Table of Contents

1. Introduction

An e-commerce company aims to optimize its sales strategy by analyzing customer purchase data collected over a one-year period, from December 2010 to December 2011.

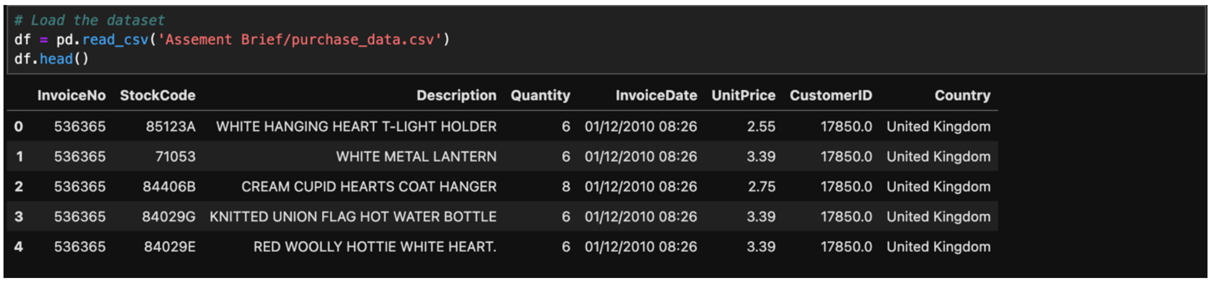
2. Methodology

For this analysis, tools such as Jupyter Notebook, Python, and the Pandas library were employed:

- **Jupyter Notebook**: An interactive platform that combines programming, data visualization, and documentation, ideal for exploratory data analysis.
- **Python**: A versatile programming language widely used for data analysis, machine learning, and web development.
- **Pandas**: A Python library designed for powerful and efficient manipulation and analysis of structured data.

2.1 Data

The dataset was read using the **pd.read_csv()** function, which is used to load Comma-Separated Values (CSV) files into a Pandas DataFrame for data manipulation and analysis. This DataFrame is stored in the variable **df**, and the **df.head()** method is then called to display the first few rows of the dataset.



```python
# Load the dataset
df = pd.read_csv('Assement Brief/purchase_data.csv')
df.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	01/12/2010 08:26	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	01/12/2010 08:26	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	01/12/2010 08:26	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	01/12/2010 08:26	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01/12/2010 08:26	3.39	17850.0	United Kingdom

Figure 2.1.1

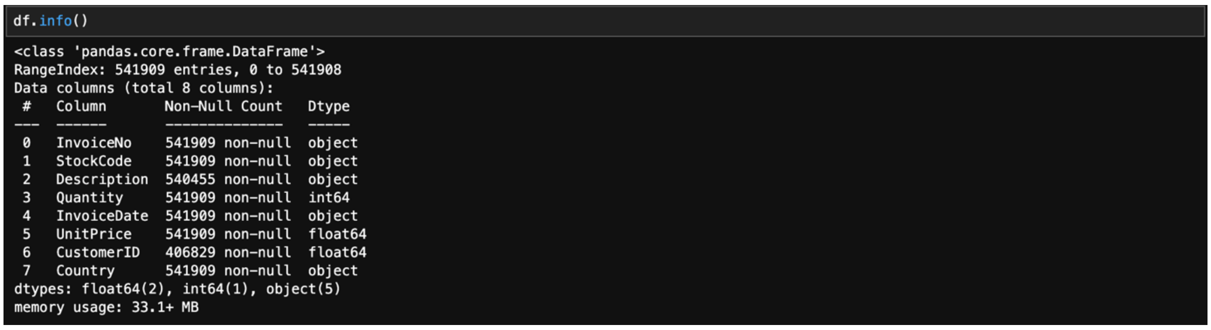
The dataset includes detailed information about transactions, featuring the following fields:

1. **InvoiceNo** – Unique identifier for each transaction.
2. **StockCode** – Unique product code for each item.
3. **Description** – Brief description of the purchased item.
4. **Quantity** – Number of items purchased in the transaction.

5. **InvoiceDate** – Date and time of the transaction.
6. **UnitPrice** – Price per unit of the product.
7. **CustomerID** – Unique identifier for each customer.
8. **Country** – The country where the transaction took place.

These fields are also known as **Features**.

Further inspection using the **df.info()** function from Pandas provides additional details about the dataset. It displays the column names, data types, and non-null counts for each column. The dataset includes both non-numerical columns (of type object) and numerical columns (int64 and float64). Additionally, it reveals that the dataset comprises 541,909 rows of transaction data.



```
df.info()

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

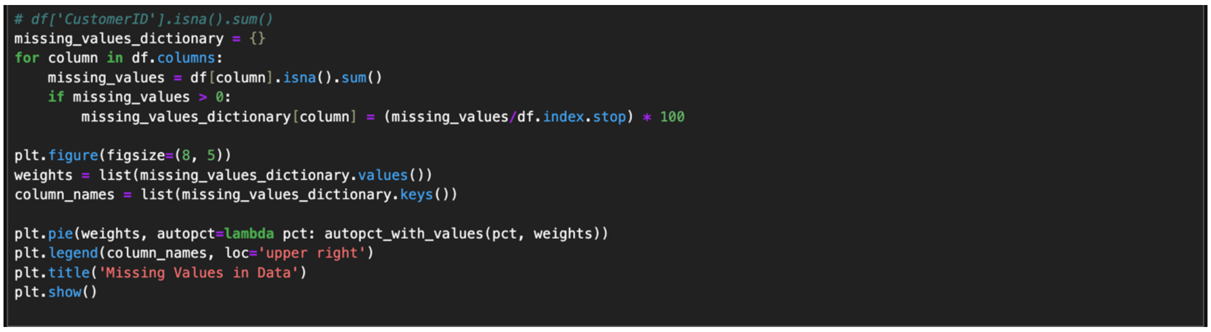
Figure 2.1.2

2.2 Data Preprocessing

2.2.1 Missing Data

However, the **CustomerID** and **Description** columns in the dataset contain missing values, necessitating data cleaning and preprocessing to ensure accurate analysis. Addressing these missing values is a critical step in maintaining data integrity and deriving meaningful insights from the dataset.

The code below iterates through all the columns in the dataset, identifying those with missing values. Columns with a percentage of missing values greater than 0 are added to the **missing_values_dictionary**. The results are then visualised using a pie chart created with the **Matplotlib** library, illustrating the proportion of rows with missing information in the data in terms of percentages.



```
# df['CustomerID'].isna().sum()
missing_values_dictionary = {}
for column in df.columns:
    missing_values = df[column].isna().sum()
    if missing_values > 0:
        missing_values_dictionary[column] = (missing_values/df.index.stop) * 100

plt.figure(figsize=(8, 5))
weights = list(missing_values_dictionary.values())
column_names = list(missing_values_dictionary.keys())

plt.pie(weights, autopct=lambda pct: autopct_with_values(pct, weights))
plt.legend(column_names, loc='upper right')
plt.title('Missing Values in Data')
plt.show()
```

Figure 2.2.1.1

This reveals that missing **CustomerID** values constitute **24.93%** of the dataset, while missing **Description** values account for only **0.27%**. Although various methods exist for handling missing data during analysis, the approach depends on the nature of the missing information.

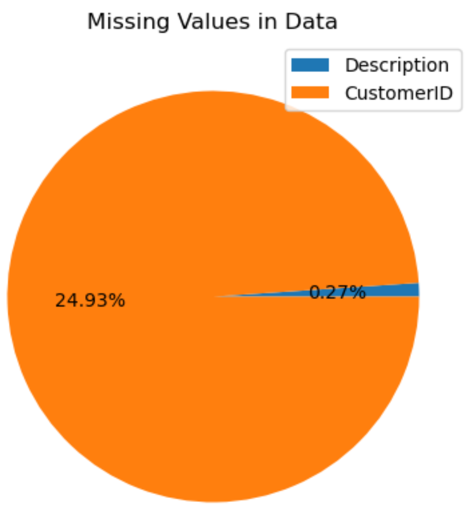
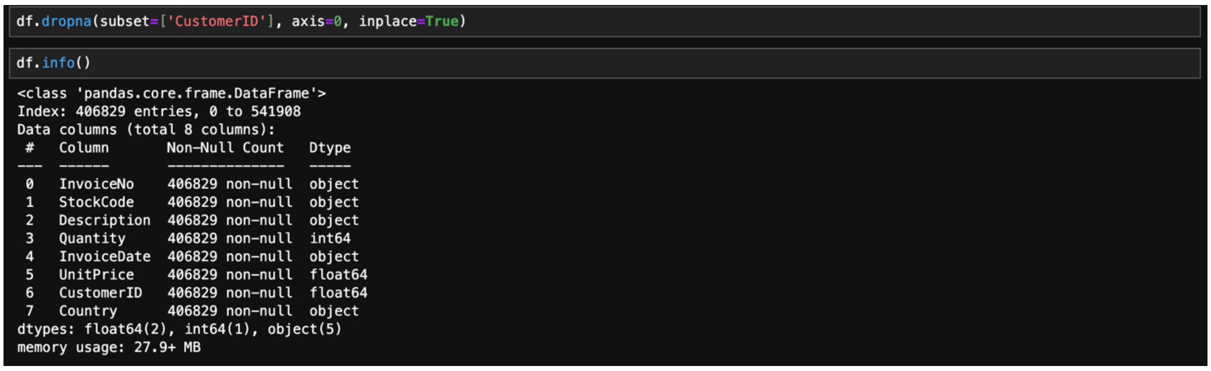


Figure 2.2.1.2

In this case, rows with missing **CustomerID** values are removed, as imputing sensitive information such as **CustomerID** could compromise the integrity of the analysis. Removing these rows ensures that the analysis remains accurate and reliable. The Pandas **dropna()** function is employed to eliminate rows containing missing values. In this scenario, the **subset** parameter specifies the columns to consider when checking for missing values, **axis=0** directs Pandas to drop rows rather than columns, and **inplace=True** ensures that the modifications are applied directly to the existing DataFrame without requiring reassignment.



```
df.dropna(subset=['CustomerID'], axis=0, inplace=True)
```

```
df.info()
```

