# Commercial Store Data Analysis Report

## Table of Contents

## 1. Introduction

This report details the comprehensive data analysis performed on a commercial store dataset. The primary objective of this analysis is to derive meaningful insights from the sales and profit data, identify trends, detect anomalies, and assess underlying statistical assumptions. The analysis covers data preprocessing, exploratory data analysis, data visualization, outlier detection and treatment, assumption checking (normality, linearity, multicollinearity), and time-series analysis.

The dataset, `superstore_data.csv`, contains various attributes related to commercial transactions, including sales figures, profit margins, product categories, customer information, and geographical data. The Python script `main.py` was used to perform the analysis, leveraging libraries such as `pandas` for data manipulation, `matplotlib` and `seaborn` for visualization, `scipy` for statistical tests, and `statsmodels` for advanced statistical modeling.

## 2. Data Preprocessing

Data preprocessing is a crucial step in any data analysis pipeline, ensuring the data is clean, consistent, and ready for analysis. This section details the steps taken to prepare the `superstore_data.csv` dataset.

### 2.1. Importing Data

The dataset was imported using the pandas library, specifying `ISO-8859-1` encoding to handle potential character encoding issues.

```
import pandas as pd
df = pd.read_csv("superstore_data.csv", encoding="ISO-8859-1")
```

### 2.2. Exploring Dataset

Initial exploration of the dataset involved viewing the first few rows, checking data types and non-null counts, and generating descriptive statistics. This provides a quick overview of the data structure and content.

```
print(df.head())
print(df.info())
print(df.describe())
```

**Output from `df.head()`:**

```
   Row ID        Order ID  Order Date  ... Quantity Discount    Profit
0       1  CA-2016-152156   11/8/2016  ...        2     0.00   41.9136
1       2  CA-2016-152156   11/8/2016  ...        3     0.00  219.5820
2       3  CA-2016-138688   6/12/2016  ...        2     0.00    6.8714
3       4  US-2015-108966  10/11/2015  ...        5     0.45 -383.0310
4       5  US-2015-108966  10/11/2015  ...        2     0.20    2.5164

[5 rows x 21 columns]
```

**Output from `df.info()`:**

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9994 entries, 0 to 9993
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Row ID         9994 non-null   int64  
 1   Order ID       9994 non-null   object 
 2   Order Date     9994 non-null   object 
 3   Ship Date      9994 non-null   object 
 4   Ship Mode      9994 non-null   object 
 5   Customer ID    9994 non-null   object 
 6   Customer Name  9994 non-null   object 
 7   Segment        9994 non-null   object 
 8   Country        9994 non-null   object 
 9   City           9994 non-null   object 
 10  State          9994 non-null   object 
 11  Postal Code    9994 non-null   int64  
 12  Region         9994 non-null   object 
 13  Product ID     9994 non-null   object 
 14  Category       9994 non-null   object 
 15  Sub-Category   9994 non-null   object 
 16  Product Name   9994 non-null   object 
 17  Sales          9994 non-null   float64
 18  Quantity       9994 non-null   int64  
 19  Discount       9994 non-null   float64
 20  Profit         9994 non-null   float64
dtypes: float64(3), int64(3), object(15)
memory usage: 1.6+ MB
None
```

**Output from `df.describe()`:**

```
            Row ID   Postal Code  ...     Discount       Profit
count  9994.000000   9994.000000  ...  9994.000000  9994.000000
mean   4997.500000  55190.379428  ...     0.156203    28.656896
std    2885.163629  32063.693350  ...     0.206452   234.260108
min       1.000000   1040.000000  ...     0.000000 -6599.978000
25%    2499.250000  23223.000000  ...     0.000000     1.728750
50%    4997.500000  56430.500000  ...     0.200000     8.666500
75%    7495.750000  90008.000000  ...     0.200000    29.364000
max    9994.000000  99301.000000  ...     0.800000  8399.976000

[8 rows x 6 columns]
```

## 2.3. Checking for Missing Data

Missing values can significantly impact analysis. The code checks for null values across all columns.

