TUNKU ABDUL RAHMAN UNIVERSITY OF MANAGEMENT AND TECHNOLOGY

FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY (FOCS)

ACADEMIC YEAR 2023/2024

BACS3013 DATA SCIENCE

ASSIGNMENT

COURSE NAME	**:**	**Data Science**
COURSE CODE	**:**	**BACS3013**
SESSION	**:**	**202401**
SUBMISSION DEADLINE	**:**	**16 September 2024 (Monday, 10:00 am)**
PRESENTATION	**:**	**Weeks 12, 13 and 14**
TOTAL MARK		**100%**
WEIGHTAGE TO FINAL MARK		**42%**
Group		**3 to 4 members**

Assignment	Marks
TOTAL (100%)	

TUNKU ABDUL RAHMAN UNIVERSITY OF MANAGEMENT AND TECHNOLOGY

FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY (FOCS)

ACADEMIC YEAR 2023/2024

BACS3013 DATA SCIENCE

ASSIGNMENT

Student's Name	Student's ID	Signature	Marks
1. JESMINE TEY KHAI JING	24WMR08866	Jesmine	
2. GAN KHAI LI	24WMR08863	Gan	
3. KIT CHIN JIE YING	24WMR08870	Kit	

Tutor's Name: **Fatin Izzati Binti Ramli**

Date of Submission: **16 September 2024**

BACS3013 Data Science

CLO1 - Apply appropriate data science concepts, statistics, and machine learning to make predictions on data.
CLO2 - Practice exploratory data analysis and visualization methods with the aid of appropriate data science tools.
CLO3 - Analyze relevant techniques, statistical methods, and machine learning algorithms for data analytics.

Criterion	Missing or Unacceptable (0-5 mark)	Poor (6-10 marks)	Good (11-15 marks)	Excellent (16-20 marks)	Marks
Data Preprocessing (CLO3)	The data preprocessing step is missing or very poor. Irrelevant, inaccurate, or inappropriate information.	The data preprocessing steps are not listed and described clearly.	The data preprocessing steps are relevant, offering details about the data science components.	The data preprocessing steps introduction are informative, succinct, and offer sufficiently specific details about the data science components.	
Descriptive Analytics and Exploratory Data Analytics (CLO2)	Analytical methods and results are not properly explained. The explanations are not aligned with the data science and business intelligence.	Analytical methods and results are explained. However, the explanations are confusing, incomplete or lacked relevance to the data science and business intelligence.	Analytical methods and results are explained. The explanations are appropriate and related to the data science and business intelligence.	Analytical methods and results are explained well. The explanations are clear, structured, appropriate and related to the data science and business intelligence.	
Graphing and Visualization (CLO2)	Graphing and visualization are omitted or not irrelevant.	Limited graphing and visualization are presented. The information is unclear or confusing.	Graphing and visualization are presented. The information is clear.	Graphing and visualization are presented systematically. The information is comprehensive and informative. All graphs and visualizations are linked to EDA and advanced analytics appropriately.	
Advanced Analytics and Discussion (CLO3)	Discussions are omitted or confusing. No or very little advanced analytics.	Little discussions are presented. Advanced analytics on the results are unclear or confusing.	Discussions of the results are presented. Advanced analytics on the results are presented.	The significance of the results of the work is discussed, sufficiently inclusive of the advanced analytics.	

				Limitations and future improvements of the studies are identified.	
Model Selection and Report Structure (CLO1)	No justification on the model selection. The structure of the report is incomprehensible, irrelevant, or confusing. Transition from one to another is awkward.	Little justifications are presented on the model selection. The structure of the is weak. Transition from one to another is weak and sometimes difficult to understand.	Adequate justifications are presented on the model selection. The structure of the report is good. Transition from one to another is smooth to show the data science concepts.	Strong justifications are presented on the model selection. The structure of the report is excellent. Transition from one to another is smooth and organized to show the data science concepts.	

TABLE OF CONTENTS

1.0 Business Understanding

1.1 Business Background

TARAMT Sdn. Bhd. is a UK-based online retailer that specializes in one-of-a-kind presents for every occasion. Established in 2004, the firm specializes in supplying a varied choice of gift products for various events such as birthdays, holidays, anniversaries, and other special occasions. TARAMT Sdn Bhd is located in Kuala Lumpur's TARAMT City Centre. It was formed by Tun Dr. Jesmine Tey Khai Jing, Tun Prof. Dr. Gan Khai Li, and Tun Sri Datin Seri Kit Chin Jie Ying with the goal of providing high-quality presents at reasonable costs to both individual consumers and wholesalers. The company serves a diverse customer base, including both retail consumers and corporate customers such as distributors. Retail customers are typically individuals looking for unique and thoughtful gifts while wholesalers purchase items in larger quantities for resale or corporate gifting purposes. This dual customer focus allows the company to reach different market segments, maximizing its reach and revenue potential. Retail consumers are often individuals searching for unique and thoughtful presents, whereas wholesalers buy products in bulk for resale or corporate giving purposes. This dual customer focus enables the organization to access several market sectors, increasing its reach and income potential.

1.2 Business Aim

The primary business aim of the company is to establish itself as a leading provider of unique, high-quality all-occasion gifts, catering to a diverse range of customers, including both individual consumers and wholesalers. The company seeks to differentiate itself in the market by offering an exclusive selection of gifts that are not only creatively designed but also made with the highest standards of craftsmanship. By fostering strong relationships with wholesalers, the company aims to expand its distribution network and reach a wider audience, ensuring that its products are available in various retail outlets across different regions. Additionally, the company is committed to continuously innovating its product line to meet the evolving tastes and preferences of its customers, while also maintaining competitive pricing. Ultimately, the business strives to enhance customer satisfaction and loyalty by delivering exceptional products and services that exceed expectations, thereby driving long-term growth and profitability.

In our pursuit of continuous improvement and strategic growth, we prioritize the collection and analysis of customer data. This process is integral to refining our business strategies and enhancing customer satisfaction. Our primary objectives for collecting customer data include understanding purchasing behaviors, identifying market trends, forecasting demand, and tailoring our offerings to better meet customer needs.

Once data is collected, we utilize advanced analytical tools and techniques to extract actionable insights. The insights gained from our data analysis are instrumental in shaping our business strategy. This enables us to optimize product offerings based on customer preferences, develop targeted marketing campaigns that resonate with specific demographics. This analysis also helps to anticipate market trends and adjust our business model accordingly to improve customer service and support. In conclusion, our commitment to meticulous data collection and analysis is a cornerstone of our strategic planning. By continually refining our approach based on customer insights, we ensure that our business remains responsive and adaptive to the evolving market landscape.

1.3 Business Objectives

The primary business objective of the company, which specializes in selling unique all-occasion gifts with a significant customer base of wholesalers, is to expand its market reach and solidify its position as a leading provider in the industry. To achieve this, the company aims to attract a broader range of wholesalers and retailers, both domestically and internationally, through targeted marketing campaigns and strategic partnerships. By continuously innovating and diversifying its product offerings, the company seeks to meet the evolving needs and preferences of customers, ensuring that its products remain attractive, competitive, and relevant for various occasions.

In addition to product innovation, the company is committed to strengthening its relationships with existing wholesalers. This will be achieved by providing exceptional customer service, offering flexible ordering options, and maintaining competitive pricing, thereby fostering long-term loyalty and repeat business. Operational efficiency is another key focus, with the company aiming to streamline its supply chain and inventory management processes. This will help reduce costs, ensure timely delivery, and maintain the high quality of its products, all of which are essential for sustaining customer satisfaction and profitability.

Furthermore, the company is dedicated to increasing its sales and profitability by implementing strategic pricing, offering bulk discounts for wholesalers, and launching seasonal promotions that cater to the needs of different markets. To enhance brand awareness, the company plans to invest in online marketing, participate in industry trade shows, and engage in other promotional activities that highlight its unique product offerings.

Lastly, the company recognizes the importance of sustainability and ethical practices in today's market. It is committed to integrating these principles into its sourcing, production, and distribution processes, appealing to the growing segment of socially conscious consumers and wholesalers. By focusing on these comprehensive objectives, the company aims to achieve sustainable growth and long-term success in the competitive gift market.

1.4 Inventory of Resources

Hardware	1. Desktop 2. Laptop These are required for daily operations such as administering the e-commerce platform, resolving customer inquiries, processing orders, and doing data analysis. Desktop computers offer powerful performance for demanding tasks, and laptop computers give flexibility for distant work and meetings.
Software	1. Visual Studio Code - Provides a clear environment for code execution, allowing team members to write the code program for data analysis in an easier way. 2. Jupyter Notebook - Provides an interactive environment for data analysis, visualization and sharing code, crucial for analyzing sales data and developing insights. 3. Google Colaboratory - Allows for cloud-based code execution and collaboration, resulting in scalable and accessible data processing and analysis without the need for local resources. 4. Google Docs - Use for collaborative documentation and reporting, allowing team members to generate, modify, and share documents quickly.
Data	1. Online Retails Data by UC Irvine Machine Learning Repository - https://archive.ics.uci.edu/dataset/352/online+retail - A comprehensive dataset for studying online retail transactions, which is critical for understanding customer behavior, sales trends, and product performance. - This freely accessible dataset facilitates data-driven decision-making and modeling.
Personnel	1. Cik Fatin Izzati Binti Ramli, Lecturer of BACS3013 Data Science - Assisting us with this assignment.

Table 1.0: Inventory of Resources

1.5 Requirements, Assumptions and Constraints

1.5.1 Schedule of Completion

This project will start from 31 July 2024 and should be done and submitted by16 September 2024.

1.5.2 Required Comprehensibility and Quality of Results

1. The results should be presented clearly and concisely.
2. The data analysis and models are accurate and validated against reliable benchmarks.
3. The results should be provided with sufficient detail in reports to support decision-making and recommendations.
4. Graphs and charts should be used effectively to communicate findings and insights.

1.5.3 Data Security Concerns

1. The dataset is made available to the public and hence there should be no copyright issues as well as any data security issues.

1.5.4 Assumptions Made by the Project

1. The dataset provided is complete and accurately represents the transactions of the online retail business.
2. External factors affecting sales and customer behavior such as economic conditions remain stable during the project period.

1.5.5 Constraints on the Project

1. Limited timeframe for completing data analysis, modeling and reporting.

1.6 Risk and Contingencies

The dataset may contain missing, incomplete, or erroneous data, affecting the reliability of the analysis and conclusions.

We can implement a contingency plan by doing preliminary data cleaning and validation to detect and fix data quality concerns. We use rigorous data pretreatment procedures to successfully manage missing or incorrect data.

1.7 Data Mining Goals

1. Discover patterns in customer transactions to understand buying behavior and preferences.
2. Group customers based on similarities in their purchasing behavior and demographics
3. Assess the performance of different products in terms of sales volume, revenue and customer feedback.
4. Use historical data to predict future sales trends and market demands
5. Identify unusual patterns or anomalies in sales data that could indicate potential issues or opportunities.
6. Analyze the impact of marketing campaigns on sales and customer behavior.

1.8 Data Mining Success Criteria

1. Each model's accuracy should be more than 80%.
2. The accuracy of each model should be more than 80%.
3. The recall of each model should be more than 80%.
4. The F1 score of each model should be more than 80%.
5. The predictive model for sales forecasting should have at least 85% accuracy.
6. The insights report should include at least three actionable recommendations that result in a 10% improvement in customer retention rates.

1.9 Project Plan

Stage	Duration	Inputs	Outputs	Dependencies
Data Understanding	1 Week	Online retail data	Data overview report and initial insights	Access to data and conduct preliminary analysis
Data Selection	1 Week	Data overview report and business requirements	Selected data subset	Completion of data understanding
Data Cleaning	2 Weeks	Selected data subset	Cleaned data	Completion of data selection
Data Transformation	2 Weeks	Cleaned data	Transformed data	Completion of data cleaning
Data Analytics	4 Weeks	Transformed data	Analytical models and key findings	Completion of data transformation
Data Interpretation and Evaluation	2 Weeks	Analytical models and key findings	Final reports and actionable insights	Completion of data analytics

Table 2.0: Schedule and Details of Stage to be executed

During the data understanding phase, it is critical to examine data features, identify significant factors, and comprehend data sources. It may require a modification in understanding based on preliminary findings and domain expert opinion. Data cleansing, handling missing numbers, and correcting mistakes are all critical during this stage of data cleaning. Re-evaluate data quality and make further adjustments as needed. Data analytics requires the development and testing of analytical models, as well as the evaluation of performance and the refinement of methodologies. Models need to be revised depending on performance indicators and validation findings. During data interpretation, it is critical to interpret analytical results, provide insights, and produce final reports.

When assessing the relationships between time schedules and risks, the following three major issues will arise. The first danger is related to data quality. It has an impact on the data cleanup and transformation stages. Delays in this step might have an influence on the next phases. To address this, do extensive data quality reviews and have backup plans for extra data cleaning. The second concern is technological issues with analytical instruments. It has an influence on the data analysis and interpretation phases. It may cause delays in generating findings. To address these challenges, tools must be updated on

a regular basis and technical help should be provided. Finally, there is a possibility of data selection delays. It has an impact on the schedule for data cleanup and transformation. It may cause delays in following phases. It is crucial to establish firm timeframes for data selection and prioritize critical data for analysis.

When risks arise, certain steps and suggestions are made. First, create a risk management strategy that outlines specific actions for each identified risk. Risk management techniques may be reviewed and updated on a regular basis as the project progresses. Second, develop contingency preparations as soon as a danger is identified. It is critical to communicate risks and mitigation measures to the project team and stakeholders in order to achieve alignment and readiness.

Following the data analysis, the final reports should include a full explanation of the project stages, resources, timeframes, risk management techniques, and assessment methodologies. To guarantee clarity and completeness, the project plan should be presented in an organized fashion with sections for each component.

1.10 Motivation

1. **Market analysis**

   Data mining is an important tool for improving numerous parts of business and research since it can extract valuable insights from enormous databases. Data mining in market analysis gives a full perspective of client behavior by analyzing purchase histories, demographics, and other profile data. This helps firms to adjust their marketing campaigns and provide individualized experiences, hence increasing client engagement and happiness. For example, Adidas uses data mining to segment their email lists and target certain client groups with appropriate discounts.

2. **Fraud detection**

   Another important application for data mining is fraud detection. It assists in detecting fraudulent activity by studying transaction patterns and abnormalities. Data mining methods are used in industries such as e-commerce and banking to spot abnormalities that might signal credit card fraud, telecommunications fraud, or computer infiltration, therefore securing financial transactions and sensitive information.

3. **Customer retention**

   Client retention is critical for sustaining corporate success, and data mining helps to identify the elements that drive client loyalty. Businesses may design ways to improve customer happiness and minimize churn rates by monitoring their purchasing habits and comments. This tailored strategy aids in retaining a loyal consumer base and generating long-term success.

4. **Production control**

   Data mining in production control helps to improve operational efficiency by evaluating production data to optimize operations and save costs. Manufacturers utilize these data to track important parameters, detect bottlenecks, and anticipate equipment breakdowns, resulting in better production management and cost savings.

5. **Scientific exploration**

   Finally, in scientific investigation, data mining makes it easier to find patterns and trends in large databases. Researchers use data mining techniques to analyze enormous amounts of data, create and test ideas, and enhance scientific understanding. This technique improves knowledge in domains like genetics and climate science, fostering innovation and discovery.

2.0 Data Understanding

2.1 Data Collection

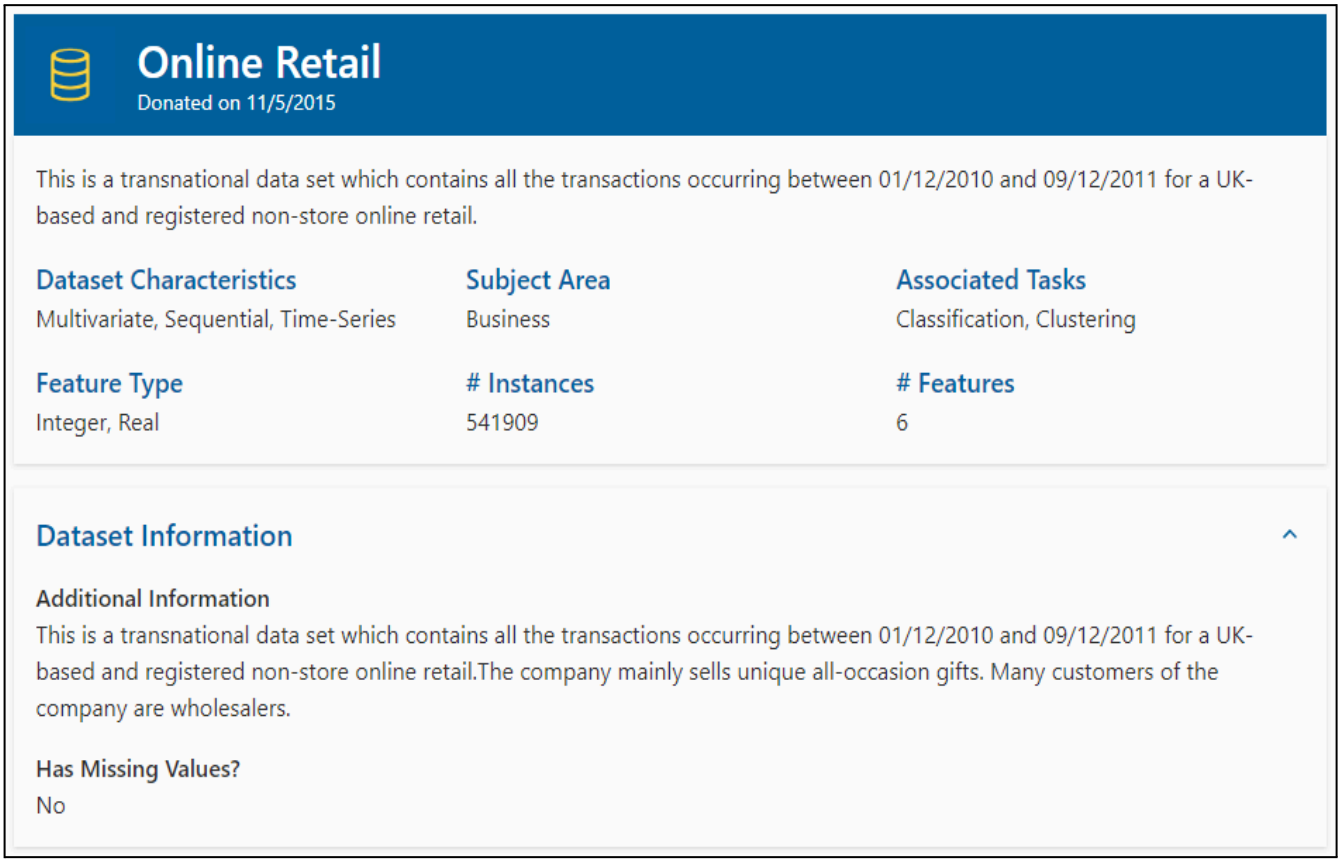


Figure 1.0: Source of Data

Data collection involves gathering information from various sources to answer questions or meet specific needs. It encompasses the processes of collecting, recording, and organizing data for analysis. During the Data Understanding phase, we initiate data collection by obtaining the 'Online Retail' dataset from the UCI Machine Learning Repository, which can be accessed at https://archive.ics.uci.edu/dataset/352/online+retail.

2.2 Data Description

2.2.1 Data Loading

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```
[1] ✓ 2.7s

Figure 2.0: Code of Importing necessary libraries

```
#Read csv file
df = pd.read_csv('onlineRetail.csv')
```
[2] ✓ 0.4s

Figure 2.1:Code of Importing Data

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	1/12/2010 8:26	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	1/12/2010 8:26	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	1/12/2010 8:26	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	1/12/2010 8:26	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	1/12/2010 8:26	3.39	17850.0	United Kingdom
...	...	...	...	...	...	...	...	...
541904	581587	22613	PACK OF 20 SPACEBOY NAPKINS	12	9/12/2011 12:50	0.85	12680.0	France
541905	581587	22899	CHILDREN'S APRON DOLLY GIRL	6	9/12/2011 12:50	2.10	12680.0	France
541906	581587	23254	CHILDRENS CUTLERY DOLLY GIRL	4	9/12/2011 12:50	4.15	12680.0	France
541907	581587	23255	CHILDRENS CUTLERY CIRCUS PARADE	4	9/12/2011 12:50	4.15	12680.0	France
541908	581587	22138	BAKING SET 9 PIECE RETROSPOT	3	9/12/2011 12:50	4.95	12680.0	France

541909 rows × 8 columns

Figure 2.2: Data Set for Online Retail

Based on the **Figure 2.0** provided, we import the necessary Python libraries commonly used for data analysis, visualization and numerical operations. We import Pandas library to work with structured data such as data frames and provide tools for data manipulation, cleaning and analysis. Besides, we import the Pyplot module from the Matplotlib library and rename it as plt. Matplotlib is a popular library for creating static, interactive and animated visualizations in Python. The Pyplot module provides a MATLAB-like interface to create plots and graphs easily. We create various types of visualizations such as line graphs, bar charts and scatter plots by using Pyplot. We also import the Seaborn library because it is a data visualization library built on top of

Matplotlib that provides more sophisticated and attractive visualizations with simpler syntax. Lastly, we import the important library which is the NumPy library because it is a good use for numerical operations in Python, especially for handling arrays and matrices as well as a large collection of mathematical functions.

We utilize the Python 'pandas' library to read a CSV file named "online_Retail.csv." This process involves using the pandas.read_csv() function to load the contents of the file into a DataFrame. This DataFrame allows us to efficiently manipulate and analyze the data for further exploration and insights. We add the CSV file into the folder so it can read the file in an easier way.

Figure 2.2 shows the data set of the online retail**.** In this dataset, there are eight variables: InvoiceNo, StockCode, Description, Quantity, InvoiceDate, UnitPrice, CustomerID, and Country. Quantity and UnitPrice are designated as the dependent variable and the remaining variables serve as independent variables. The data types in the dataset are as follows: InvoiceNo, StockCode, Description, CustomerID, and Country are categorical variables. Quantity is a numerical variable. InvoiceDate is a date variable, and UnitPrice is a continuous variable. The total number of rows in the data set is 541,909. It means that it has about 4,335,272 data cells in the data set. Since it has such a large data set, we decided to conduct some data understanding to read the basic information of the data set.

2.2.2 Basic Data Understanding

```
#Check the data types of the features in df
df.dtypes
```

```
[4]  ✓ 0.0s
```

```
InvoiceNo       object
StockCode       object
Description     object
Quantity         int64
InvoiceDate     object
UnitPrice      float64
CustomerID     float64
Country         object
dtype: object
```

Figure 2.3:Data Type of Variables

Name	**Type**	**Dependent / Independent Variable**	**Description**
InvoiceNo	Categorical	Independent Variable	A unique 6-digit integral number for each transaction. If the code is starting with letter 'c', it indicates a cancellation.
StockCode	Categorical	Independent Variable	A unique 5-digit integral number for each distinct product.
Description	Categorical	Independent Variable	A textual description of the product.
Quantity	Numerical	Dependent Variable	The quantities of each product sold in a transaction.
InvoiceDate	Date	Independent Variable	The date and time when the transaction occurs.
UnitPrice	Continuous	Dependent Variable	The price per unit of a product.
CustomerID	Categorical	Independent Variable	A unique 5-digit number for each customer.

Country	Categorical	Independent Variable	The country where the customer is located.

Table 3.0:Explanation of Variables

```
#Calculate the statistical information of df
df.describe()
```

[5] ✓ 0.1s

	Quantity	UnitPrice	CustomerID
count	541909.000000	541909.000000	406829.000000
mean	9.552250	4.611114	15287.690570
std	218.081158	96.759853	1713.600303
min	-80995.000000	-11062.060000	12346.000000
25%	1.000000	1.250000	13953.000000
50%	3.000000	2.080000	15152.000000
75%	10.000000	4.130000	16791.000000
max	80995.000000	38970.000000	18287.000000

Figure 2.4: Basic Understanding for Statistical information of the data set

To gain a better understanding of the dataset, we performed an initial statistical analysis using the `describe()` function in Pandas. The function provides a summary of key statistical metrics for each numerical column in the DataFrame.

The purpose of using df.describe() is to summarize important characteristics of the dataset, such as distribution, central tendency and variability in a quick and efficient manner. These metrics help us to identify the basic patterns, trends and potential outliers in the data before conducting data selection and data cleaning.

2.2.3 Descriptive Statistics

2.2.3.1 Descriptive Statistics of TotalSales

```
#Create a column calles 'TotalSales'
df['TotalSales'] = df['Quantity'] * df['UnitPrice']
```
[6] ✓ 0.0s

Figure 2.5:Code of calculating TotalSales

```
#Calculate the statistical information for the TotalSales
df['TotalSales'].describe()
```
[7] ✓ 0.0s

```
count    541909.000000
mean         17.987795
std         378.810824
min     -168469.600000
25%           3.400000
50%           9.750000
75%          17.400000
max      168469.600000
Name: TotalSales, dtype: float64
```

Figure 2.6: Descriptive Statistics of 'TotalSales'

```
df.head()
```
[8] ✓ 0.0s

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	TotalSales
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	1/12/2010 8:26	2.55	17850.0	United Kingdom	15.30
1	536365	71053	WHITE METAL LANTERN	6	1/12/2010 8:26	3.39	17850.0	United Kingdom	20.34
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	1/12/2010 8:26	2.75	17850.0	United Kingdom	22.00
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	1/12/2010 8:26	3.39	17850.0	United Kingdom	20.34
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	1/12/2010 8:26	3.39	17850.0	United Kingdom	20.34

Figure 2.7: The first 5 data in the DataFrame with the new variable 'TotalSales'

We created a new column called 'TotalSales'. **Figure 2.6** presents a detailed summary of the descriptive statistics for the 'TotalSales' variable, providing insights into measures such as mean, median, standard deviation, minimum, maximum, and percentiles. Based on **Figure 2.6**, the dataset consists of 541,909 records. The mean value of the TotalSales variable is approximately 17.99, with a precise calculation of 17.987795. The median of the TotalSales is 9.75. The standard deviation is 378.810824, which is approximately 378.81. The minimum and maximum values of TotalSales are -168,469.6 and 168,469.6, respectively. Additionally, the first quartile (25th percentile) is 3.40, and the third quartile (75th percentile) is 17.40. Additionally, the dataframe, illustrated in **Figure 2.7**, displays the first five rows of data, including a newly calculated variable, 'TotalSales.' TotalSales is derived by multiplying the 'Quantity' with the 'UnitPrice' for each transaction.

2.2.3.2 Descriptive Statistics of InvoiceNo

```
#Description Statistics for InvoiceNo

#Group the Invoice based on the quantity of items sold
invoice_stats = df.groupby('InvoiceNo')['Quantity'].sum().reset_index()
#Calculate the total number of Invoice
total_invoice = df['InvoiceNo'].nunique()

#Print the information found
print("Descriptive Statistics for InvoiceNo")
print("====================================")
print("Mode: ", df['InvoiceNo'].mode()[0])
print("Total Invoice: ", total_invoice)
print("====================================")
print(invoice_stats.head())
```

[9] ✓ 0.0s

Figure 2.8: Code of Descriptive Statistics for 'InvoiceNo'

```
Descriptive Statistics for InvoiceNo
====================================
Mode:  573585
Total Invoice:  25900
====================================
  InvoiceNo  Quantity
0    536365        40
1    536366        12
2    536367        83
3    536368        15
4    536369         3
```

Figure 2.9: The Descriptive Statistics of 'InvoiceNo'

The descriptive statistics of 'InvoiceNo' are shown in **Figure 2.9**. Based on this figure, we can determine the mode of InvoiceNo and the total number of invoices in the dataset.The most common InvoiceNo is 573585. There are 25,900 unique InvoiceNos in the dataset, indicating that the dataset includes 25,900 distinct invoices. The dataset contains a total of 541,909 records, each of which has an associated InvoiceNo, indicating that every record includes this information.

2.2.3.3 Descriptive Statistics of StockCode

```
#Descriptive Statistics for StockCode

#Group the StockCode by quantity of items sold
stockCode_stats = df.groupby('StockCode')['Quantity'].sum().reset_index()
#Calculate the total number of stock
total_stock = df['StockCode'].nunique()

#Print the information found for the StockCode
print("Descriptive Statistics for Stock")
print("==================================")
print("Mode: ", df['StockCode'].mode()[0])
print("Total Stock: ", total_stock)
print("==================================")
print(stockCode_stats.head())
```

[10] ✓ 0.0s

Figure 2.10: Code of Descriptive Statistics for 'StockCode'

```
Descriptive Statistics for Stock
==================================
Mode:  85123A
Total Stock:  4070
==================================
  StockCode  Quantity
0     10002      1037
1     10080       495
2     10120       193
3    10123C       -13
4    10123G       -38
```

Figure 2.11: Descriptive Statistics for 'StockCode'

The descriptive statistics of 'StockCode' are presented in **Figure 2.11**. According to the figure, we can determine the mode of the stock and the total number of stocks that the online retail store sold. The most frequent StockCode is 85123A.There are 4,070 unique stockCode, indicating that 4,070 different products are represented in the dataset. **Figure 2.11** shows the first five rows of the stock with the quantity of item sold in the data set.

2.2.3.4 Descriptive Statistics for Description

```
#Descriptive Statistics for Description

#Group the description  by the quantity of items sold
description_stats = df.groupby('Description')['Quantity'].sum().reset_index()
#Calculate the total number of stock
total_items = df['Description'].nunique()

#Print the information found for the stock
print("Descriptive Statistics for Stock (Name)")
print("=======================================")
print("Mode: ", df['Description'].mode()[0])
print("Total Stock: ", total_stock)
print("=======================================")
print(description_stats.head())
```

[11] ✓ 0.1s

Figure 2.12: Code of Descriptive Statistics for 'Description'

```
Descriptive Statistics for Stock (Name)
=======================================
Mode:  WHITE HANGING HEART T-LIGHT HOLDER
Total Stock:  4070
=======================================
                         Description  Quantity
0   4 PURPLE FLOCK DINNER CANDLES       144
1   50'S CHRISTMAS GIFT BAG LARGE      1913
2               DOLLY GIRL BEAKER      2448
3     I LOVE LONDON MINI BACKPACK       389
4     I LOVE LONDON MINI RUCKSACK         1
```

Figure 2.13: Descriptive Statistics for 'Description'

The descriptive statistics of 'Description' are presented in **Figure 2.13**. According to the figure, we can determine the mode of the stock (name) and the total number of stocks that the online retail store sold. The most frequent stock sold (name) is White Hanging Heart T-Light Holder. There are 4,070 unique stocks (name), indicating that 4,070 different products are represented in the dataset. **Figure 2.13** shows the first five rows of the stock with the quantity of item sold in the data set.

2.2.3.4 Descriptive Statistics of CustomerID

```
#Descriptive statistics for CustomerID

#Group the customer by the total quantity of items sold
customer_stats = df.groupby('CustomerID')['Quantity'].sum().reset_index()
#Calculate the total number of customer
total_customer = df['CustomerID'].nunique()

#Print the information found for the customer
print("Descriptive Statistics for Customer")
print("====================================")
print("Mode: ", df['CustomerID'].mode()[0])
print("Total Customer: ", total_customer)
print("====================================")
print(customer_stats.head())
```

[12] ✓ 0.0s

Figure 2.14: Code of Descriptive Statistics for CustomerID

```
Descriptive Statistics for Customer
====================================
Mode:  17841.0
Total Customer:  4372
====================================
   CustomerID  Quantity
0     12346.0         0
1     12347.0      2458
2     12348.0      2341
3     12349.0       631
4     12350.0       197
```

Figure 2.15: Descriptive Statistics for CustomerID

The descriptive statistics for 'CustomerID' are shown in **Figure 2.15**. From this figure, we can identify several key details which are the mode of the CustomerID and the total number of customers in the data set. Specifically, the most frequently occurring CustomerID is 17841. There are 4,372 unique CustomerIDs, indicating that there are 4,372 distinct customers recorded. **Figure 2.15** shows the first five rows of the customer with the quantity of item they bought in the data set.

2.2.3.5 Descriptive Statistics of Country

```
#Descriptive statistics for country

#Group the country by the quantity of items sold
country_stats = df.groupby('Country')['Quantity'].sum().reset_index()
#Calculate the total quantity of items sold
total_country = df['Country'].nunique()

#Print the information found for the country
print("Descriptive Statistics for Country")
print("====================================")
print("Mode: ", df['Country'].mode()[0])
print("Total Country: ", total_country)
print("====================================")
print(country_stats.head())
```

[13] ✓ 0.0s

Figure 2.16: Code of Descriptive Statistics for 'Country'

```
Descriptive Statistics for Country
====================================
Mode:  United Kingdom
Total Country:  38
====================================
     Country  Quantity
0  Australia     83653
1    Austria      4827
2    Bahrain       260
3    Belgium     23152
4     Brazil       356
```

Figure 2.17: Descriptive Statistics for 'Country'

The descriptive statistics for 'Country' are illustrated in **Figure 2.17**. From this figure, we can discern several key details which are the mode of the Country and the total number of unique Country in the data set. Specifically, the most frequently occurring country is the United Kingdom. There are 38 unique countries recorded, indicating that the dataset includes information from 38 different countries. It means there are 38 different countries in this data set.

2.3 Data Exploration

```
#Change the InvoiceDate to datetime format
df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'], format='%d/%m/%Y %H:%M')

#Extract the year, month and day from the InvoiceDate
df['Year'] = df['InvoiceDate'].dt.year
df['Month'] = df['InvoiceDate'].dt.month
df['Day'] = df['InvoiceDate'].dt.day

#Generate the day of week for each of the data
df['DayOfWeek'] = df['InvoiceDate'].dt.strftime('%A')

#Merge the year and month into one format (2011-12)
df['YearMonth'] = df['InvoiceDate'].dt.strftime('%Y-%m')
```

[14] ✓ 4.8s

Figure 3.0: Adjustment for InvoiceDate

```
df
```

[15] ✓ 0.0s Python

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	TotalSales	Year	Month	Day	DayOfWeek	YearMonth
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom	15.30	2010	12	1	Wednesday	2010-12
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom	20.34	2010	12	1	Wednesday	2010-12
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom	22.00	2010	12	1	Wednesday	2010-12

Figure 3.1: View data set

Before visualizing the data, we convert the InvoiceDate column to a proper date time format and extract useful data-related features such as year, month, day and day of the week for further analysis of trends based on the day. Besides, we also create a new column that combines the year and month in the format 'YYYY-MM' for a better understanding of the data in the specific month of the specific year.

In this subtopics, we decided to visualize all the features in the data set to get valuable insights and understand the relationships between variables. We use histograms to understand the distribution of individual features as they can show the frequency of data points within specific ranges. We also use box plots to identify the outliers and understand the distribution and spread of features. We use the correlation heatmap to display the correlation between different features to identify strongly related features. We also use the pie charts to show proportions of categorical features.

2.3.1 Relationship Exploration and Visualization Using Heatmap

```
from sklearn.preprocessing import LabelEncoder

#Copy the Dataframe df
corr_df = df.copy()

#Encoding the categorical variables
corr_df['InvoiceNo'] = corr_df['InvoiceNo'].astype(str)
corr_df['InvoiceNo'] = LabelEncoder().fit_transform(corr_df['InvoiceNo'])
corr_df['InvoiceNo'] = corr_df['InvoiceNo'].astype(int)

corr_df['StockCode'] = corr_df['StockCode'].astype(str)
corr_df['StockCode'] = LabelEncoder().fit_transform(corr_df['StockCode'])
corr_df['StockCode'] = corr_df['StockCode'].astype(int)

corr_df['Description'] = corr_df['Description'].astype(str)
corr_df['Description'] = LabelEncoder().fit_transform(corr_df['Description'])
corr_df['Description'] = corr_df['Description'].astype(int)

corr_df['Country'] = corr_df['Country'].astype(str)
corr_df['Country'] = LabelEncoder().fit_transform(corr_df['Country'])
corr_df['Country'] = corr_df['Country'].astype(int)

corr_df['DayOfWeek'] = corr_df['DayOfWeek'].astype(str)
corr_df['DayOfWeek'] = LabelEncoder().fit_transform(corr_df['DayOfWeek'])
corr_df['DayOfWeek'] = corr_df['DayOfWeek'].astype(int)

corr_df['YearMonth'] = corr_df['YearMonth'].astype(str)
corr_df['YearMonth'] = LabelEncoder().fit_transform(corr_df['YearMonth'])
corr_df['YearMonth'] = corr_df['YearMonth'].astype(int)
[16]  ✓ 0.8s
```

Figure 3.2: Code of Preparation for correlation matrix

```
#Clear the missing values
corr_df.dropna(subset = ['CustomerID'], inplace = True)
[17]  ✓ 0.0s
```

Figure 3.3: Code of clearing missing values in the data set

Before visualizing the correlation matrix, we encode the categorical variables and clean the data to prepare the data set for machine learning algorithms by using the Label Encoding. This step is essential because the machine learning models cannot handle non-numerical data.

From **Figure 3.3**, a copy of the original DataFrame df is created to avoid modifying the original data directly. The copied DataFrame is stored in `corr_df` and will be used for the correlation matrix. Categorical columns such as InvoiceNo, StockCode, Description, Country, DayOfWeek and YearMonth are converted into numerical values using the LabelEncoder. Lebel Encoding assigns a unique integer to each distinct category in these columns.

From **Figure 3.3**, we handle the missing values in the CustomerID column since CustomerID is important for customer segmentation and sales analysis. Rows with missing CustomerID values are dropped to ensure a clean and complete data for preparing the correlation matrix.

2.3.1.1 Relationship Visualization

```
#Calculate the correlation for each of the features in corr_df
correlation_matrix =  corr_df.corr()

#Plot a heatmap to visualize the correlation matrix
plt.figure(figsize=(15,15))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Heatmap of Correlation Matrix')
plt.show()
```

[18] ✓ 0.7s

Figure 3.4:Code of Heatmap of Correlation Matrix

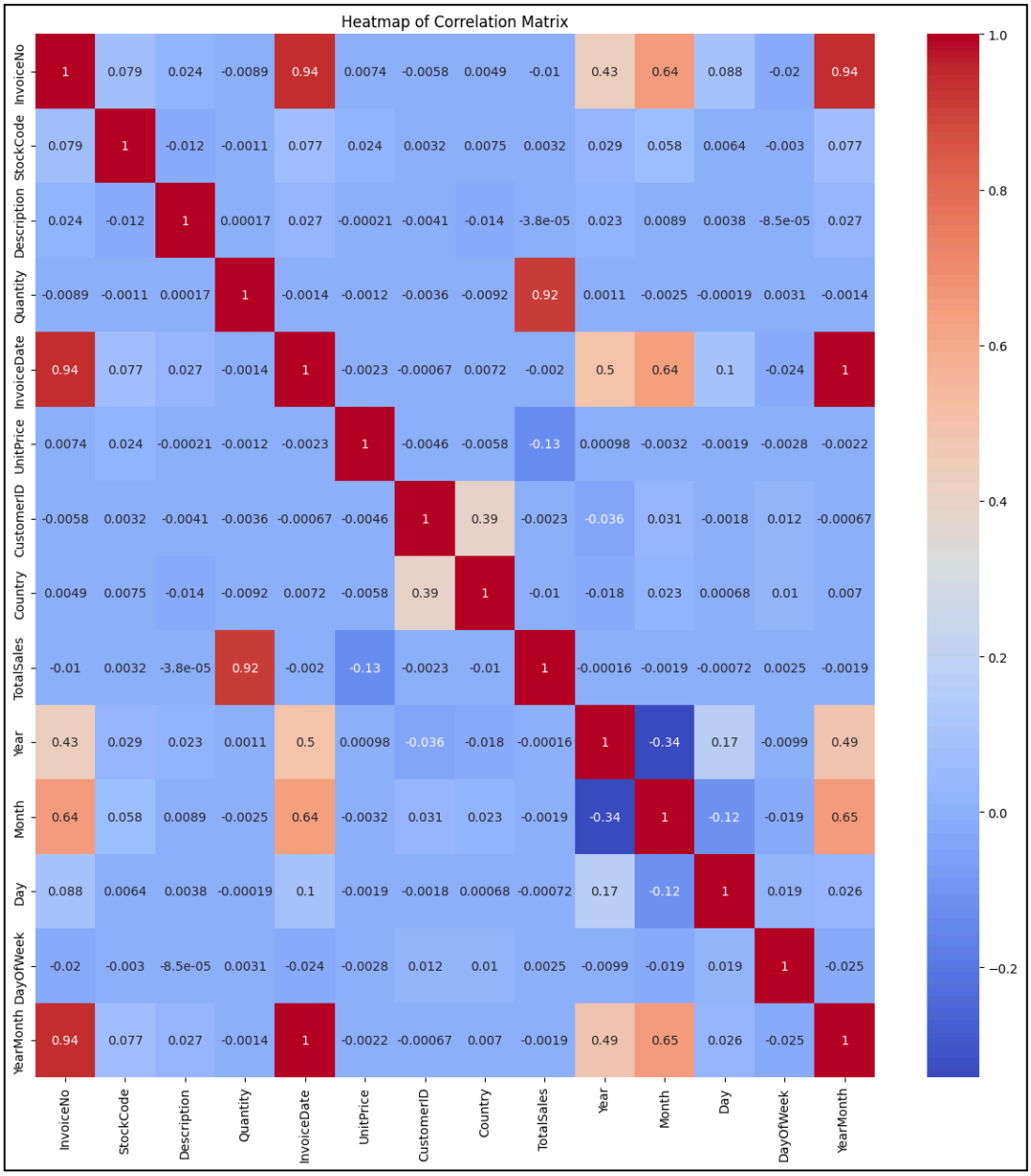


Figure 3.5:Heatmap of Correlation Matrix

2.3.1.2 Relationship Exploration

In this section, we calculate the correlation between numerical features in the dataset and visualize the results using a heatmap. This helps to identify relationships between different variables which is essential for feature selection and understanding how the features interact with each other. The Pearson correlation coefficient measures the linear relationship between two variables. It ranges from -1 to 1. A value close to 0 indicates no linear correlation between the variables.

The heatmap will show color gradients ranging from blue to red. Red indicates a strong positive correlation while blue indicates a strong negative correlation. Light shades represent weaker correlations where there is little to no linear relationship between the variables. Highly correlated features may indicate redundancy and only one of them may need to be included in the model. The heatmap helps identify potential multicollinearity between features which can be an issue in regression models.