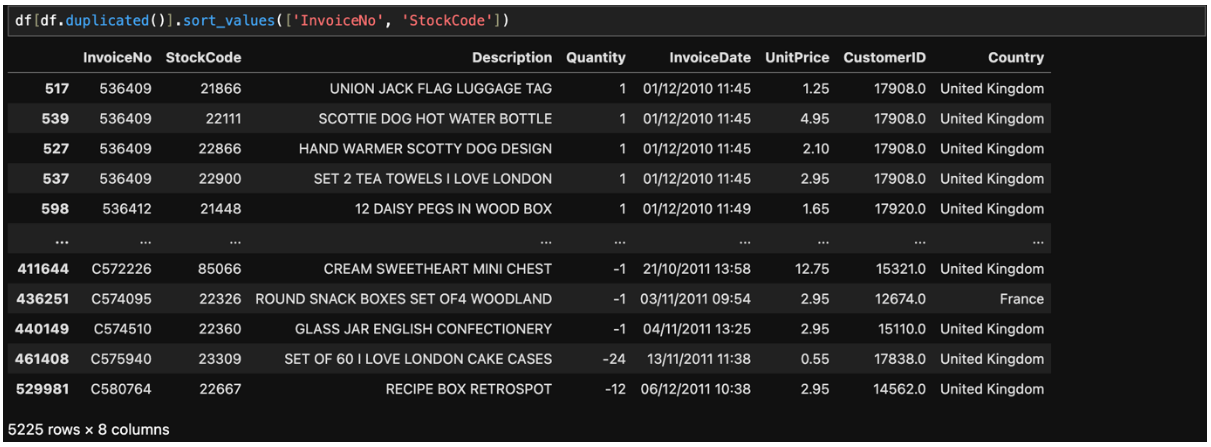
```
<class 'pandas.core.frame.DataFrame'>
Index: 406829 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   InvoiceNo    406829 non-null  object
 1   StockCode    406829 non-null  object
 2   Description  406829 non-null  object
 3   Quantity     406829 non-null  int64
 4   InvoiceDate  406829 non-null  object
 5   UnitPrice    406829 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      406829 non-null  object
dtypes: float64(2), int64(1), object(5)
memory usage: 27.9+ MB
```

Figure 2.2.1.3

It appears that dropping the rows with missing **CustomerID** also dropped the rows with missing **Description** and now there is no more missing information present in the data.

2.2.2 Duplicate Data

It is also essential to verify the presence of duplicate data in the DataFrame, as duplicates are redundant and can distort the analysis. Therefore, they need to be removed. The Pandas **duplicated()** method is used to identify duplicate rows within the DataFrame. The identified duplicate rows can then be reviewed and addressed accordingly to ensure the integrity of the dataset.



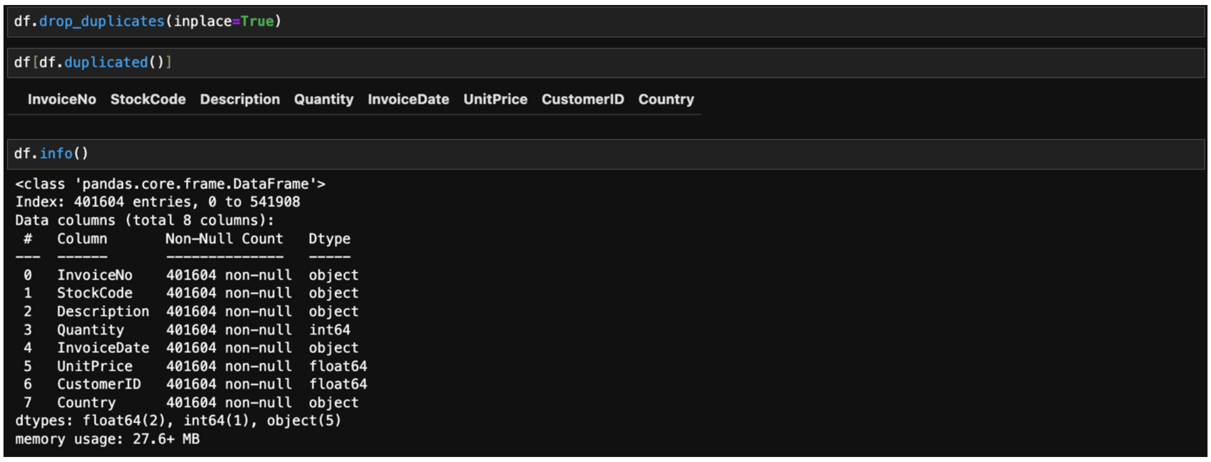
```
df[df.duplicated()].sort_values(['InvoiceNo', 'StockCode'])
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
517	536409	21866	UNION JACK FLAG LUGGAGE TAG	1	01/12/2010 11:45	1.25	17908.0	United Kingdom
539	536409	22111	SCOTTIE DOG HOT WATER BOTTLE	1	01/12/2010 11:45	4.95	17908.0	United Kingdom
527	536409	22866	HAND WARMER SCOTTY DOG DESIGN	1	01/12/2010 11:45	2.10	17908.0	United Kingdom
537	536409	22900	SET 2 TEA TOWELS I LOVE LONDON	1	01/12/2010 11:45	2.95	17908.0	United Kingdom
598	536412	21448	12 DAISY PEGS IN WOOD BOX	1	01/12/2010 11:49	1.65	17920.0	United Kingdom
...	...	...	...	...	...	...	...	...
411644	C572226	85066	CREAM SWEETHEART MINI CHEST	-1	21/10/2011 13:58	12.75	15321.0	United Kingdom
436251	C574095	22326	ROUND SNACK BOXES SET OF4 WOODLAND	-1	03/11/2011 09:54	2.95	12674.0	France
440149	C574510	22360	GLASS JAR ENGLISH CONFECTIONERY	-1	04/11/2011 13:25	2.95	15110.0	United Kingdom
461408	C575940	23309	SET OF 60 I LOVE LONDON CAKE CASES	-24	13/11/2011 11:38	0.55	17838.0	United Kingdom
529981	C580764	22667	RECIPE BOX RETROSPOT	-12	06/12/2011 10:38	2.95	14562.0	United Kingdom

5225 rows × 8 columns

Figure 2.2.2.1

The result shows that there are 5225 rows of data which are duplicates in the dataframe and to solve this, Pandas **drop_duplicates()** function is used to drop duplicated rows in the dataframe.



```
df.drop_duplicates(inplace=True)
```

```
df[df.duplicated()]
```

```
   InvoiceNo  StockCode  Description  Quantity  InvoiceDate  UnitPrice  CustomerID  Country
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 401604 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   InvoiceNo    401604 non-null  object
 1   StockCode    401604 non-null  object
 2   Description  401604 non-null  object
 3   Quantity     401604 non-null  int64
 4   InvoiceDate  401604 non-null  object
 5   UnitPrice    401604 non-null  float64
 6   CustomerID   401604 non-null  float64
 7   Country      401604 non-null  object
dtypes: float64(2), int64(1), object(5)
memory usage: 27.6+ MB
```

Figure 2.2.2.2

2.2.3 Outliers

The next step in data preprocessing involves checking for outliers. Outliers are data points that significantly deviate from the general distribution of the dataset. A boxplot is an effective tool for detecting such anomalies.

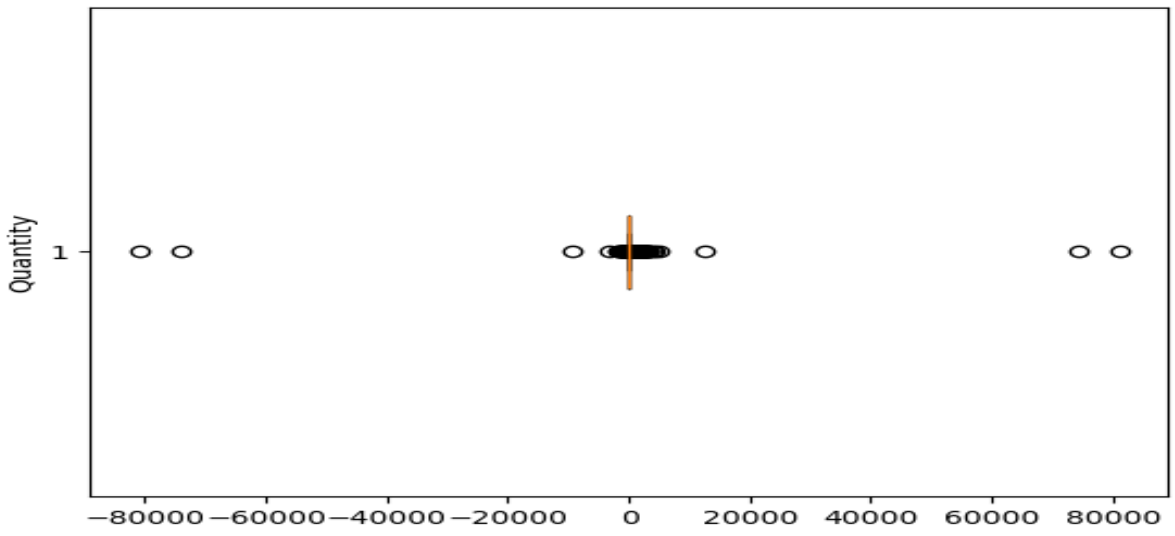
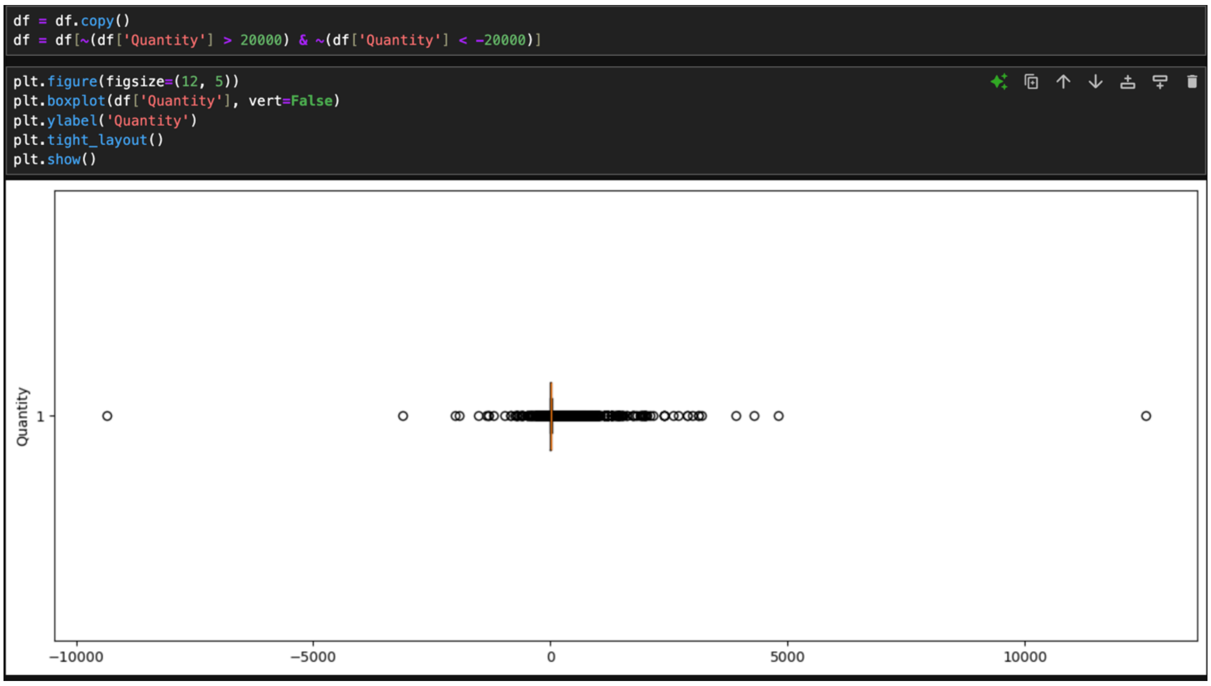


Figure 2.2.3.1

As illustrated in the plot, there are extreme values, values less than -40,000 and greater than 40,000. These points are far removed from the rest of the data. Given their rarity relative to the overall dataset, these outliers can be excluded to prevent them from skewing the analysis.

The dataframe will be filtered to exclude rows that has **Quantity** values that are greater than 20,000 and less than -20,000.