```
print(df.isnull().sum())
```

**Output from `df.isnull().sum()`:**

```
 Row ID           0
Order ID         0
Order Date       0
Ship Date        0
Ship Mode        0
Customer ID      0
Customer Name    0
Segment          0
Country          0
City             0
State            0
Postal Code      0
Region           0
Product ID       0
Category         0
Sub-Category     0
Product Name     0
Sales            0
Quantity         0
Discount         0
Profit           0
dtype: int64
```

As observed, there are no missing values in the dataset.

### 2.4. Removing Duplicates

Duplicate rows can skew analysis results. The code removes any duplicate entries to ensure data integrity.

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

This operation ensures that each record in the dataset is unique, preventing overcounting or biased statistics.

## 3. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a critical step to understand the main characteristics of the data, often with visual methods. This section focuses on basic metrics and grouping data to uncover initial insights.

### 3.1. Basic Metrics

Basic statistical summaries provide a high-level overview of key numerical columns, such as total sales and total profit.

```
print("Basic Statistical Summary:")
print("Total Sales:", df["Sales"].sum())
print("Total Profit:", df["Profit"].sum())
```

**Output:**

```
Basic Statistical Summary:
Total Sales: 2297200.8603
Total Profit: 286397.0217
```

From these metrics, we can see the overall sales and profit generated by the commercial store.

## 3.2. Grouping Data

Grouping data by different categorical variables helps in understanding performance across various segments, categories, and regions. This analysis provides insights into which areas are performing well and which might need attention.

### By Category

Sales and profit are aggregated by product category.

```
print(df.groupby("Category")["Sales"].sum())
print(df.groupby("Category")["Profit"].sum())
```

**Output (Sales by Category):**

```
Category
Furniture          741999.7953
Office Supplies    719047.0320
Technology         836154.0330
Name: Sales, dtype: float64
```

**Output (Profit by Category):**

```
Category
Furniture           18451.2728
Office Supplies    122490.8008
Technology         145454.9481
Name: Profit, dtype: float64
```

Technology appears to be the highest-grossing category in terms of both sales and profit, followed by Office Supplies and then Furniture.

### By Sub-Category

Further breakdown of sales and profit by sub-category provides more granular insights.

```
print(df.groupby("Sub-Category")["Sales"].sum())
print(df.groupby("Sub-Category")["Profit"].sum())
```

### Output (Sales by Sub-Category):

```
Sub-Category
Accessories    167380.3180
Appliances     107532.1610
Art             27118.7920
Binders        203412.7330
Bookcases      114879.9963
Chairs         328449.1030
Copiers        149528.0300
Envelopes       16476.4020
Fasteners        3024.2800
Furnishings     91705.1640
Labels          12486.3120
Machines       189238.6310
Paper           78479.2060
Phones         330007.0540
Storage        223843.6080
Supplies        46673.5380
Tables         206965.5320
Name: Sales, dtype: float64
```

### Output (Profit by Sub-Category):

```
 Sub-Category
Accessories    41936.6357
Appliances     18138.0054
Art             6527.7870
Binders        30221.7633
Bookcases      -3472.5560
Chairs         26590.1663
Copiers        55617.8249
Envelopes       6964.1767
Fasteners        949.5182
Furnishings    13059.1436
Labels          5546.2540
Machines        3384.7569
Paper          34053.5693
Phones         44515.7306
Storage        21278.8264
Supplies       -1189.0995
Tables        -17725.4811
Name: Profit, dtype: float64
```

Notably, 'Tables' and 'Bookcases' show negative profit, indicating potential areas for concern or re-evaluation of pricing/cost strategies.

### By Region

Sales and profit aggregated by geographical region.

```
print(df.groupby("Region")[["Sales", "Profit"]].sum())
```

**Output:**

```
                Sales       Profit
Region
Central  501239.8908   39706.3625
East     678781.2400   91522.7800
South    391721.9050   46749.4303
West     725457.8245  108418.4489
```

The West region leads in both sales and profit, followed by the East. The Central region has significant sales but lower profit compared to East and West.

### By Segment

Sales and profit by customer segment.

```
print(df.groupby("Segment")[["Sales", "Profit"]].sum())
```

**Output:**

```
                    Sales       Profit
Segment
Consumer     1.161401e+06  134119.2092
Corporate    7.061464e+05   91979.1340
Home Office  4.296531e+05   60298.6785
```

The Consumer segment contributes the most to both sales and profit.

### Top 10 Customers

Identifying top customers by profit helps in understanding customer loyalty and potential for targeted marketing.