Figure 3.5 illustrates a heatmap of the correlation matrix, revealing key relationships among the InvoiceNo, StockCode, Description, Quantity, InvoiceDate, UnitPrice, CustomerID, Country, TotalSales, Year, Month, Day, DayOfWeek and YearMonth. The heatmap shows a strong positive correlation **(0.94)** between **InvoiceNo and TotalSales**. This suggests that as InvoiceNo increases, the total sales also tend to increase. Besides, the correlation between **Quantity and TotalSales** is **0.92**, indicating a strong positive relationship. This makes intuitive sense because as the quantity of the items purchased increases, the total sales would naturally increase. There is a moderate positive correlation **(0.39)** between **Country and CustomerID**. This might indicate some level of geographic clustering of customer behaviors in the data set. The correlation between **Month and TotalSales** is **0.64**, which suggests that there is a seasonal trend in sales. Different months likely have varying sales performance. The correlation between **YearMonth and InvoiceNo** is very high **(0.94)**. This likely indicates that the dataset's invoices are ordered chronologically and the InvoiceNo increases as time progresses. The correlation between **Month and Day** is negative **(-0.34)**. This indicates that the day of the month has a somewhat weak inverse relationship with the month number in the dataset. Some features like **Country and UnitPrice** or **CustomerID and InvoiceNo** show very weak correlations indicating that these features do not exhibit much linear relationship with one another.

In conclusion, the heatmap highlights the strong positive correlations between features like Quantity, TotalSales, InvoiceNo and YearMonth which are key drivers of sales in this dataset. It also shows several weaker correlations between other variables which might not be as influential in predicting total sales but could still offer useful insight for feature engineering.

2.3.2 Trend and Pattern Exploration and Visualisation

2.3.2.1 Invoice Related Variables Exploration and Visualisation

```
#Visualization for Invoice

#Calculate the total sales for each of the invoice
invoice_sales = df.groupby('InvoiceNo')['TotalSales'].sum()

#Plot the bar chart and display the top 10 highest totalsales of invoice
plt.figure(figsize=(12, 6))
invoice_sales.nlargest(10).plot(kind='bar', color='salmon')
plt.title('Total Sales for Top 10 Invoice')
plt.xlabel('InvoiceNo')
plt.ylabel('Total Sales')
plt.xticks(rotation=90)
plt.show()
[19]  ✓ 0.1s
```

Figure 3.6: Code of Visualization for InvoiceNo

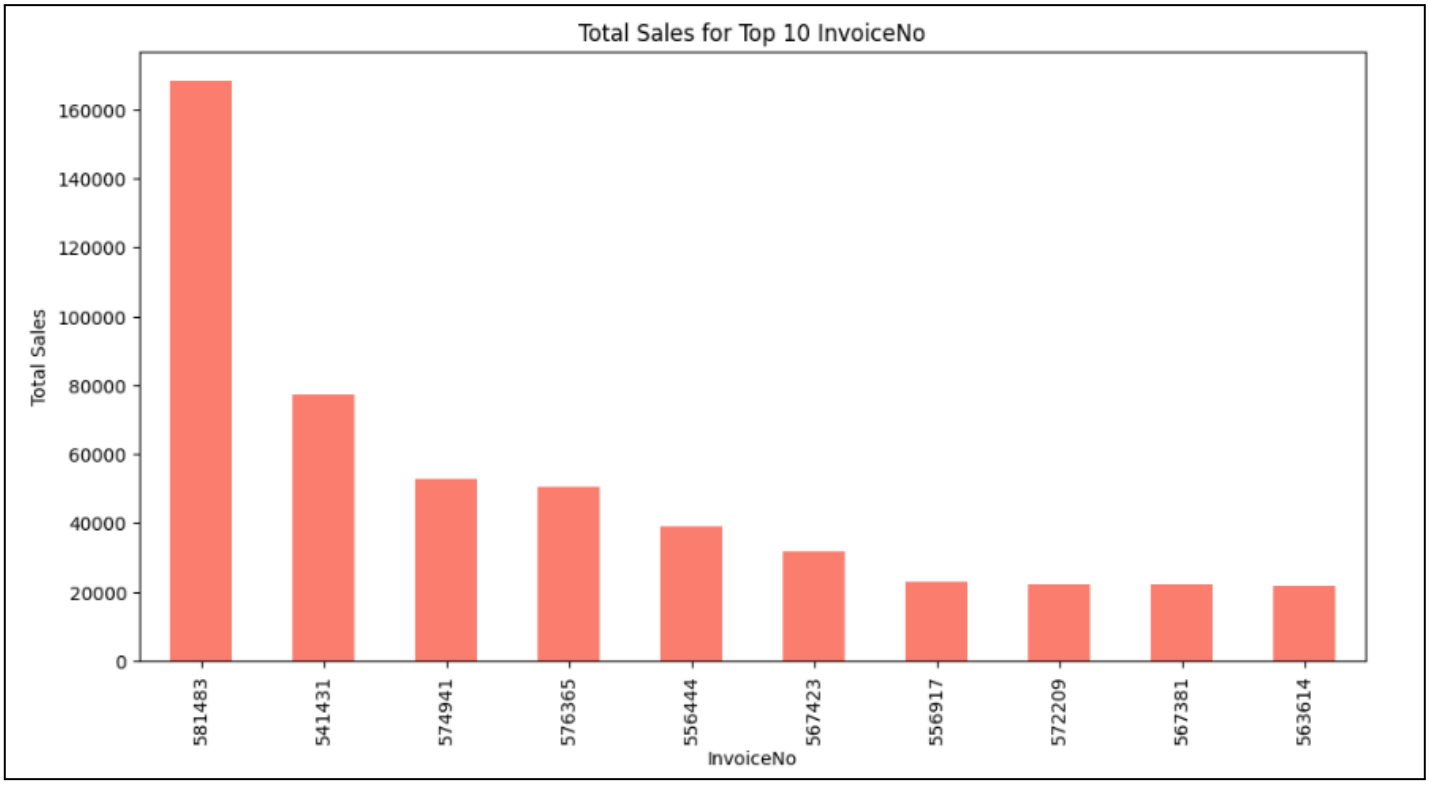


Figure 3.7: Bar Chart of Total Sales for Top 10 Invoices

Based on **Figure 3.7**, the bar chart illustrates the total sales for the top 10 invoice numbers, showing a significant disparity in sales distribution. The invoice number "581483" stands out as a clear outlier with sales nearing 170,000, far exceeding the others. Following this, invoice numbers "541431" and "574941" show sales around 70,000 and 60,000, respectively. A middle tier of invoices ("576365", "556444", "567423") ranges between 40,000 to 50,000 in

sales, while the remaining invoices (“556917”, “572209”, “567381”, “563614”) contribute less, with totals below 30,000. This distribution illustrates that the majority of total sales are concentrated among a few invoices, with a sharp decline observed after the top performers.

```
#Create a new column 'Canceled' to determine the canceled invoice and non-canceled invoice
df['Canceled'] = df['InvoiceNo'].str.startswith('C')

#Calculate the total number of canceled invoice and non-canceled invoice
canceled_invoice_count = df['Canceled'].sum()
non_canceled_invoice_count = (~df['Canceled']).sum()

#Plot a pie chart
labels = ['Canceled', 'Non-Canceled']
sizes = [canceled_invoice_count, non_canceled_invoice_count]
colors = ['lightskyblue', 'lightpink']
explode = (0.1, 0)
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')
plt.title('Pie Chart of Canceled and Non-Canceled Invoices')
plt.show()
[20]  ✓ 0.1s
```

Figure 3.8: Code of Visualization for Canceled and Non-Canceled Invoices

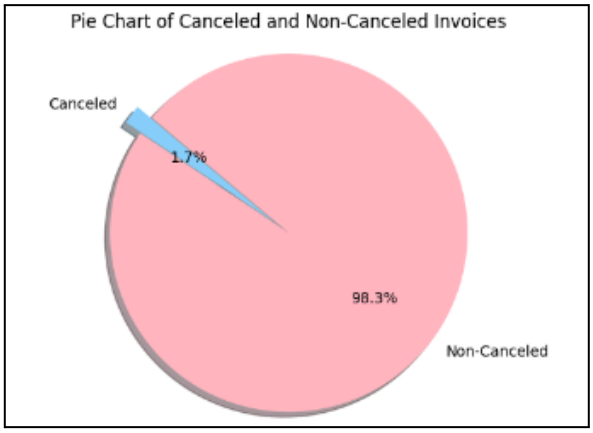


Figure 3.9:Pie Chart of the Percentage of Canceled vs. Non-Canceled Invoices

Based on **Figure 3.9**, which displays a pie chart illustrating the distribution of canceled and non-canceled invoices, we can observe that non-canceled invoices constitute 98.3% of the total, while canceled invoices represent the remaining 1.7%. This indicates a significantly higher proportion of non-canceled invoices compared to canceled ones.

```python
#Create a new column 'Manual' for the manual input invoice
df['Manual'] = df['StockCode'] == 'M'

#Calculate the total number of manual input invoice and non-manual input invoices
manual_invoice_Count = df['Manual'].sum()
non_manual_invoice_Count = (~df['Manual']).sum()

#Plot the pie chart
labels = ['Manual Input', 'Non-Manual Input']
sizes = [manual_invoice_Count, non_manual_invoice_Count]
colors = ['lightskyblue', 'lightpink']
explode = (0.1, 0)
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')
plt.title('Pie Chart of Manual Input and Non-Manual Input Invoices')
plt.show()
```

[21] ✓ 0.0s

Figure 3.10: Code of Visualization for Manual Input and Non-manual Input Invoices

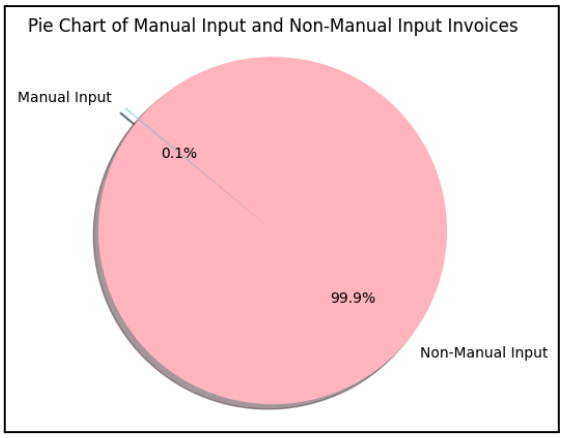


Figure 3.11:Pie Chart of the Percentage of Manual Input and Non-manual Input Invoices

Based on **Figure 3.11**, which displays a pie chart illustrating the distribution of manual input and non-manual input invoices, we can observe that non-manual input invoices constitute 99.9% of the total, while canceled invoices represent the remaining 0.1%. This indicates a significantly higher proportion of non-canceled invoices compared to canceled ones. However, we would consider dropping the manual input invoices from the data set as manual input invoices might contain human errors, leading to potential inconsistencies and noise in the dataset which could affect the accuracy of the analysis and predictions.

```python
#Create a new column 'Discount' for the discount given in the sales
df['Discount'] = df['StockCode'] == 'D'

#Calculate the total number of discount given and non-discount invoice
discount_invoice_Count = df['Discount'].sum()
non_discount_invoice_Count = (~df['Discount']).sum()

#Print the quantity of item for discount
print("Discount Given")
print("==================================")
print("Quantity of Item for Discount: ", discount_invoice_Count)

#Plot a pie chart
labels = ['Discount Item', 'Non-Discount Item']
sizes = [discount_invoice_Count, non_discount_invoice_Count]
colors = ['lightskyblue', 'lightpink']
plt.figure(figsize=(8, 6))
plt.barh(labels, sizes, color=colors)
plt.title('Comparison of Discount and Non-Discount Items')
plt.xlabel('Number of Items')
plt.show()
```

[22] ✓ 0.0s

Figure 3.12: Code of Visualization for Discount and Non-Discount Items

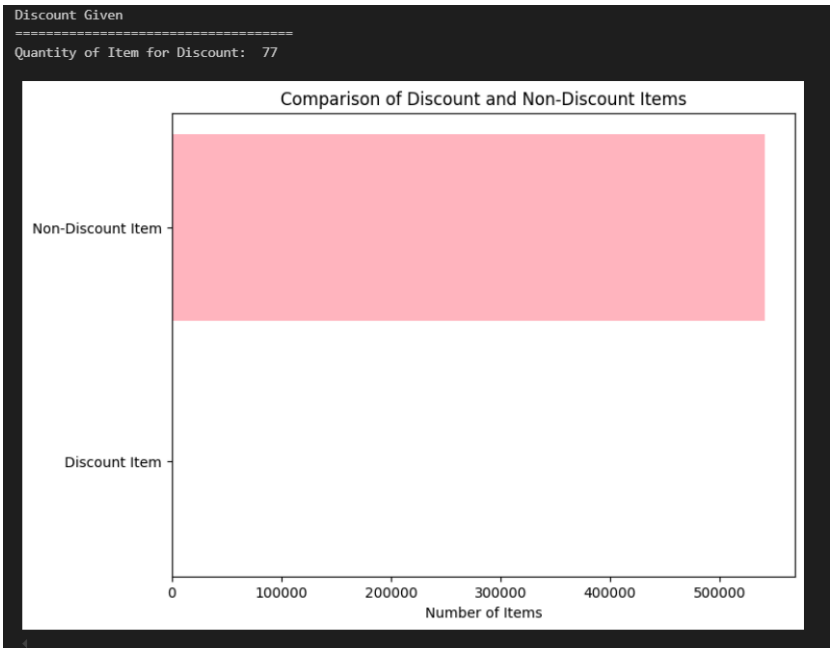


Figure 3.13: Comparison of Discount and Non-Discount Items

Based on **Figure 3.13**, which displays a bar chart illustrating the distribution of discount and non-discount invoices, we can observe that a total of 77 items were issued for discount purposes. This relatively small proportion of discount items indicates that the majority of the transactions did not involve discounts.

2.3.2.2 Stock Code Related Variable Exploration and Visualization

```
#Visualization for StockCode

#Calculate the total sales for each of the stock(Code)
stock_sales = df.groupby('StockCode')['TotalSales'].sum()

#Plot the bar chart and display the top 10 highest totalsales for the Stock (Code)
plt.figure(figsize=(12, 6))
stock_sales.nlargest(10).plot(kind='bar', color='salmon')
plt.title('Total Sales for Top 10 Stock (Code)')
plt.xlabel('StockCode')
plt.ylabel('Total Sales')
plt.xticks(rotation=90)
plt.show()
[23]  ✓ 0.1s
```

Figure 3.14: Code of Visualization for StockCode

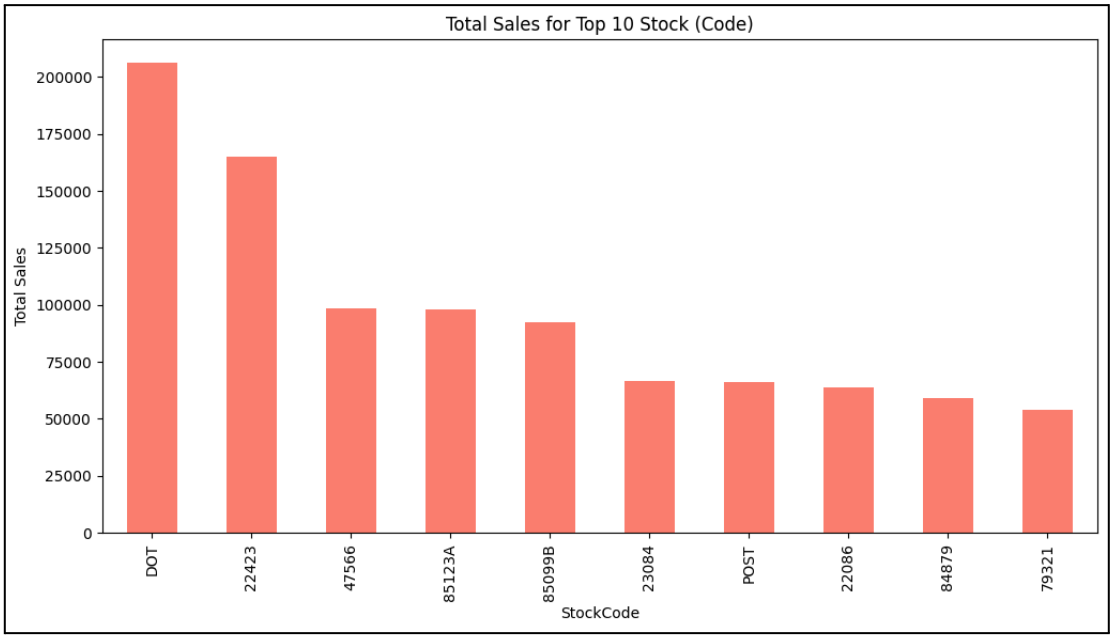


Figure 3.15: Bar Chart of Top 10 StockCode by TotalSales

Figure 3.15 presents a bar chart of the top 10 StockCodes by total sales. "DOT" leads with sales exceeding 200,000 units, indicating it is likely a high-demand or frequently purchased item. “22423” follows as the second highest, with sales around 170,000 units, reflecting its significant popularity. StockCodes like “47566”, “85123A”, and “85099B” each have sales close to the 100,000-unit mark, showing relatively strong performance, though not on par with the top two. The remaining StockCodes(“23084”, “POST”, “22086”, “84879” and “79321”) have lower total sales, ranging between 50,000 and 75,000 units, suggesting these items are popular but not as dominant as the leading StockCodes. There is a noticeable disparity in sales volumes among the top 10, with "DOT" and "22423" standing out as clear outliers with exceptionally high sales.

2.3.2.3 Item Sold Related Variable Exploration and Visualization

```
#Visualization for Description

#Calculate the total sales for each of the stock (name)
description_sales = df.groupby('Description')['TotalSales'].sum()

#Plot a bar chart and display the top 10 highest total sales for the Stock (name)
plt.figure(figsize=(12, 6))
description_sales.nlargest(10).plot(kind='bar', color='salmon')
plt.title('Total Sales for Top 10 Stock (Name)')
plt.xlabel('Stock')
plt.ylabel('Total Sales')
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
[24] ✓ 0.1s
```

Figure 3.16: Code of Visualization for Description

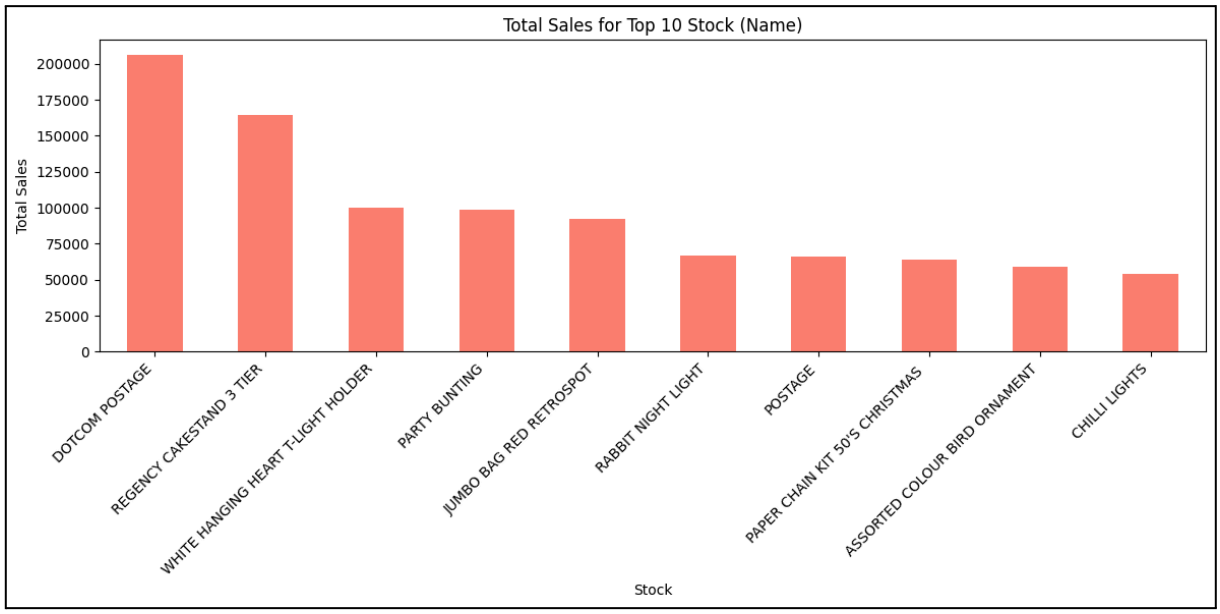


Figure 3.17: Bar Chart of Top 10 Description by TotalSales

Figure 3.17 presents a bar chart illustrating the total sales for the top 10 product descriptions. "DOTCOM POSTAGE" is the highest-selling item, with total sales exceeding 200,000 units, reflecting a trend similar to the first graph. "REGENCY CAKESTAND 3 TIER" follows closely with sales around 175,000 units, indicating its popularity among customers.Items such as "WHITE HANGING HEART T-LIGHT HOLDER", "PARTY BUNTING" and "JUMBO BAG RED RETROSPOT" have moderate sales figures, ranging between 100,000 and 125,000 units. The remaining descriptions, including "RABBIT NIGHT LIGHT", "POSTAGE", "PAPER CHAIN KIT 50'S CHRISTMAS", "ASSORTED COLOUR BIRD ORNAMENT" and "CHILLI LIGHTS" have total sales between 50,000 and 75,000 units.The data suggests that items with more specific or niche uses, such as the cake stand and retro bags, have achieved significant sales,

reflecting strong consumer preference. Additionally, the high sales of "POSTAGE" items suggest they may be frequently purchased accessories or complementary products.

2.3.2.4 Invoice Date Related Variables Exploration and Visualization

```
#Visualization for InvoiceDate

#Calculate the total sales for each of the month in 2010 and 2011
invoice_date_sales = df.groupby('YearMonth')['TotalSales'].sum()

#Plot the bar chart
plt.figure(figsize=(12, 6))
invoice_date_sales.plot(kind='bar', color='salmon')
plt.title('Total Sales by Year-Month')
plt.xlabel('Year-Month')
plt.ylabel('Total Sales')
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
[25]  ✓ 0.2s
```

Figure 3.18: Code of Visualization for Invoice Date

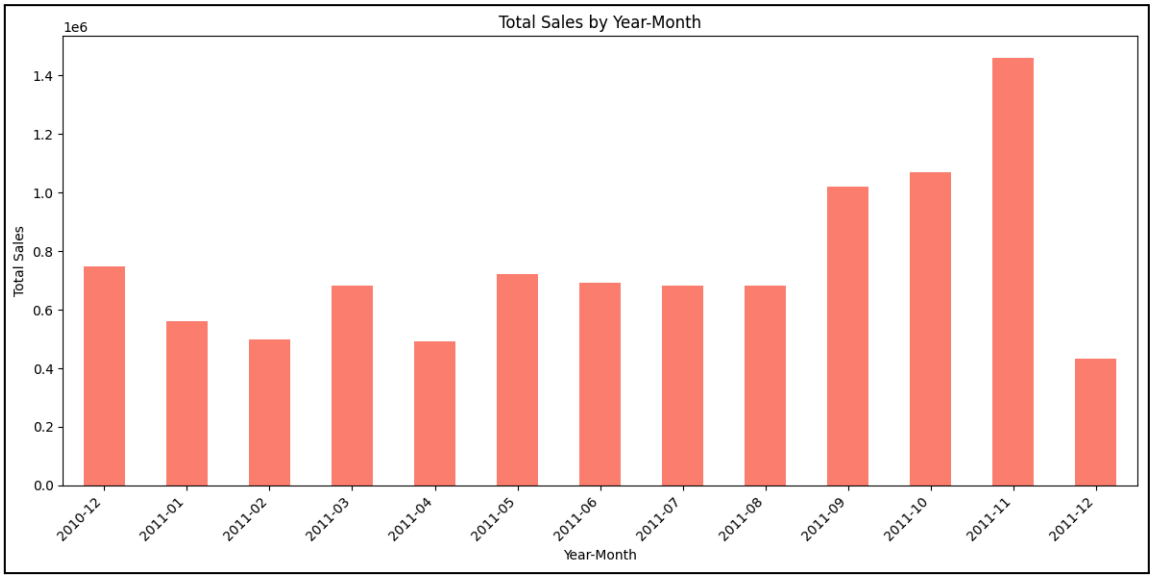


Figure 3.19: Bar Chart of Total Sales by Month

Figure 3.19 illustrates a bar chart showing the total sales broken down by month from December 2010 to December 2011. December 2010 and January 2011 display relatively low sales, at around 0.7 million and 0.6 million units, respectively. In February 2011, total sales decreased slightly to approximately 0.5 million units. March 2011 sees a rise, with sales nearing 0.7 million units, followed by a dip back to about 0.5 million units in April 2011.From May 2011 to August 2011, sales stabilize, with May reaching approximately 0.8 million units, and sales from June to August 2011 holding steady at around 0.7 million units. September 2011 marks the start of an upward trend, with sales increasing each month, peaking in November 2011 at around 1.5 million units. However, December 2011 saw a sharp decline, with total sales dropping to

approximately 0.9 million units. The significant increase in sales during the last quarter of 2011, especially in November, likely reflects holiday shopping trends.

2.3.2.5 Customer Related Variables Exploration and Visualization

```
#Visualization for CustomerID

#Calculate the total sales for each of the customer
customer_sales = df.groupby('CustomerID')['TotalSales'].sum()

#Plot a bar chart and display the top 10 total sales by the customer
plt.figure(figsize=(12, 6))
customer_sales.nlargest(10).plot(kind='bar', color='salmon')
plt.title('Total Sales by Top 10 Customer')
plt.xlabel('CustomerID')
plt.ylabel('Total Sales')
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
[26]  ✓ 0.1s
```

Figure 3.20: Code of Visualization for CustomerID

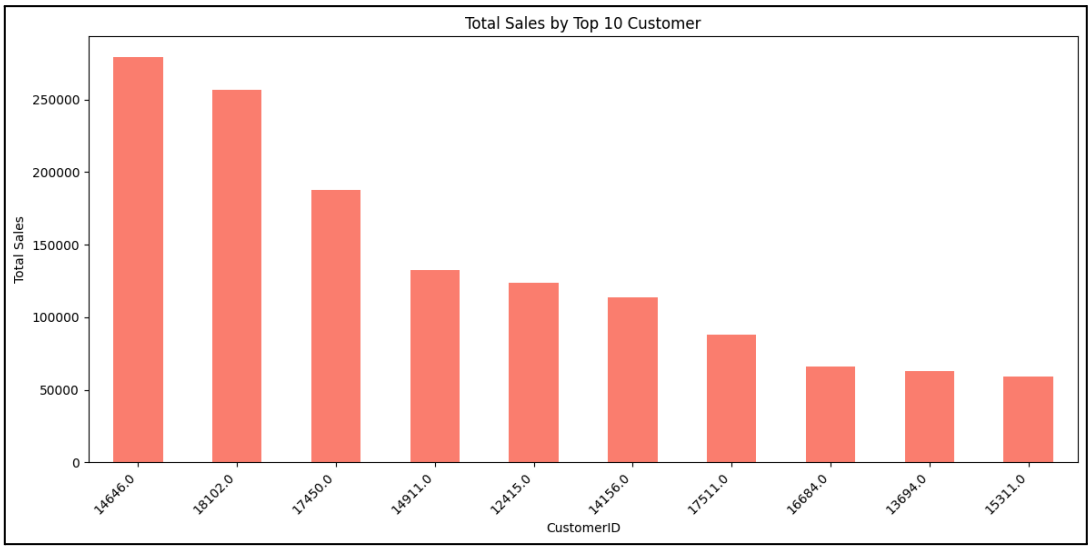


Figure 3.21: Bar Chart of Total Sales by Top 10 Customers

Figure 3.21 presents a bar chart illustrating the Total Sales contributed by the Top 10 Customers. The highest contributor is Customer ID “14646”, generating approximately 280,000 in total sales. Following closely is Customer ID “18102”, with total sales of around 250,000, and in third place is Customer ID “17450”, contributing roughly 200,000. A middle tier of customers, including Customer IDs “14911”, “12415”, “14156”, and “17511”, shows total sales ranging between 90,000 and 150,000. In contrast, the remaining customers, Customer IDs "16684", "13694", and "15311", each contribute less than 75,000 in sales. This breakdown highlights the differences in sales contributions among the top 10 customers, emphasizing the dominance of the top three and clearly distinguishing the middle and lower tiers based on their sales performance.

2.3.2.6 Country Related Variables Exploration and Visualization

```
#Visualization for country
country_sales = df.groupby('Country')['TotalSales'].sum()

#Plot a graph and display the top 10 total sales by country
plt.figure(figsize=(12, 6))
country_sales.nlargest(10).plot(kind='bar', color='salmon')
plt.title('Total Sales by Top 10 Country')
plt.xlabel('Country')
plt.ylabel('Total Sales')
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
```

[27] ✓ 0.1s

Figure 3.22: Code of Visualization for Country

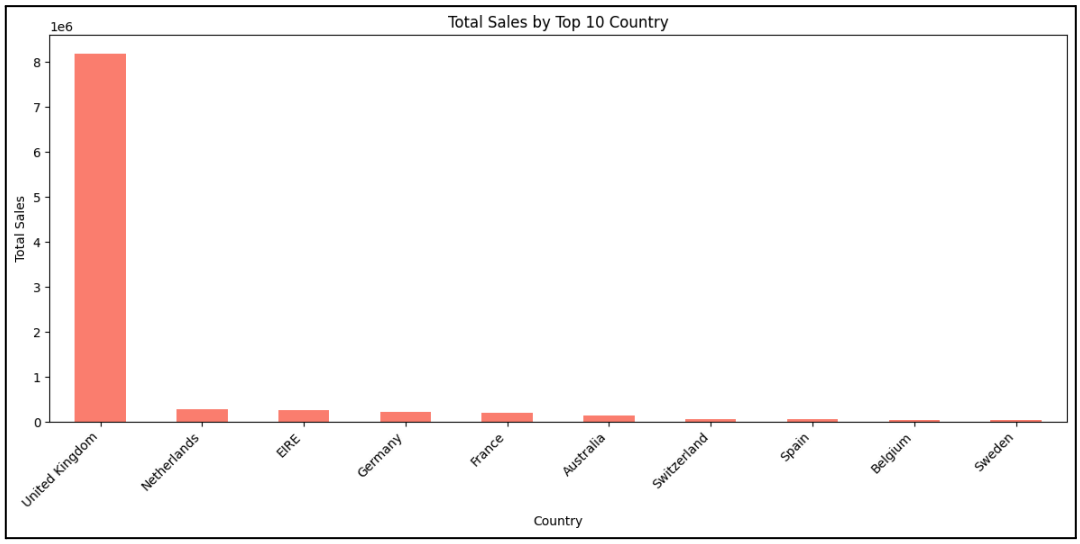


Figure 3.23: Bar Chart of Total Sales by Country

Figure 3.23 presents a bar chart showing total sales by country, with a noticeable difference between the United Kingdom and other countries. The United Kingdom stands out by contributing the majority of total sales, accounting for approximately 75% (800,000) of the overall figure. In contrast, the combined sales from other countries make up just 25% (200,000). This disparity underscores the significant contribution of the United Kingdom to the total sales in stores.

2.3.3 Distribution Exploration and Visualisation

2.3.3.1 Distribution of InvoiceDate

2.3.3.1.1 Distribution of Invoice by Year

```
#Plot a graph for distribution of invoices by Year
plt.figure(figsize=(10, 6))
sns.histplot(df['Year'], kde=True, bins=5, color='skyblue')
plt.title('Distribution of Invoices by Year')
plt.xlabel('Year')
plt.ylabel('Frequency')
plt.show()
```

[28] ✓ 3.8s

Figure 3.24: Code of Distribution of Invoices by Year

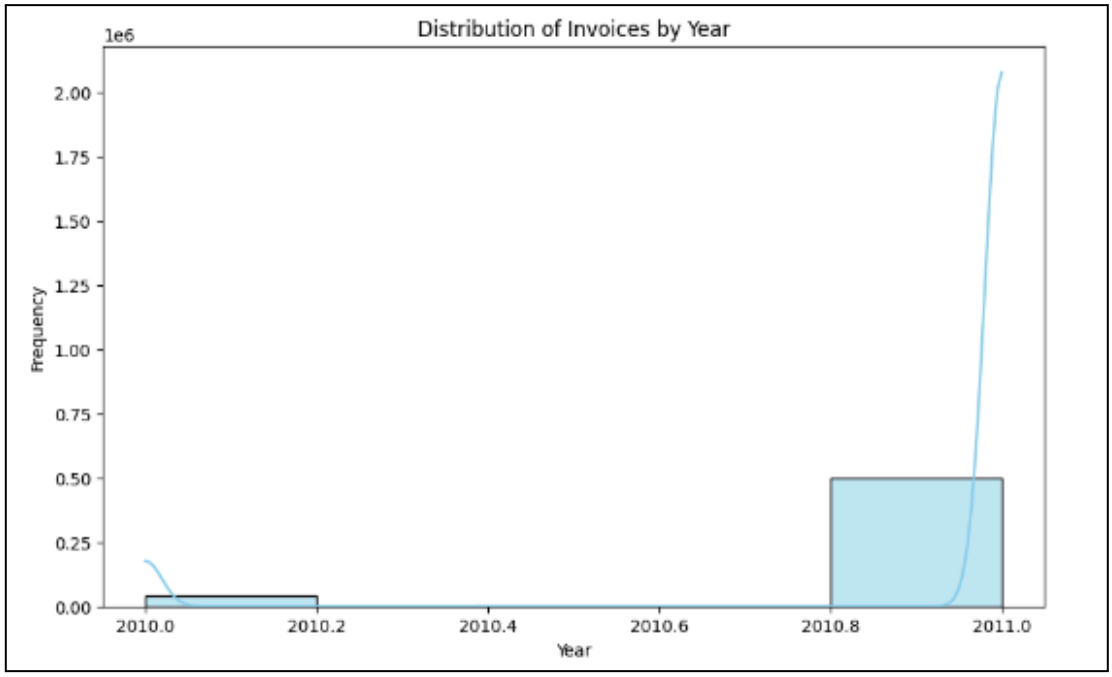


Figure 3.25: Distribution of Invoices by Year

The above **Figure 3.25** shows the distribution of invoice by year that is represented by the histogram. The histogram displays the frequency of invoices across different years, with each bar representing a range of years. The graph shows minimal activity during the early months of 2010. Starting from around mid-2010, there's a noticeable upward trend in the number of invoices. The biggest spike happened in the last quarter of 2010, especially towards the end of the year, showing a big jump in transactions during this time. The curve rose quickly at the end of 2010, possibly due to fast business growth or high seasonal demand. The graph also shows this peak extending into 2011, hinting that the trend might continue beyond the data shown.

2.3.3.1.2 Distribution of Invoices by Month

```
#Plot a graph for distribution of invoices by Month
plt.figure(figsize=(10, 6))
sns.histplot(df['Month'], kde=True, bins=5, color='skyblue')
plt.title('Distribution of Invoices by Month')
plt.xlabel('Month')
plt.ylabel('Frequency')
plt.show()
```

[29] ✓ 4.3s

Figure 3.26: Code of Distribution of Invoices by Month

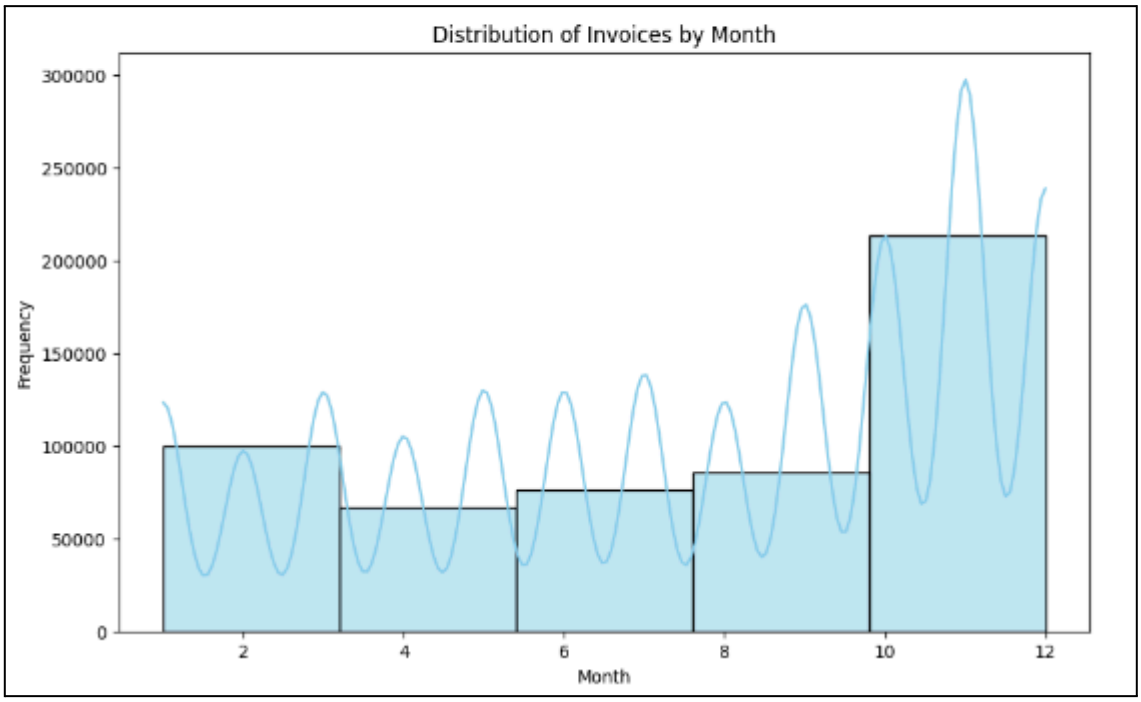


Figure 3.27: Histogram of Distribution of Invoices by Month

The above **Figure 3.27** shows the distribution of invoices over different months in a year. The histogram displays the frequency of invoices across different months, with each bar representing a range of months. The KDE line superimposed on the histogram provides a smooth estimate of the distribution. There are apparent peaks around the months of February (Month 2), May (Month 5), August (Month 8), and December (Month 12), indicating higher frequency of invoices in these months. December (Month 12) appears to have the highest frequency of invoices issued, as indicated by the tallest bar and the highest point on the KDE line. This histogram can help to visualize the seasonal trends or patterns in the invoicing data, which can show the month which have the highest or lowest frequencies of invoices.

2.3.3.1.3 Distribution of Invoices by Day

```
#Plot a graph for distribution of invoices by Day
plt.figure(figsize=(10, 6))
sns.histplot(df['Day'], kde=True, bins=5, color='skyblue')
plt.title('Distribution of Invoices by Day')
plt.xlabel('Day')
plt.ylabel('Frequency')
plt.show()
```

[30] ✓ 4.4s

Figure 3.28: Code of Distribution of Invoices by Day

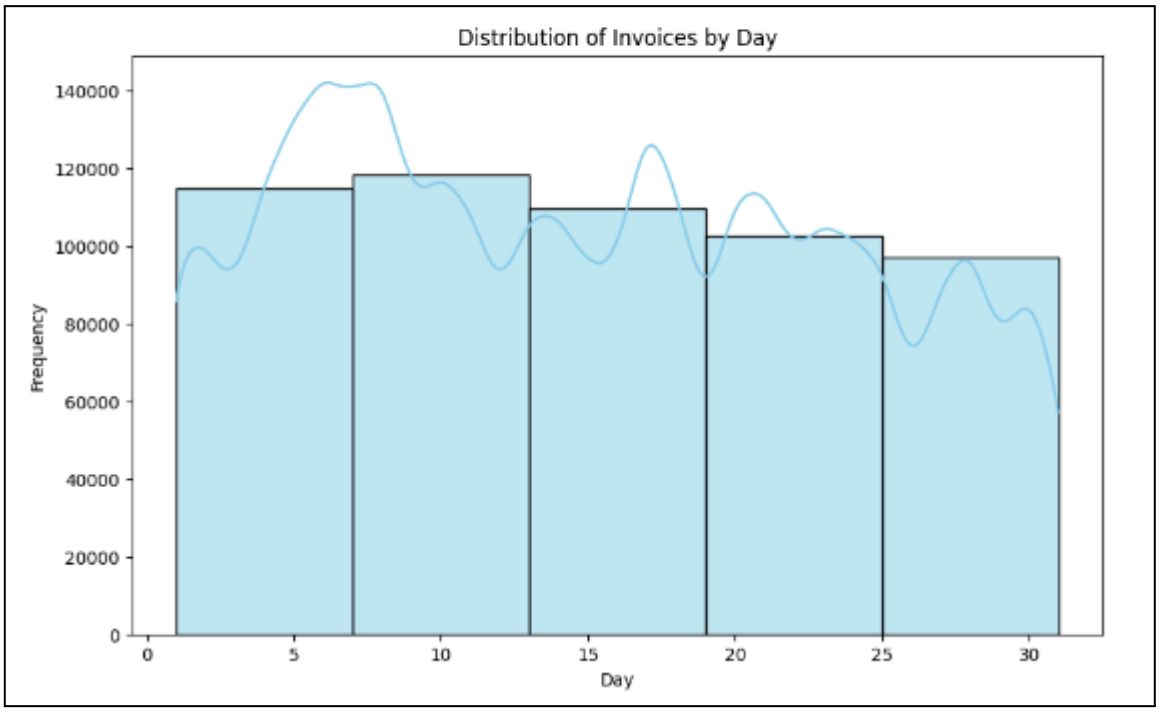


Figure 3.29: Histogram of Distribution of Invoices by Day

The above **Figure 3.29** shows the histogram of distribution of invoices by day. The graph shows the distribution of invoices over the course of the month (1 to 31 days). The bars represent how frequently invoices are issued on different days, and the line on top smooths out this distribution. The frequency of invoices appears to fluctuate between days, with the highest frequency occurring around the 5th day. While there are periodic drops, the overall pattern is fairly consistent throughout the month, with a peak in the first 10 days. After the 10th day, there's a slight decline, with some spikes around the 20th and 25th days. This graph shows invoices are not evenly distributed across the days of the month, with significant activity in the early and middle parts of the month.