```
df = df.copy()
df = df[~(df['Quantity'] > 20000) & ~(df['Quantity'] < -20000)]
```

```
plt.figure(figsize=(12, 5))
plt.boxplot(df['Quantity'], vert=False)
plt.ylabel('Quantity')
plt.tight_layout()
plt.show()
```

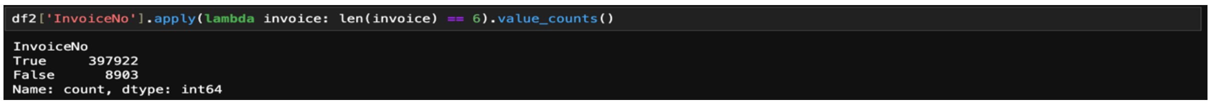
Figure 2.2.3.2

After excluding the outlier, all the other **Quantity** values are much closer to each other.

3. Feature Analysis and Engineering

Figure 2.1.1 shows **InvoiceNo** consists of six numeric characters. However, since this represents only a subset of the data, it is necessary to verify whether all **InvoiceNo** entries in the dataset follow this pattern.

To perform this check, the **InvoiceNo** column is extracted, and a function is applied to determine if each entry is exactly six characters long. The Pandas **value_counts()** function is then used to count the occurrences of different results.

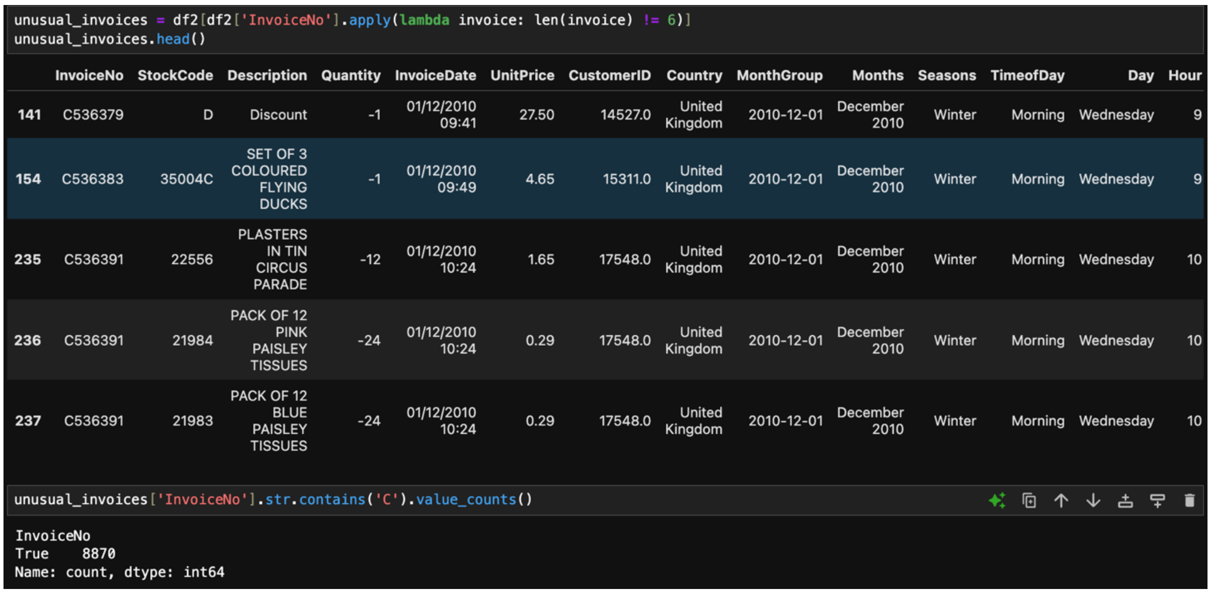


```
df2['InvoiceNo'].apply(lambda invoice: len(invoice) == 6).value_counts()

InvoiceNo
True     397922
False      8903
Name: count, dtype: int64
```

Figure 3.1

The analysis reveals that 392,730 **InvoiceNo** entries in the dataframe are exactly six characters long, while 8,870 entries deviate from this length, being either shorter or longer.

Inspecting the **InvoiceNo** that deviates from the norm show that those InvoiceNo all have a special character ‘C’ and they also contain negative Quantity in the data and some of them appears to be orders that have been cancelled or returned.



```
unusual_invoices = df2[df2['InvoiceNo'].apply(lambda invoice: len(invoice) != 6)]
unusual_invoices.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	MonthGroup	Months	Seasons	TimeofDay	Day	Hour
141	C536379	D	Discount	-1	01/12/2010 09:41	27.50	14527.0	United Kingdom	2010-12-01	December 2010	Winter	Morning	Wednesday	9
154	C536383	35004C	SET OF 3 COLOURED FLYING DUCKS	-1	01/12/2010 09:49	4.65	15311.0	United Kingdom	2010-12-01	December 2010	Winter	Morning	Wednesday	9
235	C536391	22556	PLASTERS IN TIN CIRCUS PARADE	-12	01/12/2010 10:24	1.65	17548.0	United Kingdom	2010-12-01	December 2010	Winter	Morning	Wednesday	10
236	C536391	21984	PACK OF 12 PINK PAISLEY TISSUES	-24	01/12/2010 10:24	0.29	17548.0	United Kingdom	2010-12-01	December 2010	Winter	Morning	Wednesday	10
237	C536391	21983	PACK OF 12 BLUE PAISLEY TISSUES	-24	01/12/2010 10:24	0.29	17548.0	United Kingdom	2010-12-01	December 2010	Winter	Morning	Wednesday	10

```
unusual_invoices['InvoiceNo'].str.contains('C').value_counts()
```

```
InvoiceNo
True    8870
Name: count, dtype: int64
```

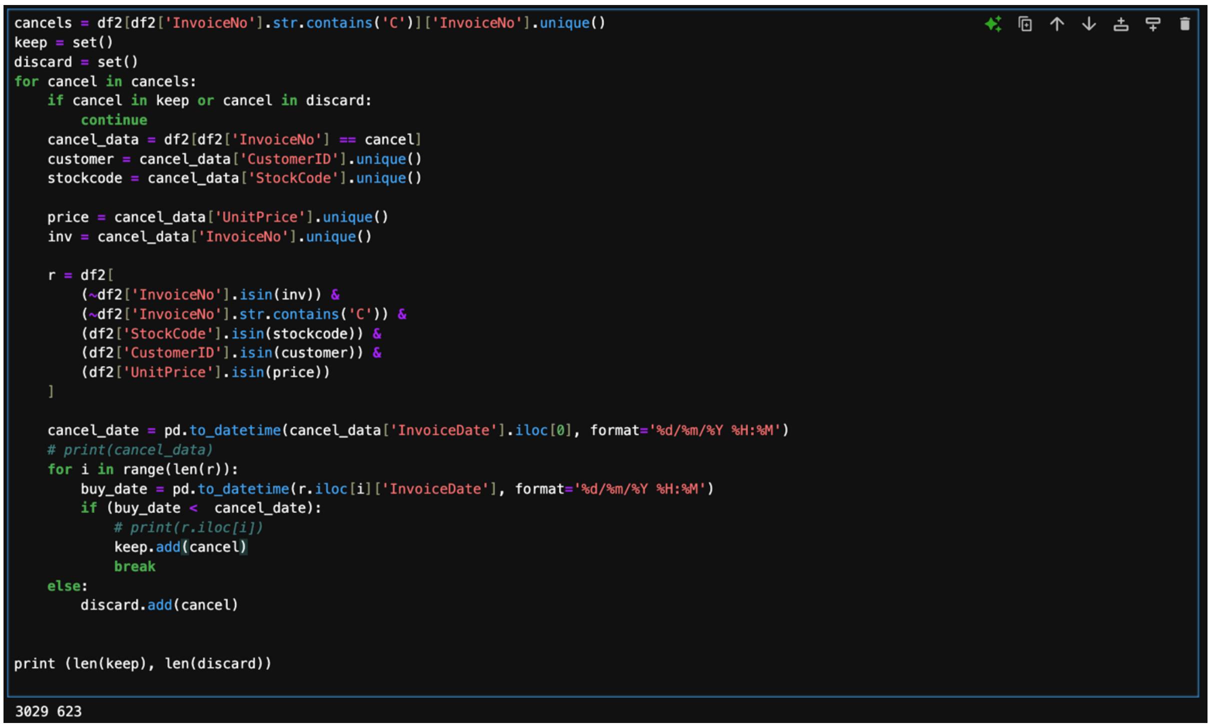
Figure 3.2

The dataframe contains a total of 3,652 cancelled **InvoiceNo**, and it is essential to determine if these cancellations are associated with prior purchases. This involves confirming whether all cancelled orders in the dataset correspond to previously purchased items.

To achieve this, all **InvoiceNo** entries containing 'C' are extracted and stored in the **cancels** variable for iteration. For each cancelled invoice, the dataframe is filtered to retrieve the corresponding cancelled invoice details, including the associated customer and products.

To verify if the cancellation is linked to a previous purchase, the dataframe is further filtered based on the customer, product (**StockCode**), and **UnitPrice** to identify matching invoices, stored in the variable r.