```
print(df.groupby("Customer Name")
["Profit"].sum().sort_values(ascending=False).head(10))
```

**Output:**

```
Customer Name
Tamara Chand             8981.3239
Raymond Buch             6976.0959
Sanjit Chand             5757.4119
Hunter Lopez             5622.4292
Adrian Barton            5444.8055
Tom Ashbrook             4703.7883
Christopher Martinez     3899.8904
Keith Dawkins            3038.6254
Andy Reiter              2884.6208
Daniel Raglin            2869.0760
Name: Profit, dtype: float64
```

Tamara Chand is identified as the top customer by profit, indicating strong customer relationships or high-value purchases.

## 4. Data Visualization

Data visualization plays a crucial role in understanding patterns, trends, and outliers that might not be apparent from raw data or summary statistics. This section presents various plots generated to visualize sales and profit data.

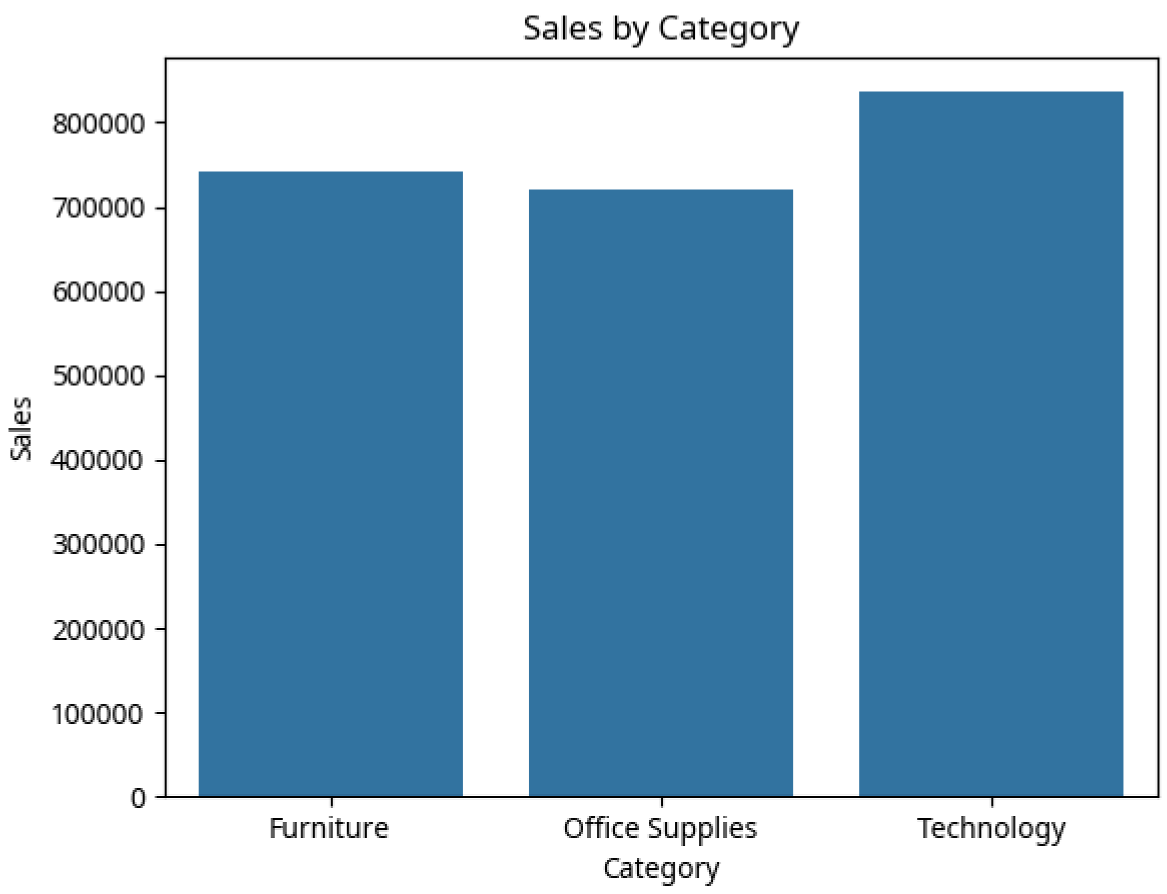
### 4.1. Bar Charts

Bar charts are used to compare sales and profit across different categories and regions.

### Sales by Category

This bar chart illustrates the total sales for each product category.