2.3.3.1.4 Distribution of Invoices by Day of Week

```
#Plot a graph for distribution of invoices by day of week
plt.figure(figsize=(10, 6))
sns.histplot(df['DayOfWeek'], kde=True, bins=5, color='skyblue')
plt.title('Distribution of Invoices by DayOfWeek')
plt.xlabel('DayOfWeek')
plt.ylabel('Frequency')
plt.show()
```

[31] ✓ 4.8s

Figure 3.30: Code of Distribution of Invoices by DayOfWeek

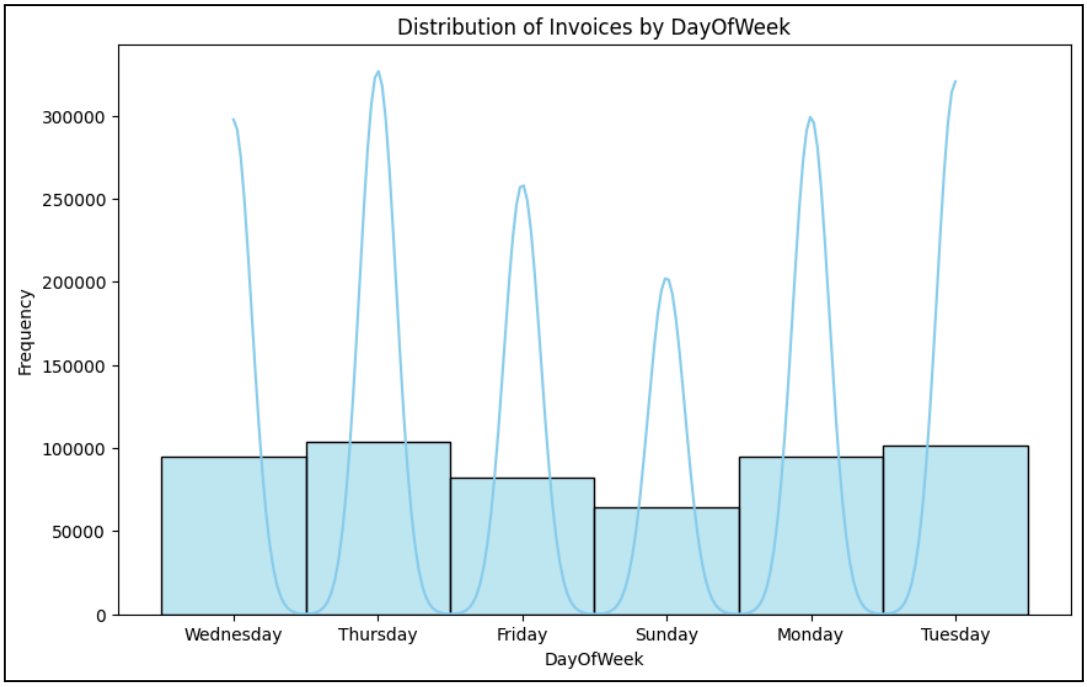


Figure 3.31: Histogram of Distribution of Invoices by DayOfWeek

The above **Figure 3.31** shows the histogram of distribution of invoices by day of week. The graph shows the distribution of invoices by day of the week (Wednesday to Tuesday). The bar represents the frequency distribution of invoices on each day and the KDE line shows the probability density of invoices being processed throughout the week. The sharp peaks indicate higher probabilities around Monday, Tuesday, and Thursday. The frequency of invoices drops significantly on Sunday, making it the day with the lowest activity. The highest invoice frequency occurs on Thursday, Monday, and Tuesday, with peaks around these days. The mid-week period (Wednesday to Friday) clearly sees the largest level of invoicing activity, with Thursday marking the top. There is a regular weekly pattern in which the busiest days are early and midweek, with a sharp drop over the weekend, particularly on Sunday.

2.3.3.2 Distribution of CustomerID

```
#Plot a graph for distribution of invoices by customer
plt.figure(figsize=(10, 8))
sns.countplot(x='CustomerID', data=df, order=df['CustomerID'].value_counts().index, palette='coolwarm')
plt.title('Distribution of Customer')
plt.xlabel('Customer')
plt.ylabel('Frequency')
plt.xticks(rotation=90)
plt.show()
[32]  ✓ 30.6s
```

Figure 3.32: Code of Distribution of CustomerID

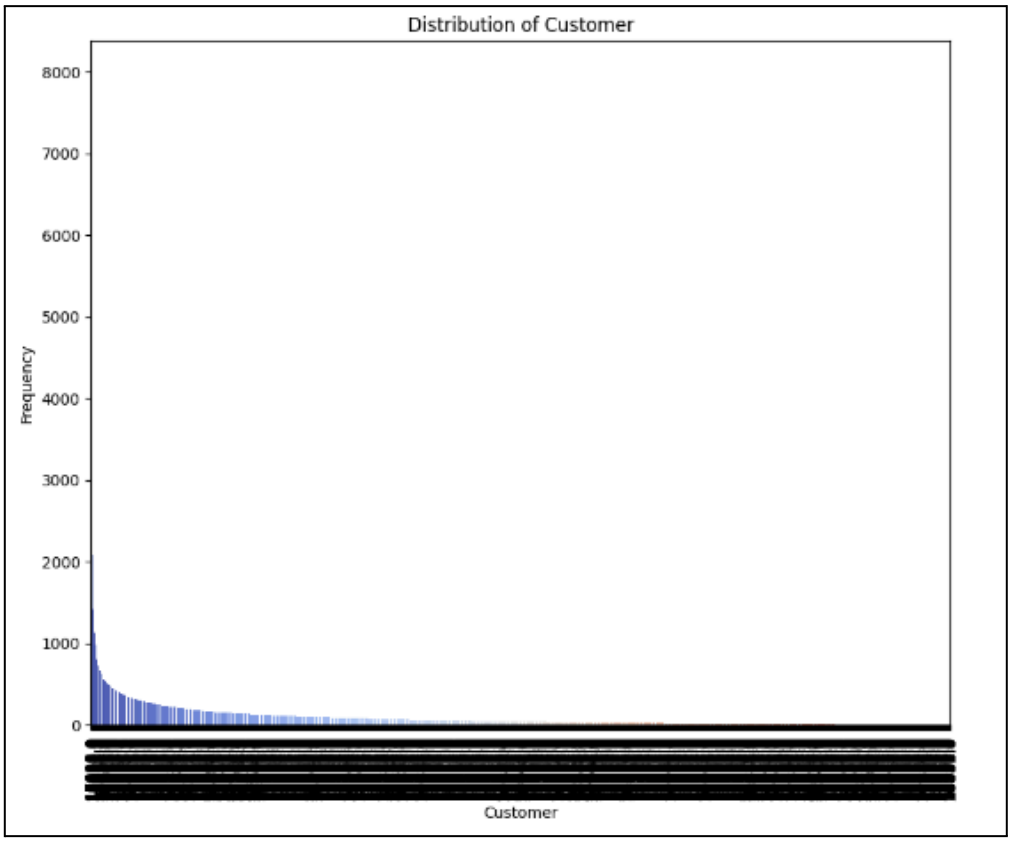


Figure 3.33: Distribution of CustomerID

Figure 3.33 above shows the frequency distribution of invoices associated with different customer IDs. The graph shows a highly skewed distribution, where a small number of customers have a very high frequency of invoices (up to 8000), while the majority of customers have very low frequencies. Most of the customer IDs have relatively few invoices, which are bunched up on the right side of the graph, creating a long tail. The left side of the graph shows a few customer IDs with a significantly higher frequency of invoices, as indicated by the tall bars. The distribution indicates that there are a few customers who generate a large number of invoices, while most customers generate only a few. This could be useful for identifying key customers who are responsible for a significant portion of the business's transactions.

2.3.3.3 Distribution of Country

```
#Plot a graph for distribution of invoices by Country
plt.figure(figsize=(10, 8))
sns.countplot(x='Country', data=df, order=df['Country'].value_counts().index, palette='coolwarm')
plt.title('Distribution of Country')
plt.xlabel('Country')
plt.ylabel('Frequency')
plt.xticks(rotation=90)
plt.show()
```

[33] ✓ 0.8s

Figure 3.34: Code of Distribution by Country

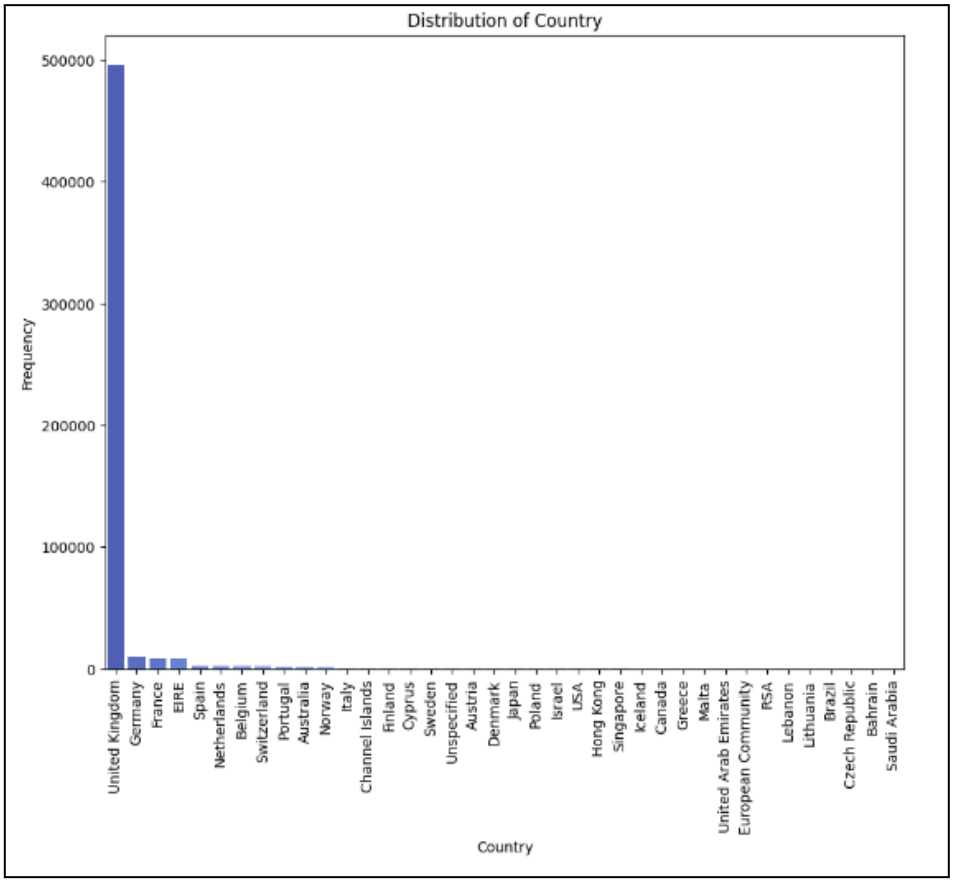


Figure 3.35: Distribution by Country

Figure 3.35 displays the distribution of invoices by country, as well as descriptive statistics for the 'Country' column in the dataset. The graph is highly skewed, with the vast majority of invoices coming from one country, the United Kingdom (UK), as indicated by the very tall bar on the left. The UK has the highest frequency of invoices, with more than 500,000 as evidenced by the tallest bar on the graph. Other countries have significantly fewer invoices, with the bars for those countries being much shorter and clustered on the right side of the graph. The distribution is highly uneven, indicating that the business is principally centered in the UK, with relatively minor activity in other countries.

2.3.3.4 Distribution of Total Sales

```
#Plot a box plot for Total Sales
plt.figure(figsize=(8,8))
plt.boxplot(df['TotalSales'])
plt.title('Boxplot of TotalSales')
plt.ylabel('TotalSales')
plt.show()
```

[34] ✓ 0.1s

Figure 3.36: Code of plotting visualization for TotalSales

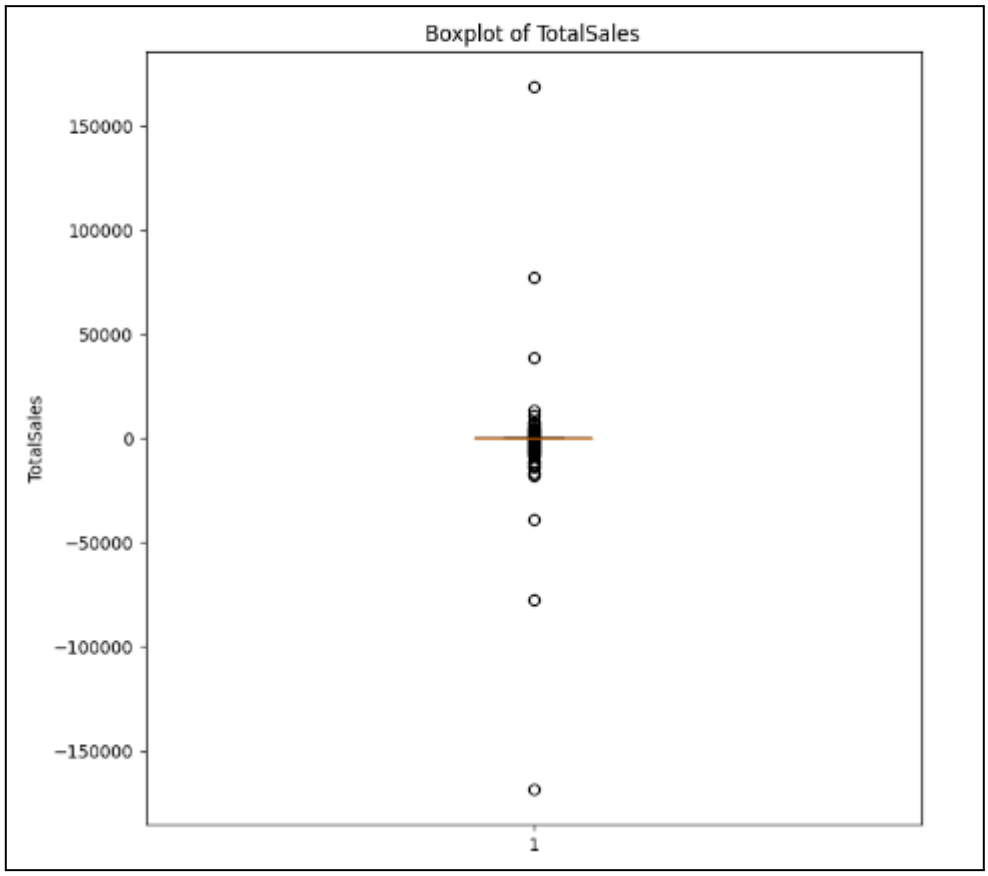


Figure 3.37: Boxplot of TotalSales

```
df['TotalSales'].describe()
[35]  ✓ 0.0s
...
count    541909.000000
mean         17.987795
std         378.810824
min     -168469.600000
25%           3.400000
50%           9.750000
75%          17.400000
max      168469.600000
Name: TotalSales, dtype: float64
```

Figure 3.38: Descriptive Statistics of TotalSales

From **Figure 3.38**, we can see the total number of observations is 541909 and the average value of TotalSales is approximately 17.99. The standard deviation is quite large at 378.81, indicating high variability in the data. The maximum value is 168469.60 and the minimum value is -168469.60 showing a wide range of values. The 25th percentile (Q1) of the TotalSales data is 3.4, meaning that 25% of the sales values are less than or equal to this amount. The median (50th percentile) is 9.75, indicating that half of the sales values fall below this number. Finally, the 75th percentile (Q3) is 17.4, showing that 75% of the sales values are less than or equal to 17.4, while the remaining 25% are higher. The data shows significant variability, with extreme outliers. Despite a median value of 9.75, the range of values extends to both large positive and large negative values. This suggests that while most of the data is clustered around lower sales figures, there are occasional extremely high and low sales that are distorting the overall distribution.

2.4 Data Quality Verification

Data quality verification is a crucial phase in data management that focuses on evaluating and ensuring the accuracy, completeness, reliability, and consistency of data by scrutinizing it to confirm it meets predefined standards and is suitable for analysis, reporting, and decision-making.

During this phase, we focus on identifying and addressing abnormal data. Abnormal data is data that deviates significantly from expected patterns, norms, or standards. Abnormal data, such as errors or outliers, can skew the results of analyses, leading to incorrect conclusions or decisions. This, in turn, affects the reliability of reports and predictions derived from the data.

By addressing abnormal data, we can improve the overall quality of the dataset. High-quality data ensures that decisions are based on accurate information, preventing flawed conclusions and suboptimal decisions. This is critical for strategic planning and operational effectiveness, as precise and accurate analysis leads to better insights and forecasts.

Accurate data is essential for valid and meaningful analysis. Verifying data quality helps ensure that the insights and conclusions drawn from the data are correct. Reducing errors, duplicates, and anomalies not only improves the efficiency of data processing and analysis but also saves time and resources.

In essence, thorough data quality verification is vital for ensuring that the insights derived from data are valid and actionable. This process helps organizations make well-informed decisions, avoid costly mistakes, and achieve better outcomes through improved strategic and operational practices.

2.4.1 Null Data Detection

Null data refers to the absence of a value or the lack of data in a dataset. In various fields such as databases, programming, and data analysis, "null" is used to signify that a particular data field does not contain any value. This absence can occur for several reasons: the data might be missing or not collected, the field may be irrelevant for a specific entry, or there could be errors or incomplete records. Unlike zeros, empty strings, or other default values, null specifically indicates an "unknown" or "not applicable" state. Properly handling null data is crucial for accurate analysis and database management, as it affects how queries are processed, calculations are performed, and data integrity is maintained.

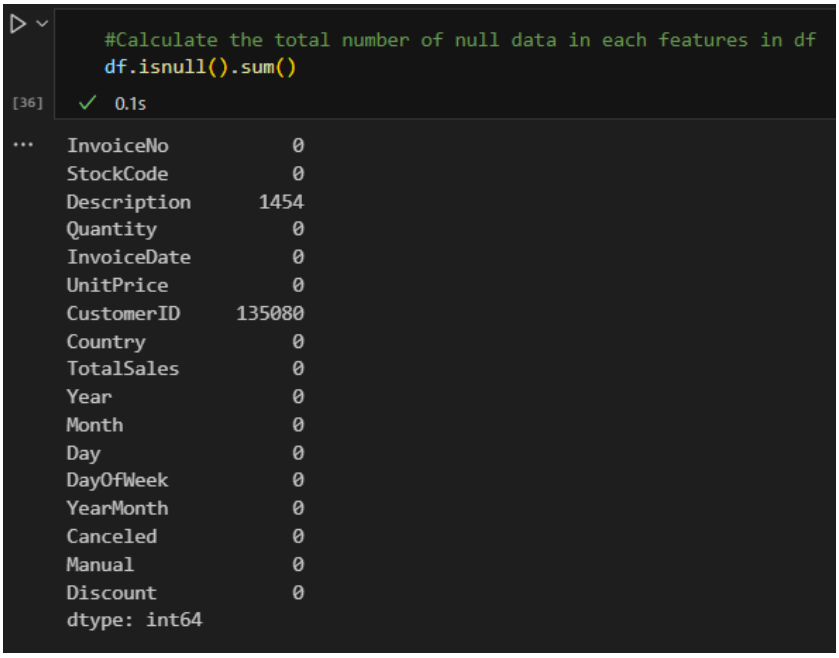


Figure 4.0: Sum of the null value for each variables

According to **Figure 4.0**, we observe that the dataset is fully complete, with no null values present. This observation is significant as it indicates that every entry in the dataset has been recorded in its entirety, with no missing data points. The absence of null values suggests that the dataset is robust and reliable for further analysis. This completeness reflects a high level of data integrity and quality, implying that the data collection and input processes were thorough and effective. As a result, the overall credibility of the dataset is strengthened. We can proceed with confidence, knowing that our analysis is based on a complete set of information. This completeness enhances the reliability of statistical measures, models, and forecasts derived from the dataset. In

summary, the absence of null values in the dataset, as illustrated in **Figure 4.0**, confirms that all data entries are intact and complete. This high level of completeness is a positive indicator of data quality and provides a solid foundation for accurate analysis and informed decision-making.

```
#Calculate the total cells in df
total_cells = np.prod(df.shape)
#Calculate the total number (in cells) of missing values
total_missing_values = df.isnull().sum().sum()

#Calculate the percentage of missing value and non-missing values
percentage_missing_values = (total_missing_values / total_cells) * 100
percentage_non_missing_values = 100 - percentage_missing_values
```
[37] ✓ 0.1s

Figure 4.1: Code of finding percentage of missing values and non-missing values in the DataFrame

```
#Print the percentage of missing values in the data set
print("Percentage of Missing Values: {:.2f}%.".format(percentage_missing_values))
```
[38] ✓ 0.0s

```
Percentage of Missing Values: 1.48%.
```

Figure 4.2: Code of printing the percentage of missing values

```
#Plot a pie chart to visualize the distribution of missing value and non-missing value
labels = ['Missing Values', 'Non-Missing Values']
sizes = [total_missing_values, total_cells - total_missing_values]
colors = ['lightskyblue', 'lightpink']
explode = (0.1, 0)
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')
plt.title('Pie Chart of Missing Values')
plt.show()
```
[39] ✓ 0.0s

Figure 4.3: Code of visualizing the percentage of missing values and non-missing values

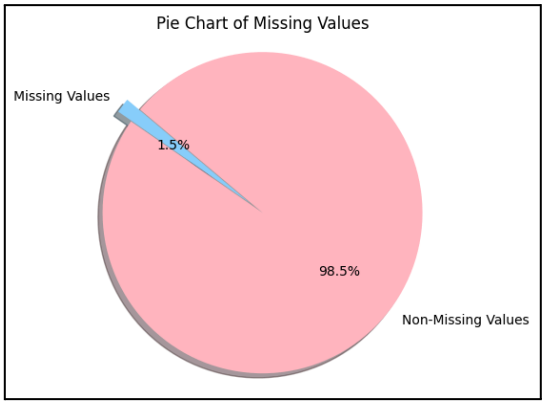
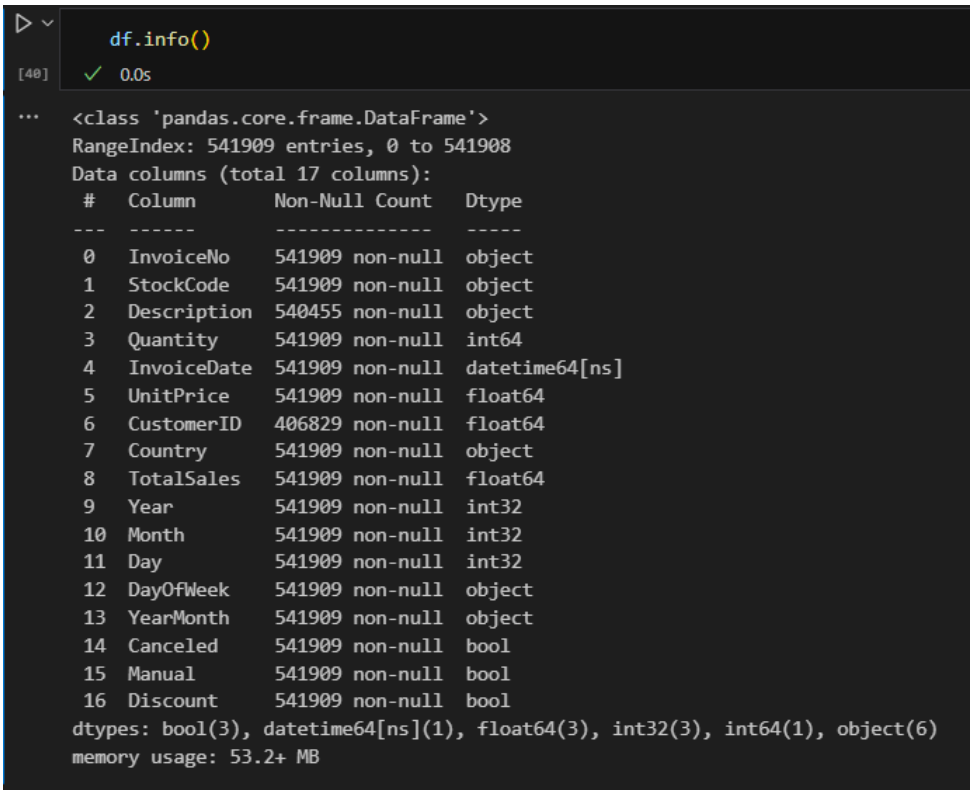


Figure 4.4: Pie Chart of Missing Values and Non-Missing Values

From **Figure 4.4**, we observe that there are 1.5% of missing values in our dataset. Since the amount of the missing values in CustomerID is too large and it is a categorical

variable that it is difficult to use statistical methods to impute it, we opt to remove those missing values in order to gain precise information.



```
df.info()
```

```
[40] ✓ 0.0s
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 17 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  datetime64[ns]
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      541909 non-null  object
 8   TotalSales   541909 non-null  float64
 9   Year         541909 non-null  int32
 10  Month        541909 non-null  int32
 11  Day          541909 non-null  int32
 12  DayOfWeek    541909 non-null  object
 13  YearMonth    541909 non-null  object
 14  Canceled     541909 non-null  bool
 15  Manual       541909 non-null  bool
 16  Discount     541909 non-null  bool
dtypes: bool(3), datetime64[ns](1), float64(3), int32(3), int64(1), object(6)
memory usage: 53.2+ MB
```

Figure 4.5: Information of each of the variables and it shapes.

From **Figure 4.5**, we can observe that there is a missing value in the features 'Description' (contains 540455 data) and 'CustomerID' (contains 406829 data). However, the complete data for the 'Description' and 'CustomerID' should be 541909.

2.4.2 Duplicate Data Detection

```
#Calculate the total number of duplicate rows in data set
duplicate_rows = df[df.duplicated()]
print('Number of duplicate rows: ', duplicate_rows.shape)
```

[41] ✓ 0.4s

```
Number of duplicate rows:  (5268, 17)
```

Figure 4.6: Code of finding the number of duplicate rows in the DataFrame

Duplicate data refers to repeated rows in a dataset that contain identical values across all columns. Detecting and handling duplicate data is crucial because duplicates can skew analysis results and lead to incorrect conclusions. It can negatively impact the performance of machine learning models.

From **Figure 4.6**, we observe that there are 5268 rows of duplicate data in our dataset. In this case, we choose to remove the duplicate row in order to obtain more reliable analyses and better-performing models.

2.4.3 Outliers Detection

```
#Plot the box plot for the features in df
plt.figure(figsize=(10,10))
df.boxplot(patch_artist=True, sym='k.')
plt.show()
```

[42] ✓ 2.9s

Figure 4.7: Code of visualizing the outlier by using boxplot

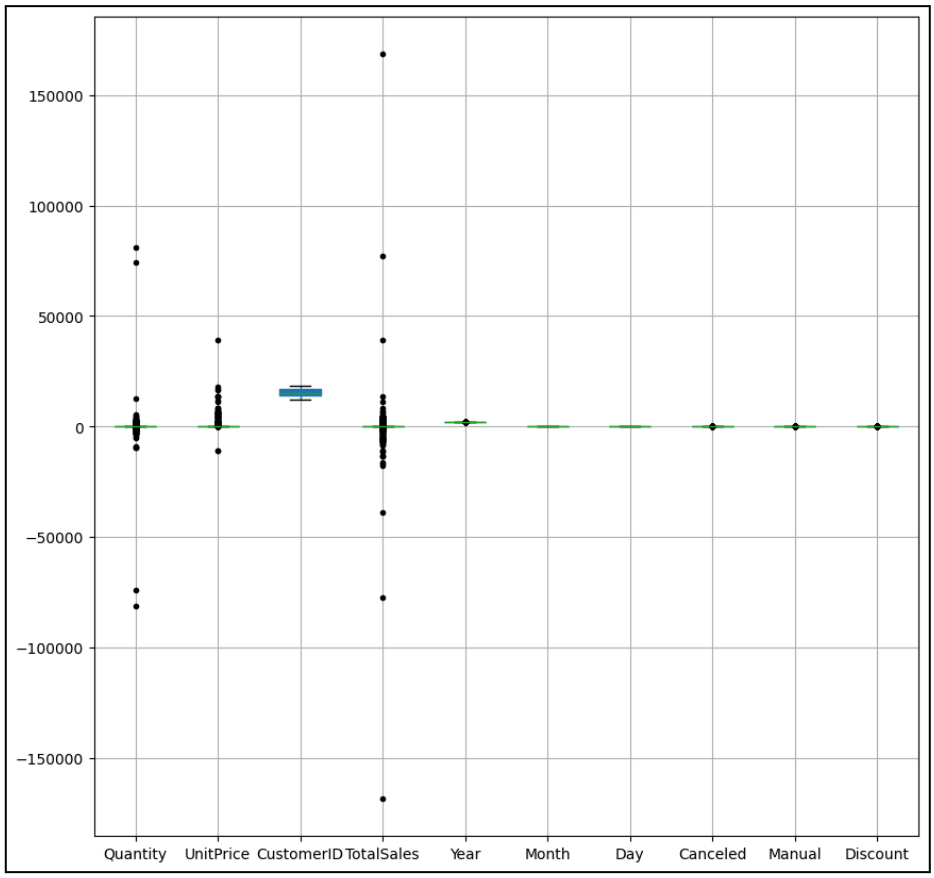


Figure 4.8: Boxplot of the continuous variables in the DataFrames

From **Figure 4.8**, we observe that there is some negative value in 'Quantity', 'UnitPrice' and 'TotalSales'. These negative values are unusual and could indicate returns, refunds or data entry errors. For the features 'Quantity', 'UnitPrice' and 'TotalSales', the distribution is skewed with negative values deviating significantly from the bulk of the data.

To maintain data integrity and ensure accurate analysis, it is important to handle these negative values. In this case, we opt to filter them out in order to get precise information for the total sales of online retail stores in the United Kingdom.

2.4.4 Zero Values Detection

```
#Calculate the zero values in'Quantity' and 'UnitPrice'
zero_Quantity = (df['Quantity'] == 0).sum()
zero_UnitPrice = (df['UnitPrice'] == 0).sum()

#Print the information found for the zero values detection
print("Total Number of Zero Values: ")
print("==============================")
print("Quantity: ", zero_Quantity)
print("Unit Price: ", zero_UnitPrice)
```

[43] ✓ 0.0s

```
Total Number of Zero Values:
==============================
Quantity:  0
Unit Price:  2515
```

Figure 4.9: Code of finding the number of zero values in Quantity, UnitPrice and TotalSales

In this stage, we detect the total number of zero values for several features which are 'Quantity' and 'UnitPrice'. Detecting the zero values is crucial as they can indicate missing data or errors in data entry which might lead to inaccurate model predictions.

From **Figure 4.9**, we observe that there is no zero value in 'Quantity'. However, there are 2515 zero values in 'UnitPrice'. These could mean that there were instances where the price per unit was not recorded or was recorded incorrectly.

The presence of zero values in 'UnitPrice'is concerning as these zeros can lead to skewed results and inaccurate predictions. To ensure the integrity of the dataset and improve the accuracy of our models, it is necessary to address these zero values appropriately.

3.0 Data Preparation

3.1 Data Selection

In this stage, the data selection process is a vital step in a data science project. It determines the quality and relevance of the data used for analysis and model development. This section outlines the data selection process undertaken for the analysis of online retail in the United Kingdom with detailing the criteria used for selecting the dataset, the sources of data and other preprocessing steps that we made to prepare the data for analysis.

We use the previous encoded data set to calculate the mutual information before proceeding to feature selection and feature engineering.

```
from sklearn.feature_selection import mutual_info_regression

#Determine the x-value and y-value
x = corr_df.drop(['InvoiceDate', 'TotalSales'], axis=1)
y = corr_df['TotalSales']

#Calculate the mutual information regression for x and y
mi = mutual_info_regression(x,y)
#Create a new data frame to store the mutual information for each of the features in df
mi_scores = pd.DataFrame(mi, index=x.columns, columns=['Mutual Information Score'])

#Sort the mutual information in descending order
mi_scores = mi_scores.sort_values(by='Mutual Information Score', ascending=False)

#Plot a graph to visualize the mutual information scores for features related to total sales
plt.figure(figsize=(10,6))
sns.barplot(x=mi_scores.index, y=mi_scores['Mutual Information Score'])
plt.title('Mutual Information Scores for Features Related to Total Sales')
plt.xlabel('Features')
plt.ylabel('Mutual Information Score')
plt.xticks(rotation=90)
plt.show()
[44]  ✓ 46.4s
```

Figure 5.0: Code of finding correlation relationship between the independent variables and dependent variables by using mutual information

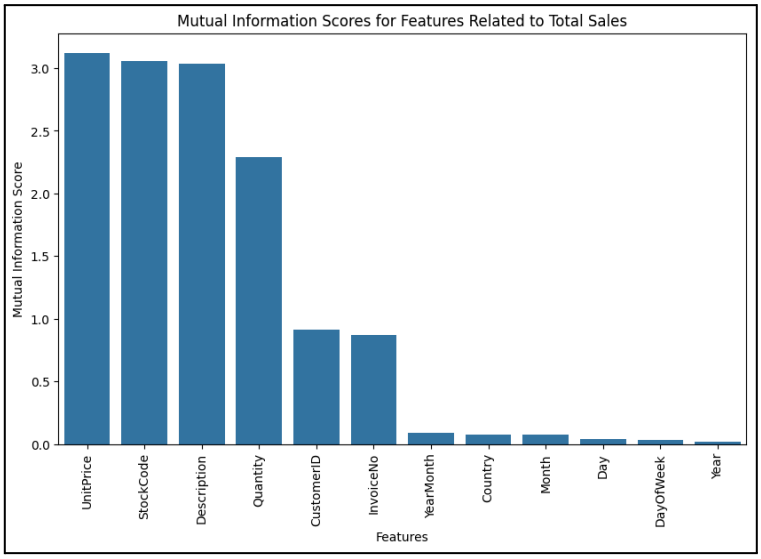


Figure 5.1:Graph of Mutual Information Scores for Features Related to TotalSales

Mutual Information (MI) is a measure from information theory that quantifies the amount of information obtained about one variable through another variable. It captures both linear and non-linear relationships between variables and makes it a versatile tool for understanding the dependency between features in a dataset.

From **Figure 5.1**, we observe that the feature 'UnitPrice' has a high MI score of approximately 3.1. This high score indicates a strong relationship between 'UnitPrice' and 'TotalSales'. This feature provides significant information about the 'TotalSales'. This suggests that 'UnitPrice' is a highly informative feature and should be considered a priority for inclusion in the model. By using 'StockCode', the model can better predict 'TotalSales', likely due to specific product codes being associated with higher sales.

In contrast, the feature 'Year' has a low MI score, indicating a weak relationship with 'TotalSales'. This means that the year in which the transaction occurred does not provide much information about the total sales. It suggests that the 'Year' might not be a critical factor in predicting sales.

```
[45]  #Drop the unecessary columns in df
      df = df.drop(columns=['InvoiceDate', 'Canceled', 'Manual', 'Discount'])
      ✓ 0.0s
```

Figure 5.2: Code for selecting data

```
[46]  df.info()
      ✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 13 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   UnitPrice    541909 non-null  float64
 5   CustomerID   406829 non-null  float64
 6   Country      541909 non-null  object
 7   TotalSales   541909 non-null  float64
 8   Year         541909 non-null  int32
 9   Month        541909 non-null  int32
 10  Day          541909 non-null  int32
 11  DayOfWeek    541909 non-null  object
 12  YearMonth    541909 non-null  object
dtypes: float64(3), int32(3), int64(1), object(6)
memory usage: 47.5+ MB
```

Figure 5.3: Selected data frame

After calculating the mutual information, we decided to drop the column 'InvoiceDate', 'Canceled', 'Manual' and 'Discount'.

The reason to drop the ‘InvoiceDate’ is because we have extracted the specific time-based features such as Year, Month, Day, DayOfWeek and YearMonth. Thus, the raw date might not provide much information directly. We dropped the ‘Canceled’ because we decided to remove all the canceled invoices from the data set. Thus, the column ‘Canceled’ is not to be used in our data modeling. We dropped the ‘Manual’ because ‘Manual’ indicates a manual entry process and it does not specify the specified stock that sold. These irrelevant features can add noise to the model and increase complexity without improving performance. We drop the ‘Discount’ because it might not contribute meaningful insights and could introduce multicollinearity.

3.2 Data Cleaning

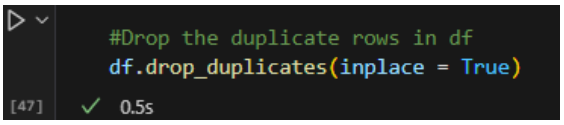


Figure 6.0: Code of dropping duplicate rows

In data cleaning, we drop the duplicate rows (as shown in **Figure 6.0**). Duplicate rows can skew the analysis and affect the model performance. To maintain the integrity of the data set, all duplicate rows were removed. This process ensures that each of the records is unique and contributes only once to the dataset. Hence, it can help to prevent any bias that might arise from repeated entries.

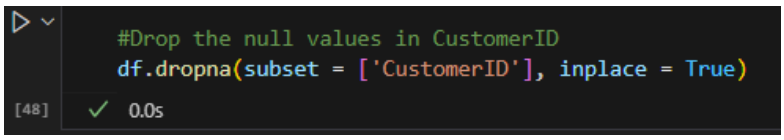


Figure 6.1: Code of dropping the rows that contain null values in CustomerID

We also handle the missing values for the CustomerID because the CustomerID field is crucial for identifying individual customers and linking transactions. Rows containing null values in the CustomerID column were removed. These missing values could lead to inaccuracies in customer-related analysis and segmentation.

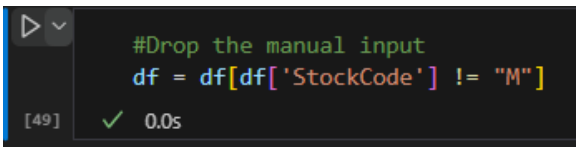


Figure 6.2: Code of dropping the manual input invoices

We handle the manual input invoices by dropping them from the data set because 'Manual' represents entries that were manually inputted and may not follow the standard data entry procedures. The rows that contain manual input are excluded from the data set to maintain the consistency and reliability in the data.

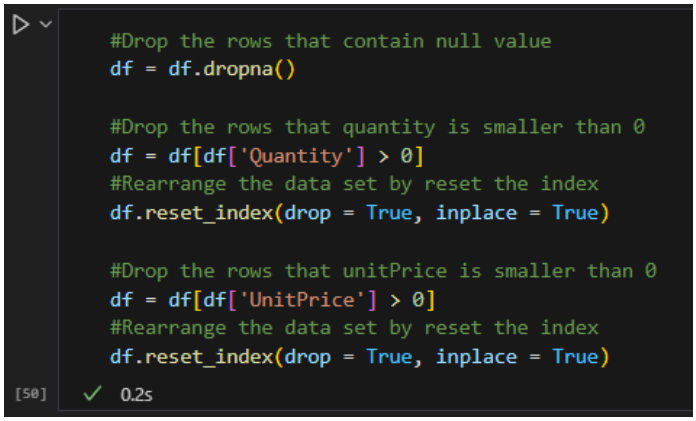


Figure 6.3: Code of dropping the zero and negative values in Quantity and UnitPrice

Lastly, we handle the null, zero or negative values in 'Quantity' and' UnitPrice'. We directly drop all the rows that contain null value in order to improve the data consistency and reliability. The rows with zero and negative values in 'Quantity' and 'UnitPrice' are removed because these data indicate the cancellation which could skew the analysis and model performance. Our data analysis purpose is to predict the total sales for the online retail store which is not including the cancellation.

3.2.1 Outliers Detection

Outliers are data points that differ significantly from other observations. Identifying outliers is essential as they can skew the results of the analysis and lead to inaccurate predictions. To visualize the outliers for each feature in our dataset, we use box plots which provide a graphical depiction of the distribution of data based on the five-number summary: minimum, first quartile, median, third quartile and maximum. Any data points outside 1.5 times the interquartile range from the quartiles are considered outliers and are plotted individually on the box plot.

```
#Plot a box plot to visualize the outlier of the data set
plt.figure(figsize=(12,8))
df.boxplot(patch_artist=True, sym='k.')
plt.show()
```

[51] ✓ 1.7s

Figure 6.4: Code of visualizing outlier of every variables in DataFrame

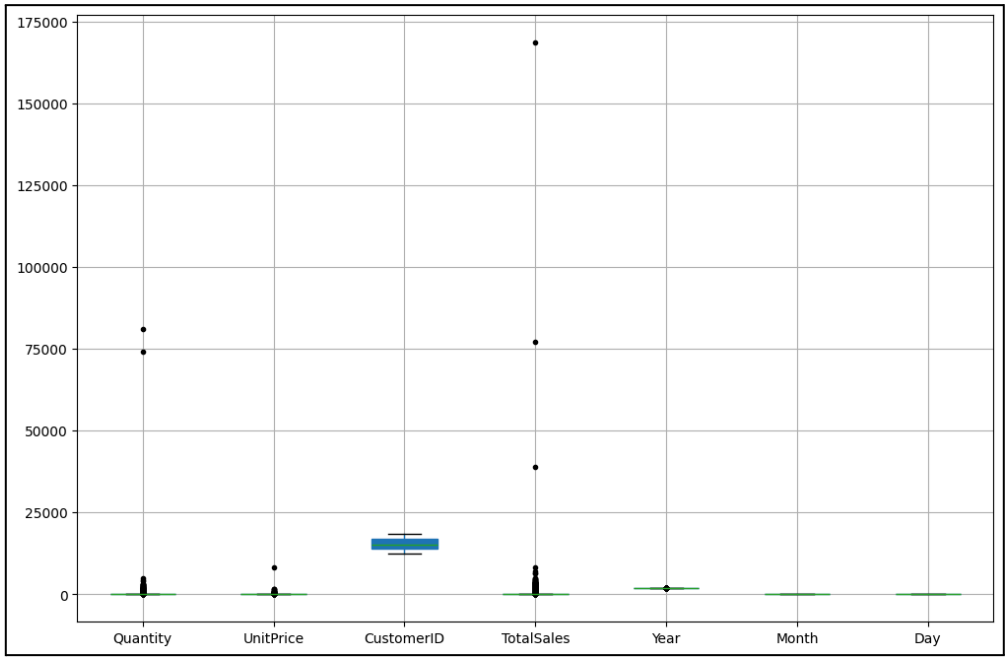


Figure 6.5: Box Plot of each of the variables in the DataFrame

3.2.2 Handling Outliers

After identifying the outliers, we choose the Z-score method to quantify the outliers. The Z-score indicates how many standard deviations an element is from the mean. For continuous features, we calculate the Z-score and remove data points with a Z-score greater than 3 as they are considered outliers.

Below is the formula of calculating Z-Score:

$$Z = \frac{x - \mu}{\sigma}$$

, where x indicates the observed value, u denotes the mean of the sample and o denotes the standard deviation of the sample.