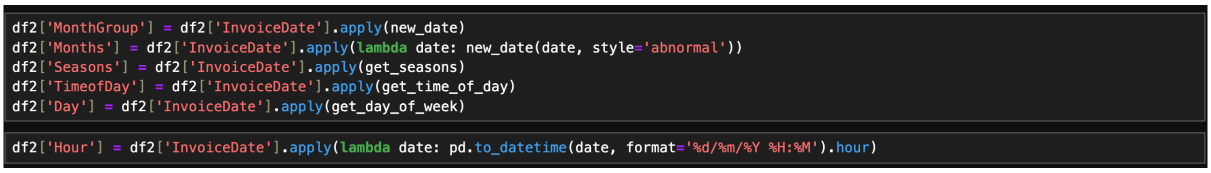
Since r might contain multiple matching entries, it is iterated over to compare the **InvoiceDate** of the cancelled stock with those of the matching invoices. Both dates are converted to datetime objects to verify that the purchase occurred before the cancellation.



```
cancels = df2[df2['InvoiceNo'].str.contains('C')]['InvoiceNo'].unique()
keep = set()
discard = set()
for cancel in cancels:
    if cancel in keep or cancel in discard:
        continue
    cancel_data = df2[df2['InvoiceNo'] == cancel]
    customer = cancel_data['CustomerID'].unique()
    stockcode = cancel_data['StockCode'].unique()

    price = cancel_data['UnitPrice'].unique()
    inv = cancel_data['InvoiceNo'].unique()

    r = df2[
        (~df2['InvoiceNo'].isin(inv)) &
        (~df2['InvoiceNo'].str.contains('C')) &
        (df2['StockCode'].isin(stockcode)) &
        (df2['CustomerID'].isin(customer)) &
        (df2['UnitPrice'].isin(price))
    ]

    cancel_date = pd.to_datetime(cancel_data['InvoiceDate'].iloc[0], format='%d/%m/%Y %H:%M')
    # print(cancel_data)
    for i in range(len(r)):
        buy_date = pd.to_datetime(r.iloc[i]['InvoiceDate'], format='%d/%m/%Y %H:%M')
        if (buy_date <  cancel_date):
            # print(r.iloc[i])
            keep.add(cancel)
            break
    else:
        discard.add(cancel)


print (len(keep), len(discard))
```

```
3029 623
```

Figure 3.3

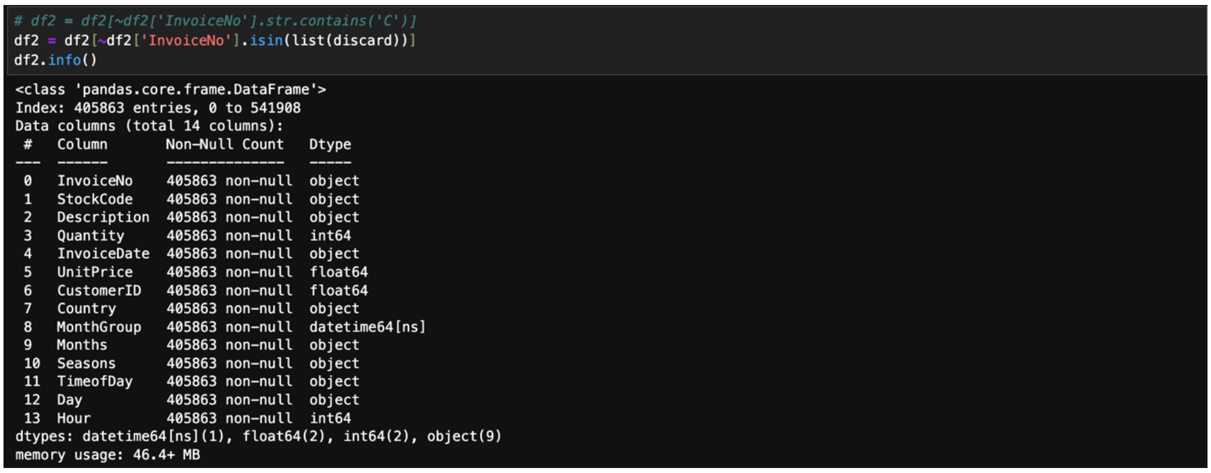


```
df2['MonthGroup'] = df2['InvoiceDate'].apply(new_date)
df2['Months'] = df2['InvoiceDate'].apply(lambda date: new_date(date, style='abnormal'))
df2['Seasons'] = df2['InvoiceDate'].apply(get_seasons)
df2['TimeofDay'] = df2['InvoiceDate'].apply(get_time_of_day)
df2['Day'] = df2['InvoiceDate'].apply(get_day_of_week)
```

```
df2['Hour'] = df2['InvoiceDate'].apply(lambda date: pd.to_datetime(date, format='%d/%m/%Y %H:%M').hour)
```

Figure 3.4

The results indicate that out of 3,652 cancelled **InvoiceNo**, 3,029 have associated orders within the dataset, while 623 do not. The cancelled **InvoiceNo** without associated orders may correspond to transactions from earlier months that are not included in this dataset. Therefore, these will be excluded from the analysis to maintain accuracy.

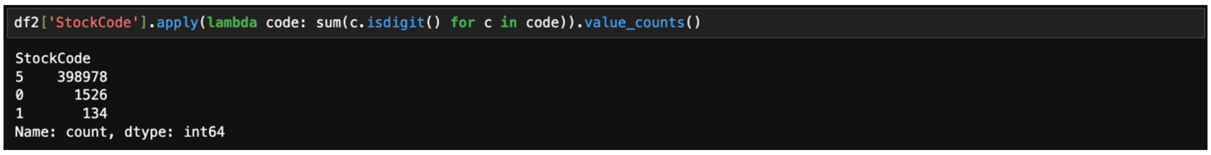


```
# df2 = df2[~df2['InvoiceNo'].str.contains('C')]
df2 = df2[~df2['InvoiceNo'].isin(list(discard))]
df2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 405863 entries, 0 to 541908
Data columns (total 14 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   InvoiceNo    405863 non-null  object
 1   StockCode    405863 non-null  object
 2   Description  405863 non-null  object
 3   Quantity     405863 non-null  int64
 4   InvoiceDate  405863 non-null  object
 5   UnitPrice    405863 non-null  float64
 6   CustomerID   405863 non-null  float64
 7   Country      405863 non-null  object
 8   MonthGroup   405863 non-null  datetime64[ns]
 9   Months       405863 non-null  object
 10  Seasons      405863 non-null  object
 11  TimeofDay    405863 non-null  object
 12  Day          405863 non-null  object
 13  Hour         405863 non-null  int64
dtypes: datetime64[ns](1), float64(2), int64(2), object(9)
memory usage: 46.4+ MB
```

Figure 3.5

Also, as shown in Figure 2.1.1, while many of the **StockCode** values contain alphabetic characters, they all appear to consist of a 5-digit numerical sequence. It is important to validate whether all **StockCode** values adhere to this pattern, as discrepancies in the format could indicate data quality issues.

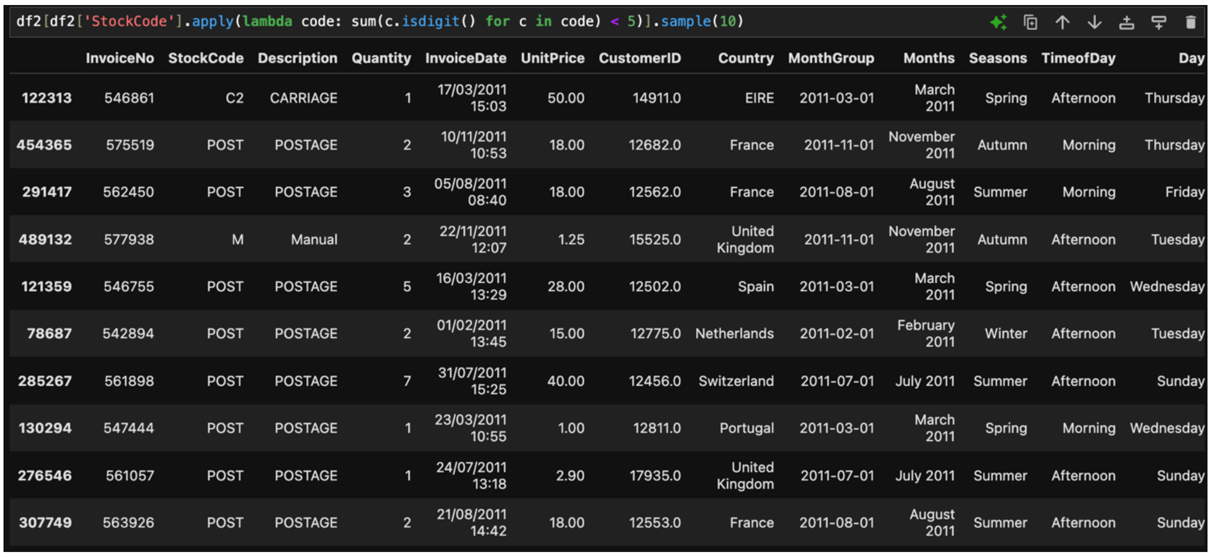


```
df2['StockCode'].apply(lambda code: sum(c.isdigit() for c in code)).value_counts()
```

```
StockCode
5    398978
0      1526
1       134
Name: count, dtype: int64
```