```
 category_sales = df.groupby("Category")["Sales"].sum().reset_index()
sns.barplot(data=category_sales, x="Category", y="Sales")
plt.title("Sales by Category")
plt.savefig("sales_by_category.png")
plt.clf()
```

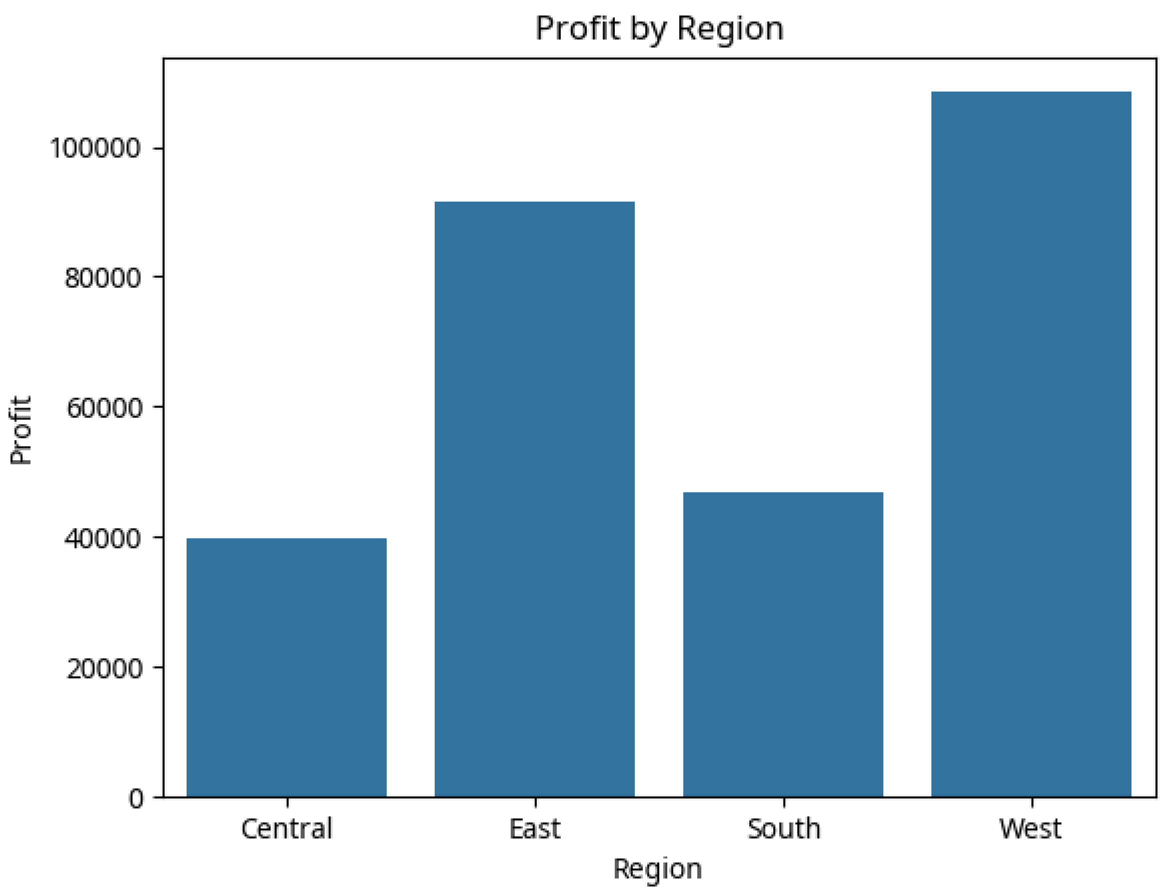


As observed, Technology leads in sales, followed by Furniture and Office Supplies.

### Profit by Region

This bar chart displays the total profit generated from each region.