In this part, we calculate the Z-score for 'Quantity', 'UnitPrice' and 'TotalSales'. Outliers in the 'Quantity' columns such as high or negative values might represent data entry errors or exceptional cases such as bulk orders. These extreme values can distort statistical analyses and model predictions by skewing the mean and variance. Extreme values can affect the performance of predictive models and lead to inaccurate forecasts and misleading insights. Prices that are far outside the norm can also skew the calculation of metrics like the average price and affect pricing strategies and analyses. Removing outliers helps ensure that price data used in models and analyses are representative of typical transactions. We also remove the outliers for the 'TotalSales' because it is the product of quantity and unit price. The outliers can significantly influence total sales figures and distort analysis outcomes.

```
[52]
from scipy import stats

#Calculate the Z-score for 'Quantity', 'UnitPrice' and 'TotalSales'
z_Score = np.abs(stats.zscore(df[['Quantity', 'UnitPrice', 'TotalSales']]))
print(z_Score)
✓ 0.0s
```

Figure 6.6: Code of calculating Z-Score for Quantity, UnitPrice and TotalSales

	Quantity	UnitPrice	TotalSales
0	0.039394	0.029995	0.023205
1	0.039394	0.025835	0.006987
2	0.028315	0.016702	0.001646
3	0.039394	0.025835	0.006987
4	0.039394	0.025835	0.006987
...	...	...	...
392407	0.006157	0.142983	0.039615
392408	0.039394	0.059903	0.031892
392409	0.050474	0.076347	0.019022
392410	0.050474	0.076347	0.019022
392411	0.056013	0.129518	0.024653

[392412 rows x 3 columns]

Figure 6.7: Z-Score for Quantity, UnitPrice and TotalSales

```
#Define the threshold
threshold = 3

#Print the indices for outliers
outlier_indices = np.where(z_Score > threshold)
print(outlier_indices)
```

[53] ✓ 0.0s

```
(array([   237,    703,    838, ..., 391616, 391946, 391946]), array([1, 0, 2, ..., 0, 0, 2]))
```

Figure 6.8: Code of finding the outlier and its results.

```
#Remove the outliers
df = df[(z_Score < threshold).all(axis=1)]
```

[54] ✓ 0.0s

Figure 6.9: Code of removing the outlier

After removing the outliers, we visualize the cleaned data using boxplots. The new boxplots show a more consistent distribution of data without extreme values, confirming the effectiveness of the Z-score method in handling outliers.

```
#Plot the box plot
plt.figure(figsize=(10,5))
df.boxplot(patch_artist=True, sym='k.')
plt.show()
```

[55] ✓ 1.8s

Figure 6.10: Code of plotting the box plot for the variables in the DataFrame

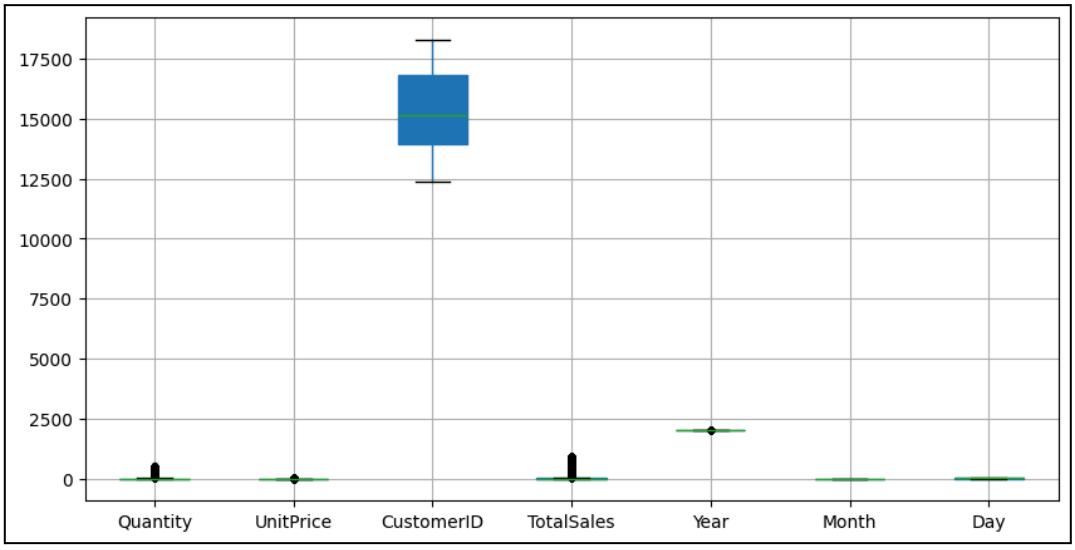


Figure 6.11: Box plot of the variables in DataFrame after removing outliers

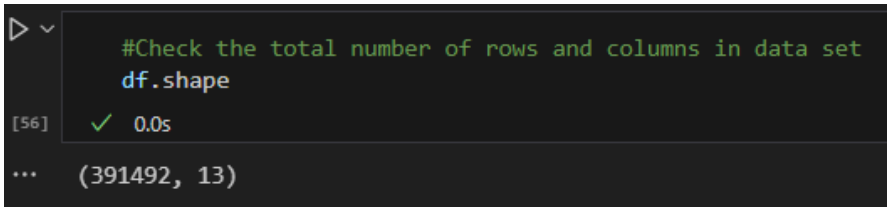


Figure 6.12: Total rows and columns in the DataFrame

In this part, we did all the data cleaning such as removing the null values, zero values and also the outliers. The data set now is clean and prepared to start the normalization process.

3.3 Data Construction

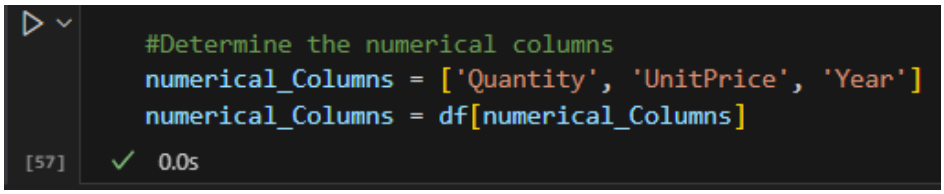


Figure 7.0: Code of constructing continuous data set

In this stage, we identified and extracted numerical columns from the data set to facilitate further analysis and modeling. The variable numerical_Columns is defined as a list containing the names of columns that are considered numerical: ['Quantity', 'UnitPrice', 'Year']. This selection is based on the nature of the data in these columns. 'Quantity' and 'UnitPrice' represent continuous numerical values related to transactions while 'Year' is a discrete numerical value indicating the year of the transaction.

We extract only the numerical columns allowing for focused analysis and preprocessing specific to numerical data. The purpose is that we also determine the 'Year' to be normalized because we want to conduct a forecast analysis. Normalization on the 'Year' will increase the data accuracy. This step is essential for tasks such as statistical analysis and feature scaling.

3.4 Data Normalisation

```
print(np.min(numerical_Columns))
print(np.max(numerical_Columns))

#Calculate the normalized values for each of the numerical data
x_normalized = (numerical_Columns - numerical_Columns.min()) / (numerical_Columns.max() - numerical_Columns.min())
```

[58] ✓ 0.0s

```
0.001
2011.0
```

Figure 8.0: Code of finding the normalized values for the continuous feature data

```
print(x_normalized.head())
```

[59] ✓ 0.0s

	Quantity	UnitPrice	Year
0	0.009276	0.056646	0.0
1	0.009276	0.075313	0.0
2	0.012987	0.061090	0.0
3	0.009276	0.075313	0.0
4	0.009276	0.075313	0.0

Figure 8.1: Viewing the normalized values for Quantity and UnitPrice

In this stage, normalization is a crucial preprocessing step for continuous data as it scales the data to a standard range, typically between 0 and 1. This ensures that features with larger scales do not dominate the model.

In this case, we choose a min-max normalization method to normalize the continuous features. The formula for this method is:

$$X_{new} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

, while X denotes the observed value, Xmin denotes the minimum value for the X, Xmax denotes the maximum value for the X.

We specifically look at the normalized values for ‘Quantity’ and ‘UnitPrice’ to ensure that they fall within the expected range.

3.5 Data Concatenation

```
#Merge the normalized data with the original data set
#by replacing it in the original data set
df['Quantity'] = x_normalized['Quantity']
df['UnitPrice'] = x_normalized['UnitPrice']
```

[60] ✓ 0.0s

Figure 9.0: Code of merging the normalized continuous feature data with the original data frame

After calculating the normalized value for continuous feature data, we merge the normalized value of continuous feature data with the original data frame to ensure the extraction of more insightful information. This step ensures that we have a complete dataset, with both normalized continuous features and encoded categorical features, ready for model training and evaluation. This merged dataset combines the benefits of both types of data. It allows the model to learn from a diverse set of features while maintaining data integrity and consistency.

3.6 Data Selection for Customer Segmentation

```
#Data Preparation for Customer Segmentation Analysis
df = pd.read_csv('onlineRetail.csv')

df['TotalSales'] = df['Quantity'] * df['UnitPrice']

df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'], format='%d/%m/%Y %H:%M')
df['Year'] = df['InvoiceDate'].dt.year
df['Month'] = df['InvoiceDate'].dt.month
df['Day'] = df['InvoiceDate'].dt.day
df['DayOfWeek'] = df['InvoiceDate'].dt.strftime('%A')
df['YearMonth'] = df['InvoiceDate'].dt.strftime('%Y-%m')
```

[ ]

Figure 10.0: Data Preparation for Customer Segmentation Analysis

```
#Find the last purchase date
last_date = df['InvoiceDate'].max()

#Calculate the recency, frequency and monetary
rfm = df.groupby('CustomerID').agg({
    'InvoiceDate': lambda x: (last_date - x.max()).days,
    'InvoiceNo': 'nunique',
    'TotalSales': 'sum'
}).reset_index()

rfm.columns = ['CustomerID', 'Recency', 'Frequency', 'Monetary']
```

[ ]

Figure 10.1: Creating RFM data frame

In this section, we would like to conduct a customer segmentation analysis. The goal of doing a customer segmentation study is multifaceted: to improve marketing effectiveness and increase customer happiness. Businesses may personalize marketing campaigns to specific client groups by segmenting them based on Recency, Frequency, and Monetary (RFM) indicators, ensuring that promotional efforts are relevant to individual preferences and habits. This focused strategy not only enhances marketing ROI, but it also boosts client engagement by sending relevant and tailored information. Furthermore, recognizing various client categories enables organizations to improve service quality, anticipate consumer wants, and solve possible difficulties proactively, resulting in increased overall customer satisfaction.

Furthermore, segmentation aids in the optimization of customer retention by identifying high-value customers and designing engagement tactics, as well as addressing at-risk categories to prevent churn. Businesses that focus on successful client groups may increase revenue and profitability while also uncovering new income streams targeted to individual demands. Overall, customer segmentation provides significant information that help firms make strategic decisions, allowing them to create segment-specific strategies that drive growth and increase their competitive advantage.

We process all the necessary data and create a data frame for RFM (Recency, Frequency and Monetary).

CustomerID	Unique identifier for each customer.
Recency (R)	Determine how recently a customer made a purchase. Measure the time elapsed since the last purchase.
Frequency (F)	Count the total number of purchases made by the customer within a given time frame.
Monetary (M)	Calculate the total monetary value of purchases made by the customer.

4.0 Modeling

In order to forecast the future outcomes, we perform the predicted modeling using machine learning modeling. Machine learning model is building algorithms that allow systems to learn patterns and make predictions or decisions based on data.

The machine learning model is divided into two categories: supervised learning and unsupervised learning. The supervised learning model is trained on labeled data, while the unsupervised learning model is used when the data is unlabeled.

In our analysis, we will use several machine learning algorithms to test and refine our predictive model. We will implement Decision Tree Regressor, Linear Support Vector Regression (LinearSVR), Random Forest, K-Nearest Neighbors (KNN) Regression and Ridge Regression. By testing these various algorithms, we aim to identify the best-performing model for our forecasting needs. It can ensure accurate and reliable predictions of future outcomes.

4.1 Train Test Split

```
#Determine y = TotalSales
#X = StockCode, Description, Quantity, UnitPrice, CustomerID, Country, Year, Month

y = df.TotalSales.values
x_Data = df.drop(['InvoiceNo', 'Day', 'YearMonth', 'TotalSales'], axis=1)
```

[61] ✓ 0.0s

Figure 11.0: Code of defining X and Y

In this section, we determine the target variable is 'TotalSales' and extract it from the DataFrame as the output we want to predict. Feature variables are selected by dropping columns that are not needed for the modeling such as 'InvoiceNo', 'Day', 'YearMonth' and 'TotalSales'.

```
from sklearn.preprocessing import LabelEncoder

#Encodes all the categorical data
le = LabelEncoder()
x_Data['StockCode'] = le.fit_transform(x_Data['StockCode'])
x_Data['Description'] = le.fit_transform(x_Data['Description'])
x_Data['CustomerID'] = le.fit_transform(x_Data['CustomerID'])
x_Data['Country'] = le.fit_transform(x_Data['Country'])
x_Data['DayOfWeek'] = le.fit_transform(x_Data['DayOfWeek'])
```

[62] ✓ 0.2s

Figure 11.1: Encoding categorical features

Categorical variables need to be converted into numerical format to be used in machine learning models. Label encoding transforms these categorical variables into numeric labels.

By defining the x with y value and encoding categorical features, we ensure that the data is in a suitable format for machine learning algorithms. This step is crucial for effective data preprocessing and allows models to work with both numerical and categorical data seamlessly.

```
#Import all the neccessary libraries that will be used for data modeling
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import mean_absolute_percentage_error, r2_score, mean_squared_error, mean_absolute_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import LinearSVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.inspection import permutation_importance
from sklearn.linear_model import Ridge
from sklearn.tree import plot_tree
```

[63] ✓ 0.0s

Figure 11.2: Importing all necessary libraries

From **Figure 11.2**, we import all the essential libraries required for data modeling and analysis. These libraries provide the tools and functions necessary for building and evaluating machine learning models, as well as for performing data preprocessing and analysis tasks. By importing these libraries, we equip ourselves with the necessary tools for building, evaluating and fine-tuning machine learning models.

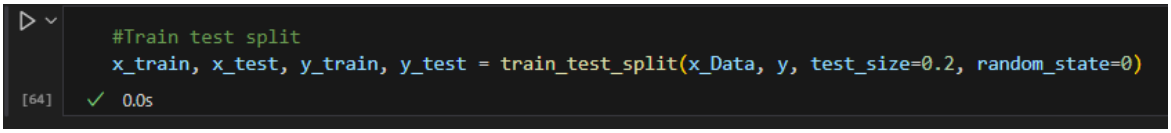


Figure 11.3: Code of splitting independent variable and dependent variable into training set and test set

The first step in modeling is to perform a train-test split. In the training set, the model learns the patterns and relationships between the independent and dependent variables. In the test set, the model's performance is evaluated after training.

We perform a 2/8 proportion of the dataset to include in the test split. 80% of the data will be used for training and the remaining 20% will be used for testing.

Variable	**Explanation**
x_train	Training subset of the features
x_test	Testing subset of the features
y_train	Training subset of the target variable
y_test	Testing subset of the target variable

Training the model on x_train and y_train allows it to learn from a substantial part of the data. Testing the model on x_test and y_test provides an unbiased evaluation of how well the model performs on new data. This process is crucial for assessing the generalization ability of the model and ensuring that it is not overfitting to the training data.

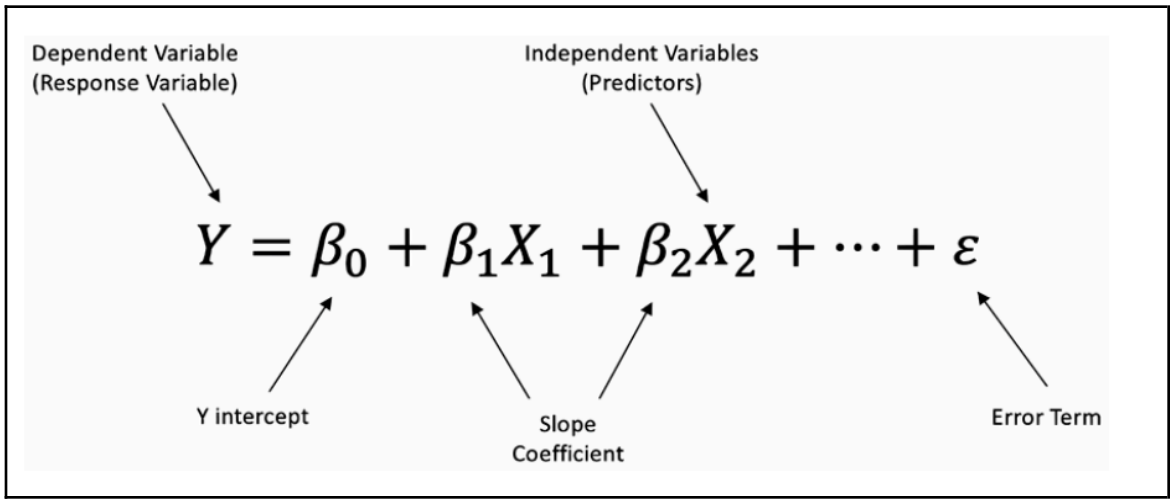
4.2 Evaluation Metrics

The target variable is total sales, which is a continuous variable. To evaluate the model's performance, we use regression evaluation metrics. These metrics include:

1. Mean Absolute Percentage Error (MAPE)
2. R-Squared (Coefficient of Determination)
3. Root Mean Square Error (RMSE)
4. Mean Absolute Error (MAE)

In this section, we use the **Linear Regression** to perform the metrics for a basic understanding of the data set. Linear Regression serves as a foundational method to assess how well the independent variables predict the continuous target variable 'TotalSales'. It aims to model the relationship between the dependent variable and the independent variables by fitting a linear equation to observed data.

Formula:



4.2.1 Mean Absolute Percentage Error (MAPE)

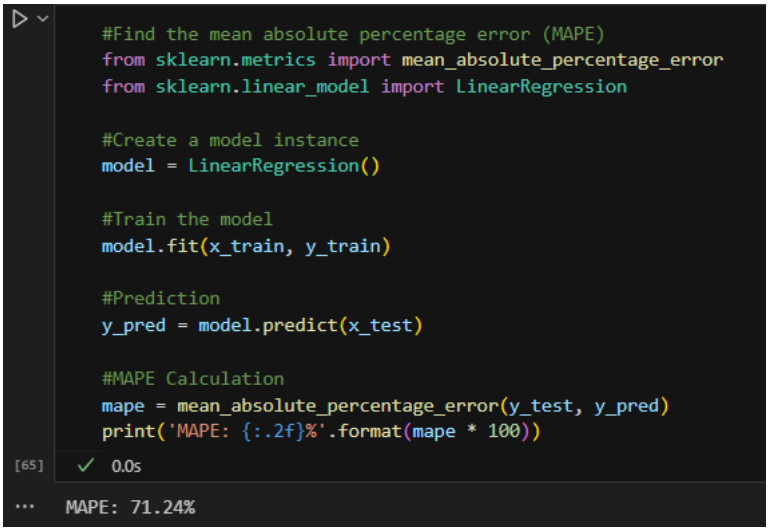


Figure 12.0: Code of Mean Absolute Percentage Error (MAPE)

The **Mean Absolute Percentage Error (MAPE)** is a metric used to evaluate regression models by measuring accuracy through the average of the absolute percentage differences between actual and predicted values.

Formula:

$\text{MAPE} = \frac{1}{n} \sum_{i=1}^{n} \frac{\lvert y_i - \hat{y}_i \rvert}{y_i} \cdot 100\%$	$y_i = actual\ value$ $\hat{y}_i = predicted\ value$ $n = number\ of\ the\ observation$

Based on the analysis shown in **Figure 12.0**, we observe that the Mean Absolute Percentage Error (MAPE) for the dataset, which measures the average magnitude of errors as a percentage of actual values, is 71.24%. This high MAPE indicates that the model's predictions are, on average, 71.24% off from the actual values, suggesting significant inaccuracies in the forecasting performance.

4.2.2 R-Squared (Coefficient of Determination)

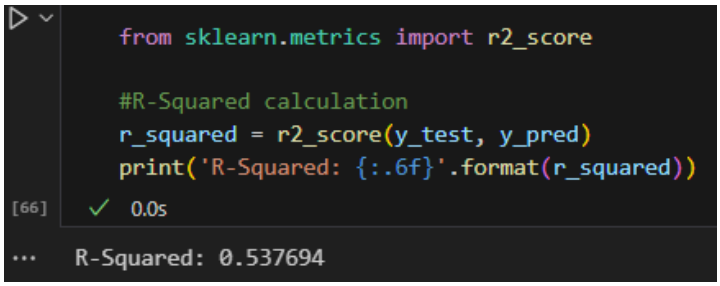


Figure 12.1: Code of R-Squared (Coefficient of Determination)

The R-Squared is a metric used to evaluate the goodness-of-fit of a regression model, representing the proportion of the variance in the dependent variable that can be explained by the independent variables.

Formula:

$R^2 = 1 - \frac{\text{sum squared regression (SSR)}}{\text{total sum of squares (SST)}}$ $= 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}.$	$y_i = actual\ value$ $\hat{y}_i = predicted\ value$ $\bar{y} = mean\ of\ actual\ value$ $n = number\ of\ the\ prediction$

Based on the analysis illustrated in **Figure 12.1**, the Coefficient of Determination (R^2) for this dataset is calculated to be 0.5377. This value suggests a moderate level of fit, meaning that approximately 53.77% of the variability in the dependent variable can be explained by the independent variable(s) in the model, while the remaining 46.23% of the variability is due to other factors not accounted for by the model.

4.2.3 Root Mean Square Error (RMSE)

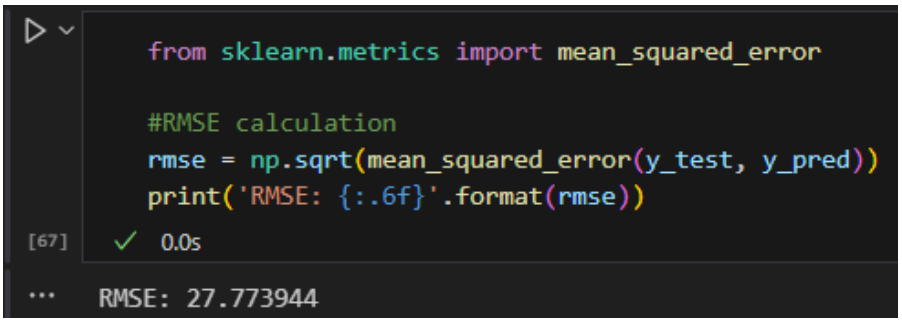


Figure 12.2: Code of Root Mean Square Error (RMSE)

The Root Mean Square Error (RMSE) is a metric used to measure the average magnitude of errors between predicted and actual values by calculating the square root of the average of the squared differences, thus giving more weight to larger errors.

Formula:

$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$	$y_i = actual\ value$ $\hat{y}_i = predicted\ value$ $n = number\ of\ the\ prediction$

Based on the analysis presented in **Figure 12.2**, the Root Mean Square Error (RMSE) for the dataset, which quantifies the average magnitude of the prediction errors, has been calculated to be 27.7739. This RMSE value indicates the standard deviation of the residuals or prediction errors, providing insight into the accuracy and precision of the model's forecasts.

4.2.4 Mean Absolute error (MAE)

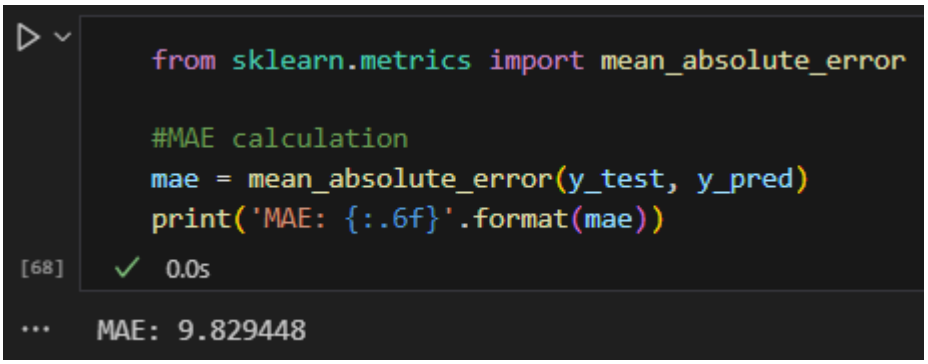


Figure 12.3: Code of Mean Absolute error (MAE)

The Mean Absolute Error (MAE) is a metric used to measure the accuracy of a regression model by representing the average magnitude of the errors between predicted and actual values, without considering the direction of the errors.

Formula:

$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} \lvert y_i - \hat{y}_i \rvert$	$y_i = actual\ value$ $\hat{y}_i = predicted\ value$ $n = number\ of\ the\ prediction$

Based on the analysis presented in **Figure 12.3**, the Mean Absolute Error (MAE) for the dataset has been determined to be 9.8294 units. This MAE value signifies that, on average, the absolute difference between the predicted values and the actual observed values in the dataset is 9.8294 units, providing a quantifiable measure of the average prediction error magnitude.

4.3 Decision Tree Regressor

4.3.1 Algorithm and Assumptions

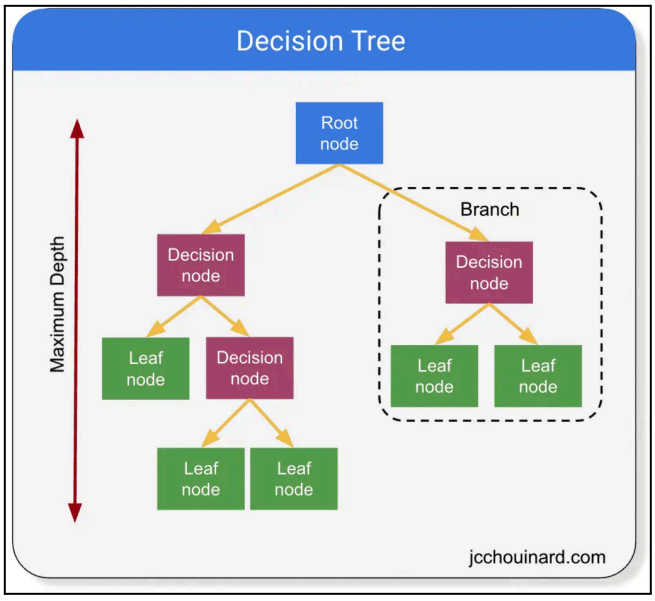


Figure 13.0: Algorithm for Decision Tree

A decision tree is a hierarchical data structure used in machine learning for both supervised and unsupervised learning tasks. It begins with a root node that represents the entire dataset. This root node is connected to one or more branch nodes, each of which splits the data based on a certain criterion. The branch nodes consist of decision nodes, which make decisions based on feature values, and eventually lead to leaf nodes. Each leaf node represents a final output or classification, indicating the outcome or label of the data at that point in the tree. Decision Trees are easy to interpret and visualize, making them useful for understanding how predictions are made.

The Decision Tree Regressor algorithm begins by selecting the best feature and split point to minimize a regression criterion, such as mean squared error (MSE). The data is then split based on this feature, and the process is recursively applied to each subset to build the tree. This splitting continues until a stopping criterion is met, such as a maximum depth of the tree or a minimum number of samples per leaf. For new instances, the algorithm follows the path from the root to a leaf node to make predictions, where the leaf node's value is used as the predicted output.

Assumptions	Explanation	Implication
Splitting based on feature values can effectively partition the data into homogenous subsets.	Decision trees recursively split the data into subsets where each subset is more homogeneous in terms of the target variable.	Can lead to overfitting if the tree is too deep or not properly pruned.
Limiting the tree's depth or the number of samples per leaf helps prevent overfitting.	Pruning the tree or setting constraints on its depth can help generalize better and reduce overfitting.	Helps in creating simpler models that generalize well but may underfit if constraints are too strict.
Greedy approach of selecting the best split at each step improves the model's accuracy.	At each node, the algorithm selects the feature and split that best improves the homogeneity of the subsets.	Can create complex models that fit the training data well, potentially leading to overfitting if not controlled.

Table 4.0: Assumption of Decision Tree Regression

4.3.2 Model Building

```
#Create a model instance
dt = DecisionTreeRegressor(random_state = 0)

#Train the model
dt.fit(x_train, y_train)

#Prediction
y_pred_dt = dt.predict(x_test)

#MAPE Calculation
mape_dt = mean_absolute_percentage_error(y_test, y_pred_dt)
print('MAPE: {:.2f}%'.format(mape_dt * 100))
```

[69] ✓ 1.1s

```
MAPE: 0.03%
```

Figure 13.1: Model Building for Decision Tree Regression

4.3.3 Model Evaluation

To build a model, we created and trained a Decision Tree Regression model using the fit method. Then, we predict values using the predict method to evaluate its performance.

We calculated the Mean Absolute Percentage Error (MAPE) for Decision Tree Regression. It used to evaluate the performance of the Decision Tree Regression model by estimating its prediction accuracy. We trained a Decision Tree Regression model using the training database. The predicted values are compared against the actual values to assess the model's performance.

The MAPE for Decision Tree Regression is 0.03%, which means that it is strongly accurate and easy to interpret and understand. We will conduct a hyperparameter tuning to train again the testing model to ensure the data accuracy and quality to improve the model performance and generalizability although the model is already accurate.

4.3.4 Hyperparameter Tuning

```
#Determine the parameters for hyperparameter tuning
parameter_grid = {
    'max_features': ['sqrt'],
    'max_depth': [5],
    'criterion': ['squared_error'],
    'min_samples_leaf': [2],
    'min_samples_split': [3]
}

#Use grid search to fit the data
grid_search = GridSearchCV(DecisionTreeRegressor(random_state=0), parameter_grid, cv=5)
grid_search.fit(x_train, y_train)

#Determine the best estimator by using grid search
best_model = grid_search.best_estimator_
y_pred_dt = best_model.predict(x_test)

#Calculate the accuracy for test set and training set
dt_accurate_test = (best_model.score(x_test, y_test)) * 100
dt_accurate_train = (best_model.score(x_train, y_train)) * 100

#Print the accuracy for test set and training set
print('Decision Tree accuracy for test set: {:.2f}%'.format(dt_accurate_test))
print('Decision Tree accuracy for training set: {:.2f}%'.format(dt_accurate_train))

[70]  ✓ 0.9s

...  Decision Tree accuracy for test set: 82.77%
     Decision Tree accuracy for training set: 83.12%
```

Figure 13.2: Code of Hyperparameter Tuning for Decision Tree Regression and Calculation for data accuracy

In this section, we perform hyperparameter tuning for a Decision Tree Regressor to optimize its performance. Hyperparameter tuning helps us find the best configuration for the model which can improve its accuracy and effectiveness.

We define a grid of hyperparameters to search over which includes:

1. **`max_features`** which is used to specify the number of features to consider when looking for the best split. We set 'sqrt' to use the square root of the total number of features.
2. **`max_depth`** which is used to specify the maximum depth of the tree. We set it to 5 to limit the complexity and prevent overfitting.
3. **`criterion`** which is used to measure the quality of a split. We set it to 'squared_error' to minimize the mean squared error.
4. **`min_samples_leaf`** which is used to specify the minimum number of samples required to be at a leaf node. We set it to 2 to ensure that each leaf has a sufficient number of samples.

5. **`min_samples_split`** which is used to specify the minimum number of samples required to split an internal node. We set it to 3 to control the tree's growth.

```
#Plot the decision tree
plt.figure(figsize=(30,20))
plot_tree(best_model, feature_names=x_train.columns)
plt.title('Decision Tree Visualization')
plt.show()
```

[72] ✓ 1.0s

Figure 13.3: Code of Building Decision Tree

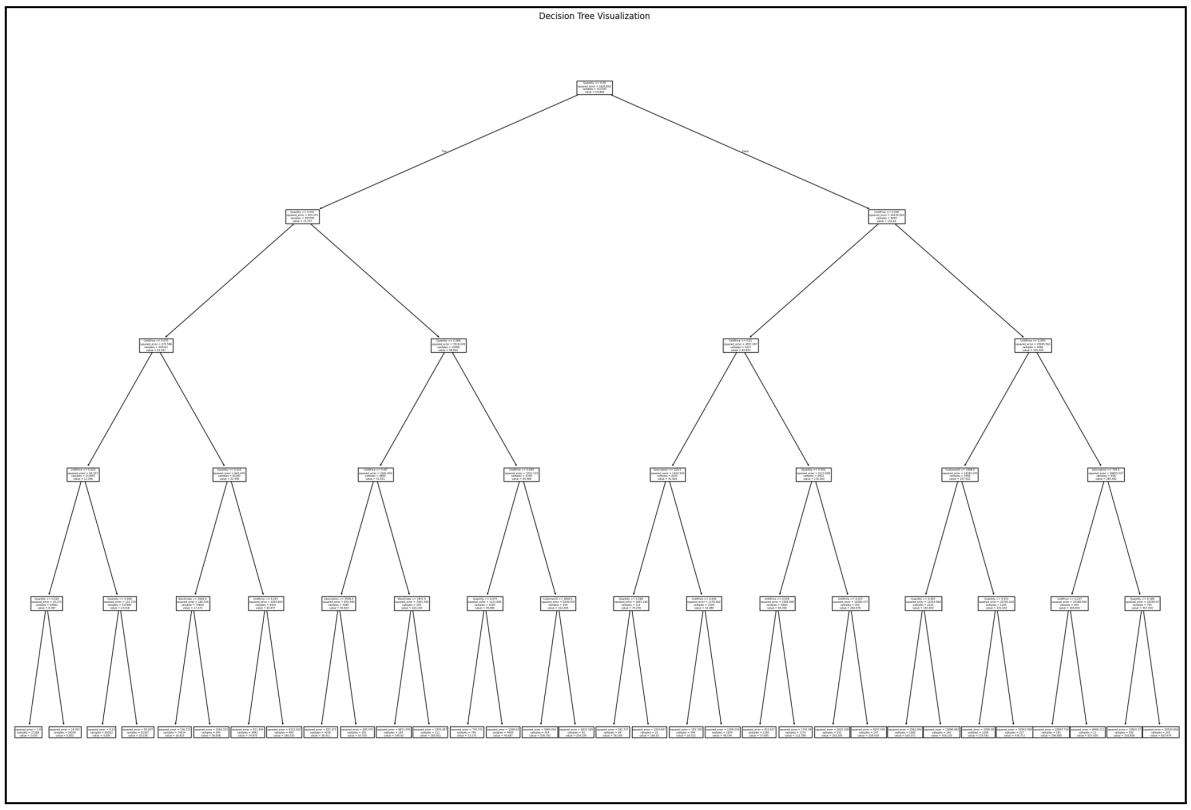


Figure 13.4: Decision Tree Model

In this part, we build a decision tree model to understand the structure and decisions made by the Decision Tree model.We visualize the trained tree using a plot. This visualization helps in interpreting the model's decision-making process and examining how the features contribute to the predictions.

4.3.5 Model Revaluation

```
# Calculate MAPE
dt_mape = mean_absolute_percentage_error(y_test, y_pred_dt)
print('MAPE for Decision Tree Regressor: {:.2f}%'.format(dt_mape * 100))

# Calculate R-Squared
dt_r2 = r2_score(y_test, y_pred_dt)
print('R-Squared for Decision Tree Regressor: {:.6f}'.format(dt_r2))

# Calculate RMSE
dt_rmse = np.sqrt(mean_squared_error(y_test, y_pred_dt))
print('RMSE for Decision Tree Regressor: {:.6f}'.format(dt_rmse))

# Calculate MAE
dt_mae = mean_absolute_error(y_test, y_pred_dt)
print('MAE for Decision Tree Regressor: {:.6f}'.format(dt_mae))
```

[71] ✓ 0.0s

```
MAPE for Decision Tree Regressor: 67.25%
R-Squared for Decision Tree Regressor: 0.827693
RMSE for Decision Tree Regressor: 16.956007
MAE for Decision Tree Regressor: 7.275265
```

Figure 13.5: Descriptive statistics for Decision Tree Regression

In this section, we evaluate the performance of the Decision Tree Regressor using several regression metrics. These metrics provide insights into the accuracy and effectiveness of the model's predictions.

From **Figure 13.5**, a MAPE of 67.25% indicates that, on average, the model's predictions deviate by 67.25% from the actual values. This suggests that there is a considerable amount of error in the predictions relative to the scale of the data.

An R-Squared value of 0.8277 indicates that approximately 82.77% of the variance in the target variable TotalSales is explained by the model. This suggests a strong correlation between the predicted and actual values.

An RMSE of 16.9560 indicates the average magnitude of the prediction error in the same units as the target variable. This value provides a measure of how far off the model's predictions are from the actual values on average.

An MAE of 7.2753 indicates the average absolute error in the predictions. This value provides insight into the average magnitude of errors in the model’s predictions.

```
#Scatter plot of Actual and predicted values by using Decision Tree Regressor
plt.scatter(y_test, y_pred_dt)
#Plot the diagonal line (perfect prediction)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--', lw=2)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Decision Tree Regressor: Actual vs Predicted')
plt.show()
```

[73] ✓ 0.2s

Figure 13.6: Code of plotting scatter plot for Decision Tree Regressor: Actual vs Predicted

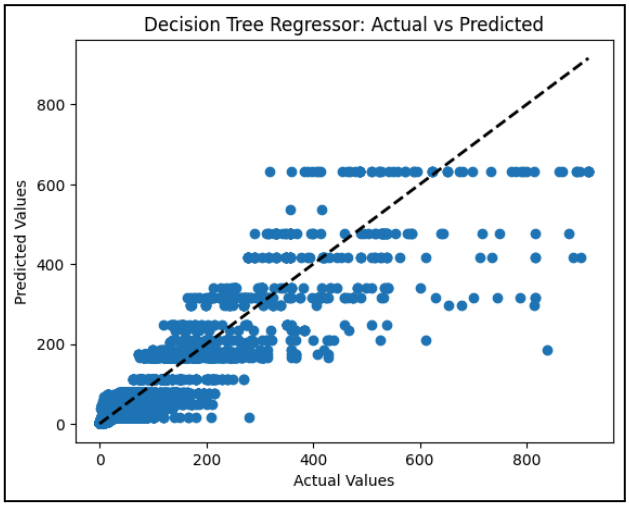


Figure 13.7: Scatter plot for Decision Tree Regressor: Actual vs Predicted

The scatter plot shows individual points where each point represents an observation. The x-axis represents the actual values from the test dataset and the y-axis represents the predicted values generated by the Decision Tree Regression. The dashed black line represents the ideal scenario where the predicted values equal the actual values. This line is the reference line that can help to visualize how close or far the predicted values are from the actual values. The line is drawn from the minimum to the maximum of the actual values and serves as a benchmark for perfect prediction.

From **Figure 13.7**, most of the points are almost lying on the diagonal line. It indicates that the model's predictions are strongly accurate. The points are scattered closer from the line, which indicates that the model's predictions have no significant errors. The scatter plot helps visualize the performance of the Decision Tree Regression model by showing how well the predicted values align with the actual values. It provides a quick visual assessment of model accuracy and helps identify any patterns or areas where the model may be underperforming.

```
#Calculate the residuals
residuals = y_test - y_pred_dt

#Plot a residual plot for decision tree regressor
plt.scatter(y_pred_dt, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot for Decision Tree Regressor')
plt.show()
```
[74] ✓ 0.2s

Figure 13.8: Code of Plotting the Residual Plot for Decision Tree Regression

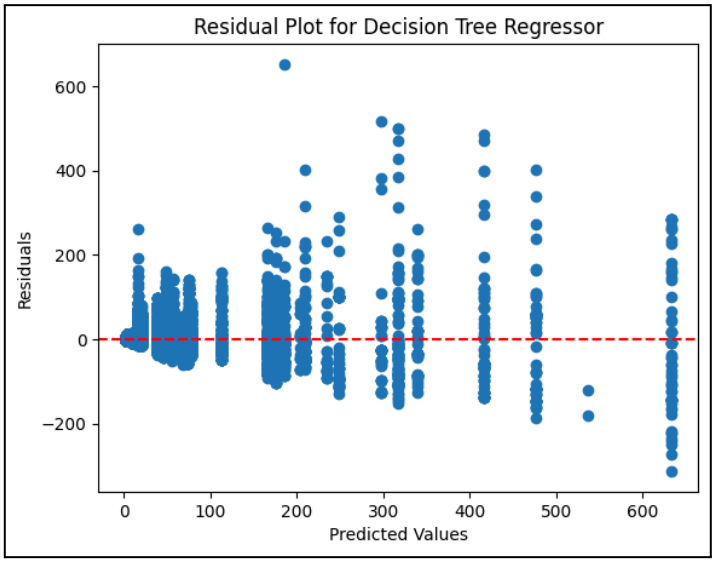


Figure 13.9: Residual Plot for Decision Tree Regression

We calculate the residuals which is the difference between the actual values and the predicted values. This provides a measure of the error in the model's prediction.

From **Figure 13.9**, the scatter plot shows residuals on the y-axis and predicted values on the x-axis. Each point represents an observation from the test dataset. The plot helps visualize whether the residuals are randomly distributed. The red dashed line represents the baseline where the residuals are zero. This helps to see how far the residuals deviate from zero.

From **Figure 13.9**, we can see that the residuals are heteroscedasticity scattered around the horizontal line. It indicates that the model's errors are random and there are no obvious patterns in the residuals. The residuals get larger as the predicted moves from small to large. This suggests a worse-fitted model. There are potential problems with the model's assumptions and it is needed to guide further improvement.

```
#find the importance for each of the feature
importance= best_model.feature_importances_

#Sort the importance in descending order
indices = np.argsort(importance)[::-1]
#Determine the feature name
feature_name = x_Data.columns

#Print the feature ranking
print("Feature ranking for Decision Tree Regressor:")
for f in range(x_Data.shape[1]):
    print("%d. feature %d %s (%f)" % (f + 1, indices[f], feature_name[indices[f]], importance[indices[f]]))

#Plot the bar chart for the feature importances
plt.figure(figsize=(10, 6))
plt.title("Feature importances for Decision Tree Regressor")
plt.bar(range(x_Data.shape[1]), importance[indices], align="center")
plt.xticks(range(x_Data.shape[1]), feature_name)
plt.xlim([-1, x_Data.shape[1]])
plt.show()
[75]  ✓ 0.0s
```

Figure 13.10: Code of Feature importances for Decision Tree Regression

```
Feature ranking for Decision Tree Regressor:
1. feature 2 Quantity (0.678082)
2. feature 3 UnitPrice (0.310820)
3. feature 1 Description (0.004537)
4. feature 0 StockCode (0.003874)
5. feature 4 CustomerID (0.002687)
6. feature 8 DayOfWeek (0.000000)
7. feature 5 Country (0.000000)
8. feature 7 Month (0.000000)
9. feature 6 Year (0.000000)
```

Figure 13.11: Feature ranking for Decision Tree Regression

From **Figure 13.11**:

Feature Name	Feature Importance	Interpretation
Quantity	0.678082	This feature has the highest importance score. It indicates that it is the most influential variable in predicting the total sales. A score of 0.678082 suggests that Quantity contributes significantly to the model's decisions.
UnitPrice	0.310820	The UnitPrice feature is the second most important, with a score of 0.310820. This indicates that UnitPrice also plays a substantial role in predicting sales, though less so than Quantity.
Description	0.004537	The Description feature has a very low importance score of 0.004537. This suggests that Description contributes minimally to the model's predictions.