Figure 3.6

To confirm this, the **StockCode** column was examined to check for values that do not contain exactly 5 numerical digits. The result indicates that the majority of **StockCode** values indeed consist of 5 digits, but a small number contain fewer digits (such as 1 or 0 digits). These anomalies are likely related to charges or other non-transactional data, which are not relevant for this analysis.



```
df2[df2['StockCode'].apply(lambda code: sum(c.isdigit() for c in code) < 5)].sample(10)
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	MonthGroup	Months	Seasons	TimeofDay	Day
122313	546861	C2	CARRIAGE	1	17/03/2011 15:03	50.00	14911.0	EIRE	2011-03-01	March 2011	Spring	Afternoon	Thursday
454365	575519	POST	POSTAGE	2	10/11/2011 10:53	18.00	12682.0	France	2011-11-01	November 2011	Autumn	Morning	Thursday
291417	562450	POST	POSTAGE	3	05/08/2011 08:40	18.00	12562.0	France	2011-08-01	August 2011	Summer	Morning	Friday
489132	577938	M	Manual	2	22/11/2011 12:07	1.25	15525.0	United Kingdom	2011-11-01	November 2011	Autumn	Afternoon	Tuesday
121359	546755	POST	POSTAGE	5	16/03/2011 13:29	28.00	12502.0	Spain	2011-03-01	March 2011	Spring	Afternoon	Wednesday
78687	542894	POST	POSTAGE	2	01/02/2011 13:45	15.00	12775.0	Netherlands	2011-02-01	February 2011	Winter	Afternoon	Tuesday
285267	561898	POST	POSTAGE	7	31/07/2011 15:25	40.00	12456.0	Switzerland	2011-07-01	July 2011	Summer	Afternoon	Sunday
130294	547444	POST	POSTAGE	1	23/03/2011 10:55	1.00	12811.0	Portugal	2011-03-01	March 2011	Spring	Morning	Wednesday
276546	561057	POST	POSTAGE	1	24/07/2011 13:18	2.90	17935.0	United Kingdom	2011-07-01	July 2011	Summer	Afternoon	Sunday
307749	563926	POST	POSTAGE	2	21/08/2011 14:42	18.00	12553.0	France	2011-08-01	August 2011	Summer	Afternoon	Sunday

Figure 3.7

Given their limited relevance, the StockCode values with fewer than 5 digits will be excluded from the dataset to ensure the accuracy and integrity of the analysis. The Country column in the dataframe consists of many countries but the United Kingdom accounts for most. This could be due to the company being based in the United Kingdom or other factors.

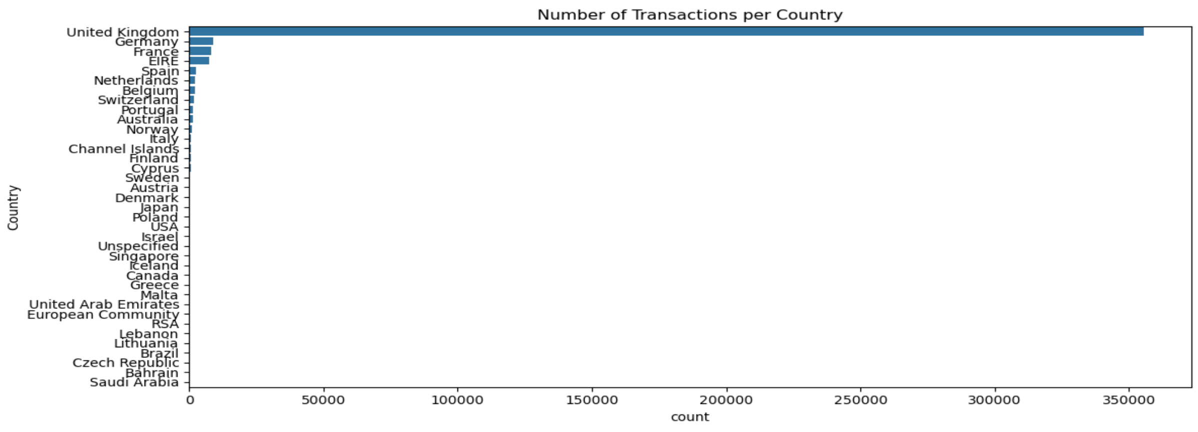
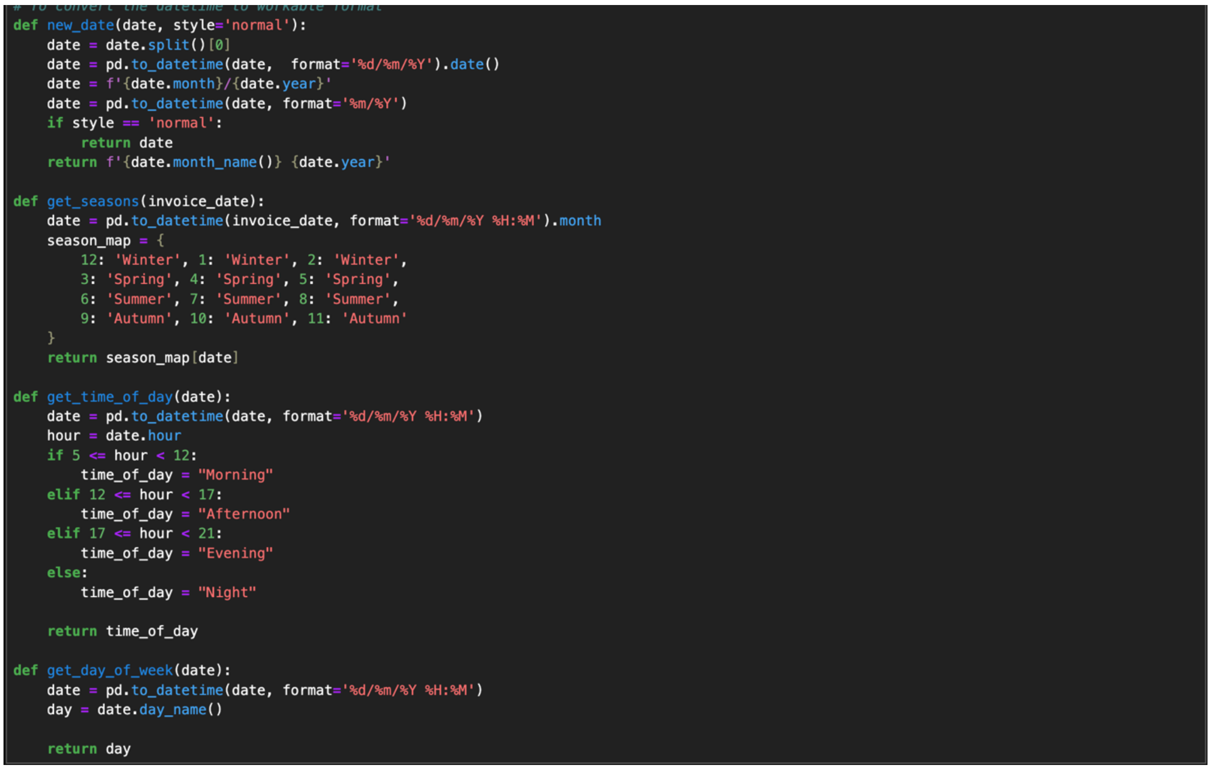


Figure 3.8

To effectively manipulate and analyse the dataset, additional derived information is added to the dataframe. This new information is generated from the existing data and includes the following:

1. **MonthGroup**: Extracts the month and year of a transaction, excluding the time, to facilitate efficient grouping.
2. **Months**: Captures only the month when each transaction occurred, represented numerically.
3. **Seasons**: Categorises transactions into their respective seasons (e.g., Spring, Summer, Autumn, Winter) based on the transaction date.
4. **TimeofDay**: Classifies transactions based on the time they occurred (e.g., Morning, Afternoon, Evening).
5. **Day**: Identifies the day of the week on which the transaction took place (e.g., Monday, Tuesday).
6. **Hour**: Extracts the specific hour during which the transaction occurred.

These additional columns enrich the dataset, enabling more nuanced analysis and insights.



```python
def new_date(date, style='normal'):
    date = date.split()[0]
    date = pd.to_datetime(date,  format='%d/%m/%Y').date()
    date = f'{date.month}/{date.year}'
    date = pd.to_datetime(date, format='%m/%Y')
    if style == 'normal':
        return date
    return f'{date.month_name()} {date.year}'

def get_seasons(invoice_date):
    date = pd.to_datetime(invoice_date, format='%d/%m/%Y %H:%M').month
    season_map = {
        12: 'Winter', 1: 'Winter', 2: 'Winter',
        3: 'Spring', 4: 'Spring', 5: 'Spring',
        6: 'Summer', 7: 'Summer', 8: 'Summer',
        9: 'Autumn', 10: 'Autumn', 11: 'Autumn'
    }
    return season_map[date]

def get_time_of_day(date):
    date = pd.to_datetime(date, format='%d/%m/%Y %H:%M')
    hour = date.hour
    if 5 <= hour < 12:
        time_of_day = "Morning"
    elif 12 <= hour < 17:
        time_of_day = "Afternoon"
    elif 17 <= hour < 21:
        time_of_day = "Evening"
    else:
        time_of_day = "Night"

    return time_of_day

def get_day_of_week(date):
    date = pd.to_datetime(date, format='%d/%m/%Y %H:%M')
    day = date.day_name()

    return day
```

Figure 3.8

4. Analysis

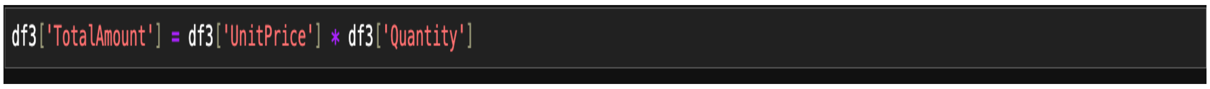
4.1 Monthly Transactions and Revenue

The dataset does not directly provide the total revenue for each month, so it must be calculated based on the available information. The total revenue for a given month is determined by summing the **TotalAmount** for each transaction within that month.

Since the dataset includes both **UnitPrice** and **Quantity** of the items purchased, the **TotalAmount** for each transaction can be computed using the following formula:

$$TotalAmount = Quantity * UnitPrice$$

Once the **TotalAmount** is calculated for each transaction, the sum of these values within a given month will give the total revenue for that month.

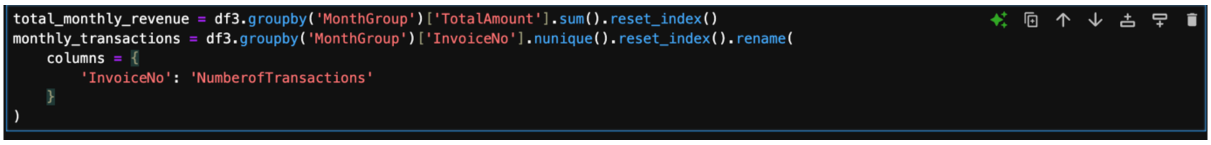


```python
df3['TotalAmount'] = df3['UnitPrice'] * df3['Quantity']
```

Figure 4.1.1

To calculate both the revenue and the number of transactions for each month, the pandas **groupby([column names])** function is used. This function groups the data based on the specified column(s), in this case, by the '**MonthGroup**'. Each group contains rows that share the same values in the selected columns, allowing us to calculate aggregate values (such as sum and count) for each group.

By grouping the data by month, we can then compute the total revenue (sum of **TotalAmount**) and the number of transactions (count of unique **InvoiceNo**) for each month.



```
total_monthly_revenue = df3.groupby('MonthGroup')['TotalAmount'].sum().reset_index()
monthly_transactions = df3.groupby('MonthGroup')['InvoiceNo'].nunique().reset_index().rename(
    columns = {
        'InvoiceNo': 'NumberofTransactions'
    }
)
```

Figure 4.1.2

The results indicate a consistent rise in both the number of transactions and the total sales revenue from April 2011 to November 2011. However, a sharp decline is observed in December 2011, which could be attributed to the dataset not capturing transactions for the entire month as data provided for that month ranges from December 1 to December 9.

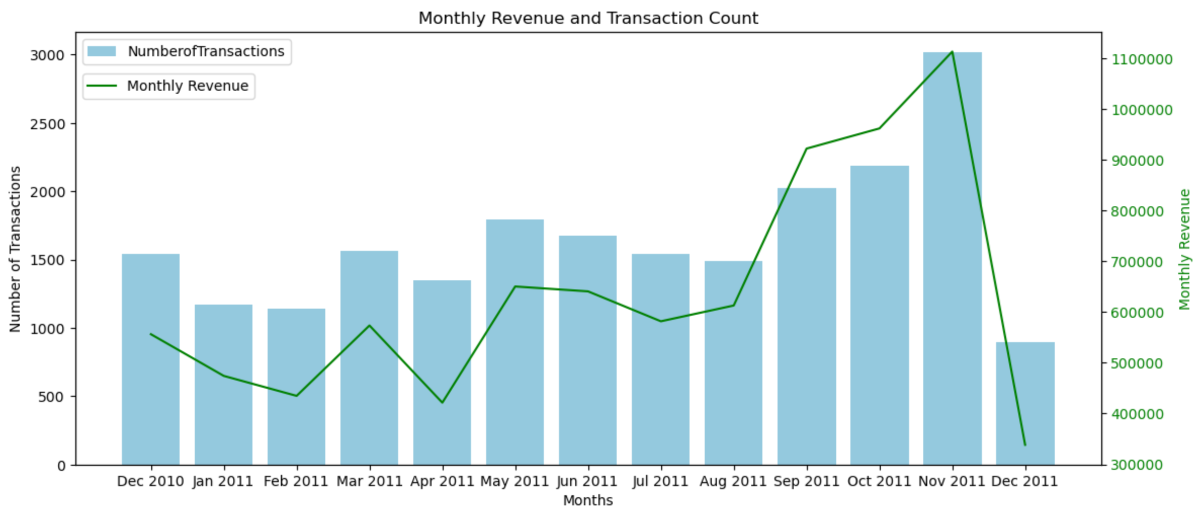
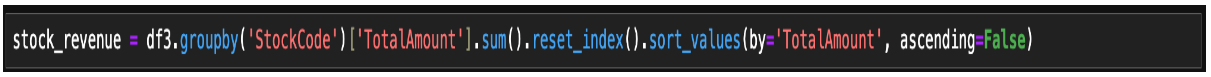


Figure 4.1.3

4.2 Highest Total Revenue

To determine the Stocks with highest revenue, the Pandas **groupby()** function is used once again to group the data according to **StockCode** and then calculate the sum of the **TotalAmount** each of those stocks has generated.