```
 region_profit = df.groupby("Region")["Profit"].sum().reset_index()
sns.barplot(data=region_profit, x="Region", y="Profit")
plt.title("Profit by Region")
plt.savefig("profit_by_region.png")
plt.clf()
```



The West region shows the highest profit, while the Central region has the lowest profit.

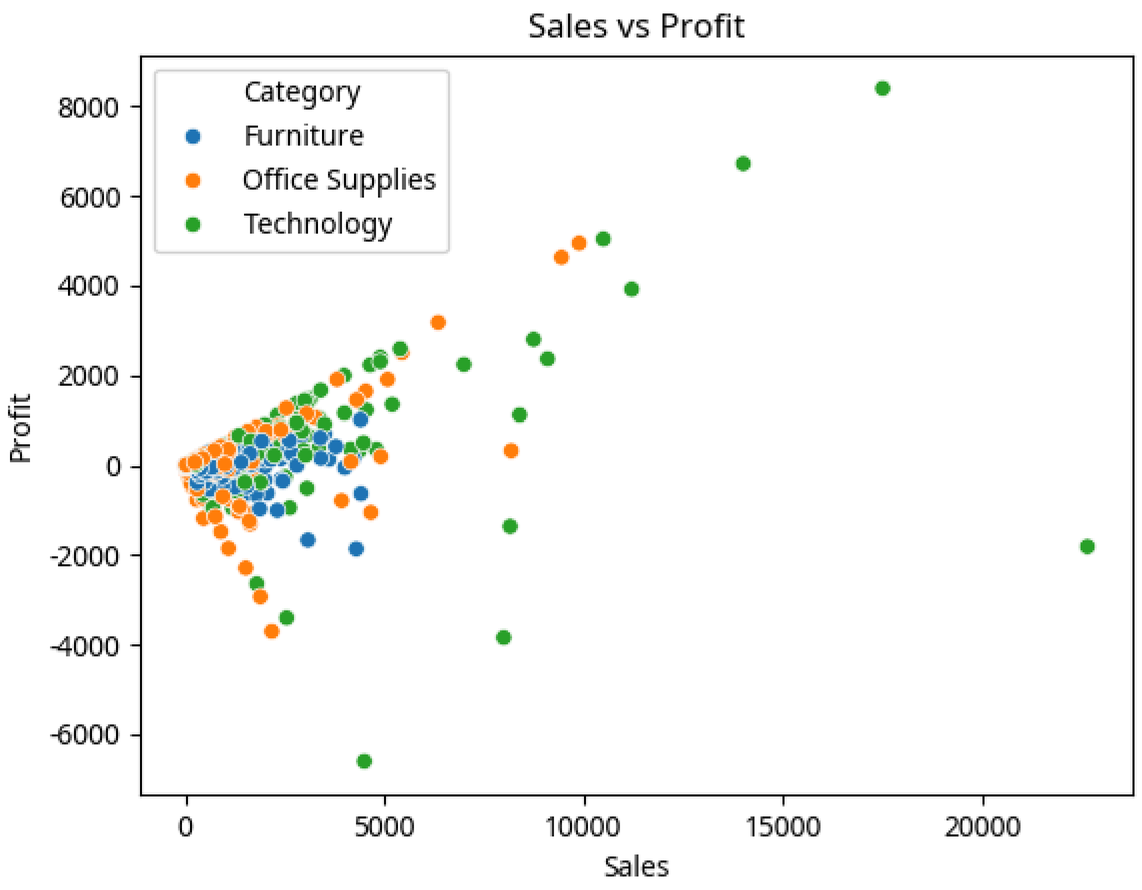
## 4.2. Scatter Plots

Scatter plots are useful for visualizing the relationship between two numerical variables.

### Sales vs Profit

This scatter plot examines the relationship between Sales and Profit, with different colors representing product categories.