StockCode	0.003874	The StockCode feature has a score of 0.003874, indicating that it has a very limited impact on the model's predictions.
CustomerID	0.002687	With a score of 0.002687, CustomerID also contributes very little to the model's predictions, suggesting its limited relevance.
DayOfWeek	0.000000	The DayOfWeek feature has an importance score of 0.000000, meaning it does not contribute at all to the model's predictions.
Country	0.000000	Similarly, the Country feature has an importance score of 0.000000, indicating that it has no impact on the model's predictions.
Month	0.000000	The Month feature also scores 0.000000, showing that it does not influence the predictions.
Year	0.000000	Finally, the Year feature has an importance score of 0.000000, indicating that it is not relevant for predicting TotalSales.

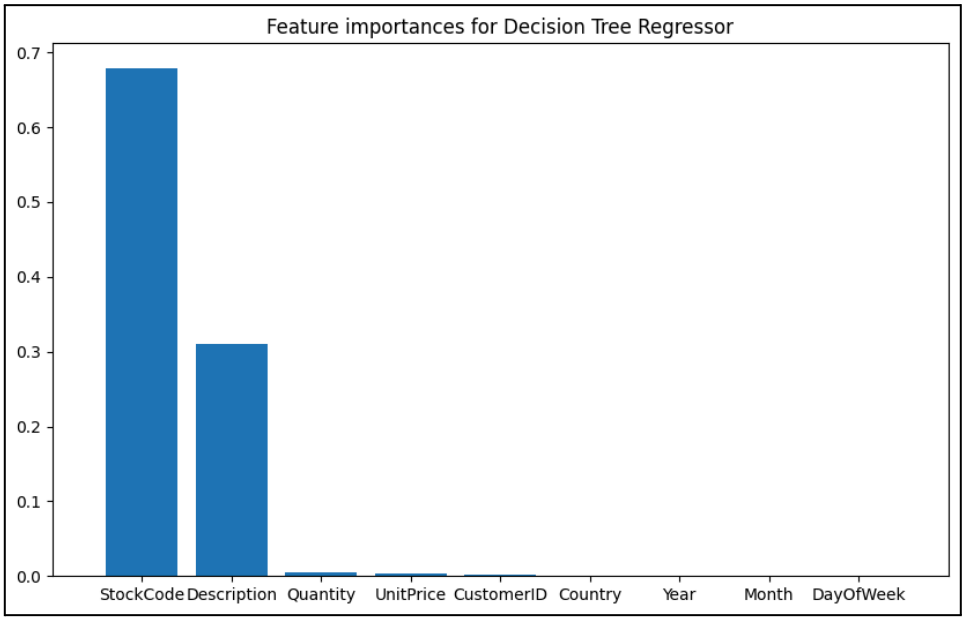


Figure 13.12: Bar chart of feature importances for Decision Tree Regression

Figure 13.12 provides a bar plot visualizing the importance of each feature used in the Decision Tree Regressor model. The height of each bar represents the magnitude of the feature's importance score.

Quantity and UnitPrice are the most significant features. It suggests that the quantity and unit price of items sold are crucial for predicting sales.

Features such as Description, StockCode, CustomerID, and others with very low or zero importance scores do not contribute meaningfully to the model's predictions.

Features with zero importance (DayOfWeek, Country, Month, and Year) have no impact on the model and may be considered for removal or further investigation.

4.4 Linear Support Vector Regression (LinearSVR)

4.4.1 Algorithm and Assumptions

Linear Support Vector Regression (LinearSVR) is a regression technique based on the principles of Support Vector Machines (SVMs). Unlike traditional regression methods that minimize the prediction error, SVR aims to fit the data within a certain margin, allowing for some flexibility or tolerance to error. This technique seeks a balance between model complexity and accuracy by minimizing a cost function, which penalizes deviations from a certain threshold. LinearSVR is particularly useful when dealing with high-dimensional data or when the relationship between features and the target variable is approximately linear.

LinearSVR is efficient for large-scale datasets, as it approximates the solution using linear kernels. It doesn't provide the flexibility of nonlinear kernels, but it is faster and more interpretable, making it suitable for situations where linearity in the data is assumed. This algorithm can handle both regression and classification tasks, but in the context of regression, it seeks to minimize the margin of error while keeping the model simple.

The algorithm behind LinearSVR can be summarized in a few key steps. First, the model attempts to find a hyperplane that best fits the data points within a defined epsilon-insensitive margin. Data points within this margin are considered correctly predicted, even if there's some error. The goal is to minimize the errors for data points outside this margin, while also minimizing the model’s complexity.

Mathematically, the objective function involves minimizing the following expression:

$$\frac{1}{2}||w||^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*)$$

Figure 14.0: Objective Function of Linear Support Vector Regression

where ||w||2||w||^2||w||2 controls the flatness of the model, CCC is the regularization parameter that balances margin width and prediction error, and ξi,ξi*\xi_i, \xi_i^*ξi,ξi* are the slack variables that measure the deviation from the margin. The optimization problem is solved using quadratic programming techniques, where the cost is minimized while keeping the number of errors as low as possible.

Algorithms	Explanation	Implication
Linear relationship between features and target	LinearSVR assumes that there is a linear relationship between the features and the target variable.	The algorithm may not perform well if the relationship between variables is nonlinear without preprocessing.
Independence of observations	It assumes that the observations are independent of each other, and the errors are not correlated.	Violation of this assumption can lead to biased predictions and lower model accuracy.
Homoscedasticity	LinearSVR assumes constant variance of errors.	Heteroscedasticity can lead to underestimation or overestimation of certain data points.
No multicollinearity	Features are assumed to be independent of each other.	High multicollinearity between features can result in unreliable estimates of coefficients and model instability.
Normal distribution of residuals	Assumes that residuals (errors) are normally distributed around the margin.	Non-normal residuals can indicate model misspecification and can affect the reliability of the model's predictions.

Table 5.0: Assumption of LinearSVR

4.4.2 Model Building

```
#Create a model instance
svr = LinearSVR()
svr.fit(x_train, y_train)

#Prediction
y_pred_svr = svr.predict(x_test)

#MAPE Calculation
mape_svr = mean_absolute_percentage_error(y_test, y_pred_svr)
print('MAPE for Linear Support Vector Regression: {:.2f}%'.format(mape_svr * 100))
```

[76] ✓ 1m 33.8s

Figure 14.1: Model Building for LinearSVR

```
MAPE for Linear Support Vector Regression: 341.84%
```

Figure 14.2: MAPE for LinearSVR

4.4.3 Model Evaluation

We calculated the Mean Absolute Percentage Error (MAPE) for LinearSVR. It used to evaluate the performance of the LinearSVR model by estimating its prediction accuracy. We trained a LinearSVR model using the training database. The predicted values are compared against the actual values to assess the model's performance.

The MAPE for ridge regression is 341.84%, which means that it is strongly inaccurate and difficult to interpret and understand. We will conduct a hyperparameter tuning to train again the testing model to ensure the data accuracy and quality.

4.4.4 Hyperparameter Tuning

```
#Determine the parameter used
parameter_grid = {
    'C': [None, 0.1, 1, 10],
    'epsilon': [None, 0.1, 0.2],
    'loss': ['epsilon_insensitive', 'squared_epsilon_insensitive']
}

#Use grid search to fit the data
svr_grid = GridSearchCV(LinearSVR(random_state=0), parameter_grid, cv=5)
svr_grid.fit(x_train, y_train)

#Determine the best estimator for the SVR
best_model = svr_grid.best_estimator_
y_pred_svr = best_model.predict(x_test)

#Calculation off accuracy for test set and training set
svr_accurate_test = best_model.score(x_test, y_test) * 100
svr_accurate_train = best_model.score(x_train, y_train) * 100

print('Linear Support Vector Regression accuracy for test set: {:.2f}%'.format(svr_accurate_train))
print('Linear Support Vector Regression accuracy for training set: {:.2f}%'.format(svr_accurate_test))
[77]  ✓ 36m 45.5s
```

Figure 14.3: Code of calculating LinearSVR accuracy for test set and training set

```
Linear Support Vector Regression accuracy for test set: 1.50%
Linear Support Vector Regression accuracy for training set: 1.61%
```

Figure 14.4: Result of LinearSVR accuracy for test set and training set

To optimize the performance of the Linear Support Vector Regression (SVR) model, we conducted hyperparameter tuning using GridSearchCV. We apply the parameter such as:

1. **`C`** indicates the regularization parameter which helps to control the trade-off between achieving a low error on the training data and minimizing the model complexity.
2. **`epsilon`** indicates the epsilon in the epsilon-SVR model which defines the margin of tolerance where no penalty is given for errors.
3. **`loss`** indicates the loss function to be used. In this case, we decided to use epsilon_insensitive and squared_epsilon_insensitive.

We used GridSearchCV to fit the data with these parameters, allowing us to determine the best parameter combination for the SVR model.

After fitting the SVR model with the best parameters, we evaluated its performance on both the test and training sets.

The accuracy for both the test and training sets is very low, indicating that the Linear SVR model performs poorly on this dataset. This suggests that the model is not capturing the relationships in the data effectively.

The reason that the accuracy is too low including the features used for modeling might not be adequately capturing the underlying patterns in the data. Linear SVR might not be the best choice for this problem, and exploring other regression models or more complex SVR kernels might yield better results.

In this case, we decided to consider using different regression models such as Ridge Regression or ensemble methods such as Random Forest.

4.4.5 Model Revaluation

```
svr_mape = mean_absolute_percentage_error(y_test, y_pred_svr)
print('MAPE for Linear Support Vector Regression: {:.2f}%'.format(svr_mape * 100))

svr_r2 = r2_score(y_test, y_pred_svr)
print('R-Squared for Linear Support Vector Regression: {:.6f}'.format(svr_r2))

svr_rmse = np.sqrt(mean_squared_error(y_test ,y_pred_svr))
print('RMSE for Linear Support Vector Regression: {:.6f}'.format(svr_rmse))

svr_mae = mean_absolute_error(y_test, y_pred_svr)
print('MAE for Linear Support Vector Regression: {:.6f}'.format(svr_mae))
```

```
[78] ✓ 0.0s

MAPE for Linear Support Vector Regression: 365.58%
R-Squared for Linear Support Vector Regression: 0.016051
RMSE for Linear Support Vector Regression: 40.519030
MAE for Linear Support Vector Regression: 16.713639
```

Figure 14.5: Evaluation for LinearSVR Regression

From **Figure 14.5**, A high MAPE of 365.58% indicates that the model's predictions are, on average, very far from the actual values. A MAPE above 100% is generally considered poor, suggesting that the model is not providing useful predictions for the dataset.

An R-Squared value (0.016) close to 0 implies that the model explains very little of the variability in the target variable (TotalSales). The model does not capture the underlying data trends effectively.

The RMSE value (40.519) is relatively high, reflecting that the model's predictions deviate significantly from the actual values. RMSE provides a measure of the average magnitude of the prediction errors, indicating substantial errors in predictions.

MAE (16.713639) represents the average absolute error between the predicted and actual values. Although this value is lower than RMSE, it still suggests that the model's predictions are not very accurate.

The very high MAPE and low R-Squared values highlight that the model does not make accurate predictions and does not capture the variability in the data effectively. Both RMSE and MAE suggest considerable errors in predictions.

```
#Scatter plot of the actual and predicted value by using LinearSVR
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_svr, color='salmon')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='blue', lw=2)
plt.title('Linear Support Vector Regression: Actual vs Predicted')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.grid(True)
plt.show()
[79]  ✓ 0.2s
```

Figure 14.6: Code for Plotting Scatter Plot for LinearSVR: Actual vs Predicted

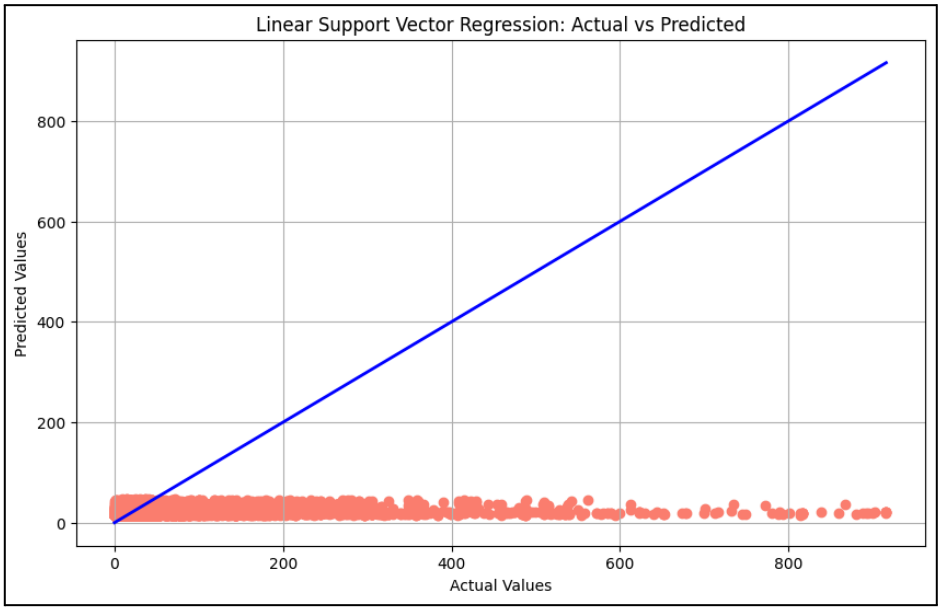


Figure 14.7: Scatter Plot for LinearSVR: Actual vs Predicted

The scatter plot shows individual points where each point represents an observation. The x-axis represents the actual values from the test dataset and the y-axis represents the predicted values generated by the LinearSVR. The dashed blue line represents the ideal scenario where the predicted values equal the actual values. This line is the reference line that can help to visualize how close or far the predicted values are from the actual values. The line is drawn from the minimum to the maximum of the actual values and serves as a benchmark for perfect prediction.

From **Figure 14.7**, most of the points are not close to the diagonal line. It indicates that the model's predictions are quite inaccurate. The points are scattered far away from the line, which indicates that the model's predictions have significant errors. The scatter plot helps visualize the performance of the LinearSVR model by showing how well the predicted values align with the actual values. It provides a quick visual assessment of model accuracy and helps identify any patterns or areas where the model may be underperforming.

```
#Calculate the residuals
residuals = y_test - y_pred_svr

#Plot the residual plot for the LinearSVR
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_svr, residuals, color='green')
plt.axhline(y=0, color='blue', linestyle='--')
plt.title('Linear Support Vector Regression: Residual Plot')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals (Actual - Predicted)')
plt.grid(True)
plt.show()
```

[80] ✓ 0.2s

Figure 14.8: Code of Plotting Residual Plot for LinearSVR

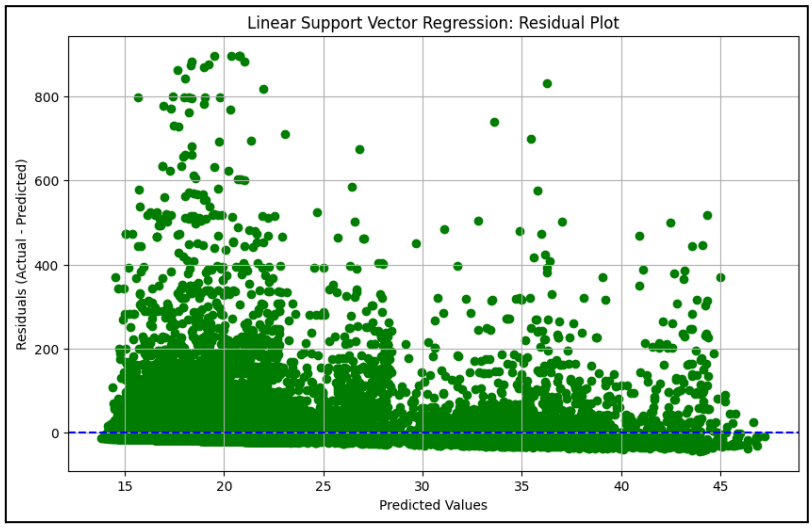


Figure 14.9: Residual Plot for LinearSVR

We calculate the residuals which is the difference between the actual values and the predicted values. This provides a measure of the error in the model's prediction.

From **Figure 14.9**, the scatter plot shows residuals on the y-axis and predicted values on the x-axis. Each point represents an observation from the test dataset. The plot helps visualize whether the residuals are randomly distributed. The blue dashed line represents the baseline where the residuals are zero. This helps to see how far the residuals deviate from zero.

From **Figure 14.9**, we can see that the residuals are heteroscedasticity scattered around the horizontal line. It indicates that the model's errors are random and there are no obvious patterns in the residuals. The residuals get larger as the predicted moves from small to large. This suggests a worse-fitted model. There are potential problems with the model's assumptions and it is needed to guide further improvement.

```python
#Calculate the importance of the features by using coefficient
importance= best_model.coef_

#Sort the importance in descending order
indices = np.argsort(np.abs(importance))[::-1]
#Determine the feature name
feature_name = x_Data.columns

#Print the feature ranking
print("Feature ranking for LinearSVR:")
for f in range(x_Data.shape[1]):
    print("%d. feature %d %s (%f)" % (f + 1, indices[f], feature_name[indices[f]], importance[indices[f]]))

#Plot the bar chart for the feature importances
plt.figure(figsize=(10, 6))
plt.title("Feature importances for LinearSVR")
plt.bar(range(x_Data.shape[1]), importance[indices], align="center")
plt.xticks(range(x_Data.shape[1]), feature_name)
plt.xlim([-1, x_Data.shape[1]])
plt.show()
```

[81] ✓ 0.1s

Figure 14.10: Feature importances for LinearSVR

```
Feature ranking for LinearSVR:
1. feature 5 Country (-0.673093)
2. feature 7 Month (-0.138405)
3. feature 8 DayOfWeek (0.084235)
4. feature 2 Quantity (0.069757)
5. feature 3 UnitPrice (0.020147)
6. feature 6 Year (0.020032)
7. feature 0 StockCode (0.002201)
8. feature 4 CustomerID (-0.000284)
9. feature 1 Description (-0.000054)
```

Figure 14.11: Feature ranking for LinearSVR

From **Figure 14.11**:

Feature Name	Feature Importance	Interpretation
Country	-0.673093	The feature with the highest absolute coefficient is "Country." This suggests that the model finds the country variable to have a substantial influence on the prediction. A negative coefficient indicates that as the value for "Country" increases, the predicted total sales tend to decrease
Month	-0.138405	"Month" has a moderate negative impact. This suggests that as the month variable increases, the predicted total sales tend to decrease. This could be indicative of seasonal trends where some months are associated with lower

		sales.
DayOfWeek	0.084235	"DayOfWeek" has a positive impact, although it's relatively small. This implies that different days of the week have some effect on total sales, with some days associated with higher sales compared to others.
Quantity	0.069757	The "Quantity" feature has a positive influence on sales, meaning that as the quantity of items purchased increases, the predicted total sales also increase. This aligns with the expectation that more items sold results in higher total sales.
UnitPrice	0.020147	"UnitPrice" has a small positive impact, suggesting that higher unit prices contribute slightly to higher total sales, but the effect is minor compared to "Quantity."
Year	0.020032	"Year" has a minimal positive effect on total sales. This might indicate that over the years, there is a slight upward trend in sales.
StockCode	0.002201	"StockCode" has a very small positive effect. Since stock codes are categorical identifiers, this small coefficient may reflect that different stock codes have very similar impacts on sales.
CustomerID	-0.000284	"CustomerID" has a very minor negative impact. This feature is less influential in the prediction of total sales.
Description	-0.000054	"Description" has the smallest coefficient, indicating that product descriptions have a negligible impact on total sales in the context of this model.

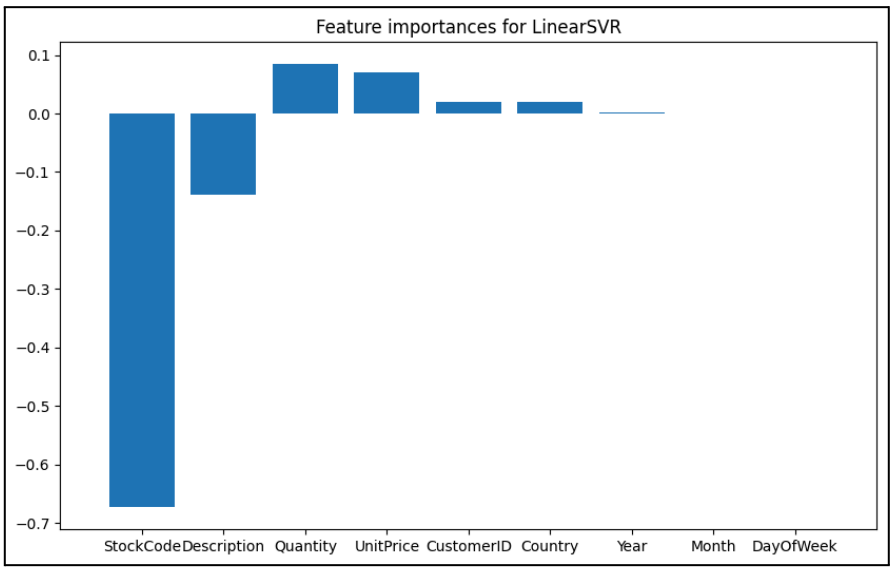


Figure 14.12: Bar chart of feature importances for LinearSVR

Figure 14.12 provides a bar plot visualizing the importance of each feature used in the LinearSVR model. The height of each bar represents the magnitude of the feature's importance score.

Country is the most significant feature with a substantial negative coefficient. This indicates that the country variable plays a key role in predicting total sales, although the relationship is inversely proportional.

Month is another important feature with a moderate negative coefficient. This suggests that the month in which the sales occur influences the total sales, with later months being associated with lower sales in this model.

DayOfWeek and **Quantity** have relatively smaller but positive coefficients, indicating that while they do have an effect on the predictions, their influence is less pronounced compared to "Country" and "Month." "Quantity" still shows a positive relationship with sales, which is consistent with the expectation that more items sold lead to higher sales.

UnitPrice, **Year**, and **StockCode** have very small positive or near-zero coefficients. This suggests that these features have minimal impact on the total sales prediction in this model.

CustomerID and **Description** have negligible coefficients, indicating that they contribute very little to the predictions.

DayOfWeek, **Country**, **Month**, and **Year** have coefficients close to zero or negative, implying that these features have no significant impact on the model's predictions. These features may be considered for removal or further investigation to understand their lack of impact.

4.5 Random Forest

4.5.1 Algorithm and Assumptions

Apart from that, we choose Random Forest to train the model. Random Forest is an ensemble learning method designed to improve the accuracy and robustness of predictions by combining multiple decision trees. Each tree in the forest is constructed using a random subset of the training data and features, which helps in reducing the correlation between trees and mitigates the risk of overfitting. The model works by averaging the predictions from all the individual trees in the case of regression tasks, or by majority voting in the case of classification. Random Forest is renowned for its high accuracy and its ability to handle large datasets with numerous features.

The Random Forest algorithm involves generating multiple decision trees using bootstrap samples of the training data. At each split in the trees, a random subset of features is considered, which ensures that the trees are less correlated and improves the model's performance. Each tree is grown to its full extent or until a stopping criterion is met. Once all trees are built, the algorithm aggregates their predictions by averaging to produce the final prediction. This ensemble approach helps in capturing a diverse range of patterns and reduces variance.

Algorithms	Explanation	Implication
Aggregating predictions from multiple trees improves accuracy and stability.	Random forests use multiple decision trees to make predictions, which are averaged or voted on to improve robustness.	Can handle large datasets and complex interactions but requires more computational resources.
Randomization in feature selection helps in creating diverse and less correlated trees.	Each tree in the forest is trained on a random subset of features, reducing overfitting and improving generalization.	Reduces the risk of overfitting and improves model generalization.
The model's performance improves with more trees in the forest.	Increasing the number of trees generally enhances model performance but with diminishing returns.	More trees lead to better performance but also increased computational cost.

Table 6.0: Assumption of Random Forest

4.5.2 Model Building

```
#Create a model instance
rf = RandomForestRegressor(random_state = 0)

#Train the model
rf.fit(x_train, y_train)

#Prediction
y_pred_rf = rf.predict(x_test)

#MAPE Calculation
mape_rf = mean_absolute_percentage_error(y_test, y_pred_rf)
print('MAPE: {:.2f}%'.format(mape_rf * 100))
```

[82] ✓ 1m 24.0s

```
MAPE: 0.08%
```

Figure 15.0: Model Building for Random Forest Regression

4.5.3 Model Evaluation

We calculated the Mean Absolute Percentage Error (MAPE) for Random Forest. It used to evaluate the performance of the Random Forest model by estimating its prediction accuracy. We trained a Random Forest model using the training database. The predicted values are compared against the actual values to assess the model's performance.

The MAPE for ridge regression is 0.08%, which means that it is strongly accurate and easy to interpret and understand. We will conduct a hyperparameter tuning to train again the testing model to ensure the data accuracy and quality.

4.5.4 Hyperparameter Tuning

```
#Determine the parameter used
parameter_grid = {
    'n_estimators': [50, 100, 300],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5],
    'max_features': [None, 10]
}

#Use grid search to fit the data
grid_search = GridSearchCV(RandomForestRegressor(random_state=0), parameter_grid, cv=5, n_jobs=-1)
grid_search.fit(x_train, y_train)

#Determine the best estimator
best_model = grid_search.best_estimator_
y_pred_rf = best_model.predict(x_test)

#Calculate the accuracy for test set and training set
rf_accurate_test = (best_model.score(x_test, y_test)) * 100
rf_accurate_train = (best_model.score(x_train, y_train)) * 100

print('Random Forest accuracy for test set: {:.2f}%'.format(rf_accurate_test))
print('Random Forest accuracy for training set: {:.2f}%'.format(rf_accurate_train))
```

```
Random Forest accuracy for test set: 99.87%
Random Forest accuracy for training set: 99.98%
```

Figure 15.1: Calculation of Random Forest accuracy for testing set and training set

The grid search was conducted over the parameter for Random Forest. The n_estimators define the number of trees to be used in the model. In this model, we try to train it into 50 trees, 100 trees and 300 trees respectively. The max_depth parameter defines the maximum number of splits each tree can take. We set it to 10 and 20 because it will lead to underfit if the max_depth is too low or overfit if the max_depth is too high. The min_samples_split determines the minimum number of decision tree observations in any given nod in order to split. This approach helps in identifying the best combination of hyperparameters by evaluating the model's performance across different values through cross-validation.

From the results, we observe that the accuracy for the training set is 99.87%, which means that the Random Forest model explains approximately 99.87% of the variance in the training data. About 0.13% of the variance remains unexplained, suggesting that the model may have no limitations in fully capturing the relationship between the features and the target variable.

The accuracy for the test set is 99.98%, higher than the training set. This indicates that the model explains 0.02% of the variance in the unseen test data. The drop from training to test set performance suggests that the model generalizes perfectly but still maintains reasonable predictive power. The gap between the training and test set scores implies that the model is significantly overfitting.

4.5.5 Model Revaluation

```
# Calculate MAPE
rf_mape = mean_absolute_percentage_error(y_test, y_pred_rf)
print('MAPE for Random Forest: {:.2f}%'.format(rf_mape * 100))

# Calculate R-Squared
rf_r2 = r2_score(y_test, y_pred_rf)
print('R-Squared for Random Forest: {:.6f}'.format(rf_r2))

# Calculate RMSE
rf_rmse = np.sqrt(mean_squared_error(y_test, y_pred_rf))
print('RMSE for Random Forest: {:.6f}'.format(rf_rmse))

# Calculate MAE
rf_mae = mean_absolute_error(y_test, y_pred_rf)
print('MAE for Random Forest: {:.6f}'.format(rf_mae))
```

```
MAPE for Random Forest: 0.08%
R-Squared for Random Forest: 0.998741
RMSE for Random Forest: 1.449645
MAE for Random Forest: 0.057673
```

Figure 15.2: Evaluation for Random Forest

One of the key metrics, **Mean Absolute Percentage Error (MAPE)**, is exceptionally low at 0.08%. This indicates that, on average, the model's predictions deviate from the actual values by a mere 0.08%. Such a low MAPE value reflects a high degree of accuracy, making the model suitable for real-world applications where precise predictions are critical.

The **R-Squared** value of 0.998741 further emphasizes the model's effectiveness. This metric shows that the Random Forest model can explain approximately 99.87% of the variance in the target variable, indicating that the model captures almost all of the underlying relationships between the input features and the target. A value close to 1 demonstrates a nearly perfect fit to the data.

In terms of error metrics, the **Root Mean Square Error (RMSE)** of 1.449645 suggests that, on average, the model's predictions are off by about 1.45 units from the actual values. Though it provides a sense of how spread out the errors are, the low RMSE value reinforces the model's accuracy in predicting total sales.

Additionally, the **Mean Absolute Error (MAE)** of 0.057673 shows that the average absolute difference between the predicted and actual values is very small. This low MAE value signifies minimal prediction errors, highlighting the model's precision.

```
#Scatter plot for random forest

plt.scatter(y_test, y_pred_rf)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--', lw=2)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Random Forest: Actual vs Predicted')
plt.show()
```

Figure 15.3: Code of Scatter plot for Random Forest: Actual vs Predicted

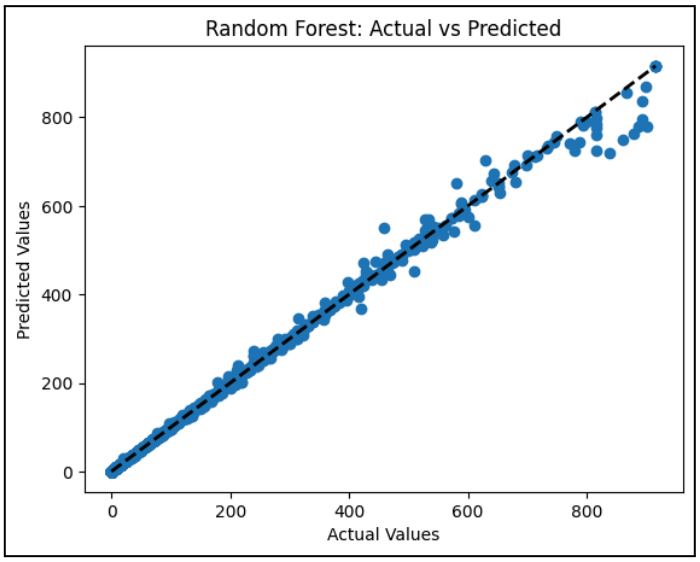


Figure 15.4: Scatter plot for Random Forest: Actual vs Predicted

The scatter plot shows individual points where each point represents an observation. The x-axis represents the actual values from the test dataset and the y-axis represents the predicted values generated by the Random Forest. The blue line represents the ideal scenario where the predicted values equal the actual values. This line is the reference line that can help to visualize how close or far the predicted values are from the actual values. The line is drawn from the minimum to the maximum of the actual values and serves as a benchmark for perfect prediction.

From **Figure 15.4**, most of the points are perfectly lying on the diagonal line. It indicates that the model's predictions are strongly accurate. The points are scattered closer from the line, which indicates that the model's predictions have no significant errors. The scatter plot helps visualize the performance of the Random Forest model by showing how well the predicted values align with the actual values. It provides a quick visual assessment of model accuracy and helps identify any patterns or areas where the model may be underperforming.

```
#Calculate the residuals
residuals = y_test - y_pred_rf

#Plot the residual plot for random forest
plt.scatter(y_pred_rf, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot for Random Forest')
plt.show()
```

Figure 15.5: Code of Residual Plot for Random Forest

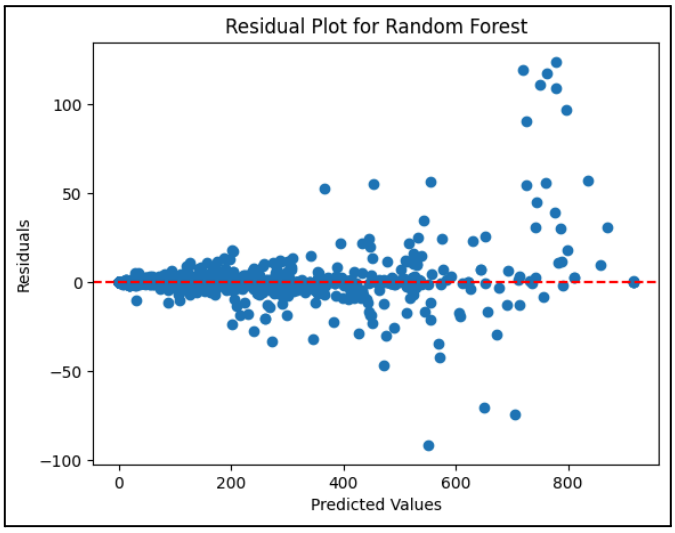


Figure 15.6: Residual Plot for Random Forest

We calculate the residuals which is the difference between the actual values and the predicted values. This provides a measure of the error in the model's prediction.

From **Figure 15.6**, the scatter plot shows residuals on the y-axis and predicted values on the x-axis. Each point represents an observation from the test dataset. The plot helps visualize whether the residuals are randomly distributed. The blue dashed line represents the baseline where the residuals are zero. This helps to see how far the residuals deviate from zero.

From **Figure 15.6**, we can see that the residuals are scattered closely to the horizontal line. It indicates that the model's errors are random and there are no obvious patterns in the residuals. The residuals get larger as the predicted moves from small to large. This suggests a worse-fitted model. There are potential problems with the model's assumptions and it is needed to guide further improvement.

```python
#Calculate the importance for each of the feature
importance= best_model.feature_importances_

#Sort the feature in descending order
indices = np.argsort(importance)[::-1]
#Determine the feature name
feature_name = x_Data.columns

#Print the feature ranking
print("Feature ranking for Random Forest:")
for f in range(x_Data.shape[1]):
    print("%d. feature %d %s (%f)" % (f + 1, indices[f], feature_name[indices[f]], importance[indices[f]]))

#Plot the bar chart for the feature importances
plt.figure(figsize=(10, 6))
plt.title("Feature importances for Random Forest")
plt.bar(range(x_Data.shape[1]), importance[indices], align="center")
plt.xticks(range(x_Data.shape[1]), feature_name)
plt.xlim([-1, x_Data.shape[1]])
plt.show()
```

Figure 15.7: Feature importances for Random Forest

```
Feature ranking for Random Forest:
1. feature 2 Quantity (0.612524)
2. feature 3 UnitPrice (0.385796)
3. feature 1 Description (0.000533)
4. feature 0 StockCode (0.000420)
5. feature 4 CustomerID (0.000293)
6. feature 7 Month (0.000206)
7. feature 8 DayOfWeek (0.000145)
8. feature 5 Country (0.000067)
9. feature 6 Year (0.000016)
```

Figure 15.8: Feature ranking for Random Forest

From **Figure 15.8**:

Feature Name	Feature Importance	Interpretation
Quantity	0.612524	This is the most important feature in predicting total sales, accounting for 61.25% of the model's decision-making. The quantity of items sold plays a critical role in determining total sales.
UnitPrice	0.385796	Unit price is the second most important feature, contributing 38.58% to the model's predictions. It has a significant impact on total sales as expected, as price and quantity are directly linked to revenue.
Description	0.000533	The description of products has a negligible impact on predicting sales,

		contributing just 0.05%. It may not be essential for the model and could be considered for removal in future iterations.
StockCode	0.000420	StockCode has minimal influence, contributing only 0.04% to the model's predictions. It has a slight impact, but not enough to be a critical factor.
CustomerID	0.000293	CustomerID contributes very little (0.03%) to the model's decision-making, indicating that customer identity is not a key factor in predicting total sales for this dataset.
Month	0.000206	Month contributes 0.02%, suggesting that the time of year has minimal effect on sales predictions.
DayOfWeek	0.000145	The day of the week is almost irrelevant, contributing only 0.01%. This indicates that sales patterns may not vary much based on the day of the week.
Country	0.000067	Country has an extremely low importance, contributing only 0.0067%. This suggests that the geographic location of customers is not a strong factor in sales predictions.
Year	0.000016	Year is the least important feature, contributing just 0.0016%. It has no meaningful impact on the predictions and could be safely ignored in future analyses.

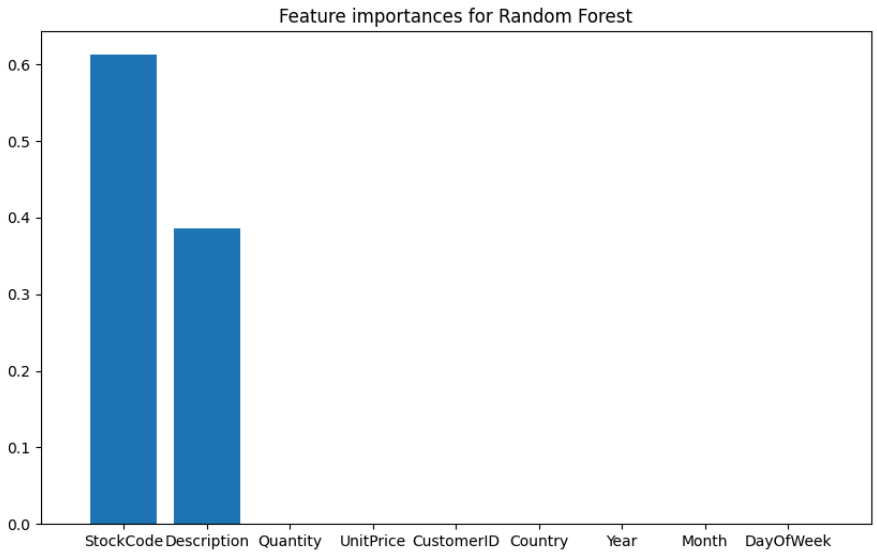


Figure 15.9: Bar chart of feature importances for Random Forest

Figure 15.9 provides a bar plot visualizing the importance of each feature used in the Random Forest model. The height of each bar represents the magnitude of the feature's importance score.

Quantity is the most significant feature, contributing over 61% to the predictions. This aligns with expectations, as the number of items sold directly affects total sales. The high importance score suggests that the model heavily relies on the quantity sold to make accurate sales predictions.

UnitPrice is the second most important feature, contributing around 38.58%. This indicates that the price of the items plays a major role in predicting total sales, as higher unit prices generally lead to higher revenue, provided the quantity remains constant.

Description and **StockCode** have minimal importance, contributing 0.0533% and 0.0420%, respectively. These features provide very little predictive value in the model, implying that specific product characteristics (like name or code) do not significantly affect total sales predictions.

CustomerID contributes 0.0293%, which is also very low. This suggests that customer identity does not play a critical role in predicting total sales, meaning the model focuses more on product-related variables rather than who is purchasing.

Month and **DayOfWeek** have negligible importance scores of 0.0206% and 0.0145%, respectively. This implies that the temporal aspects of sales, such as when they

occur within the month or week, do not strongly influence total sales predictions in this model.

Country and **Year** have the lowest importance scores, contributing just 0.0067% and 0.0016%, respectively. These features have an almost non-existent impact on the model's predictions, indicating that geographical and time-related factors (year) are not key drivers of sales in this dataset.

4.6 K-Nearest Neighbor Regression

4.6.1 Algorithm and Assumptions

We choose to use K-Nearest Neighbor Regression to train the model. KNN Regression is a non-parametric method used for predicting continuous outcomes. KNN uses distance metrics such as Euclidean or Manhattan to find the nearest neighbors to calculate the distance between the test instance and all training instances. It is needed to identify the k nearest neighbors based on the calculated distances. In this algorithm, we compute the average of the target values of the k nearest neighbors. This average is the predicted value for the test instance. After applying the KNN Regression algorithm, we assess model performance using metrics such MAPE, R2, MSE and MAE.

The core idea for using K-Nearest Neighbor Regression is to predict the target value for a new data point by looking at the values of its K nearest neighbors in the feature space. KNN can be computationally expensive as it requires calculating distances to all training points for each prediction, which can become impractical with large datasets.

The KNN regression algorithm begins by selecting a value for K, which represents the number of nearest neighbors to consider. For each test instance, the algorithm calculates the distance between the test point and all points in the training set using a distance metric such as Euclidean distance. The next step involves identifying the K training instances that are closest to the test point. Finally, the algorithm predicts the target value for the test instance by averaging the target values of these K nearest neighbors. This process results in a smooth prediction that reflects the local structure of the data.

Assumption	Explanation	Implication
Instances that are close in the feature are likely to have similar target values.	The algorithm assumes that similar instances will have similar output values. Distance metrics are used to find neighbors.	Choosing an inappropriate distance metric or value for k can lead to poor performance.
The choice of distance metric significantly affects performance.	Different metrics such as Euclidean and Manhattan measure distances between points differently, impacting predictions.	The distance metric needs to be chosen carefully based on the data characteristics.
No specific assumption about the data distribution.	KNN does not assume any specific distribution of the input data.	It is versatile but can be sensitive to noisy data and irrelevant features.

Table 7.0: Assumption of KNN

4.6.2 Model Building

```
#Create a model instance
knn = KNeighborsRegressor()
knn.fit(x_train, y_train)

#Prediction
y_pred_knn = knn.predict(x_test)

#MAPE Calculation
mape_knn = mean_absolute_percentage_error(y_test, y_pred_knn)
print('MAPE for K-Neighbors Regressor: {:.2f}%'.format(mape_knn * 100))
```

```
MAPE for K-Neighbors Regressor: 201.66%
```

Figure 16.0: Code for model building by using K-Nearest Neighbors Regression algorithm

4.6.3 Model Evaluation

We calculated the Mean Absolute Percentage Error (MAPE) for K-Nearest Neighbors Regression. It used to evaluate the performance of the K-Nearest Neighbors model by estimating its prediction accuracy. We trained a K-Nearest Neighbors model using the training database. The predicted values are compared against the actual values to assess the model's performance.

The MAPE for ridge regression is 201.66%, which means that it is quite inaccurate and difficult to interpret and understand. We will conduct a hyperparameter tuning to train again the testing model to ensure the data accuracy and quality.