```
stock_revenue = df3.groupby('StockCode')['TotalAmount'].sum().reset_index().sort_values(by='TotalAmount', ascending=False)
```

Figure 4.2.1

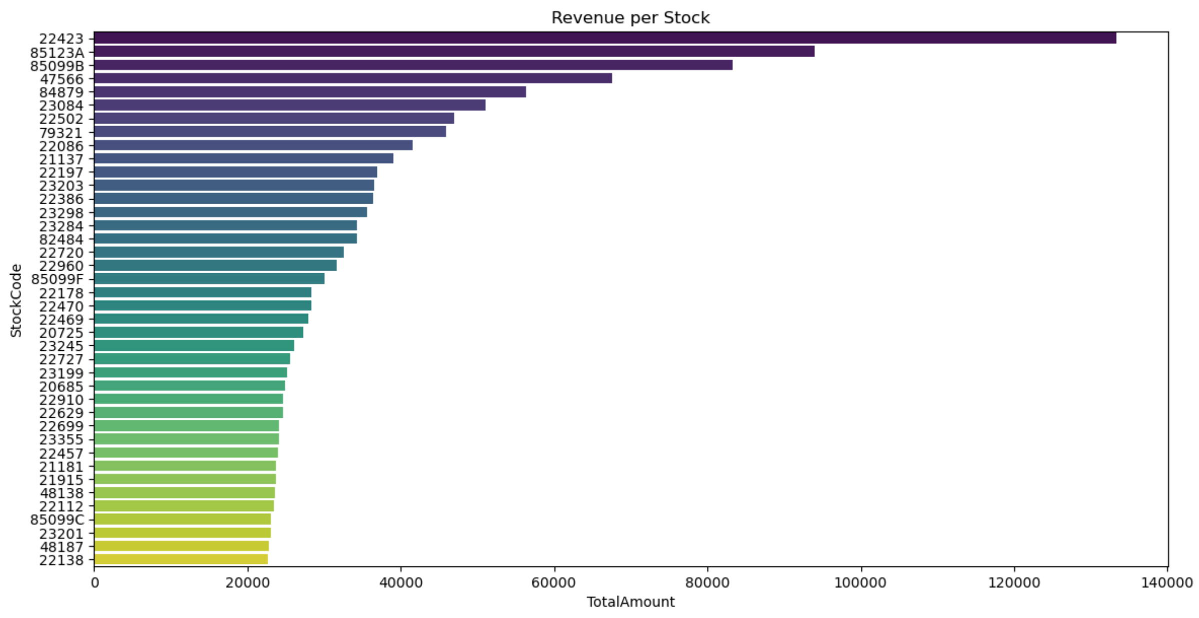


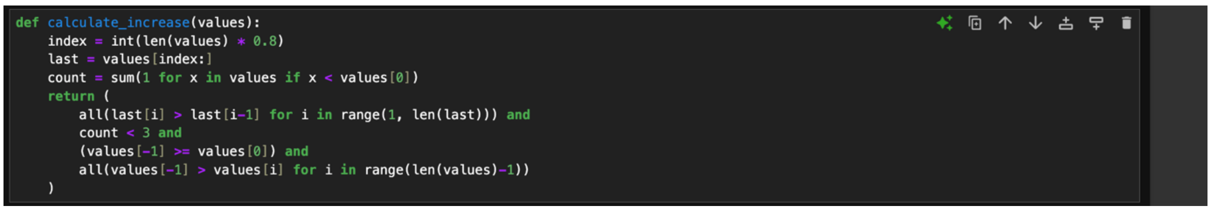
Figure 4.2.2

To identify **StockCodes** with consistent revenue growth, the function **calculate_increase()** was defined. This function returns True if a **StockCode** exhibits a steady increase in revenue over time. Given that data for December 2011 is incomplete, revenue for this month is excluded from the analysis.

The function evaluates consistent growth by focusing on the following criteria:

1. Last 20% Trend: It checks if the revenue in the last 20% of the months shows a monotonic increase.
2. Start vs End Revenue: Ensures that the revenue at the end of the period is not lower than at the start.
3. No Drops Below Start: Confirms that the revenue in the last month is not lower than any previous month's revenue.

By applying these checks, the function identifies stocks with sustained revenue growth, ensuring reliable and meaningful analysis.



