phase, is used to evaluate the final model's performance.

5.Modeling:

In this step, an appropriate forecasting model is selected based on the nature of the data and the specific forecasting task. The architecture of the model should be configured, and training should commence. It's crucial to monitor the training process to ensure that the model is learning effectively without overfitting.

6.Evaluation:

After training, the model's performance should be evaluated using relevant metrics such as Root Mean Squared Error (RMSE) or Mean Absolute Error (MAE). Visualization techniques, such as plotting actual vs. predicted values, can provide insights into the model's performance and identify areas for improvement.

7.Forecasting:

Once the model is evaluated, it can be used to make predictions on future time steps. It's essential to consider confidence intervals or prediction intervals to quantify the uncertainty associated with the forecasts, which can be valuable for decision-making.

8.Deployment:

Finally, the trained model should be prepared for deployment, ensuring it can be integrated into existing systems. This may involve setting up an API or other interfaces for real-time predictions, along with monitoring mechanisms to ensure the model continues to perform well over time.

References

• Hochreiter, S. & Schmidhuber, J. (1997) 'LSTM', Neural Computation.

• Cho, K., et al. (2014) 'Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation'.

• Borovykh, A., et al. (2017) 'Conditional Time Series Forecasting with Convolutional Neural Networks'.

• Bai, S., Zhan, J. & Kolter, J.Z. (2018) 'An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling'.

• Tulip Lab. (n.d.). *SIT742: M05 Data Analytics*. GitHub. Available at: https://github.com/tulip-lab/sit742/tree/c7814205f14d75ead557457f0a0b58b53f5be0f4/Jupyter/M05-DataAnalytics

Question:

How you and your team member collaborate on this assignment?

Answer:

Our team collaborated effectively using Google Meet for discussions and brainstorming sessions. We created a shared Google Drive folder where all code, documents, and resources were uploaded for easy access. This central repository allowed team members to stay updated on each other's progress and contribute efficiently. By maintaining open communication through scheduled meetings, we ensured that everyone was on the same page, enabling us to address challenges collectively and share insights seamlessly throughout the project.

Question:

What have you learned with your team member from the second assignment?

Answer:

Through this assignment, we learned the importance of teamwork and effective communication in a collaborative environment. Engaging in discussions helped us clarify concepts and share diverse perspectives, enhancing our understanding of the subject matter. Additionally, we gained practical experience in using collaborative tools like Google Drive and Google Meet, which improved our ability to manage and execute projects remotely. The assignment also reinforced our technical skills, as we worked together to solve problems & debug code.

Question:

What is the contribution of each the team member for finishing the second assignment?

Answer:

We split the questions among ourselves, assigning specific tasks to each team member based on their strengths. Each person focused on their assigned section, ensuring that we made steady progress. Whenever someone encountered a challenge or got stuck, we scheduled calls to discuss the issue and provide support. This collaborative approach fostered an environment of teamwork, allowing us to troubleshoot problems together and share insights. By helping each other out, we not only enhanced our understanding of the material but also ensured that the project was completed efficiently and effectively.