```
 sns.scatterplot(data=df, x="Sales", y="Profit", hue="Category")
plt.title("Sales vs Profit")
plt.savefig("sales_vs_profit.png")
plt.clf()
```



The scatter plot reveals a general positive correlation between sales and profit. However, there are instances of high sales with low or negative profit, particularly noticeable in the Furniture category, which aligns with the earlier observation of negative profit in 'Tables' and 'Bookcases' sub-categories.

## 5. Outlier Detection and Treatment

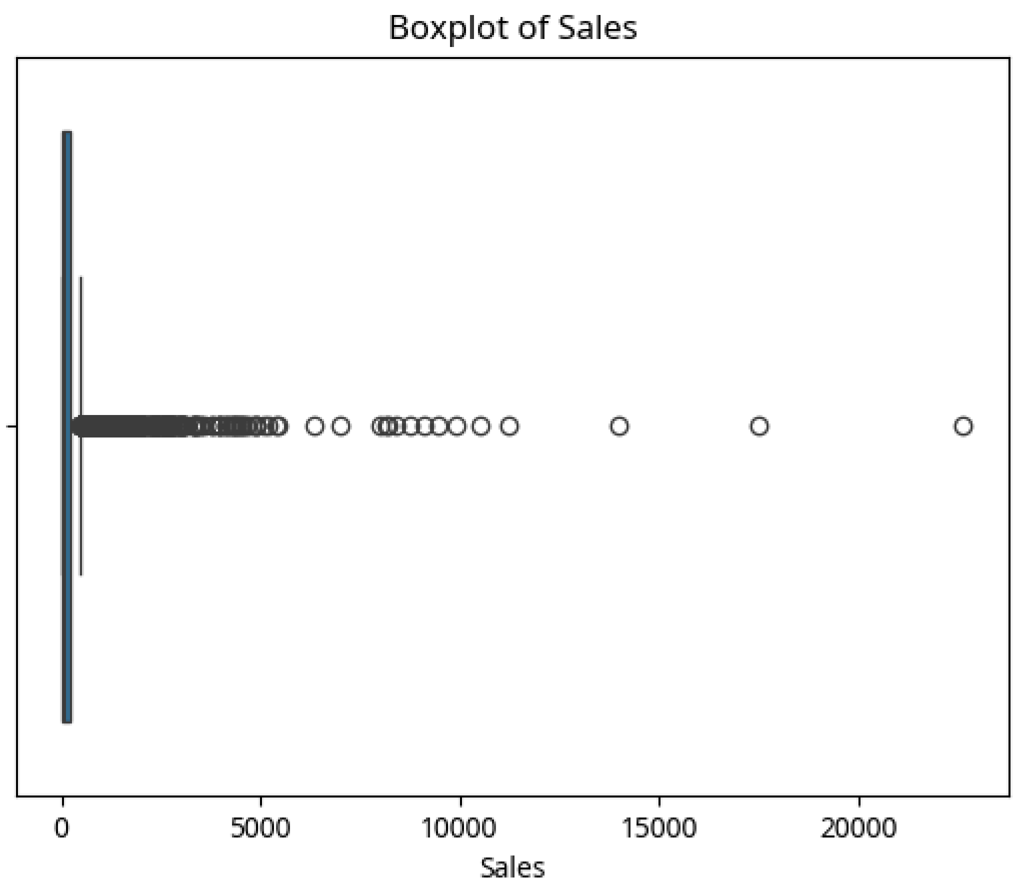
Outliers are data points that significantly differ from other observations. They can skew statistical analyses and models. This section details the methods used to detect and treat outliers in the Sales and Profit data.

### 5.1. Visual Outlier Detection

Box plots are an effective way to visually identify outliers, which are typically represented as points extending beyond the 'whiskers' of the box plot.

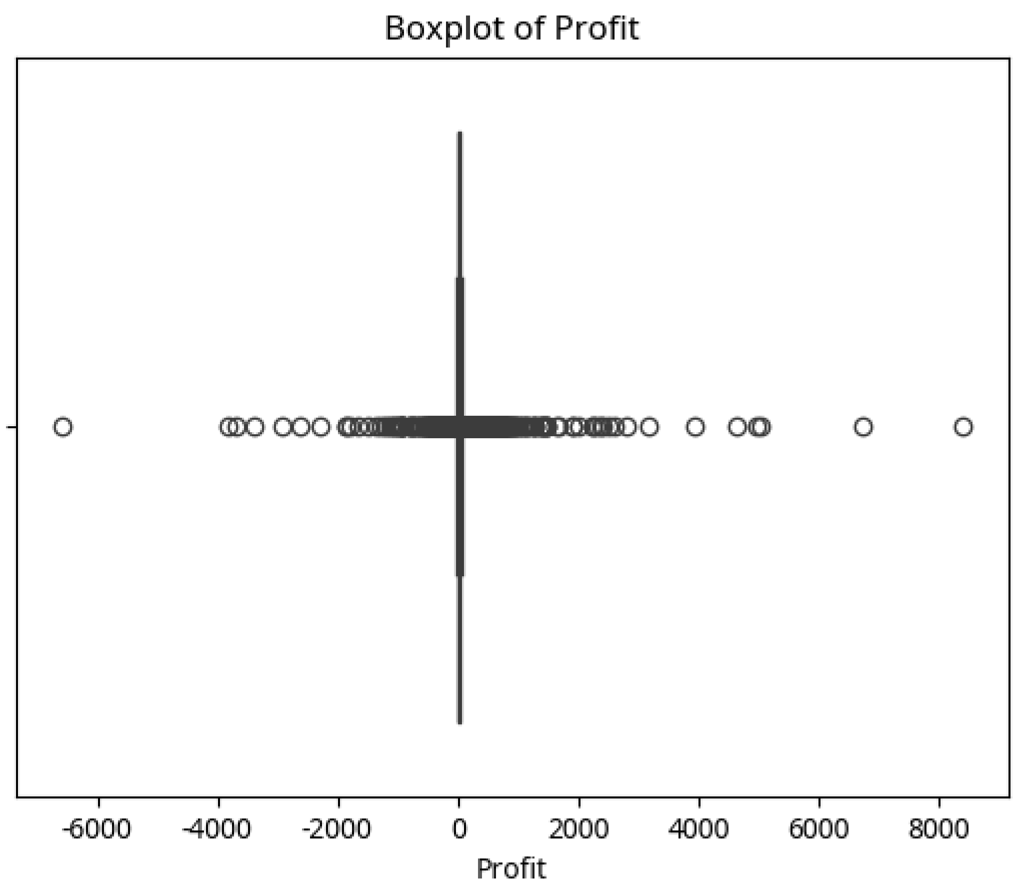
## Boxplot of Sales