```
def calculate_increase(values):
    index = int(len(values) * 0.8)
    last = values[index:]
    count = sum(1 for x in values if x < values[0])
    return (
        all(last[i] > last[i-1] for i in range(1, len(last))) and
        count < 3 and
        (values[-1] >= values[0]) and
        all(values[-1] > values[i] for i in range(len(values)-1))
    )
```

Figure 4.2.3

The analysis identified 33 StockCodes with consistent revenue growth. Among these, some of the best-performing stocks are highlighted below.

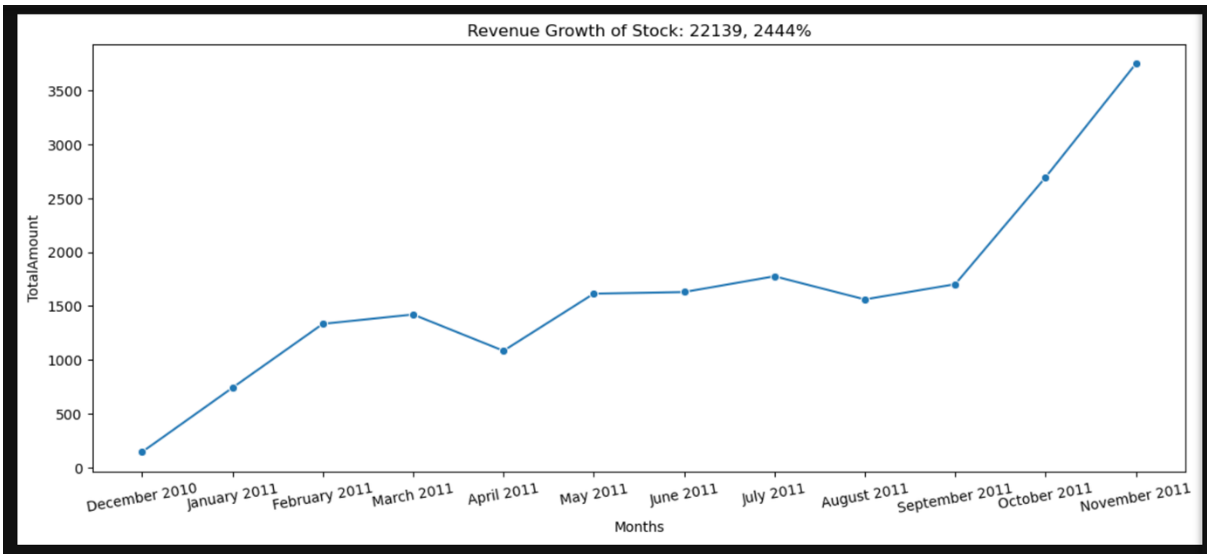


Figure 4.2.4

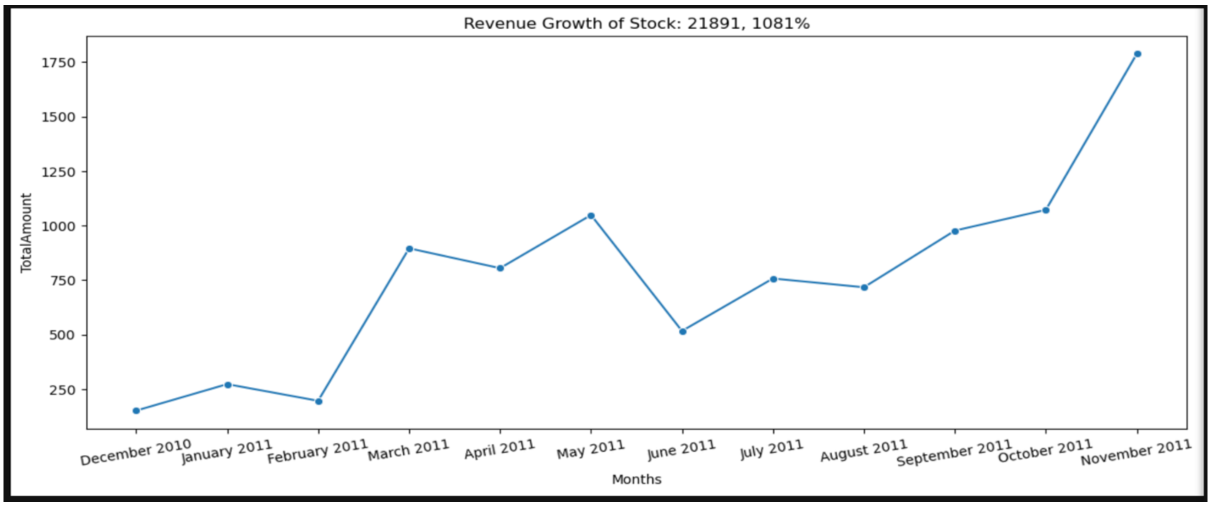


Figure 4.2.5

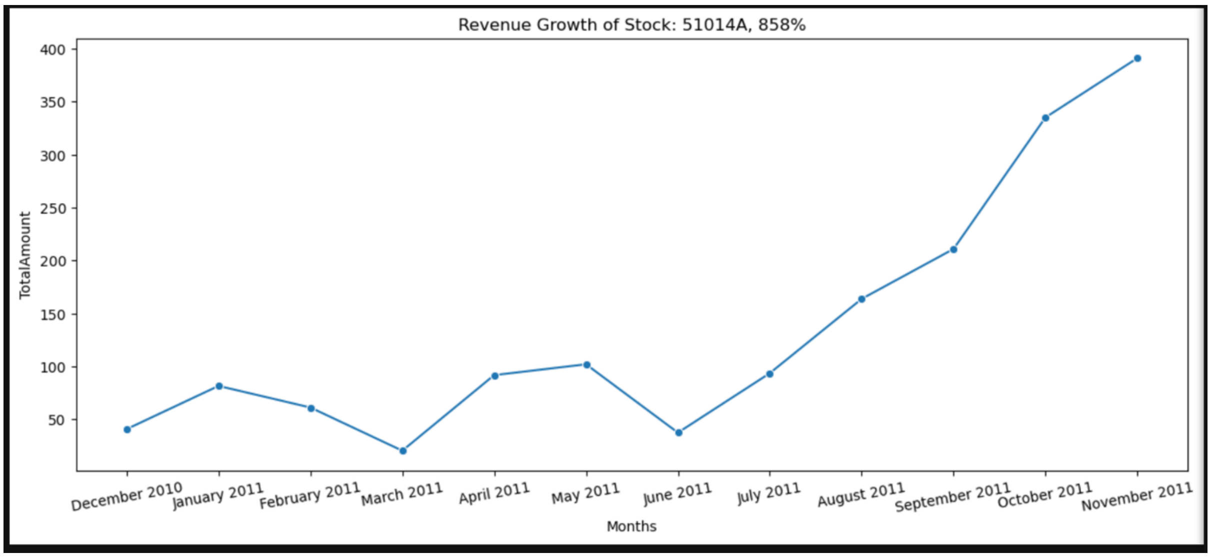


Figure 4.2.6

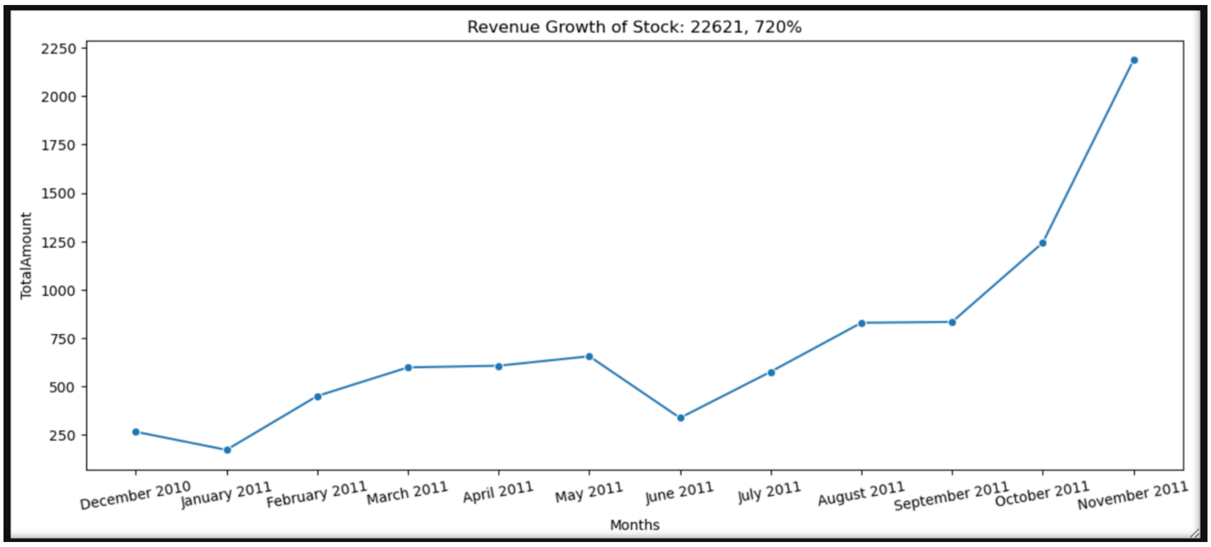


Figure 4.2.7

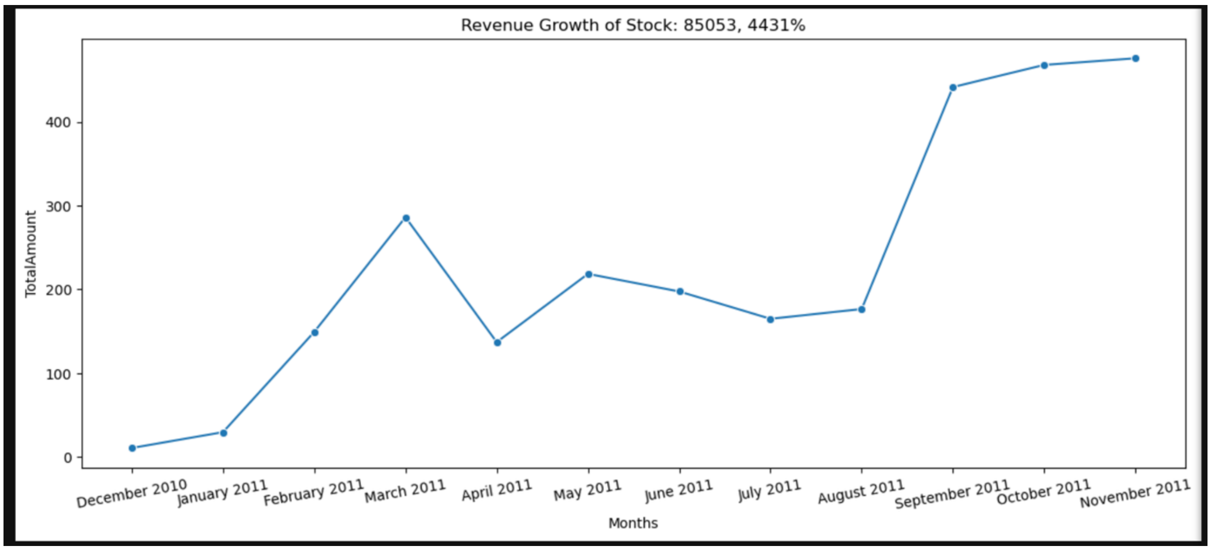


Figure 4.2.8

4.3 Seasonal Variation

To be able to properly identify which **StockCodes** perform better at different seasons, each transaction has been put into one of the following seasons.

1. Winter
2. Summer
3. Fall
4. Autumn

This categorisation is based on the transaction date, allowing for a seasonal analysis of sales performance. By assigning each transaction to a specific season, patterns in revenue or transaction frequency across seasons can be identified, offering insights into seasonal trends for different **StockCodes**.

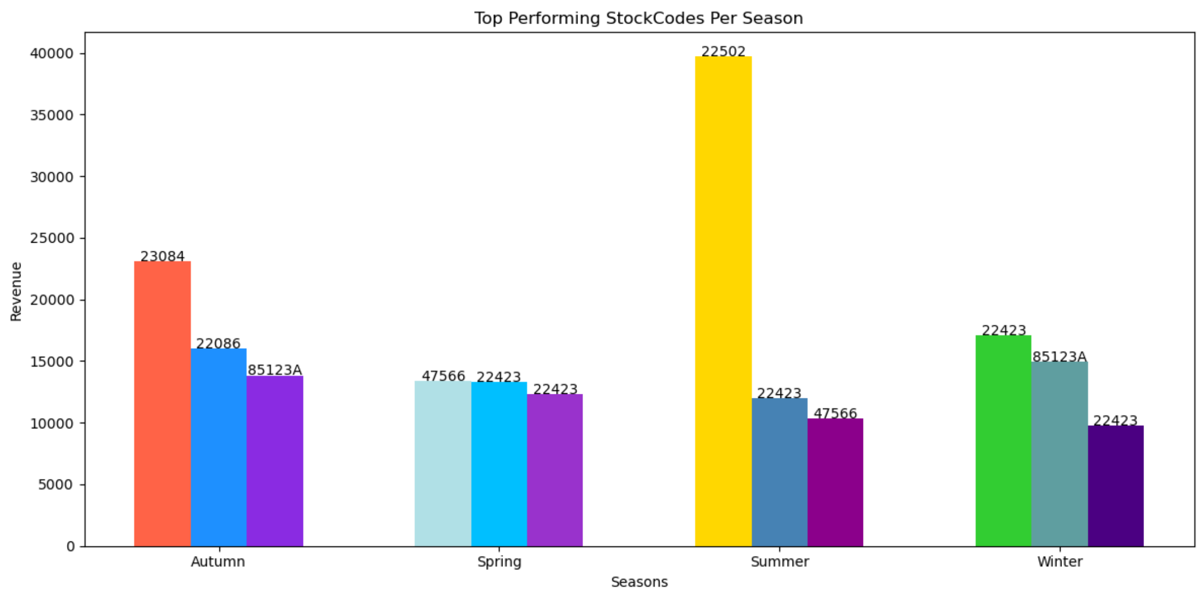


Figure 4.3.1

Some **StockCodes**, such as '22502', show a spike in revenue during Summer season. This indicates that these products might cater to seasonal needs or trends. **StockCodes** like '22423' and '22197' demonstrate consistent revenue and purchase frequency across all seasons. These products may cater to general or everyday needs, making them less sensitive to seasonal trends.
These items could be considered staple products, contributing to steady cash flow throughout the year.

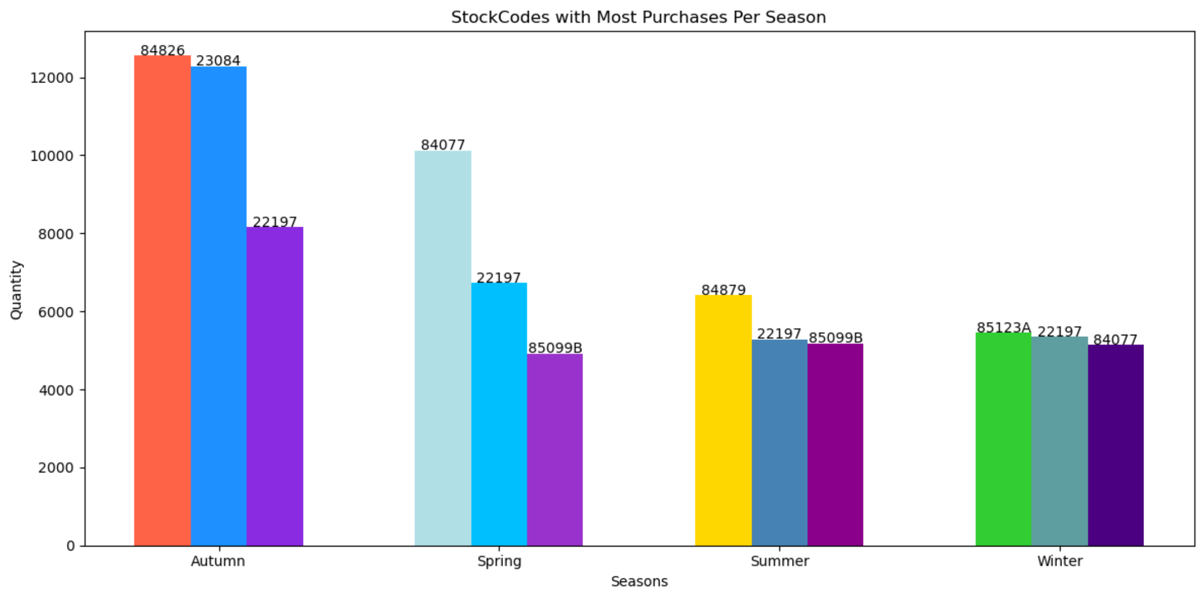


Figure 4.3.2

4.4 Customer Purchasing Behaviour

For analysing shifts in customer purchasing behaviour, some objectives must be defined.

- How often customers buy
- Peak periods of purchases
- Changes in Monthly average spend?