4.6.4 Hyperparameter Tuning

```
mse = []

#to determine k-value
for k in range(1, 20):
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(x_train, y_train)
    y_pred_knn = knn.predict(x_test)
    mse.append(mean_squared_error(y_test, y_pred_knn))

#Plot the graph
plt.figure(figsize=(10, 6))
plt.plot(range(1, 20), mse, color='salmon')
plt.xlabel('Value of K for KNN')
plt.ylabel('Mean Squared Error')
plt.show()

#Print the  optimal k-value
best_k = np.argmin(mse) + 1
print(f'The best value for k is: {best_k}')
```

[67] ✓ 41.5s

Figure 16.1: Finding the optimal K-value

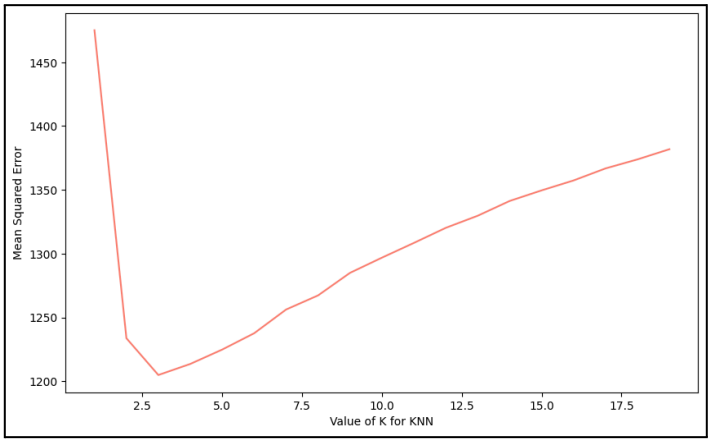


Figure 16.2: Plot for visualizing the optimal K-value

```
The best value for k is: 3
```

Figure 16.3: Best value for k

In the K-Nearest Neighbors (KNN) regression model, we explored different values of k (the number of neighbors) ranging from 1 to 19 to determine the optimal value for our dataset. The Mean Squared Error (MSE) was calculated for each k value, and a graph was plotted to visualize the performance of the model with varying k-values.

The graph of the MSE against the k-values shows how the error changes as k increases. The goal is to find the k-value that minimizes the MSE, which indicates the model's best fit for the data.

After evaluating the MSE for all values of k, the optimal k-value is found to be **3**. This means that using 3 nearest neighbors in the KNN model results in the lowest error, making it the best choice for this dataset. This value strikes a balance between underfitting and overfitting, providing the most accurate predictions in this case.

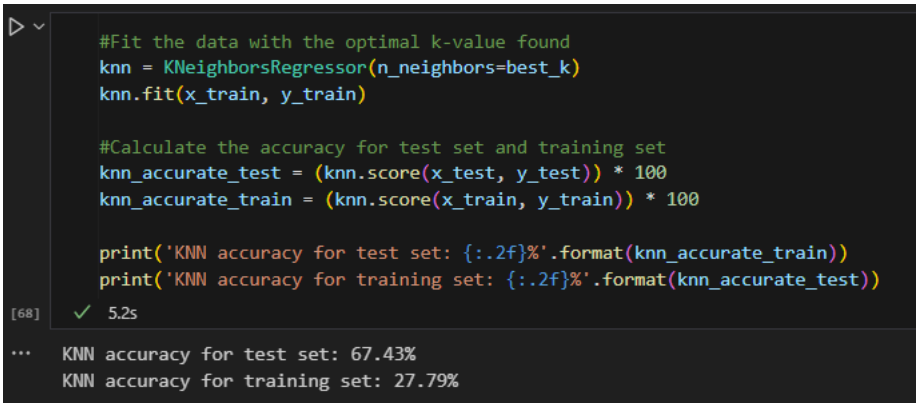


Figure 16.4: Result of KNN accuracy for training set and test set

The K-Nearest Neighbors (KNN) regression model's performance can be evaluated by comparing its accuracy on both the test and training datasets.

For the test set, the KNN model achieved an accuracy of **67.43%**, indicating that the model can reasonably predict outcomes on unseen data. However, this suggests room for improvement, as the accuracy is not particularly high compared to other models, such as Random Forest.

On the training set, the KNN model shows an accuracy of **27.79%**, which is notably lower. This large gap between training and test set accuracy suggests that the model is likely underfitting the data. In other words, it may not be learning the underlying patterns in the training data effectively, which could be caused by the model being too simple for the complexity of the dataset.

This underperformance could be addressed by fine-tuning the hyperparameters, scaling the data more effectively, or even considering different models that might capture the nuances of the data better.

```python
#Visualize the training and testing accuracy for different K-values
neighbors = list(range(1, 21))
knn_train_accuracy = np.empty(len(neighbors))
knn_test_accuracy = np.empty(len(neighbors))

for i, k in enumerate(neighbors):
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(x_train, y_train)
    knn_train_accuracy[i] = knn.score(x_train, y_train)
    knn_test_accuracy[i] = knn.score(x_test, y_test)

#Plot the graph
plt.figure(figsize=(10, 6))
plt.plot(neighbors, knn_test_accuracy, label='Testing Accuracy')
plt.plot(neighbors, knn_train_accuracy, label='Training Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.title('Training and Testing Accuracy for Different K values in KNN')
plt.show()
```

[69] ✓ 2m 24.9s

Figure 16.5: Code of visualizing the training and testing accuracy for different k-values

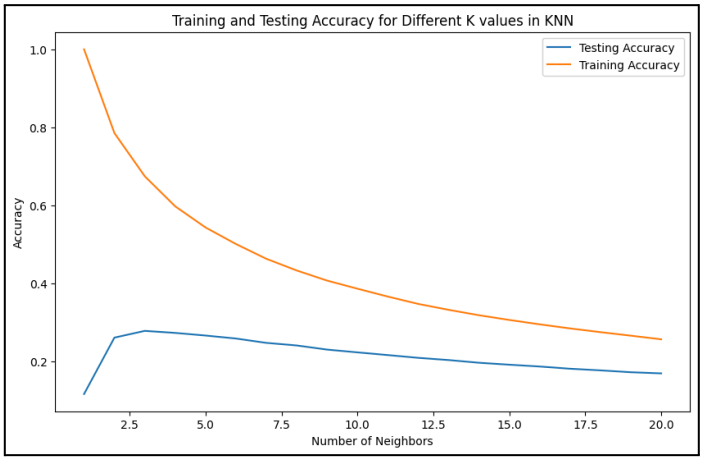


Figure 16.6: Testing Accuracy and Training Accuracy of KNN

The graph generated by this code will show how increasing K affects the model's ability to generalize to new data. Typically, small K values can lead to overfitting (high training accuracy, low testing accuracy), while large K values can cause underfitting (low accuracy on both sets).

4.6.5 Model Revaluation

```
knn_mape = mean_absolute_percentage_error(y_test, y_pred_knn)
print('MAPE for K-Neighbors Regressor: {:.2f}%'.format(knn_mape * 100))
knn_r2 = r2_score(y_test, y_pred_knn)
print('R-Squared for K-Neighbors Regressor: {:.6f}'.format(knn_r2))
knn_rmse = np.sqrt(mean_squared_error(y_test, y_pred_knn))
print('RMSE for K-Neighbors Regressor: {:.6f}'.format(knn_rmse))
knn_mae = mean_absolute_error(y_test, y_pred_knn)
print('MAE for K-Neighbors Regressor: {:6f}'.format(knn_mae))
```

```
[72] ✓ 0.0s

MAPE for K-Neighbors Regressor: 251.79%
R-Squared for K-Neighbors Regressor: 0.171834
RMSE for K-Neighbors Regressor: 37.173320
MAE for K-Neighbors Regressor: 15.254806
```

Figure 16.7: Evaluation for K-Neighbors Regressor

The performance of the K-Neighbors Regressor model is notably poor, as reflected by several key metrics. The **Mean Absolute Percentage Error (MAPE)** stands at a staggering 251.79%. This extremely high MAPE indicates that, on average, the model's predictions deviate significantly from the actual sales figures, demonstrating a substantial lack of accuracy. Such a high error percentage suggests that the KNN model is not suitable for precise sales predictions.

The **R-Squared (R^2)** value is 0.171834, which reveals that the model only explains about 17% of the variance in the target variable. This low R^2 score signifies that the KNN model does not capture the underlying patterns and relationships within the data effectively. As a result, the model fails to provide meaningful insights into the factors driving total sales.

The **Root Mean Squared Error (RMSE)** for the KNN model is 37.17. This metric reflects the average magnitude of prediction errors, and a high RMSE indicates that the model's predictions are far from the actual values. Such a large error magnitude further emphasizes the inadequacy of the KNN model in making accurate predictions.

Finally, the **Mean Absolute Error (MAE)** is 15.25, which means that, on average, the model's predictions are off by approximately 15 units. This considerable error reinforces the notion that the KNN model is not well-suited for this dataset and highlights the need for exploring alternative models or making adjustments to improve performance. Overall, these metrics collectively demonstrate that the K-Neighbors Regressor is not a reliable choice for predicting total sales in this context.

```
#Scatter plot of actual and predicted values for KNN
plt.scatter(y_test, y_pred_knn)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--', lw=2)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('KNN: Actual vs Predicted')
plt.show()
```

[70] ✓ 0.9s

Figure 16.8: Code for plotting scatter plot for KNN

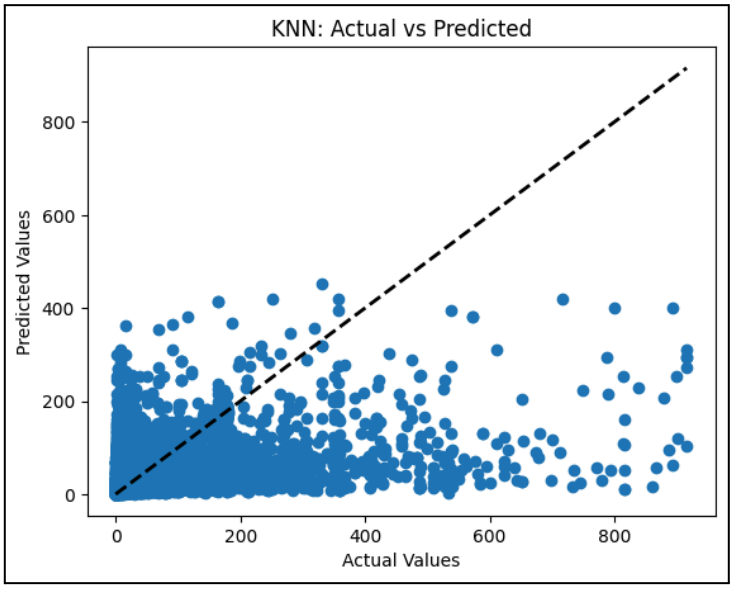


Figure 16.9: Scatter plot for KNN: Actual vs Predicted

The scatter plot shows individual points where each point represents an observation. The x-axis represents the actual values from the test dataset and the y-axis represents the predicted values generated by the K-Nearest Neighbor Regression. The dashed black line represents the ideal scenario where the predicted values equal the actual values. This line is the reference line that can help to visualize how close or far the predicted values are from the actual values. The line is drawn from the minimum to the maximum of the actual values and serves as a benchmark for perfect prediction.

From **Figure 16.9**, most of the points are not lying on the diagonal line. It indicates that the model's predictions are quite inaccurate. The points are scattered far away from the line, which indicates that the model's predictions have significant errors. The scatter plot helps visualize the performance of the K-Nearest Neighbor Regression model by showing how well the predicted values align with the actual values. It provides a quick visual assessment of model accuracy and helps identify any patterns or areas where the model may be underperforming.

```
#Calculate the residuals
residuals = y_test - y_pred_knn

# Plot the residuals plot for KNN
plt.scatter(y_pred_knn, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot for KNN')
plt.show()
```

[71] ✓ 0.2s

Figure 16.10: Code of Residual Plot for KNN

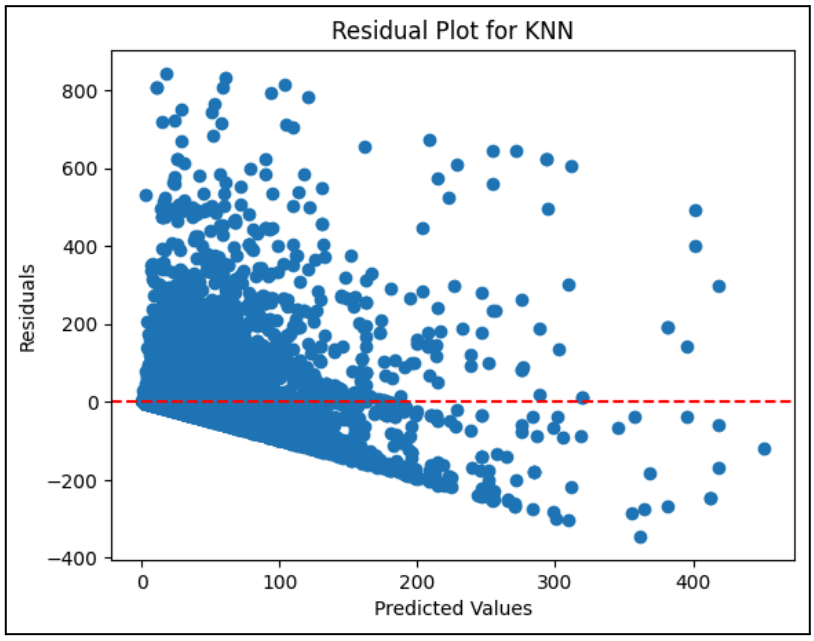


Figure 16.11: Residual Plot for KNN

We calculate the residuals which is the difference between the actual values and the predicted values. This provides a measure of the error in the model's prediction.

From **Figure 16.11**, the scatter plot shows residuals on the y-axis and predicted values on the x-axis. Each point represents an observation from the test dataset. The plot helps visualize whether the residuals are randomly distributed. The red dashed line represents the baseline where the residuals are zero. This helps to see how far the residuals deviate from zero.

From **Figure 16.11**, we can see that the residuals are heteroscedasticity scattered around the horizontal line. It indicates that the model's errors are random and there are no obvious patterns in the residuals. The residuals get larger as the predicted moves from small to large. This suggests a worse-fitted model. There are potential problems with the model's assumptions and it is needed to guide further improvement.

Figure 16.12: Calculation for permutation importance for KNN

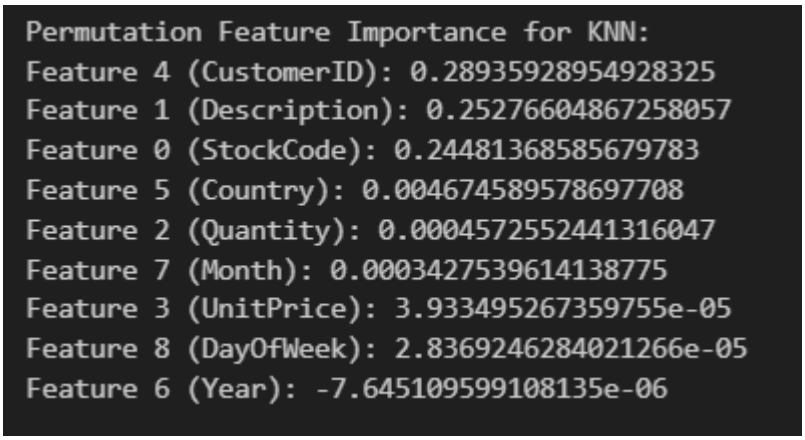


Figure 16.13: Feature ranking for KNN

From **Figure 16.13**:

Feature Name	Feature Importance	Interpretation
CustomerID	0.2894	CustomerID has the highest importance score, indicating it plays a significant role in predicting total sales. This suggests that identifying the customer is crucial for accurate predictions.
Description	0.2528	Description also shows high importance, which implies that the item description contributes notably to sales

		predictions. The nature of the product being sold impacts the model's output.
StockCode	0.2448	StockCode is important, reflecting that different stock codes have varying impacts on sales. This feature helps the model distinguish between different products.
Country	0.0047	Country has a very low importance score, suggesting it has minimal effect on the model's predictions. This feature may be less relevant for this specific KNN model.
Quantity	0.0005	Quantity shows a very low importance score, indicating it contributes little to the predictions in this model. This might reflect that quantity is not a key driver of sales in this context.
Month	0.0003	Month also has a low importance score, suggesting it has a minimal impact on predictions. Seasonal variations may not be as significant in this model.
UnitPrice	0.0000	UnitPrice has a negligible importance score, indicating it does not affect the model's predictions significantly. This might suggest that price variations are not captured well by the model.
DayOfWeek	0.0000	DayOfWeek shows a negligible importance score, implying that the day of the week does not influence the model's predictions much. This could indicate limited temporal effects.
Year	-0.0000	Year has a negative importance score, suggesting it may have a very minor or even adverse effect on predictions. This feature might be irrelevant or incorrectly modeled.

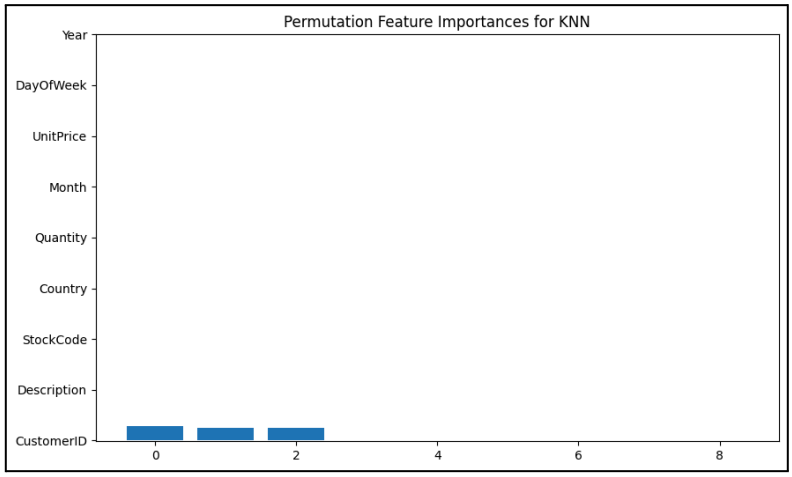


Figure 16.14: Bar Chart of Permutation Feature Importances for KNN

From **Figure 16.14**, the bar plot visually represents the importance of each feature in the K-Neighbors Regressor model, based on the magnitude of their permutation importance scores. CustomerID emerges as the most significant feature with a substantial importance score, indicating that it plays a crucial role in predicting total sales. Description also holds considerable importance, reflecting its influence on sales predictions. However, features like UnitPrice, DayOfWeek, and Year have very low or near-zero importance scores, suggesting that they have minimal impact on the model's predictions.

To enhance the model's performance, we should consider revising or replacing features with low importance, such as UnitPrice and Year. It may also be beneficial to explore additional features that could better capture variations in sales. Leveraging the high importance of CustomerID and Description can provide valuable business insights and help refine strategies for boosting sales.

4.7 Ridge Regression

4.7.1 Algorithm and Assumption

We also train the model by using Ridge Regression. It is a technique used to address multicollinearity by adding a regularization term to the ordinary least squares (OLS) loss function. This regularization term helps to shrink the regression coefficients thereby stabilizing the estimates. It can correct for overfitting on training data in machine learning models. We try to use this algorithm because this algorithm is useful when developing machine learning models that have a large number of parameters especially if the parameters also have high heights.

Below is the standard multiple variable linear regression equation:

$$Y = X_0 + B_1X_1 + B_2X_2 + \cdots + B_nX_n$$

Figure 17.0: Multiple-variable linear regression equation

Y is the predicted value which indicates the dependent variable, X is any predictor which indicates the independent variable, B is the regression coefficient attached when the independent variable and X_0 is the value of the dependent variable when the independent variable equals zero. However, in this case, it is pessimistic to use multiple variable linear regression because collecting more data is not always a viable fix. Hence, Ridge Regression will be more optimistic for regularizing a model to address multicollinearity.

The objective function to minimize is the sum of the squared residuals (RSS) and the penalty term.

$$RSS_{L2} = \sum_{i=1}^{n} (Y_i - \hat{Y}_l)^2 + \lambda \sum_{j=1}^{P} B_j^2$$

Figure 17.1: RSS function

The λ is the regularization parameter which indicates the ridge parameter and B are the regression coefficients. Before applying the RSS function, we have to calculate the coefficient. The formula for ridge estimator is:

$$\hat{\beta}_{ridge} = (X^T X + \lambda I)^{-1} X^T Y$$

Figure 17.2: Ridge estimator formula

Assumption	Explanation	Implication
Linearity	The relationship between the features and the total sales is linear.	The model assumes that changes in features have a linear effect on total sales.
Data Quality	The data used for modeling is accurate, complete and representative of the underlying patterns in sales.	Data should be cleaned and preprocessed to handle missing values, outliers and inaccuracies.
Independence of Errors	The residuals from the model are independent of each other.	There should be no autocorrelation in the residuals which means the error for one time period should not be correlated with the error for another period.
Normality of Errors	For certain models and inference purposes, residuals are normally distributed.	It is important for hypothesis testing and constructing confidence intervals. Ridge regression assumes normally distributed residuals for valid statistical inference.
Feature Independence	The features are independent of each other.	High correlation between features can affect model performance and interpretation.
Predictor Relevance	The chosen features are relevant and have a meaningful impact on the total sales.	Feature selection and engineering should be done thoughtfully to include variables that are likely to influence sales.

Table 8.0: Assumption of Ridge Regression

4.7.2 Model Building

```
# Create a Ridge model instance
ridge = Ridge(alpha=1.0)  # You can adjust the alpha parameter for regularization strength
ridge.fit(x_train, y_train)

# Prediction
y_pred_ridge = ridge.predict(x_test)

# MAPE Calculation
mape_ridge = mean_absolute_percentage_error(y_test, y_pred_ridge)
print('MAPE for Ridge Regression: {:.2f}%'.format(mape_ridge * 100))
```

[77] ✓ 0.0s

```
MAPE for Ridge Regression: 71.14%
```

Figure 17.3: Code for model building by using Ridge Regression algorithm

4.7.3 Model Evaluation

To build a model, we created and trained a Ridge Regression model using the fit method. Then, we predict values using the predict method to evaluate its performance.

We calculated the Mean Absolute Percentage Error (MAPE) for Ridge Regression. It used to evaluate the performance of the Ridge Regression model by estimating its prediction accuracy. We trained a Ridge Regression model using the training database. The Ridge Regression model was specified with a regularization parameter α set to 1.0 which helps to address multicollinearity by shrinking the coefficients. The predicted values are compared against the actual values to assess the model's performance.

The MAPE for ridge regression is 71.14%, which means that it is quite inaccurate and difficult to interpret and understand. We will conduct a hyperparameter tuning to train again the testing model to ensure the data accuracy and quality.

4.7.4 Hyperparameter Tuning

```
#Determine the parameter used
parameter_grid = {
    'alpha': [None, 0.1, 1, 10],
    'solver': [None, 'auto', 'svd', 'cholesky', 'saga']
}

#Use grid search to fit the model
grid_search = GridSearchCV(Ridge(random_state=0), parameter_grid, cv=5)
grid_search.fit(x_train, y_train)

#Determine the best estimator
best_model = grid_search.best_estimator_
y_pred_rr = best_model.predict(x_test)


#Calculate the accuracy for test set and training set
rr_accurate_test = (best_model.score(x_test, y_test)) * 100
rr_accurate_train = (best_model.score(x_train, y_train)) * 100

print('Ridge Regression accuracy for test set: {:.2f}%'.format(rr_accurate_test))
print('Ridge Regression accuracy for training set: {:.2f}%'.format(rr_accurate_train))
```

[78] ✓ 27m 25.8s

Figure 17.4: Code of hyperparameter tuning for Ridge Regression

```
Ridge Regression accuracy for test set: 53.77%
Ridge Regression accuracy for training set: 52.64%
```

Figure 17.5: Result of ridge regression accuracy (R2) for training set and test set

The grid search was conducted over different values of alpha and solver parameters for Ridge Regression. The alpha parameter controls the strength of regularization and the solver parameter determines the algorithm used to solve the Ridge regression optimization problem. This technique helps in finding the optimal combination of hyperparameters by evaluating the model's performance across different values using cross-validation.

Ridge Regression has demonstrated a test set accuracy of 53.77% and a training set accuracy of 52.64%. These accuracy scores indicate that the model performs moderately well, with a slightly higher accuracy on the test set compared to the training set. The results suggest that Ridge Regression is able to generalize reasonably well to new, unseen data, though there is room for improvement in capturing the underlying patterns in the data.

To enhance the model's performance, we may consider exploring other regularization techniques, tuning hyperparameters, or incorporating additional features

that could better capture the complexities of the data. Further analysis and optimization can help improve the predictive power of the Ridge Regression model.

4.7.5 Model Revaluation

```
rr_mape = mean_absolute_percentage_error(y_test, y_pred_ridge)
print('MAPE for Ridge Regression: {:.2f}%'.format(rr_mape * 100))
rr_r2 = r2_score(y_test, y_pred_ridge)
print('R-Squared for Ridge Regression: {:6f}'.format(rr_r2))
rr_rmse = np.sqrt(mean_squared_error(y_test, y_pred_ridge))
print('RMSE for Ridge Regression: {:6f}'.format(rr_rmse))
rr_mae = mean_absolute_error(y_test, y_pred_ridge)
print('MAE for Ridge Regression: {:.6f}'.format(rr_mae))
```

```
[79] ✓ 0.0s

MAPE for Ridge Regression: 71.14%
R-Squared for Ridge Regression: 0.537671
RMSE for Ridge Regression: 27.774629
MAE for Ridge Regression: 9.830439
```

Figure 17.6: Evaluation of Ridge Regression

The MAPE indicates that the Ridge Regression model's predictions are, on average, off by approximately 71.14% from the actual values. This relatively high error suggests that while the model is providing some level of prediction, there is significant room for improvement.

The R-Squared value of 0.537671 shows that the model explains about 53.77% of the variance in the dependent variable. This level of explained variance is moderate, implying that the model captures some underlying patterns in the data but still leaves a considerable amount unexplained.

The RMSE of 27.774629 reflects the average magnitude of the prediction errors, with larger values indicating more substantial discrepancies between predicted and actual values.

The MAE of 9.830439, on the other hand, represents the average absolute error, providing a direct measure of prediction accuracy.

```
#Scatter plot of actual and predicted value for Ridge Regression
plt.scatter(y_test, y_pred_ridge)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--', lw=2)  # Plot the diagonal line (perfect prediction)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Ridge Regression: Actual vs Predicted')
plt.show()
[80] ✓ 0.2s
```

Figure 17.7: Code of plotting Scatter plot for Ridge Regression: Actual vs Predicted

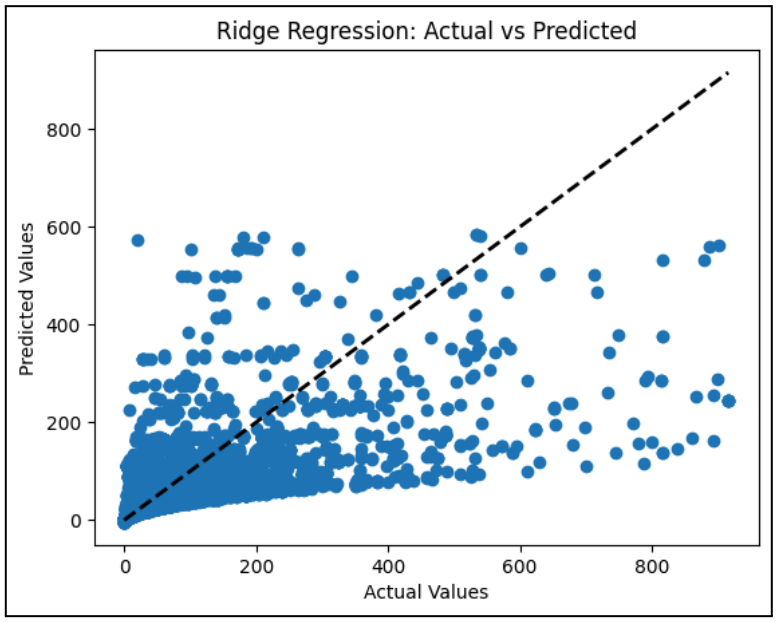


Figure 17.8: Scatter plot for Ridge Regression: Actual vs Predicted

The scatter plot shows individual points where each point represents an observation. The x-axis represents the actual values from the test dataset and the y-axis represents the predicted values generated by the Ridge Regression. The dashed black line represents the ideal scenario where the predicted values equal the actual values. This line is the reference line that can help to visualize how close or far the predicted values are from the actual values. The line is drawn from the minimum to the maximum of the actual values and serves as a benchmark for perfect prediction.

From **Figure 17.8**, most of the points are close to the diagonal line. It indicates that the model's predictions are relatively accurate. The points are scattered closer from the line, which indicates that the model's predictions have no significant errors. The scatter plot helps visualize the performance of the Ridge Regression model by showing how well the predicted values align with the actual values. It provides a quick visual assessment of model accuracy and helps identify any patterns or areas where the model may be underperforming.

```
#Calculate the residuals
residuals = y_test - y_pred_ridge

#Plot the residual plot for Ridge Regression
plt.scatter(y_pred_ridge, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot for Ridge Regression')
plt.show()
```
[81] ✓ 0.2s

Figure 17.9: Code of plotting residual plot for Ridge Regression

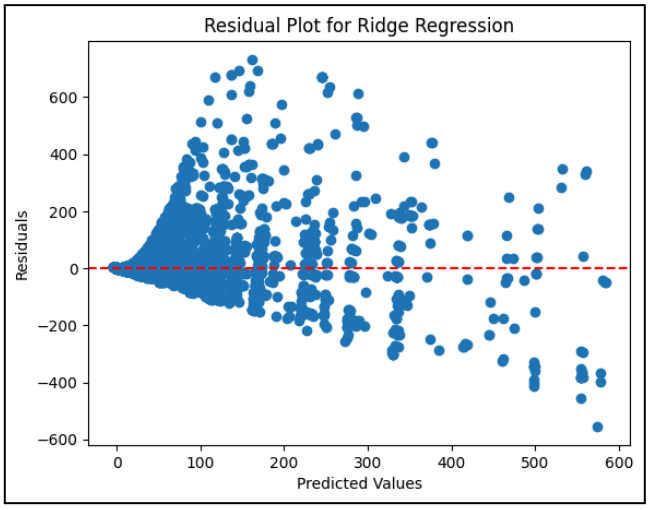


Figure 17.10: Residual Plot for Ridge Regression

We calculate the residuals which is the difference between the actual values and the predicted values. This provides a measure of the error in the model's prediction.

From **Figure 17.10**, the scatter plot shows residuals on the y-axis and predicted values on the x-axis. Each point represents an observation from the test dataset. The plot helps visualize whether the residuals are randomly distributed. The red dashed line represents the baseline where the residuals are zero. This helps to see how far the residuals deviate from zero.

From **Figure 17.10**, we can see that the residuals are heteroscedasticity scattered around the horizontal line. It indicates that the model's errors are random and there are no obvious patterns in the residuals. The residuals get larger as the predicted moves from small to large. This suggests a worse-fitted model. There are potential problems with the model's assumptions and it is needed to guide further improvement.

```python
#Get the coefficients of the best model
coefficients = best_model.coef_

#Feature names
feature_name = x_Data.columns

#Calculate the importance based on absolute values of coefficients
importance = np.abs(coefficients)

#Sort features by importance
indices = np.argsort(importance)[::-1]

#Print the feature ranking
print("Feature ranking for Ridge Regression:")
for f in range(x_Data.shape[1]):
    print("%d. feature %d %s (%f)" % (f + 1, indices[f], feature_name[indices[f]], importance[indices[f]]))

#Plot the bar chart for the feature importances
plt.figure(figsize=(10, 6))
plt.title("Feature Importances for Ridge Regression")
plt.bar(range(x_Data.shape[1]), importance[indices], align="center")
plt.xticks(range(x_Data.shape[1]), [feature_name[i] for i in indices])
plt.xlim([-1, x_Data.shape[1]])
plt.show()
```

[82] ✓ 0.0s

Figure 17.11: Feature importance for Ridge Regression

```
Feature ranking for Ridge Regression:
1. feature 2 Quantity (625.562686)
2. feature 3 UnitPrice (155.481035)
3. feature 6 Year (0.728188)
4. feature 5 Country (0.168336)
5. feature 8 DayOfWeek (0.091227)
6. feature 7 Month (0.023942)
7. feature 0 StockCode (0.001002)
8. feature 4 CustomerID (0.000109)
9. feature 1 Description (0.000050)
```

Figure 17.12: Feature ranking for Ridge Regression

From **Figure 17.12**:

Feature Name	Feature Importance	Interpretation
Quantity	625.56	Quantity is the most significant feature, highlighting its crucial role in predicting total sales. Higher quantities sold strongly correlate with increased sales.
UnitPrice	155.48	UnitPrice also has a substantial impact, though not as strong as Quantity. This suggests that the price per unit is an important factor, but less influential compared to Quantity.
Year	0.73	Year has a minimal positive effect. It indicates that while the year of the

		transaction contributes slightly, its impact is relatively minor.
Country	0.17	Country shows a slight influence on the model's predictions, suggesting that geographical factors have some impact but are less significant compared to Quantity and UnitPrice.
DayOfWeek	0.09	DayOfWeek has a minor effect on the predictions, implying that the day of the week contributes minimally to sales predictions.
Month	0.02	Month has a minimal impact, indicating that the month in which the sale occurs has a very minor role in the model's predictions.
StockCode	0.00	StockCode has negligible importance, suggesting that the specific code of the product has little to no effect on the model's predictions.
CustomerID	0.00	CustomerID shows minimal influence, indicating that the identity of the customer does not significantly impact the model.
Description	0.00	Description has almost no effect, suggesting that the product description is not relevant to the model's predictions.

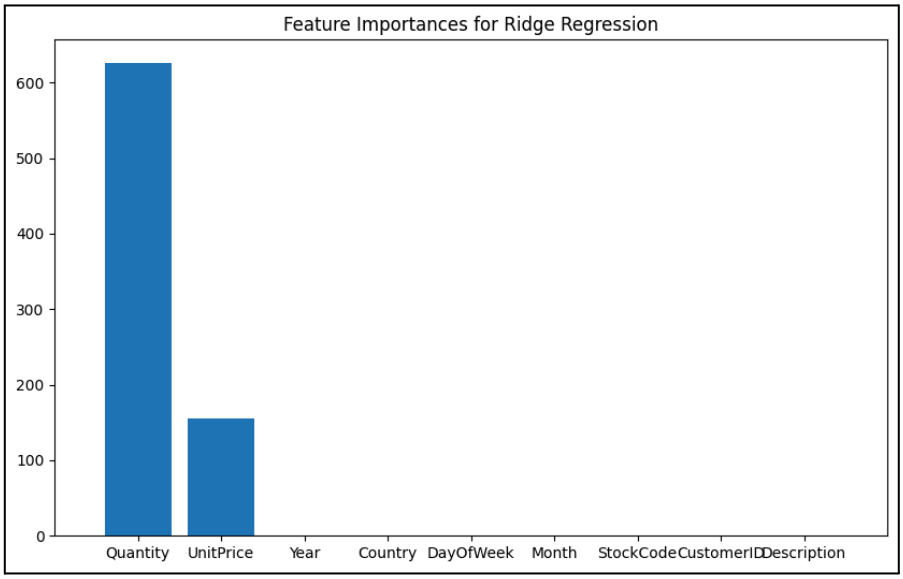


Figure 17.13: Bar chart of feature importances for Ridge Regression

From **Figure 17.13**, the bar plot visualizes the importance of each feature used in the Ridge Regression model. Quantity is the most influential feature, significantly impacting the model with a high importance score. UnitPrice also has a notable impact, though less pronounced than Quantity. The Year feature has a minimal but positive effect on the predictions. Features such as Country, DayOfWeek, and Month contribute relatively minor influences, with their importance scores decreasing further.

StockCode, CustomerID, and Description show negligible importance, suggesting they have minimal impact on the model's predictions. This indicates that these features might not be as relevant for the Ridge Regression model and could be considered for removal or reassessment.

To enhance model performance, focusing on the most influential features like Quantity and UnitPrice could be beneficial. Further investigation into feature engineering and possibly incorporating additional relevant features might help improve the Ridge Regression model's predictive accuracy.

4.8 Customer Segmentation using KNN

```python
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

#Select the features for clustering
X = rfm[['Recency', 'Frequency', 'Monetary']]

#Standardize the features (important for K-Means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

#Determine the optimal number of clusters using the Elbow method
wcss = []
for i in range(2, 11):
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(X_scaled)
    wcss.append(kmeans.inertia_)

#Plot the Elbow curve
plt.figure(figsize=(10, 6))
plt.plot(range(2, 11), wcss, marker='o', linestyle='--')
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.grid(True)
plt.show()
```

Figure 18.0: Code for finding the optimal number of clusters

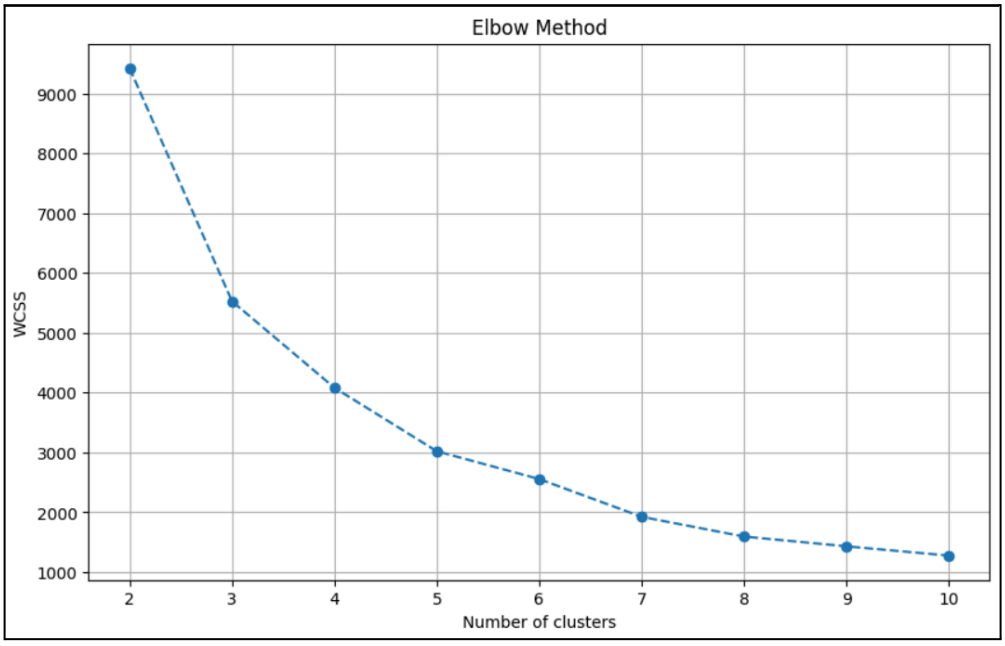


Figure 18.1: Graph of the number of clusters by within-cluster-sum-of-square by using Elbow Method

In this part, we try to apply K-means, a distance-based unsupervised clustering technique, to choose the best cluster for customer segmentation study. We chose the elbow approach because it provides a graphical depiction of the best K-value in k-means clustering.

However, based on **Figure 18.1**, the elbow points at K=2, K=3, and K=4 appear to be acceptable. It is unclear which K-value should be used for the elbow point. Hence, we opt to use the Silhouette method to find the optimal K-value.

```
from sklearn.metrics import silhouette_score

#Calculate the Silhouette Scores
silhouette_scores = []
for i in range(2, 11):
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(X_scaled)
    score = silhouette_score(X_scaled, kmeans.labels_)
    silhouette_scores.append(score)

#Plot Silhouette Scores
plt.figure(figsize=(10, 6))
plt.plot(range(2, 11), silhouette_scores, marker='o', linestyle='--')
plt.title('Silhouette Score Method')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.grid(True)
plt.show()
```

Figure 18.2: Code for finding the silhouette scores

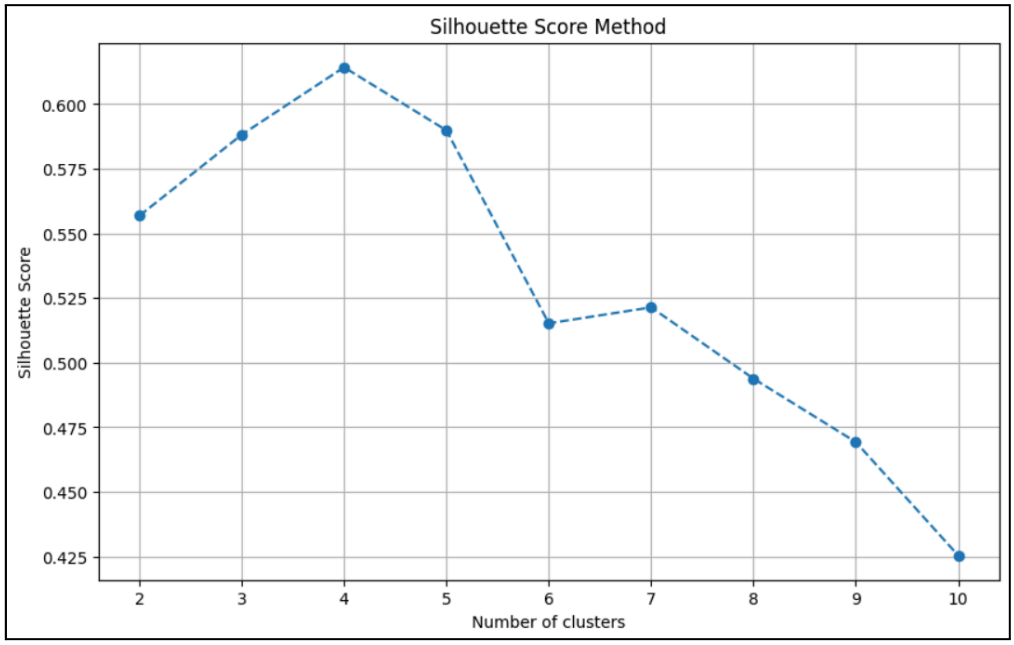


Figure 18.3: Graph for visualizing the Silhouette score of each number of clusters

Figure 18.3 shows that the maximum Silhouette score is K=2, while the lowest is K=10. A higher Silhouette score signifies better data clustering outcomes. This score is generated by comparing each data point to its cluster and determining how distinct it is from other clusters. The Silhouette score is frequently used to evaluate the effectiveness of clustering techniques such as K-means clustering. However, the graph for determining the optimal number of clusters by using the Elbow method and Silhouette method is slightly different. Hence, we need to apply an appropriate approach to determine the optimal value of K-value. We opted to combine both graphs to find the optimal K-value.

```
import numpy as np

#define the cluster for testing
cluster_range = range(2,11)

#relative change in wcss
wcss_diff = np.diff(wcss) / np.max(wcss)

#normalize the silhouette score
silhouette_scores_norm = (silhouette_scores - np.min(silhouette_scores)) / (np.max(silhouette_scores) - np.min(silhouette_scores))

wcss_diff = np.append(wcss_diff,0)
combined_scores = silhouette_scores_norm - np.abs(wcss_diff)
optimal_clusters = cluster_range[np.argmax(combined_scores)]

print(f"Optimal number of clusters based on combined criteria: {optimal_clusters}")
```

```
Optimal number of clusters based on combined criteria: 4
```

Figure 18.4: Calculation for the optimal number of clusters based on combined criteria.

In clustering analysis, determining the optimal number of clusters is crucial for achieving meaningful and interpretable results. The approach we used involves a combination of two key metrics: the Within-Cluster Sum of Squares (WCSS) and the Silhouette Scores.