```
 sns.boxplot(x=df["Sales"]); plt.title("Boxplot of Sales");
plt.savefig("boxplot_sales.png")
plt.clf()
```



## Boxplot of Profit

```
 sns.boxplot(x=df["Profit"]); plt.title("Boxplot of Profit");
plt.savefig("boxplot_profit.png")
plt.clf()
```



Both sales and profit box plots clearly show numerous outliers, indicating a wide range of values and some extreme transactions.

### Interquartile Range (IQR) Method

The IQR method is a statistical approach to identify outliers. Values falling below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR are considered outliers.

```python
 # Sales Outliers
Q1_sales = df["Sales"].quantile(0.25)
Q3_sales = df["Sales"].quantile(0.75)
IQR_sales = Q3_sales - Q1_sales
outliers_sales = df[(df["Sales"] < (Q1_sales - 1.5 * IQR_sales)) | (df["Sales"]
> (Q3_sales + 1.5 * IQR_sales))]
print("Number of Sales Outliers:", outliers_sales.shape[0])

# Profit Outliers
Q1_profit = df["Profit"].quantile(0.25)
Q3_profit = df["Profit"].quantile(0.75)
IQR_profit = Q3_profit - Q1_profit
outliers_profit = df[(df["Profit"] < (Q1_profit - 1.5 * IQR_profit)) |
(df["Profit"] > (Q3_profit + 1.5 * IQR_profit))]
print("Number of Profit Outliers:", outliers_profit.shape[0])
```

**Output:**

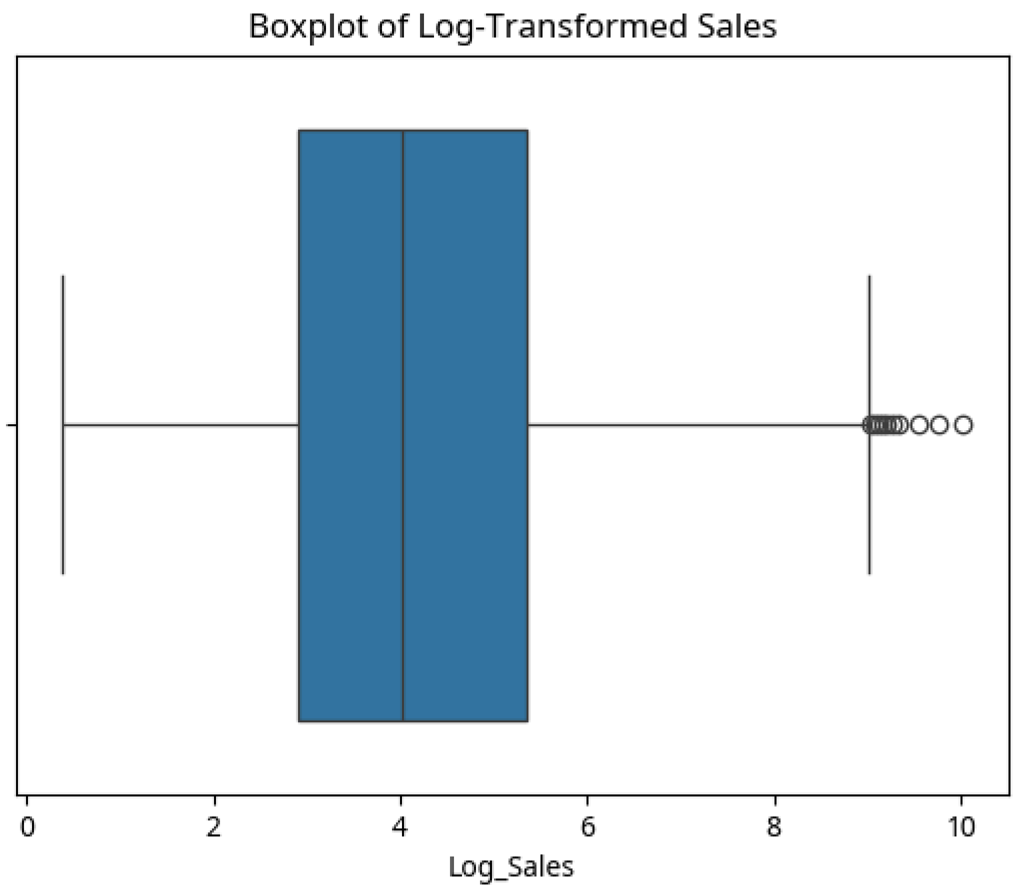