How often customers buy is also known as Purchase Frequency and it can be calculated by getting the total number of unique transactions in a month, divided by the total number of unique customers in that month.

$$Purchasing\ Frequency = \frac{Total\ Number\ of\ Unique\ Transactions}{Number\ of\ Unuque\ Customers}$$

The data shows a sharp decline in purchase frequency in January 2010. This might be attributed to customers taking a break from spending after the festive shopping period in December 2010. Another notable decline in purchase frequency occurred in December 2011, which could be due to the data capturing only the first 9 days of the month (from December 1 to December 9), limiting the scope of transactions for that period.

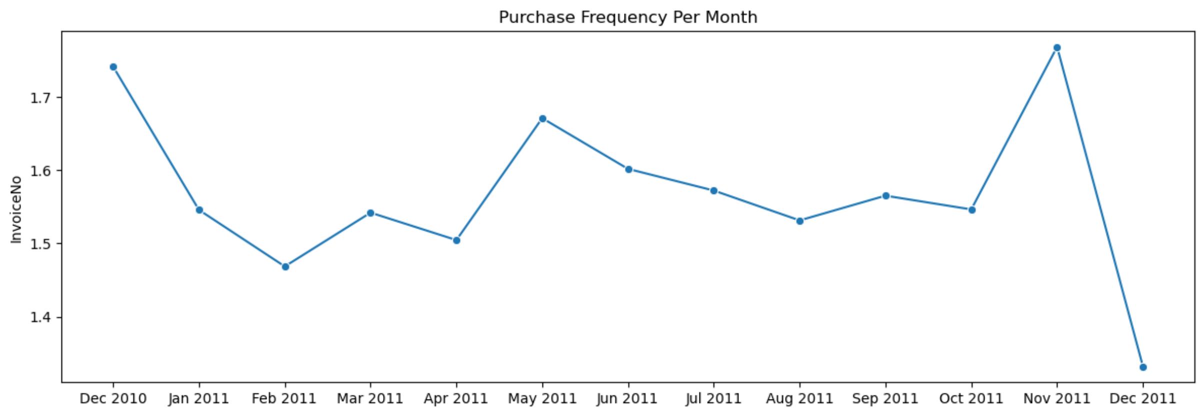


Figure 4.4.1

Aside from these drops, the data indicates a steady rise in purchase frequency from February 2011 to November 2011, suggesting a consistent increase in the frequency with which customers made purchases during this period.

The data indicates that customers do not make purchases on Saturdays, which is likely due to the store being closed on that day. Additionally, the data reveals that the highest volume of purchases occurs on Thursdays.

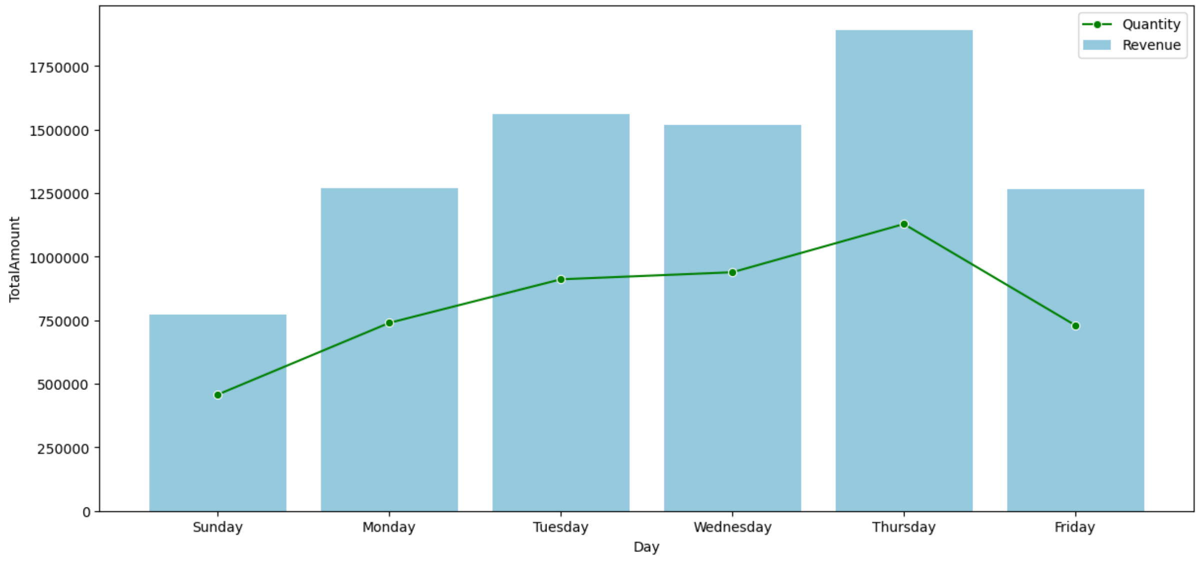


Figure 4.4.2

The data further reveals that the majority of purchases take place between 10 a.m. and 3 p.m., indicating peak shopping hours during the late morning and early afternoon.

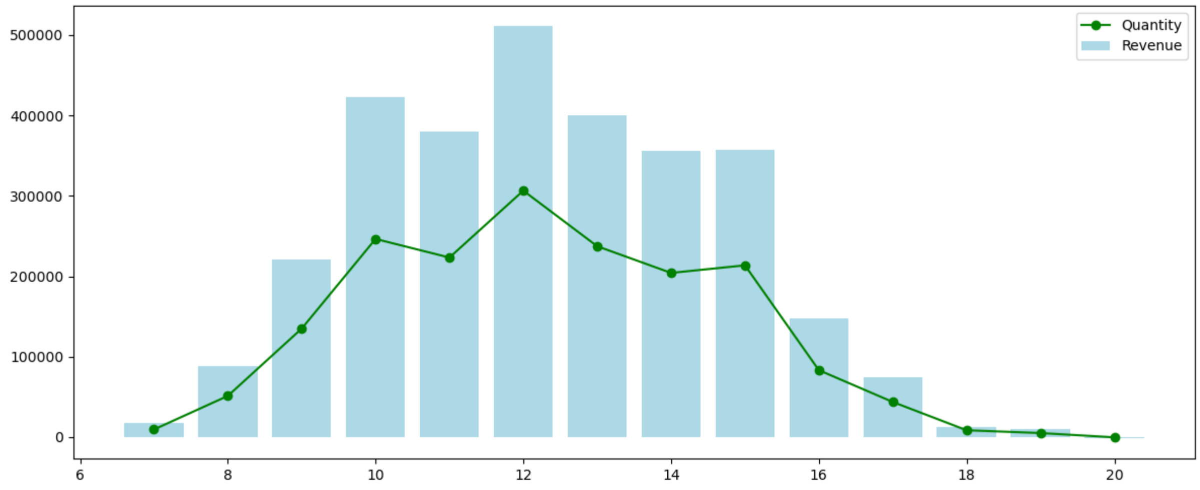
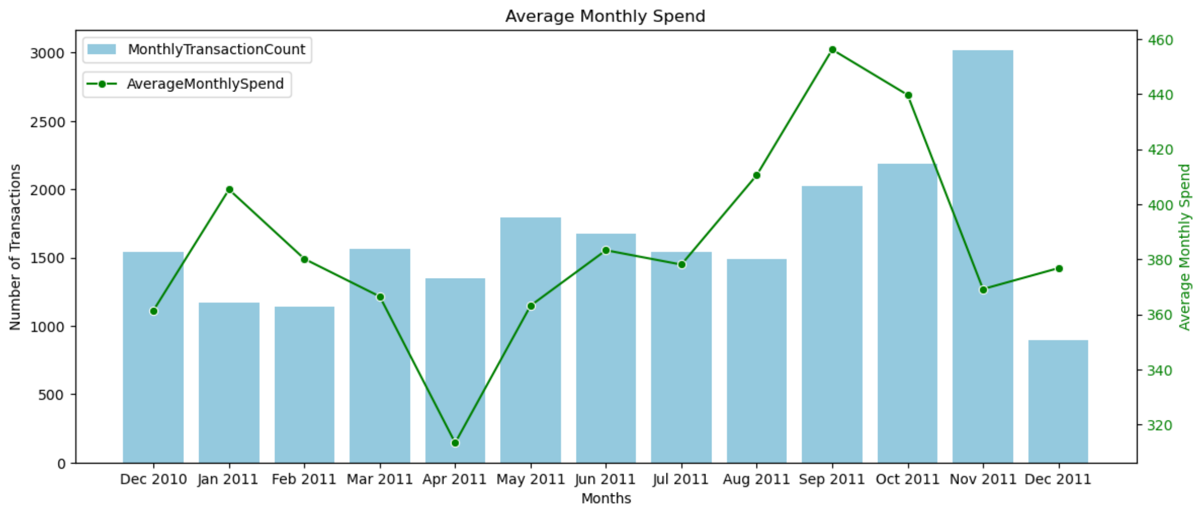


Figure 4.4.3

The data suggests a potential correlation between MonthlyTransactionCount and AverageMonthlySpend. From April 2011 to September 2011, both metrics experienced a steady rise. However, a sharp decline in AverageMonthlySpend was observed in the subsequent months, despite the continued increase in

MonthlyTransactionCount. This trend indicates that while customers were making more transactions, they were likely purchasing lower-cost items.


Average Monthly Spend
MonthlyTransactionCount
AverageMonthlySpend
Number of Transactions
Average Monthly Spend
Months
Dec 2010
Jan 2011
Feb 2011
Mar 2011
Apr 2011
May 2011
Jun 2011
Jul 2011
Aug 2011
Sep 2011
Oct 2011
Nov 2011
Dec 2011