Combining these metrics involves calculating the relative change in WCSS and normalizing the Silhouette Scores. We then compute a combined score for each number of clusters by subtracting the absolute value of the relative WCSS change from the normalized Silhouette Scores. This combined approach ensures that both the internal compactness and the separation of clusters are considered.

The optimal number of clusters is determined by identifying the cluster count that maximizes this combined score. In this case, the number of clusters that achieved the highest combined score was selected as the optimal choice.

This methodology ensures a balanced evaluation of clustering quality by integrating both compactness and separation criteria. By using this combined criteria approach, we obtain a robust and well-rounded assessment of the optimal number of clusters, leading to more meaningful and effective clustering results.

From **Figure 18.4**, it is evident that the optimal number of clusters for the K-means algorithm is 4. This conclusion is based on a comprehensive evaluation of the cluster validity measures.

By analyzing the combined criteria of the normalized silhouette scores and the relative change in the within-cluster sum of squares (WCSS), we determined that 4 clusters offer the best balance between intra-cluster cohesion and inter-cluster separation. This optimal k value was chosen because it maximizes the silhouette score while minimizing the relative change in WCSS, thus providing the most stable and well-defined clusters for the given data.

```
#Fit the K-means with the optimal number of cluster
kmeans = KMeans(n_clusters=optimal_clusters, random_state=42)
rfm['Cluster'] = kmeans.fit_predict(X_scaled)

cluster_summary = rfm.groupby('Cluster').agg({
    'Recency': ['mean', 'std'],
    'Frequency': ['mean', 'std'],
    'Monetary': ['mean', 'std']
}).reset_index()

plt.figure(figsize=(12, 8))
sns.scatterplot(x='Recency', y='Frequency', hue='Cluster', data=rfm, palette='viridis')
plt.title('Clusters Visualization')
plt.show()
```

Figure 18.5: Code of fitting the optimal number of cluster

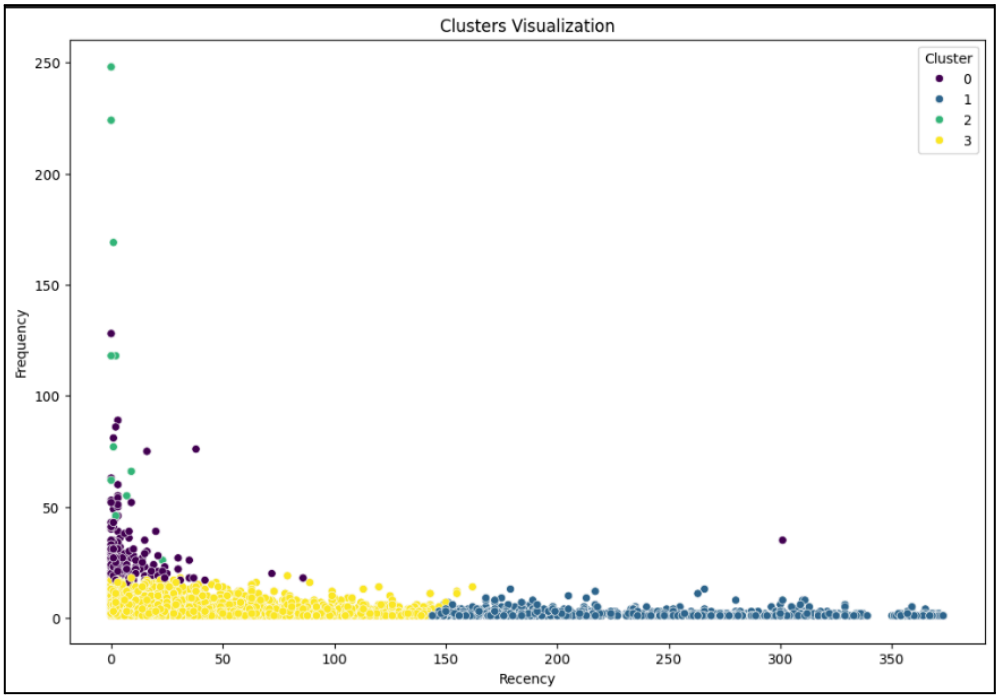


Figure 18.6: Cluster graph of Customer Segmentation

After determining the optimal number of clusters using a combined criteria approach, we proceeded to fit the K-Means algorithm with the selected cluster count. In this case, K-Means was configured to use the optimal number of clusters, as identified in the previous step.

We applied the K-Means algorithm with the optimal number of clusters to the scaled RFM data (Recency, Frequency, Monetary). Each data point in the dataset was assigned to one of the clusters, and the cluster assignments were added to the rfm dataframe.

To better understand the characteristics of each cluster, we calculated summary statistics for each cluster. The summary includes the mean and standard deviation of the Recency, Frequency, and Monetary values. This summary provides insights into the typical behavior of customers within each cluster, highlighting differences in their recency, purchase frequency, and monetary contributions.

The scatter plot allows us to visually assess the distribution of clusters based on Recency and Frequency. It provides a clear view of how customers are grouped into distinct clusters, showing the spread and separation of clusters in the feature space.

5.0 Evaluation

5.1 Evaluation of Models

```
#Create arrays
model_Name = ['Decision Tree', 'LinearSVR', 'Random Forest', 'KNN', 'Ridge Regression']
model_Accuracy = [dt_accurate_test, svr_accurate_test, rf_accurate_test, knn_accurate_test, rr_accurate_test]
mape_of_All_Models = [dt_mape, svr_mape, rf_mape, knn_mape, rr_mape]
r2_of_All_Models = [dt_r2, svr_r2, rf_r2, knn_r2, rr_r2]
rmse_of_All_Models = [dt_rmse, svr_rmse, rf_rmse, knn_rmse, rr_rmse]
mae_of_All_Models = [dt_mae, svr_mae, rf_mae, knn_mae, rr_mae]
[102] ✓ 0.0s
```

Figure 19.0: Code of Create Arrays

```
#Create a DataFrame
results_df = pd.DataFrame({
    'Model Name': model_Name,
    'Accuracy (%)': model_Accuracy,
    'MAPE (%)': mape_of_All_Models,
    'R2 Score': r2_of_All_Models,
    'RMSE': rmse_of_All_Models,
    'MAE': mae_of_All_Models
})

#Display the DataFrame
print(results_df)
[103] ✓ 0.0s
```

Figure 19.1: Code of Create a DataFrame

	Model Name	Accuracy (%)	MAPE (%)	R2 Score	RMSE	MAE
0	Decision Tree	82.769346	0.672539	0.827693	16.956007	7.275265
1	LinearSVR	1.605087	3.655809	0.016051	40.519030	16.713639
2	Random Forest	99.874056	0.000797	0.998741	1.449645	0.057673
3	KNN	27.788656	2.517935	0.171834	37.173320	15.254806
4	Ridge Regression	53.767105	0.711395	0.537671	27.774629	9.830439

Figure 19.2: Evaluation of Machine Learning Model Using Various Performance Metrics

Based on the analysis in **Figure 19.2**, several machine learning models were evaluated for their performance, including Decision Tree, LinearSVR, Random Forest, KNN and Ridge Regression. The metrics that were used to assess the performance are Accuracy, Mean Absolute Percentage Error (MAPE), R-Square Score, Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE). Among these models, Random Forest emerged as the best performing model, with an extremely high accuracy of 99.87% and the lowest error rates,

indicated by a low RMSE(1.45%) and MAE(0.06%). In contrast, the LinearSVR and KNN models showed significantly lower accuracy and higher error metrics, making them the least effective models in this analysis. The Ridge Regression and Decision Tree performed moderate, but still outperformed by the Random Forest model. Overall, the Random Forest model proved to be the most suitable choice for this analysis, accurately predicting the target variable with minimal errors.

5.1.1 Accuracy Evaluation

```
#Visualization for Data Accuracy
plt.figure(figsize=(10, 6))
plt.bar(model_Name, model_Accuracy, color=['skyblue', 'lightcoral', 'seagreen', 'mediumpurple', 'yellow'])
plt.xlabel('Algorithms')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.show()

print(model_Accuracy)
```

[104] ✓ 0.0s

Figure 19.3: Code of Visualization for Data Accuracy

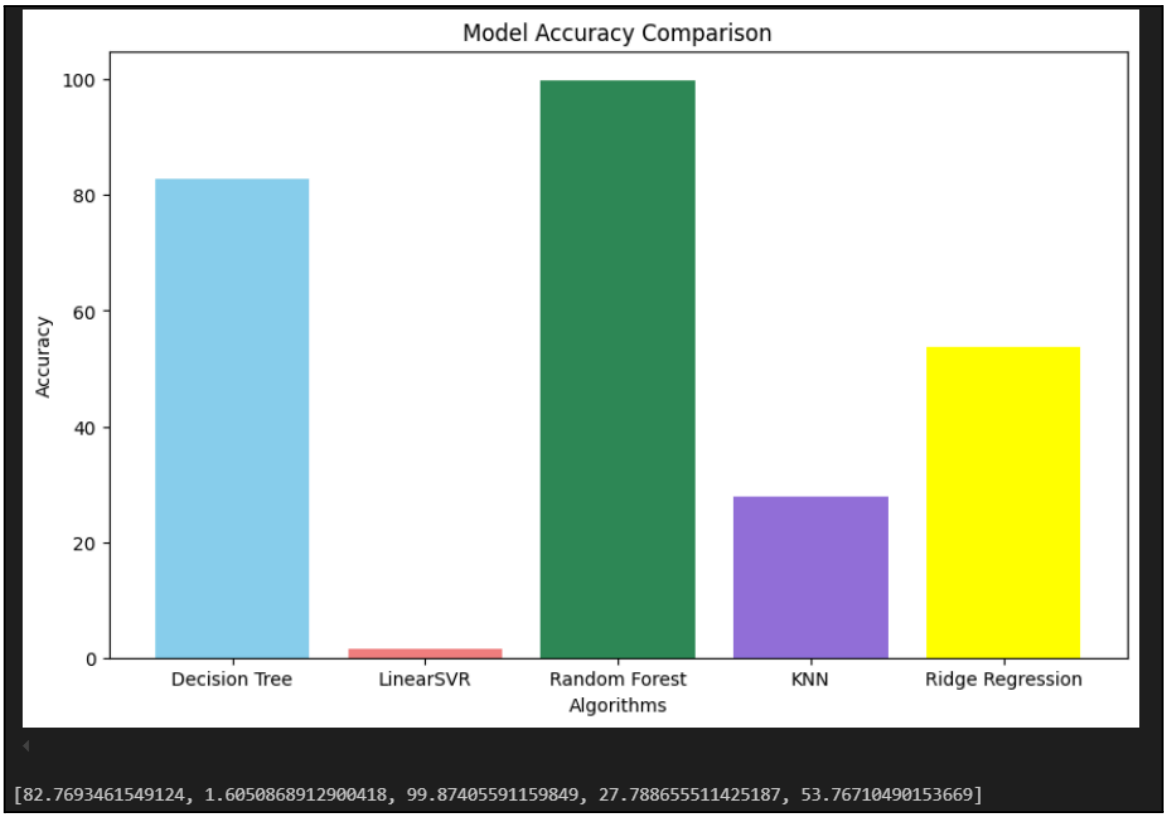


Figure 19.4: Accuracy Comparison of Machine Learning Models

In **Figure 19.4**, the accuracy of various machine learning models is visually compared using a bar chart. The models evaluated include Decision Tree, LinearSVR,

Random Forest, KNN and Ridge Regression. The chart shows that the Random Forest model has the highest accuracy, reaching close to 100%, which demonstrates that it is the most effective model in terms of accuracy. The Decision Tree and Ridge Regression also perform relatively well, though their accuracy is significantly lower than the Random Forest. On the other hand, LinearSVR and KNN models show particularly lower accuracy, indicating that they are less reliable for this analysis. The bar chart provides a straightforward visual representation, making it easy to see which models are performing better.

5.1.2 Mean Absolute Percentage Error (MAPE)

```
#Visualization for MAPE
plt.figure(figsize=(10, 6))
plt.bar(model_Name, mape_of_All_Models, color=['skyblue', 'lightcoral', 'seagreen', 'mediumpurple', 'yellow'])
plt.xlabel('Algorithms')
plt.ylabel('MAPE')
plt.title('Model MAPE Comparison')
plt.show()

print(mape_of_All_Models)
[105]  ✓ 0.0s
```

Figure 19.5: Code of Visualization for MAPE

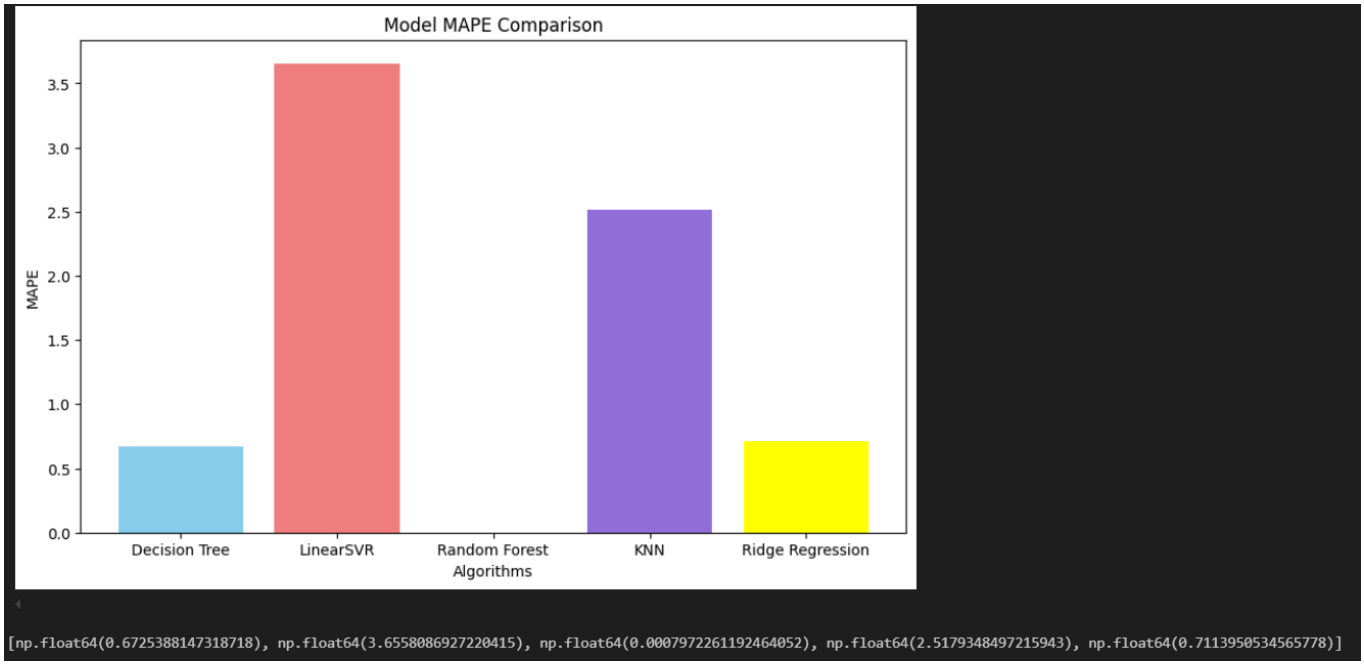


Figure 19.6: MAPE Comparison of Machine Learning Models

Figure 19.6 shows the Mean Absolute Percentage Error (MAPE) of various machine learning models is compared using a bar chart, which visually demonstrates the prediction accuracy of each model in terms of percentage error. The Linear Support Vector Regression (LinearSVR) model has the highest MAPE among the models, showing a

significantly higher prediction error. This model struggles significantly with accuracy in percentage makes it the least reliable among the models evaluated.

The KNN model has one of the higher MAPE values, showing its difficulties with forecast accuracy, making it less reliable in terms of prediction performance. Ridge Regression shows a relatively low MAPE, indicating this model effectively minimizes percentage errors. The Decision Tree model has a very low MAPE, indicating strong performance with minimal percentage error and the model is quite accurate in making predictions. The MAPE of the Random Forest model typically exhibits strong performance and low MAPE, indicating precise predictions.

In summary, this bar chart clearly illustrates that LinearSVR, KNN and Ridge Regression Perform the least accurate predictions, models like Decision Tree and Random Forest are more effective at producing accurate predictions with minimal percentage error.

5.1.3 R-Squared (Coefficient of Determination)

```
#Visualization for R-Squared
plt.figure(figsize=(10, 6))
plt.bar(model_Name, r2_of_All_Models, color=['skyblue', 'lightcoral', 'seagreen', 'mediumpurple', 'yellow'])
plt.xlabel('Algorithms')
plt.ylabel('R-Squared')
plt.title('Model R-Squared Comparison')
plt.show()

print(r2_of_All_Models)
```

[106] ✓ 0.0s

Figure 19.7: Code of Visualization for R-Squared

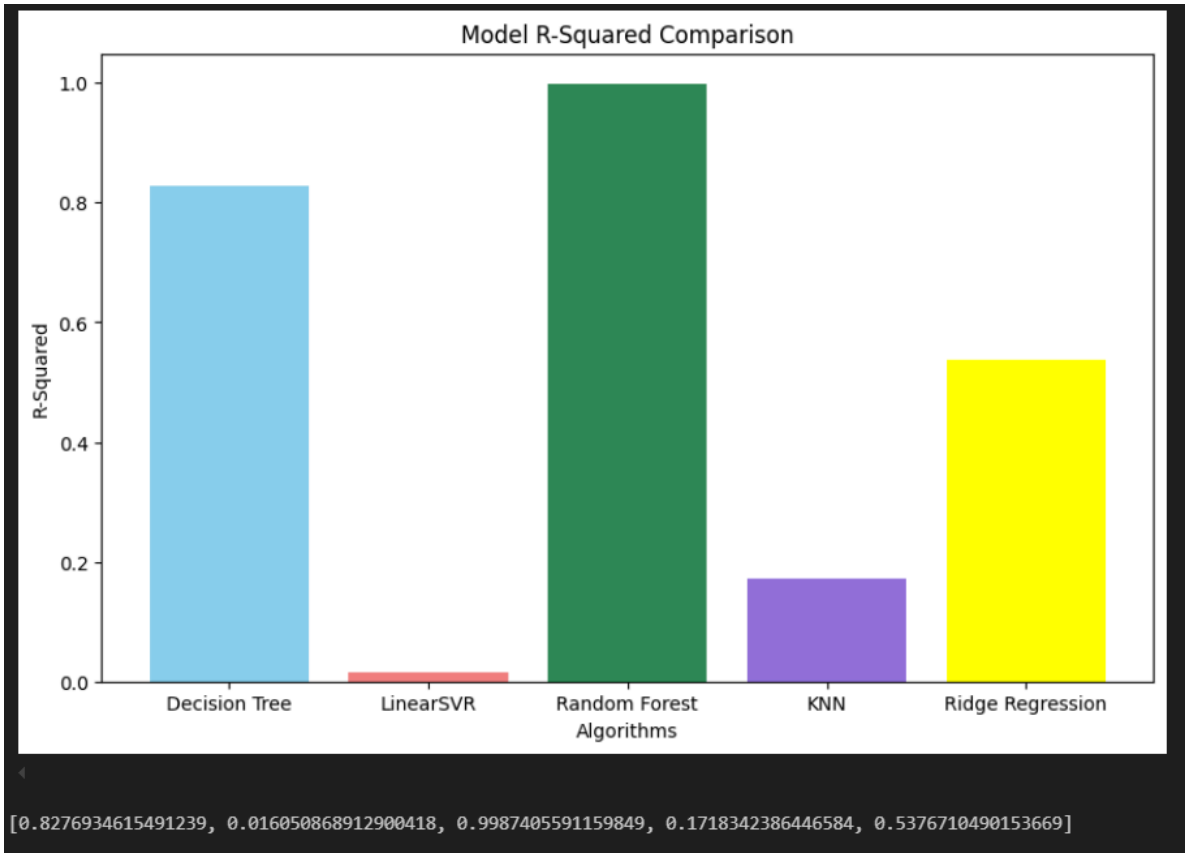


Figure 19.8: R-Squared Comparison of Machine Learning Models

The bar chart in **Figure 19.8** provides a comparison of the R-squared values for five distinct machine learning algorithms, offering insight into their performance on the given dataset. From the chart, it is clear that the Random Forest algorithm exhibits the highest R-squared value, nearing 1.0, which signifies outstanding predictive power and an ability to explain nearly all the variance in the data. Following closely is the Decision Tree model, with an R-squared value of approximately 0.82. This indicates a very high level of predictive accuracy, though slightly lower than Random Forest.

In contrast, the Ridge Regression model demonstrates a moderate R-squared value of around 0.54. This suggests it can explain only about half of the variance in the dataset, performing better than both LinearSVR and K-Nearest Neighbors (KNN) but worse than Random Forest and Decision Tree. The LinearSVR model has the lowest R-squared value, around 0.016, indicating poor performance and a limited ability to capture the underlying patterns in the data. Similarly, K-Nearest Neighbors (KNN) performs poorly, with an R-squared value of approximately 0.17, signifying a weak fit to the data.

This bar chart serves as a crucial tool for model evaluation, making it easy to identify the best-performing algorithm. In this case, Random Forest is the top-performing model, followed closely by Decision Tree, while Ridge Regression provides a moderate fit. The significantly lower performance of LinearSVR and KNN further emphasizes the importance of model selection based on the specific characteristics of the dataset. Ultimately, this comparison allows for a data-driven selection of the most appropriate algorithm, ensuring optimal predictive performance.

5.1.4 Root Mean Square Error (RMSE)

```
#Visualization for RMSE
plt.figure(figsize=(10, 6))
plt.bar(model_Name, rmse_of_All_Models, color=['skyblue', 'lightcoral', 'seagreen', 'mediumpurple', 'yellow'])
plt.xlabel('Algorithms')
plt.ylabel('RMSE')
plt.title('Model RMSE Comparison')
plt.show()

print(rmse_of_All_Models)
```

[107] ✓ 0.0s

Figure 19.9: Code of Visualization for RMSE

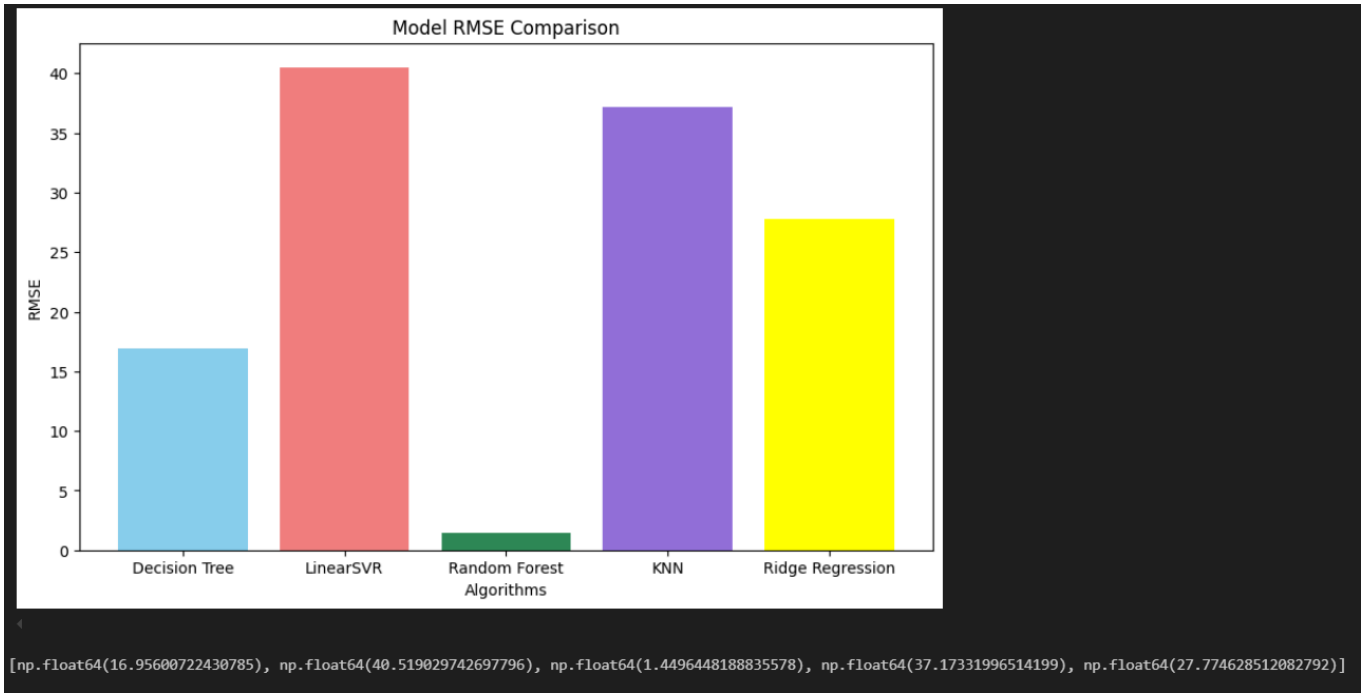


Figure 19.10: RMSE Comparison of Machine Learning Models

The bar chart in **Figure 19.10** provides a detailed comparison of the Root Mean Square Error (RMSE) across five different machine learning models. The Random Forest model stands out with the lowest RMSE of approximately 1.45, indicating its exceptional accuracy and reliability in predicting the target variable. This makes it the most effective model among those evaluated. The Decision Tree model, with an RMSE of approximately 16.96, also demonstrates strong predictive accuracy, positioning it as the second-best model in this comparison.

In contrast, the Ridge Regression model, which has an RMSE of about 27.77, performs moderately well. Although it is more accurate than the LinearSVR and KNN models, it is still less reliable than the Random Forest and Decision Tree models. The Linear Support Vector Regression (SVR) model shows a significantly higher RMSE of

approximately 40.52, indicating poor performance, with predictions that deviate substantially from the actual values. Similarly, the K-Nearest Neighbors (KNN) model, with the highest RMSE of approximately 37.17, demonstrates the least accuracy among the models, reflecting considerable deviation from the true values.

This bar chart serves as a critical tool for evaluating and comparing model performance. It clearly highlights the Random Forest as the top-performing model, closely followed by the Decision Tree. Ridge Regression offers a middle-ground solution, while LinearSVR and KNN lag behind significantly, underscoring the importance of selecting the most suitable model based on the specific characteristics of the dataset. This data-driven comparison enables the selection of the most appropriate algorithm, ensuring optimal predictive performance for the given task.

5.1.5 Mean Absolute Error (MAE)

```
#Visualization for MSE
plt.figure(figsize=(10, 6))
plt.bar(model_Name, mae_of_All_Models, color=['skyblue', 'lightcoral', 'seagreen', 'mediumpurple', 'yellow'])
plt.xlabel('Algorithms')
plt.ylabel('MAE')
plt.title('Model MAE Comparison')
plt.show()

print(mae_of_All_Models)
```

[108] ✓ 0.0s

Figure 19.11: Code of Visualization for MSE

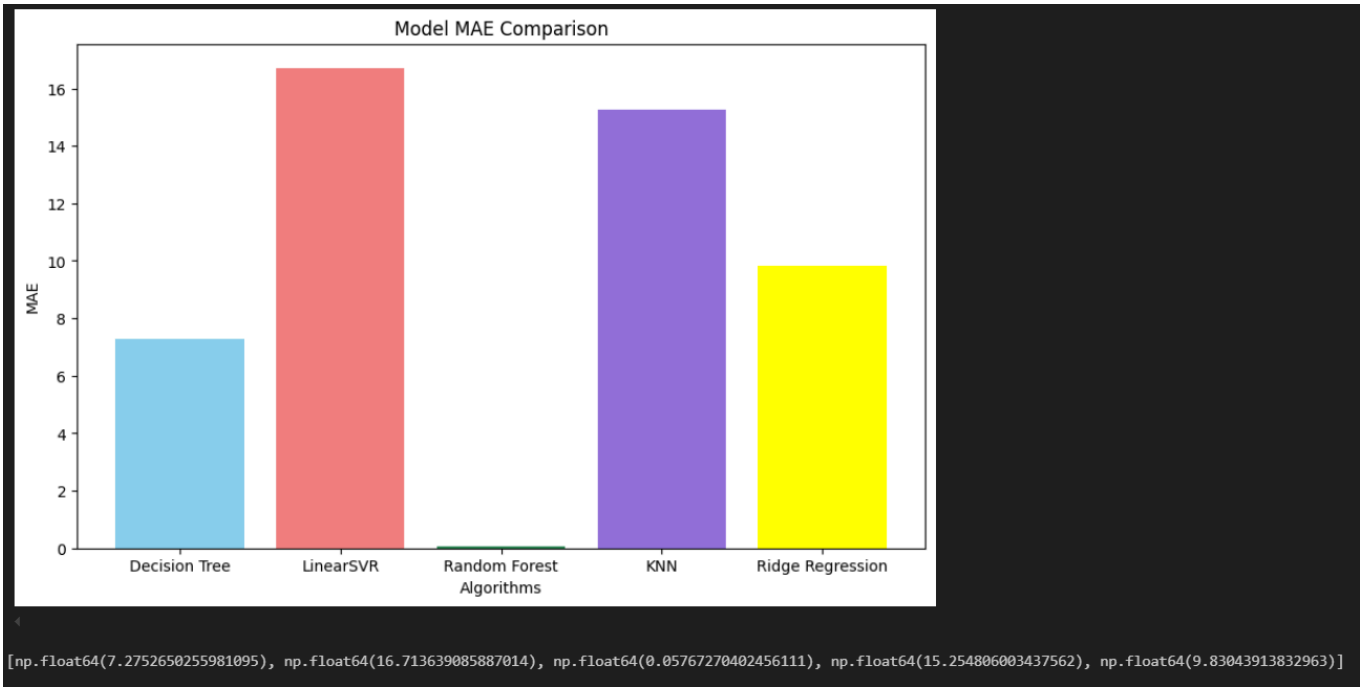


Figure 19.12: MAE Comparison of Machine Learning Models

The bar chart in **Figure 19.12** compares the Mean Absolute Error (MAE) of different machine learning models. Among these, the Random Forest model stands out with the lowest MAE, approximately 0.6, indicating it is the best performer in terms of prediction accuracy. This low MAE suggests that the Random Forest model consistently makes predictions that are close to the actual values. The Decision Tree model also shows strong performance, with a low MAE of around 7.28, slightly higher than the Random Forest. This indicates that the Decision Tree model is highly accurate, though not as precise as the Random Forest model.

The Ridge Regression model has a moderate MAE of approximately 9.83, placing it between the best-performing models (Random Forest and Decision Tree) and the worst. While Linear Regression provides reasonably accurate predictions, it is not as effective as

the top models. The KNN model, with an MAE of about 15.25, has a relatively high error, indicating less accurate predictions compared to the other models. The Linear SVR model has the highest MAE, around 16.71, making it the worst performer in this comparison. This high MAE suggests that the Linear SVR model's predictions are significantly less accurate, making it the least suitable for this particular problem.

In summary, the bar chart illustrates that the Random Forest model is the best option, closely followed by the Decision Tree. The Ridge Regression model occupies a middle ground with moderate accuracy. The KNN and LinearSVR models, with the highest MAE values, are the least accurate and therefore the least recommended for this task. This comparison helps in selecting the most appropriate model based on prediction accuracy.

5.2 Determination of Model with Best Performance

Based on the above analysis, we can determine that the Random Forest model is the best-performing model to analyze the data. There are several strong reasons to support our decision. Firstly, the Random Forest model achieves a high accuracy of 99.874056%, which is the highest among all the models evaluated. This indicates that it correctly predicts the target variable almost all the time., making it the most reliable in terms of prediction accuracy. Besides, the Random Forest model consists of an excellent R-squared score. The R-squared score of the Random Forest model is 0.998741, which is the highest among the models. This suggests that the model explains 99.87% of the variance in the target variable, indicating a very strong relationship between the predictors and the outcome.

Furthermore, the Random Forest model has both the lowest RMSE (1.449645) and the lowest MAE (0.057673) out of all the models assessed. This indicated that the Random Forest model consistently makes smaller errors in its predictions compared to the other models. The low RMSE value shows that the model's predictions are on average, very close to the actual values. The MAPE of the Random Forest model is 0.000797, which is the smallest value among the models, reflecting a very low percentage error. This means that, on average, the model's predictions are very close to the actual values, maintaining a high level of precision.

The Random Forest model demonstrates resilience to overfitting, a key advantage of its ensemble method. By combining the predictions of multiple decision trees, it reduces the risk of overfitting, which is particularly beneficial when working with complex datasets. Single decision trees can often overfit, leading to poor generalization on unseen data. However, the Random Forest model mitigates this issue, as evidenced by its high accuracy and R^2 score. These metrics indicate that the model not only generalizes well to new data but also maintains robust predictive power, making it a reliable choice for various applications.

However, to guarantee that the model's performance translates into meaningful business value, it is critical to evaluate how these predictions affect actual business results. For example, precise sales forecasting might result in better inventory management, lower holding costs, and more customer satisfaction. Conversely, it is critical to discover any potential flaws in the model. Evaluating if the model's elements are comprehensive and relevant to the company objectives might help identify opportunities for development. Furthermore, verifying the model's performance in real-world settings or during a pilot phase can guarantee that it retains its high accuracy and dependability when applied to new, unknown data or operating conditions.

To confirm the Random Forest model's efficacy, do pilot testing or deploy it in a constrained real-world application. This will give insights into how effectively the model works in real-world business scenarios, as well as enable long-term monitoring of its accuracy and dependability. Implementing the model in a controlled environment allows you to test its effectiveness in real-world settings and guarantee that it stays trustworthy under varying conditions. Continuous performance monitoring is required to ensure the model's efficacy. Businesses can guarantee that their models continue to provide accurate and dependable forecasts by tracking their performance over time. If the model's performance deteriorates or new data becomes available, modifications can be made to keep it effective.

The data mining outcomes, such as client segmentation and sales projections, should be reevaluated to verify that they effectively satisfy the original business objectives. Evaluating the relevance and influence of these findings on corporate strategies can help decide whether the insights presented are useful and practical. Furthermore, discovering any unexpected insights or trends discovered through data mining might lead to new possibilities or difficulties. For example, consumer segmentation may identify new market niches that may be targeted with particular incentives.

In conclusion, the Random Forest model emerges as the best-performing model for analyzing the data due to its superior accuracy, strong R-squared score, and minimal error rates across various metrics. Its ability to consistently deliver accurate predictions, couples with its robustness against overfitting, makes it an exceptional choice for modeling complex datasets. The model’s high precision, as indicated by the lowest MAPE, RMSE, and MAE values, further underscores its effectiveness in making reliable and precise predictions. The higher accuracy and R-squared score, combined with its lowest RMSE and MAE, reinforce the Random Forest model’s reliability and effectiveness, highlighting its overall superiority in both accuracy and error minimization, making it the most reliable and powerful predictive tools for this analysis.

5.3 Evaluation of Customer Segmentation

```
sns.pairplot(rfm, hue='Cluster', vars=['Recency', 'Frequency', 'Monetary'], palette='viridis')
plt.show()

print(cluster_summary)
```

[115] ✓ 2.8s

Figure 20.1: Code of the Pair plot for Cluster Summary

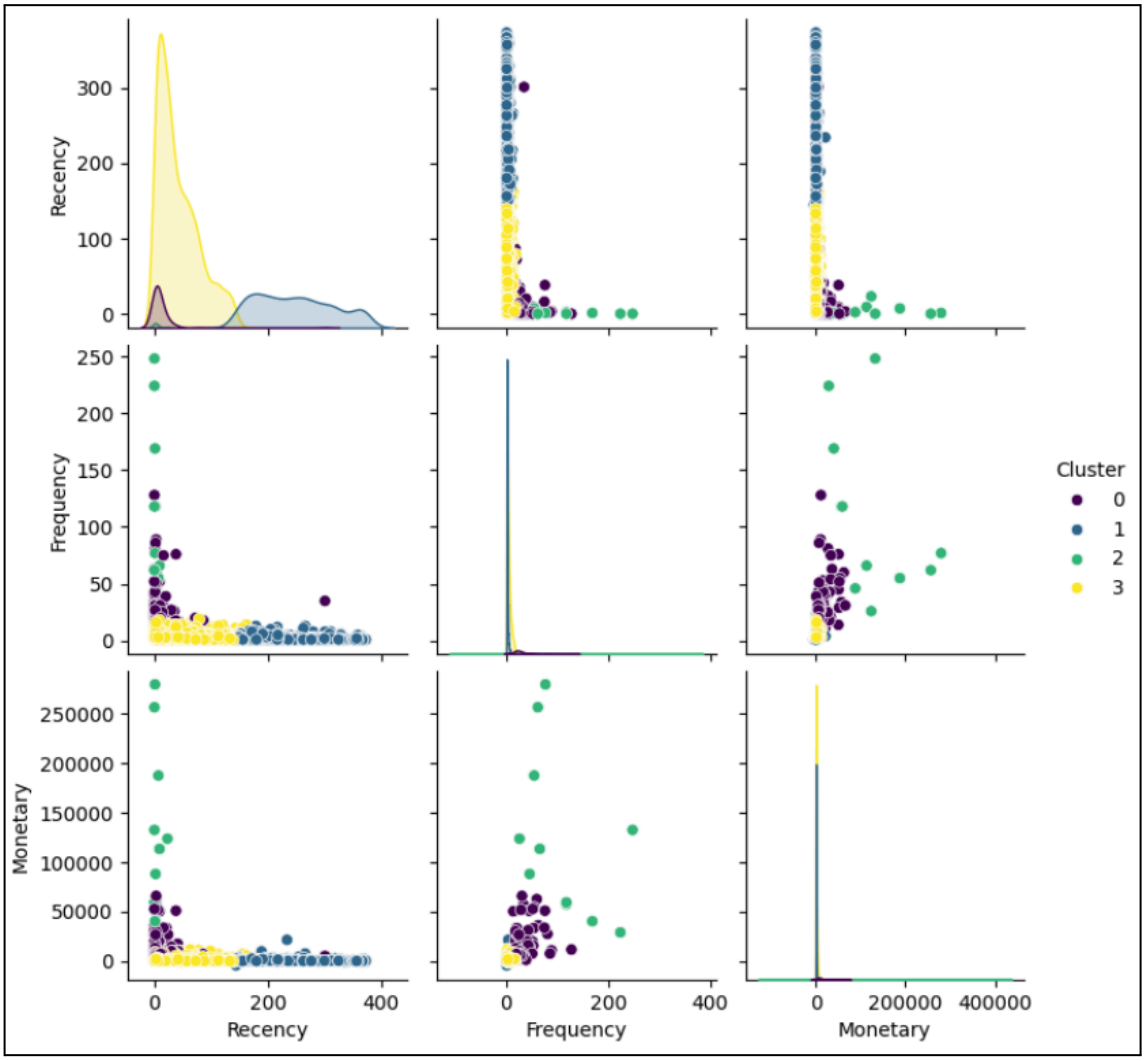


Figure 20.2: Cluster Summary

Figure 20.1 displays the pairplot that performs the visualizing of relationships between the variables used in the customer segmentation evaluation using the RFM model (Recency, Frequency, Monetary) analysis.

Figure 20.2 is a pair plot graph showing the relationship between RFM model and four clusters which are labeled in four colors. The clusters 0,1,2,3 are color-coded in the plot, likely representing different customer segments derived from a clustering algorithm such as K-Means. Each point on the graph corresponds to a customer.

Firstly is the interpretation of the diagonal subplots. These subplots show the distribution of the individual variables (Recency, Frequency and Monetary) for each cluster. These are represented either as histograms or kernel density plots, showing insights into how each cluster is distributed across the three variables. Based on **Figure 20.2**:

➢ Recency

It measures how recently a customer made a purchase. Lower values indicate more recent purchases, and higher values indicate that the purchase was made further in the past.

1. Cluster 0 (Purple):	- Cluster 0 is similar to Cluster 2 in that it has a higher recency, indicating that these customers also haven't made purchases recently. However, its density is lower compared to the other clusters, meaning there are fewer customers in this category.
2. Cluster 1 (Blue):	- This cluster has a wider spread in recency values but is still concentrated on the lower end of the recency axis, indicating somewhat recent purchases. - This spread indicated customers who are still fairly active but might not be as recent as those in Cluster 3. The density tails after 100 days and so on, meaning these customers are less likely to be very recent purchasers.
3. Cluster 2 (Green):	- Cluster 2's distribution shifts towards higher recency values, meaning these customers haven't purchased in a while. - The density increases as the recency values go up, which implies that this group consists of customers who may have stopped purchasing and haven't returned for a long time.

4. Cluster 3 (Yellow)	- Cluster 3 concentrates in the lower recency range, indicating that most customers in this cluster have made recent purchases. - The high density near the lower recency values suggests that this cluster could represent very active or newly acquired customers who are still engaged with the business.

➢ Frequency

It measures how often a customer makes purchases, with higher value representing more frequent buyers.

1. Cluster 0 (Purple) & Cluster 1 (Blue):	- These clusters have a lower frequency with most customers making between 1 to 10. - Cluster 0 and Cluster 1 show fairly low density for higher frequency purchases, indicating that these customers don't buy very often. This suggests they could be infrequent buyers who may not be highly engaged.
2. Cluster 2 (Green):	- This cluster has the highest frequency and is the most spread out across different frequency values, indicating a diverse group of frequent buyers. - The density curve suggests that many customers in this cluster buy relatively often, with the majority having a frequency of up to around 100 purchases. This indicates that this cluster could represent high value customers who buy frequently and may be loyal to the business.
3. Cluster 3 (Yellow):	- Cluster 3 is tightly clustered near zero frequency, meaning most customers in this cluster have made very few or even just a single purchase. - This suggests that this cluster of customers consists of new or one time buyers who have not yet

	demonstrated loyalty or consistent purchasing behavior.

➢ Monetary

It measures how much a customer has spent, with higher values representing higher total spending.

1. Cluster 0 (Purple), Cluster 1 (Blue) & Cluster 3 (Yellow):	- These clusters are much more tightly distributed around lower monetary values. The majority of customers in these clusters spend significantly less than Cluster 2. - Cluster 3 has a particularly low monetary range, indicating that most customers in this group spend little, possibly due to making few purchases. - Cluster 1 and Cluster 0 are similar, with the majority of their customers falling into the low spending category, but with slightly higher spending than CLuster 3. These customers may be casual buyers or those who make smaller purchases
4. Cluster 2 (Green):	- This cluster contains customers who spend the most, and it has a longer tail in the monetary distribution. Some customers in this cluster are high spenders - The widespread suggests that Cluster 2 contains high value customers who contribute significantly to revenue. These customers are likely to be the business's high spending customers.

Next is to interpret the off-diagonal scatter plots for each pair of variables in more detail. This helps to understand how two variables interact with each other across customer clusters.

Interpretation of data:

Recency vs Frequency:	➢ Cluster 2 (Green) appears in the upper-left corner, meaning customers who buy frequently and have more recent purchases. ➢ Cluster 0 and Cluster 3 are more towards the bottom, indicating lower frequency and higher recency (less recent purchases) ➢ Cluster 1 show low frequency (a small number of purchases) but with somewhat recent purchases (lower recency)
Recency vs Monetary	➢ Cluster 2 spends the most money and is slightly more spread out across different recency values, indicating some high spending customers who haven't made a recent purchase. ➢ Cluster 0 and Cluster 1 are more concentrated with lower monetary values. ➢ Cluster 3 is concentrated at the low end of both recency and monetary, indicating customers who spend very little but make recent purchases.
Frequency vs Monetary	➢ Cluster 2 shows customers who not only spend more but also buy more frequently. ➢ Cluster 0, Cluster 1 and Cluster 3 are concentrated near lower values in both frequency and monetary, meaning they purchase less frequently and spend less overall.

	Cluster	Recency		Frequency		Monetary	\
		mean	std	mean	std	mean	
0	0	9.752577	23.982952	28.510309	15.017171	12168.264691	
1	1	247.927577	67.071082	1.805942	1.430692	455.110716	
2	2	4.090909	6.963541	109.909091	74.266351	124312.306364	
3	3	41.780906	36.245015	4.370550	3.591324	1320.981506	

	std
0	11895.936188
1	907.428611
2	84756.843941
3	1471.360742

Figure 20.3: Summary of the Descriptive Statistics for the Clusters

Cluster 0:

Recency	The average recency of Cluster 0 is quite low, around 9.75, indicating that these customers made purchases very recently.
Frequency	The average frequency of Cluster 0 is relatively high, with an average 28 purchases.
Monetary	The monetary value of Cluster 0 is high, with an average of 12168, indicating the customers spend a lot.
This group of customers likely consists of loyal and high-value individuals.	

Cluster 1:

Recency	The high average recency in Cluster 1, around 247.92, indicates that these customers have not made recent purchases.
Frequency	The average frequency in Cluster 1 is low, with an average of 1.80 purchases.
Monetary	The average monetary value in Cluster 1 is around 455, indicating low spending.
This group of customers might be one-time buyers or inactive customers.	

Cluster 2:

Recency	Cluster 2 has a very low average recency, around 4.09, suggesting that these customers have made very recent purchases.

Frequency	The purchase frequency in Cluster 2 is very high, averaging around 109.91 purchases.
Monetary	The monetary value in Cluster 2 is extremely high, with an estimated average spend of 124,312.
This group of customers is likely a segment of top customers who make frequent purchases and spend the most.	

Cluster 3:

Recency	The average recency in Cluster 3 is moderate, approximately 41.78, indicating that these customers have made purchases at a moderate frequency.
Frequency	The purchase frequency in Cluster 3 is low, averaging around 4.37 purchases.
Monetary	The average monetary value in Cluster 3 is low to moderate, approximately 1,320.
This group of customers might be infrequent buyers who are still somewhat engaged, given their occasional purchases.	

These clusters help identify customer groups that need different marketing strategies. For instance, Cluster 1 might need special retention efforts, while Cluster 2 could benefit from loyalty rewards.

6.0 Deployment

6.1 Model Deployment

The goal of this project is to build and deploy a web-based interface for predicting total sales and performing customer segmentation. The deployment uses a Random Forest model for sales prediction and K-Means clustering for customer segmentation. The application is built using Streamlit, a Python framework for creating interactive web applications and the entire project is developed and tested in Visual Studio Code.

We use Streamlit to build the user-friendly web interface. The reason that we opt to use Streamlit to build the web interface is because Streamlit is an open-source framework that simplifies the processing of creating web-based applications directly from Python scripts. There is no need to separate front-end and back-end development. In a few lines of code, we can create an interactive and data-driven application. Besides, Streamlit supports a variety of widgets such as input boxes and drop down menus that are perfect for building user-friendly interfaces. These widgets enable users to input data and instantly see the model's predictions. Streamlit also provides a seamless deployment process, whether it is on local machines or cloud platforms. This reduces the time to bring the application to users and ensures fast and effective deployment.

In this project, we use Visual Studio Code as a development environment for writing and testing the application. VS Code has a wide range of extensions such as Python extensions that provide enhanced support for debugging, code formatting and linting. It integrates seamlessly with Jupyter notebooks, Streamlit and other necessary tools. VS Code's built-in terminal allows running Streamlit applications directly within the IDE. Its powerful debugging tools simplify the process of identifying and fixing issues which is essential for ensuring model and UI performance. We also use Pickle for saving and loading trained models to and from the files.

In the model training, we use Random Forest model to train the prediction model by using sales data including the features like StockCode, Description, Quantity, UnitPrice, CustomerID and Country. The model is designed to predict total sales based on these features. Hyperparameter tuning for Random Forest is performed using GridSearchCV to find the optimal parameters to maximize the model's prediction accuracy. In this project, we found that the best optimal parameters for Random Forest is as shown below:

```
parameter_grid = {
    'n_estimators': [300],
    'max_depth': [20],
    'min_samples_split': [2],
    'max_features': [None]
}
```

Figure 21.0: Code of Parameter Grid

The K-means model is trained on RFM values which are Recency, Frequency and Monetary to segment customers into distinct groups. Recency is indicating how recently a customer made a purchase. Frequency is indicating how often a customer makes a purchase while Monetary is indicating how much money a customer has spent in total. By using K-means clustering, customers are grouped into distinct segments based on their purchasing behavior. The number of clusters is determined by evaluating metrics such as Within-Cluster Sum of Squares(WCSS) and Silhouette scores to optimize the cluster configuration.

After training, both models are saved into files using Pickle. This allows the models to be used without retraining, enhancing performance and deployment efficiency. Besides, we also save the scalers and encoders for data preprocessing including label encoders for categorical variables and scalers for numerical features. Supporting data such as stock code descriptions and country lists are also saved in a Pickle file to enhance the user experience by providing more context to inputs.

For the user interface design, the web interface is divided into two tabs which are:

1. Prediction for Total Sales
2. Customer Segmentation Analysis

For the total sales prediction tab, users input relevant data such as StockCode, UnitPrice, Year, Month, Country and CustomerID. These input fields are familiar to sales teams and business analysts. After entering the inputs, users click the Predict button which invokes the Random Forest model to predict total sales. The result is displayed on the interface in real-time to enable faster decision-making for sales teams and business managers. By making the interface intuitive, users can easily provide the necessary data and get predictions without technical knowledge.

For the customer segmentation tab, users can enter the RFM values to predict which customer segment a user belongs to. After submitting the RFM data, the K-means model predicts the customer segment which helps the business understand customer behavior patterns with a visualization graph for better understanding. This can be used to

personalize marketing strategies and improve customer retention. The interface can be easily adapted and extended to include more advanced customer analytics, making it a powerful tool for customer behavior analysis.

The user inputs necessary data points through the interface. The data is preprocessed and fed into the models. In the Total Sales Prediction Tab, the Random Forest model calculates the predicted total sales. In the Customer Segmentation Tab, the K-Means model determines the customer segment based on RFM values. The predictions are displayed immediately after submission, offering users quick insights based on their input.

We test the application locally in Visual Studio Code to ensure accuracy and performance. The model predictions were compared against known values for validation and hyperparameter tuning was done using GridSearchCV for optimal performance.

After testing, the application can be deployed on a web server or cloud platform for public access. Streamlit's simple deployment options make this step straightforward.

After local testing, the deployed application provides a robust and user-friendly platform for both prediction total sales and segmenting customers efficiently utilizing machine learning models to offer valuable insight for business decision-making. Streamlit offers a seamless deployment process, making it easy to push the application to platforms like Streamlit Cloud or AWS.

In this project, we made a strategic decision to separate the model deployment process from the data analysis and model training tasks by using two distinct files:

1. ModelDeployment.ipynb (for training)
   This file is responsible for the full data analysis process and model training. This includes the data preprocessing, feature engineering, training the models, hyperparameter tuning and saving the models. This Jupyter notebook is mainly used for exploratory data analysis (EDA), model experimentation and training. Since training large machine learning models can be time-consuming, we run this task separately from deployment.

```
#Importing necessary libraries
import pandas as pd
import numpy as np
from scipy import stats
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestRegressor
import pickle
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

#Data Loading
df = pd.read_csv('onlineRetail.csv')

#Calculate Total Sales
df['TotalSales'] = df['Quantity'] * df['UnitPrice']

#Adjust for InvoiceDate
df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'], format='%d/%m/%Y %H:%M')
df['Year'] = df['InvoiceDate'].dt.year
df['Month'] = df['InvoiceDate'].dt.month
df['Day'] = df['InvoiceDate'].dt.day
df['DayOfWeek'] = df['InvoiceDate'].dt.strftime('%A')
df['YearMonth'] = df['InvoiceDate'].dt.strftime('%Y-%m')
```

Figure 21.1: Data Processing

```
#Data Cleaning
df.drop_duplicates(inplace=True)
df.dropna(subset=['CustomerID'], inplace=True)
df = df[df['StockCode'] != "M"]

df = df.dropna()
df = df[df['Quantity'] > 0]
df.reset_index(drop=True, inplace=True)

df = df[df['UnitPrice'] > 0]
df.reset_index(drop=True, inplace=True)

#Remove Outliers by using Z-Score
z_Score = np.abs(stats.zscore(df[['Quantity', 'UnitPrice', 'TotalSales']]))
threshold = 3
df = df[(z_Score < threshold).all(axis=1)]

#Normalize continuous data
numerical_Columns = ['Quantity', 'UnitPrice', 'Year']
x_normalized = (df[numerical_Columns] - df[numerical_Columns].min()) / (df[numerical_Columns].max() - df[numerical_Columns].min())
df['Quantity'] = x_normalized['Quantity']
df['UnitPrice'] = x_normalized['UnitPrice']
```

Figure 21.2: Data Processing

At the beginning, we import all the necessary libraries to support the data analysis, model training and deployment. This section ensures that the tools required for both the training and deployment phases are available. We proceed with data loading and preprocessing. Preprocessing is a critical step to prepare raw data for model training. We also handle the missing data to enhance the data accuracy.

```
#Data Selection
y = df.TotalSales.values
x_Data = df.drop(['InvoiceNo', 'InvoiceDate', 'Day', 'DayOfWeek', 'YearMonth', 'TotalSales'], axis=1)

#Find the last purchase date
last_date = df['InvoiceDate'].max()

#Calculate the recency, frequency and monetary values (RFM)
rfm = df.groupby('CustomerID').agg({
    'InvoiceDate': lambda x: (last_date - x.max()).days,
    'InvoiceNo': 'nunique',
    'TotalSales': 'sum'
}).reset_index()

rfm.columns = ['CustomerID', 'Recency', 'Frequency', 'Monetary']

#Save the column names
column_names = x_Data.columns
with open('column_names.pkl', 'wb') as file:
    pickle.dump(column_names, file)
```

Figure 21.3: Feature Engineering

```
#Group the stockCode and description
stockcode_description = df.groupby('StockCode')['Description'].unique().to_dict()

#Save the grouping for stockCode and description
with open('stockcode_description.pkl', 'wb') as file:
    pickle.dump(stockcode_description, file)

#Save the country list
with open('country_list.pkl', 'wb') as file:
    pickle.dump(df['Country'].unique(), file)

#Encoding the categorical data
le_stockcode = LabelEncoder()
le_description = LabelEncoder()
le_customerid = LabelEncoder()
le_country = LabelEncoder()

x_Data['StockCode'] = le_stockcode.fit_transform(x_Data['StockCode'])
x_Data['Description'] = le_description.fit_transform(x_Data['Description'])
x_Data['CustomerID'] = le_customerid.fit_transform(x_Data['CustomerID'])
x_Data['Country'] = le_country.fit_transform(x_Data['Country'])
```

Figure 21.4: Feature Engineering

```
#Save the LabelEncoders for stockCode, description, customerID and country
with open('label_encoder_stockcode.pkl', 'wb') as file:
    pickle.dump(le_stockcode, file)

with open('label_encoder_description.pkl', 'wb') as file:
    pickle.dump(le_description, file)

with open('label_encoder_customerid.pkl', 'wb') as file:
    pickle.dump(le_customerid, file)

with open('label_encoder_country.pkl', 'wb') as file:
    pickle.dump(le_country, file)
```

Figure 21.5: Feature Engineering

In this step, we create new features from existing data. We create the date-time values by extracting Year and Month from the date column and an RFM score for customer segmentation. Numerical features are scaled using StandardScaler or MinMaxScaler to normalize the data. This is particularly important for algorithms like K-Means that are sensitive to the scale of data. For sales prediction, we scale features like Quantity and UnitPrice to avoid bias in the model. We save all the label encoder and necessary information in a Pickle file in order to improve the ease of use in the model deployment process.

```
#Train Test Split
x_train, x_test, y_train, y_test = train_test_split(x_Data, y, test_size=0.2, random_state=0)

#Data Modeling using Random Forest Algorithm
#We use the best estimator that we found in testing for data modeling
parameter_grid = {
    'n_estimators': [300],
    'max_depth': [20],
    'min_samples_split': [2],
    'max_features': [None]
}

#Train the model
model = GridSearchCV(RandomForestRegressor(random_state=0), parameter_grid, cv=5, n_jobs=-1)
model.fit(x_train, y_train)

#Save the model
with open('model.pkl', 'wb') as file:
    pickle.dump(model, file)
```

Figure 21.6: Training the model with hyperparameter tuning for prediction for total sales

```
#Select the features for customer segmentation clustering
X = rfm[['Recency', 'Frequency', 'Monetary']]

#Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
kmeans = KMeans(n_clusters=4, n_init=10, random_state=42)
kmeans.fit(X_scaled)

#Save the Customer Segmentation model
with open('customer_segmentation.pkl', 'wb') as file:
    pickle.dump(kmeans, file)

#Save the Scaler
with open('scaler.pkl', 'wb') as file:
    pickle.dump(scaler, file)
```

Figure 21.7: Training the model with hyperparameter tuning for customer segmentation

In this step, we use the train test split to separate the training set and training set. We use the Random Forest algorithm to train the model with hyperparameter tuning. We also use K-means clustering to train a model for customer segmentation. After training both models, we save those into a Pickle file.

2. ModelDeployment.py (for deployment)
   This file is responsible for deploying the trained models and serving them via a web-based interface using Streamlit. The previously saved Random Forest and K-Means models are loaded from their Pickle files. This allows the deployment file to bypass the need for retraining, saving significant computation time. We use Streamlit to build a user-friendly site with two tabs which are the Total Sales Prediction tab and Customer Segmentation tab.

```
1   import pandas as pd
2   import numpy as np
3   import pickle
4   import streamlit as st
5   import matplotlib.pyplot as plt
6   import seaborn as sns
7   from sklearn.preprocessing import StandardScaler
8
```

Figure 21.8: Importing necessary libraries

```python
#Load necessary files
def load_model():
    with open('model.pkl', 'rb') as file:
        return pickle.load(file)

def load_customer_segmentation_model():
    with open('customer_segmentation.pkl', 'rb') as file:
        return pickle.load(file)

def load_label_encoder(file_name):
    with open(file_name, 'rb') as file:
        return pickle.load(file)

def load_column():
    with open('column_names.pkl', 'rb') as file:
        return pickle.load(file)

def load_scaler():
    with open('scaler.pkl', 'rb') as file:
        return pickle.load(file)

def load_stockcode_description():
    with open('stockcode_description.pkl', 'rb') as file:
        return pickle.load(file)

def load_country_list():
    with open('country_list.pkl', 'rb') as file:
        return pickle.load(file)

def load_encoded_stockcode():
    with open('label_encoder_stockcode.pkl', 'rb') as file:
        return pickle.load(file)

def load_encoded_description():
    with open('label_encoder_description.pkl', 'rb') as file:
        return pickle.load(file)

def load_encoded_customerid():
    with open('label_encoder_customerid.pkl', 'rb') as file:
        return pickle.load(file)

def load_encoded_country():
    with open('label_encoder_country.pkl', 'rb') as file:
        return pickle.load(file)
```

Figure 21.9: Loading all the saved model and label encoder

```
#Load models and encoders
model = load_model()
rfm_model = load_customer_segmentation_model()
le_stockcode = load_encoded_stockcode()
le_description = load_encoded_description()
le_customerid = load_encoded_customerid()
le_country = load_encoded_country()
column_names = load_column()
scaler = load_scaler()

#Load the grouping of stockcode and description
stockcode_description = load_stockcode_description()

#List of countries for the dropdown
countries = load_country_list()

```

Figure 21.10: Save the loaded model and label encoder into variable

In this step, we import the essential libraries. These libraries enable us to load the saved models and label encoders, manage user inputs via the web interface and ensure that the predictions are made efficiently. After importing the required libraries, the next step is to load the pre-trained models and label encoders from the files in which they were saved. This is crucial as it allows us to utilize the models without retraining them. It can ensure faster response times during deployment. After that, we assign those pickle modules to several variables. These variables are then used throughout the application for prediction tasks.

```
#Add unseen labels dynamically
def add_unseen_label(encoder, value):
    if value not in encoder.classes_:
        new_classes = np.append(encoder.classes_, value)
        encoder.classes_ = new_classes
    return encoder.transform([value])[0]

#Prediction function for total sales
def prediction(model, x):
    x = x.reindex(columns=column_names, fill_value=0)
    return model.predict(x)

#Customer segmentation prediction function
def predict_segmentation(rfm_values, scaler, model):
    rfm_scaled = scaler.transform([rfm_values])
    return model.predict(rfm_scaled)

```

Figure 21.11: Create necessary function for prediction

In this step, we need to create functions that will use the loaded models to predict total sales and segment customers to perform predictions in the deployed application. The

function will take user inputs for total sales prediction, preprocess the inputs and return the predicted value using the Random Forest model or K-means clustering model. The reason that we create a function to add the unseen labels dynamically is to maintain the robustness of the model during deployment.

```python
#Main Streamlit app
def main():
    #Create tabs
    tab1, tab2 = st.tabs(["Prediction for Total Sales", "Customer Segmentation Analysis"])

    #Predict Total Sales tab
    with tab1:
        st.header("Prediction for Total Sales")

        #Input field for StockCode
        stockcode_input = st.text_input("Enter the Stock Code:")

        #Automatically retrieve the corresponding description
        description_input = ""
        if stockcode_input:
            if stockcode_input in stockcode_description:
                #Fetch the first description associated with the StockCode
                description_input = stockcode_description[stockcode_input][0]
                st.write(f"Product Name: {description_input}")
            else:
                st.warning("StockCode not found in the database.")

        #Input fields for UnitPrice, Quantity, CustomerID, and Year, Month
        unitprice_input = st.text_input("Enter the Unit Price (Pounds):")
        quantity_input = st.text_input("Enter the Quantity Sold:")
        customerid_input = st.text_input("Enter the Customer ID:")

        #Country dropdown for selection
        country_input = st.selectbox("Select the Country:", countries)

        year_input = st.text_input("Enter the Year:")
        month_input = st.text_input("Enter the Month (1-12):")
```

Figure 21.12: Code of Streamlit App

```python
        #Perform validation and prediction
        if st.button("Predict Total Sales"):
            try:
                #Convert inputs
                unit_price = float(unitprice_input)
                quantity = int(quantity_input)
                year = int(year_input)
                month = int(month_input)

                #Handle unseen labels for StockCode, Description, CustomerID, and Country
                stockcode_encoded = add_unseen_label(le_stockcode, stockcode_input)
                description_encoded = add_unseen_label(le_description, description_input)
                customerid_encoded = add_unseen_label(le_customerid, customerid_input)
                country_encoded = add_unseen_label(le_country, country_input)

                #Create input DataFrame
                X = pd.DataFrame([[stockcode_encoded, description_encoded, unit_price, quantity, customerid_encoded, country_encoded, year, month]],
                                 columns=['StockCode', 'Description', 'UnitPrice', 'Quantity', 'CustomerID', 'Country', 'Year', 'Month'])

                #Predict total sales
                result = prediction(model, X)
                st.success(f"The Predicted Total Sales is: {result[0]}")

            except ValueError as e:
                st.error(f"Error: {e}")
```

Figure 21.13: Code of Streamlit App

```python
    #Customer Segmentation tab
    with tab2:
        st.header("Customer Segmentation")

        recency_input = st.number_input("Enter Recency:", min_value=0)
        frequency_input = st.number_input("Enter Frequency:", min_value=0)
        monetary_input = st.number_input("Enter Monetary Value:", min_value=0)

        if st.button("Predict Customer Segment"):
            rfm_values = np.array([recency_input, frequency_input, monetary_input])
            segment = predict_segmentation(rfm_values, scaler, rfm_model)
            st.write(f"Customer belongs to segment: {segment[0]}")

            #Plot Customer Segmentation
            st.subheader("Customer Segmentation Visualization")

            #Generate data for visualization
            rfm = pd.DataFrame({
                'Recency': [recency_input],
                'Frequency': [frequency_input],
                'Monetary': [monetary_input],
                'Segment': [segment[0]]
            })
```

Figure 21.14: Code of Streamlit App

```python
            #Reload data again to gain a more accurate data
            X = pd.read_csv('onlineRetail.csv')
            X['TotalSales'] = X['Quantity'] * X['UnitPrice']
            X['InvoiceDate'] = pd.to_datetime(X['InvoiceDate'], format='%d/%m/%Y %H:%M')
            last_date = pd.to_datetime(X['InvoiceDate']).max()
            rfm_existing = X.groupby('CustomerID').agg({
                'InvoiceDate': lambda x: (last_date - x.max()).days,
                'InvoiceNo': 'nunique',
                'TotalSales': 'sum'
            }).reset_index()
            rfm_existing.columns = ['CustomerID', 'Recency', 'Frequency', 'Monetary']
            rfm_existing['Segment'] = rfm_model.predict(scaler.transform(rfm_existing[['Recency', 'Frequency', 'Monetary']]))

            #plotting cluster
            plt.figure(figsize=(10, 6))
            sns.scatterplot(data=rfm_existing, x='Recency', y='Monetary', hue='Segment', palette='viridis', alpha=0.7)
            plt.scatter(rfm['Recency'], rfm['Monetary'], color='red', s=100, label='New Customer')
            plt.title('Customer Segmentation')
            plt.xlabel('Recency')
            plt.ylabel('Monetary Value')
            plt.legend()
            st.pyplot(plt)

if __name__ == '__main__':
    main()
```

Figure 21.15: Code of Streamlit App

Lastly, we create the Streamlit app in the main function. The user interface for this project is designed to be intuitive and user-friendly. It provides two main functionalities which are predicting total sales and customer segmentation analysis. The user enters a valid stock code. If the stock code is recognized, the corresponding product description is automatically retrieved and displayed. This provides the user with contextual information about the product they're working with. The user selects the country from a dropdown list, which is encoded to handle unseen labels during prediction. The modular structure of the

code ensures that all saved models (Random Forest for total sales and K-Means for segmentation) and encoders are loaded once, reducing training time and improving the user experience by providing instant results. This design allows the application to run smoothly, delivering real-time predictions and insights, making it a powerful tool for sales forecasting and customer segmentation.

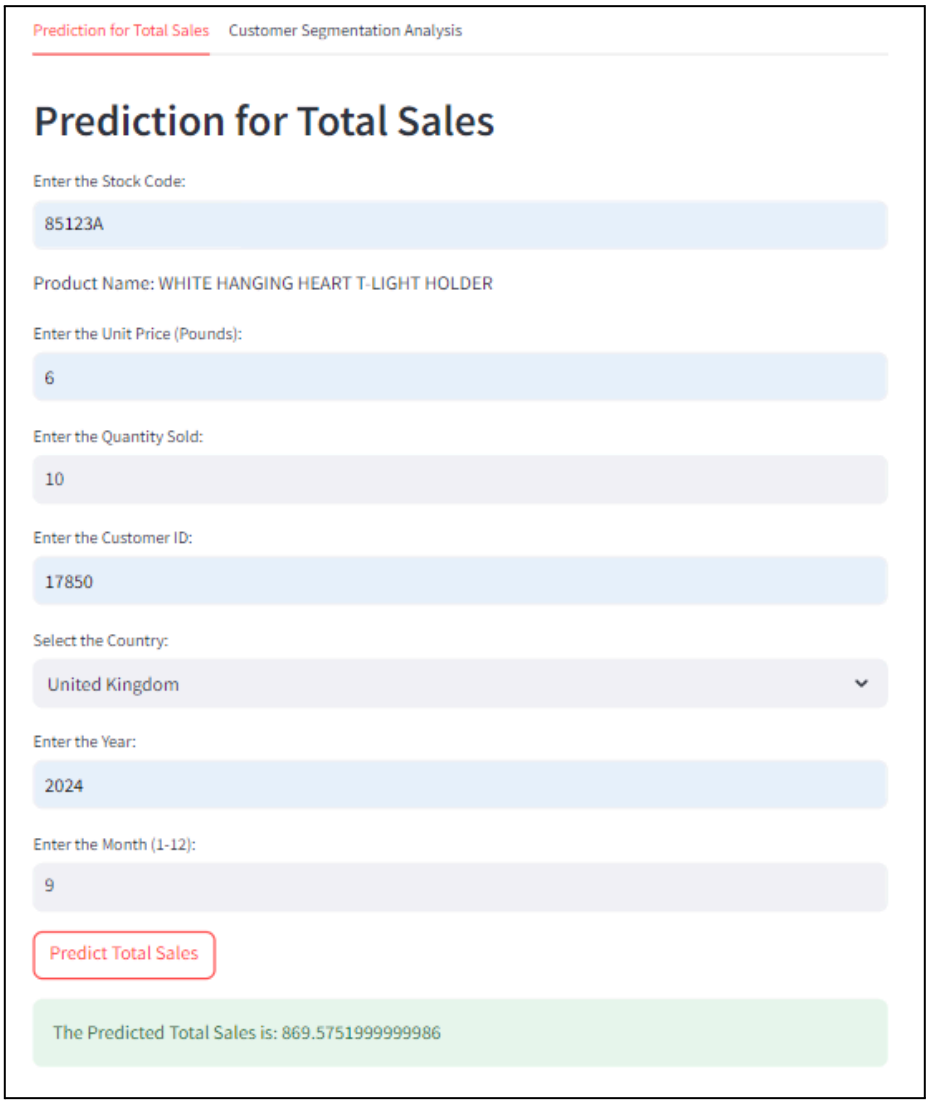


Figure 21.16: Example output for Prediction for Total Sales [1]

Figure 21.16 shows the example output for prediction for total sales. The description is generated automatically after the user inputs the stock code. After filling in all the field, user can press the button 'Predict Total Sales' and get the predicted total sales at the bottom.

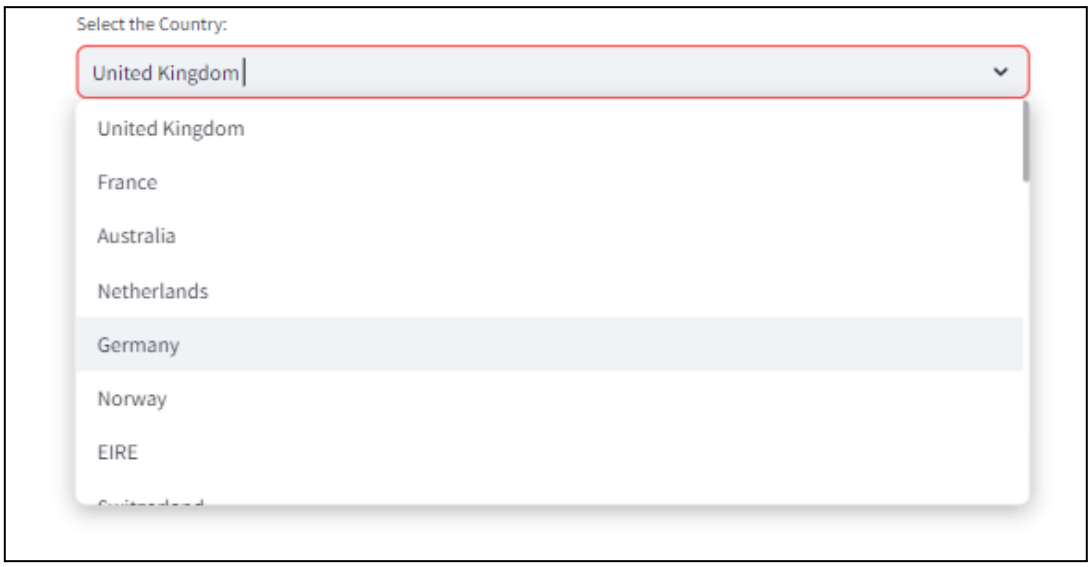


Figure 21.17: Example of user interface of Streamlit app

Figure 21.17 shows one of the user interface designs in our app. We use a drop down menu to let the user choose the country in order to increase the data accuracy and data consistency. The country list is taken from the csv file so that it can ensure the model can use a correct input to predict the total sales.

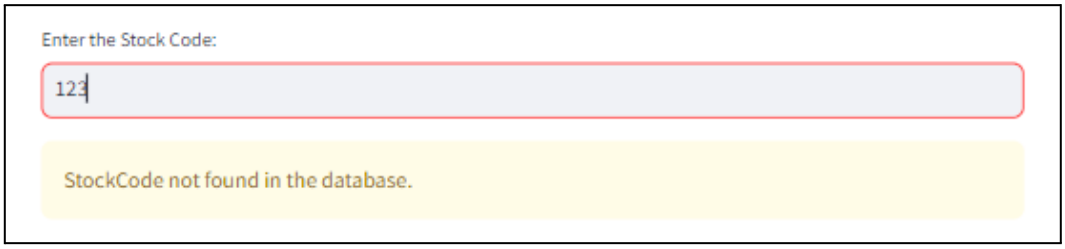


Figure 21.18: Example of user interface of Streamlit app

Figure 21.18 shows the validation for the stock. If the user enters a stock code that has never appeared in the database, it will show an error message.

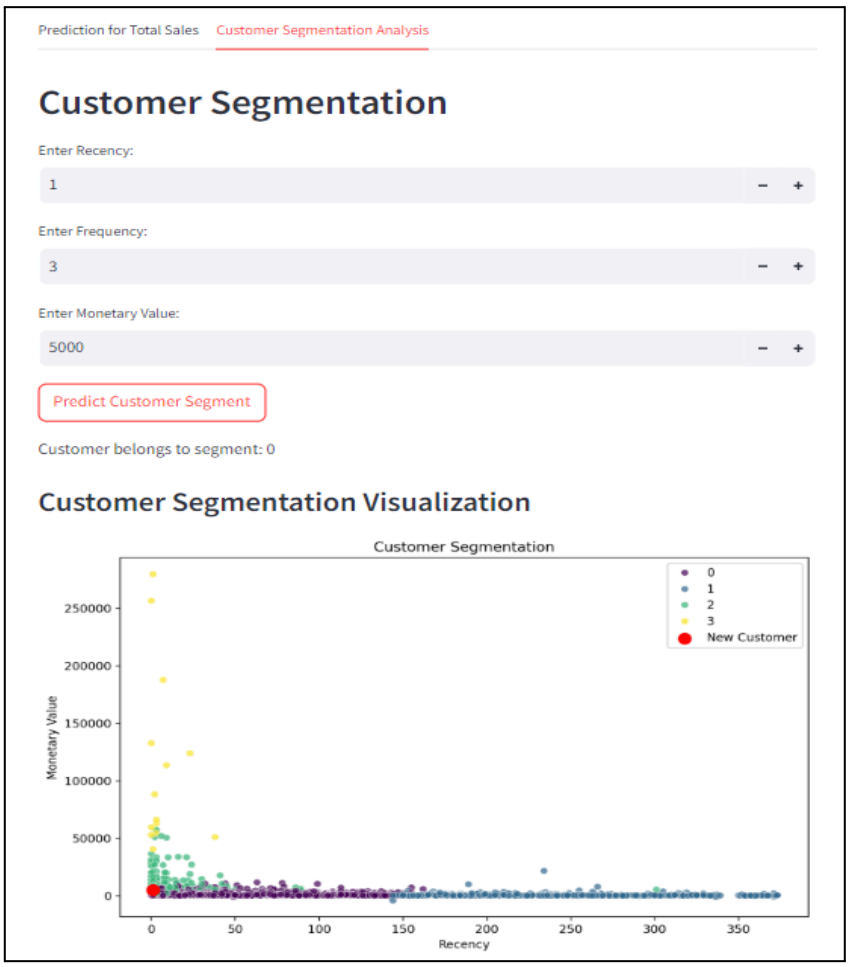


Figure 21.19: Example output for Customer Segmentation Analysis [1]

Figure 21.19 shows the example output for customer segmentation analysis. User needs to input the Recency, Frequency and Monetary to predict the customer segment.

From **Figure 21.19**, we can see that the new customer belongs to segment 0. Sales teams and business analysts can use this information to make an efficient decision making to the new customer. The red dot represents the segment that the new customer belongs to.

Recency	1
Frequency	3
Monetary	5000
Customer segmentation	0

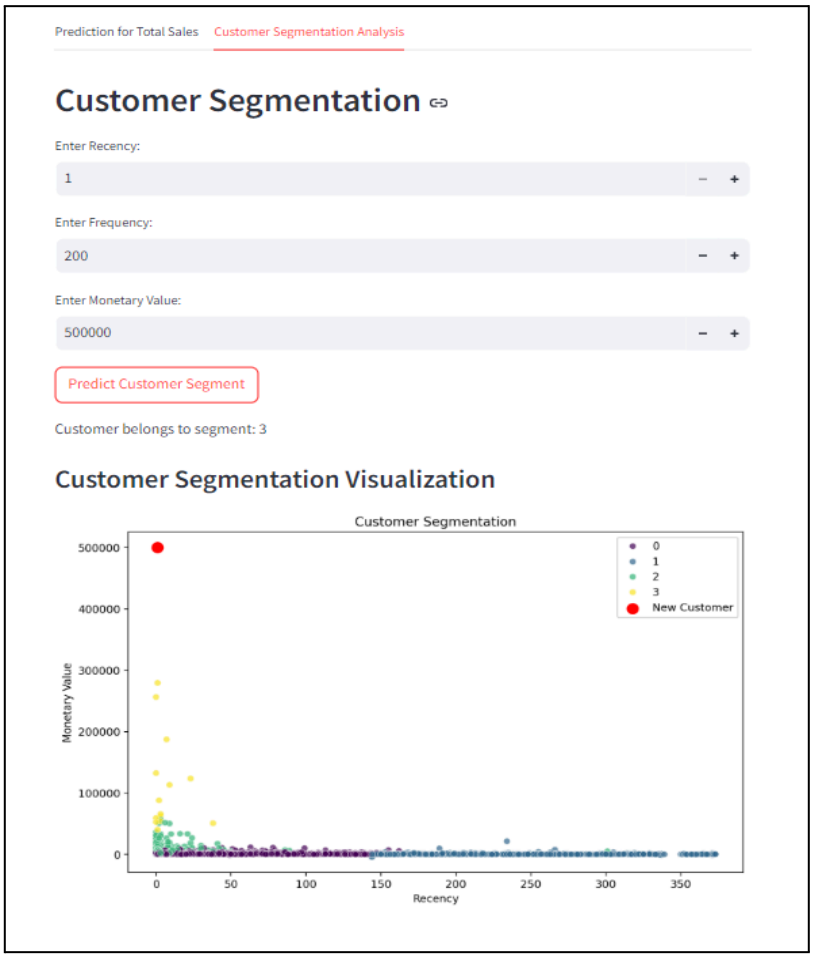


Figure 21.20: Example output for Customer Segmentation Analysis [2]

Figure 21.20 shows the example output for customer segmentation analysis. User needs to input the Recency, Frequency and Monetary to predict the customer segment.

From **Figure 21.20**, we can see that the new customer belongs to segment 0. Sales teams and business analysts can use this information to make an efficient decision making to the new customer. The red dot represents the segment that the new customer belongs to.

Recency	1
Frequency	3
Monetary	5000
Customer segmentation	0

The whole system is substantially more efficient as a result of separating the training and deployment phases. Model training can be computationally expensive, but once trained and saved, the deployment step is substantially faster because it merely loads and applies the model. The software may be instantly deployed and utilized by end users, without the need for reprocessing or retraining. Separating the training and deployment phases increases the project's modularity. It also makes maintenance easier because modifications to the model or interface don't alter the underlying data analysis. The most significant is that in real-world applications, training and development are frequently separated to decrease the strain on production systems. Our technique replicates this by guaranteeing that training takes place offline while forecasts are supplied in real time.

6.2 Project Review and Future Improvements

The project successfully created and implemented a machine learning system for predicting total sales and customer segmentation. Using a Random Forest model, the team predicted sales based on a variety of characteristics including StockCode, Description, Quantity, UnitPrice, CustomerID, Country, Year, and Month. In addition, K-Means clustering was used for consumer segmentation, with RFM analysis applied to efficiently classify customers. The Streamlit program made user engagement easier by offering distinct tabs for sales forecast and customer segmentation, allowing users to enter data and examine outcomes live.

Several major milestones contributed to the project's success. The implementation of the add_unseen_label() method solved the difficulty of managing new, unseen categories in the data, assuring the model's robustness throughout deployment. Data cleaning efforts were also critical, as they entailed dealing with negative values and outliers to improve model accuracy. Model performance was assessed using measures such as Mean Absolute Percentage Error (MAPE), R-Squared, Root Mean Square Error (RMSE), and Mean Absolute Error (MAE), with hyperparameter tweaking carried out via GridSearchCV to improve the Random Forest model.

However, the project encountered significant difficulties, such as the necessity to manage invisible labels and clean complicated data. Despite these challenges, the methods provided efficiently addressed them, resulting in a practical and dependable machine learning application.

In the future, various improvements can be made to the project. Exploring new feature engineering strategies for model augmentation might bring more insights and increase predicted accuracy. Testing newer algorithms like Gradient Boosting Machines (GBM) or XGBoost may also result in higher performance than the existing Random Forest model.

To handle more complicated cases, improving the add_unseen_label() method to properly manage additional categories and exploring incremental learning approaches might aid in adjusting to changing data without needing complete retraining. Enhancements to the user interface might include more interactive and informative graphics to help consumers comprehend predictions and segmentation findings more easily. Furthermore, enhancing the user experience with features such as data validation and feedback will make the Streamlit app more intuitive.

Scalability and deployment will need improving the Streamlit application's performance in order to efficiently manage larger datasets and greater user traffic.

Integrating with other corporate systems or APIs might help automate data entry and improve the model's real-world applicability.

Finally, installing continuous monitoring tools to track the model's performance over time and detect possible decline is critical. Regular upgrades and maintenance will keep the model relevant and effective in meeting developing company demands.

7.0 Conclusion

This project aims to meet essential business goals by creating a powerful machine learning solution for predicting total sales and customer segmentation. From the outset, it was evident that the aim was to improve decision-making capabilities by delivering accurate sales predictions and detailed customer segmentation. These insights were supposed to help make better strategic decisions, optimize marketing activities, and boost overall operational efficiency.

During the data understanding phase, a thorough study of the information identified critical variables that influence sales and consumer behavior. This entailed investigating elements such as StockCode, Quantity, UnitPrice, and CustomerID, as well as comprehending the distribution and quality of the information. The data preparation stage was critical for cleaning and preparing the data to assure its appropriateness for modeling. To preserve the model's robustness, this includes coping with negative values, outliers, categorical variables, and unseen labels.

The modeling step included using Random Forest to estimate total sales and K-Means clustering to segment customers. Random Forest was chosen for its robustness and capacity to handle complicated feature interactions, whilst K-Means clustering gave useful insights into consumer behavior by categorizing customers based on recency, frequency, and monetary worth. During this phase, critical activities such as hyperparameter tuning and model improvement were carried out to improve accuracy and performance.

The models were evaluated using several metrics, including Mean Absolute Percentage Error (MAPE), R-Squared, Root Mean Square Error (RMSE), and Mean Absolute Error (MAE). These indicators offered a clear picture of the model's ability to anticipate sales and segment customers. The findings showed that the models worked effectively and met the business objectives established at the start of the project.

Finally, the deployment process entailed integrating the models into a user-friendly Streamlit interface. This application's easy interface allowed users to interact with data, examine forecasts, and study consumer segments. The addition of features such as dynamic handling of unseen labels and a well-designed user interface guaranteed that the program was both practical and accessible.

The Random Forest model for sales prediction revealed notable capabilities, including the capacity to handle complicated feature interactions and to withstand overfitting. The model produced trustworthy and accurate predictions by utilizing numerous decision trees, as shown by favorable assessment measures such as Mean

Absolute Percentage Error (MAPE), R-Squared, Root Mean Square Error (RMSE), and Mean Absolute Error (MAE). The model's performance was improved by thorough hyperparameter tweaking, which helped to achieve its high accuracy.

Despite these benefits, the models did have certain drawbacks. The Random Forest model, while resilient, can be computationally demanding, necessitating extensive resources for big datasets. Furthermore, it may not always detect extremely small patterns in the data, thereby limiting its forecast accuracy in some instances.

The K-Means clustering technique for customer segmentation successfully classified clients based on recency, frequency, and monetary values, providing meaningful insights into customer behavior. The ideal number of clusters, defined by WCSS and silhouette scores, enabled meaningful segmentation that can influence targeted marketing tactics and boost consumer engagement.

The K-Means clustering model, on the other hand, implies that clusters are spherical and evenly sized, which may not necessarily correspond to the underlying structure of the data. This assumption may result in inadequate segmentation if the real customer categories are not well-separated or differ considerably in size. Furthermore, the requirement to predefine the number of clusters might be a constraint because it does not always capture the most natural groups in the data.

Overall, the project achieved its objectives by providing actionable insights and predictive capabilities that are in line with business requirements. The process of understanding the business needs and delivering the final solution guaranteed that the models were well-prepared, precisely evaluated, and efficiently executed. Future enhancements will build on this foundation, improving the model's capabilities and the application's functionality to accommodate changing business requirements and data dynamics.

8.0 Reference

1. Padhma. (2024, September 4). *A comprehensive introduction to evaluating regression models.* Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/10/evaluation-metric-for-regression-models/ [viewed by 7 September 2024]

2. *Numeracy, Maths and Statistics - Academic Skills kit.* (n.d.). https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/coefficient-of-determination-r-squared.html#:~:text=To%20calculate%20R2%20you,into%20the%20regression%20line%20equation [viewed by 7 September 2024]

3. Chouinard, J. (2023, September 25). *Decision Trees in Machine Learning (with Python Examples)*. JC Chouinard. https://www.jcchouinard.com/decision-trees-in-machine-learning/ [viewed by 8 September 2024]

Appendix

1. Data Science Assignment
   Folder including Jupyter Notebook Source code, Python file, Pickle file and CSV file.