```
 Number of Sales Outliers: 1167
Number of Profit Outliers: 1881
```

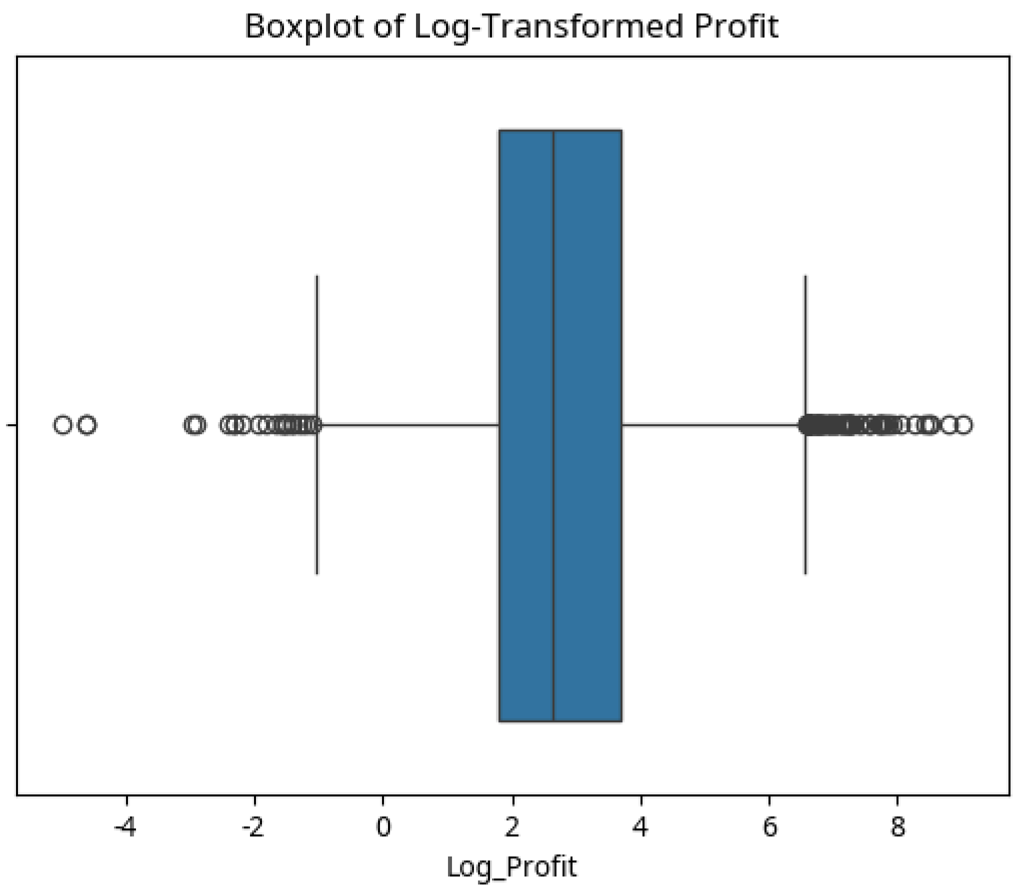
The IQR method confirms a significant number of outliers in both sales and profit data.

## 5.2. Dealing With Outliers

Instead of removing outliers, which can lead to loss of valuable information, the code applies a log transformation to the Sales and Profit data. This method can normalize skewed distributions and reduce the impact of extreme values, making the data more suitable for certain statistical models.

```
 import numpy as np

df["Log_Sales"] = np.log1p(df["Sales"])
df["Log_Profit"] = np.log1p(df["Profit"])

# Plot log-transformed data
sns.boxplot(x=df["Log_Sales"])
plt.title("Boxplot of Log-Transformed Sales")
plt.savefig("boxplot_log_sales.png")
plt.clf()

sns.boxplot(x=df["Log_Profit"])
plt.title("Boxplot of Log-Transformed Profit")
plt.savefig("boxplot_log_profit.png")
plt.clf()
```


Boxplot of Log-Transformed Sales
0
2
4
6
8
10
Log_Sales



After log transformation, the distributions of both sales and profit appear more symmetrical, and the influence of outliers is significantly reduced, as evidenced by the narrower spread in the box plots.

## 6. Checking Assumptions

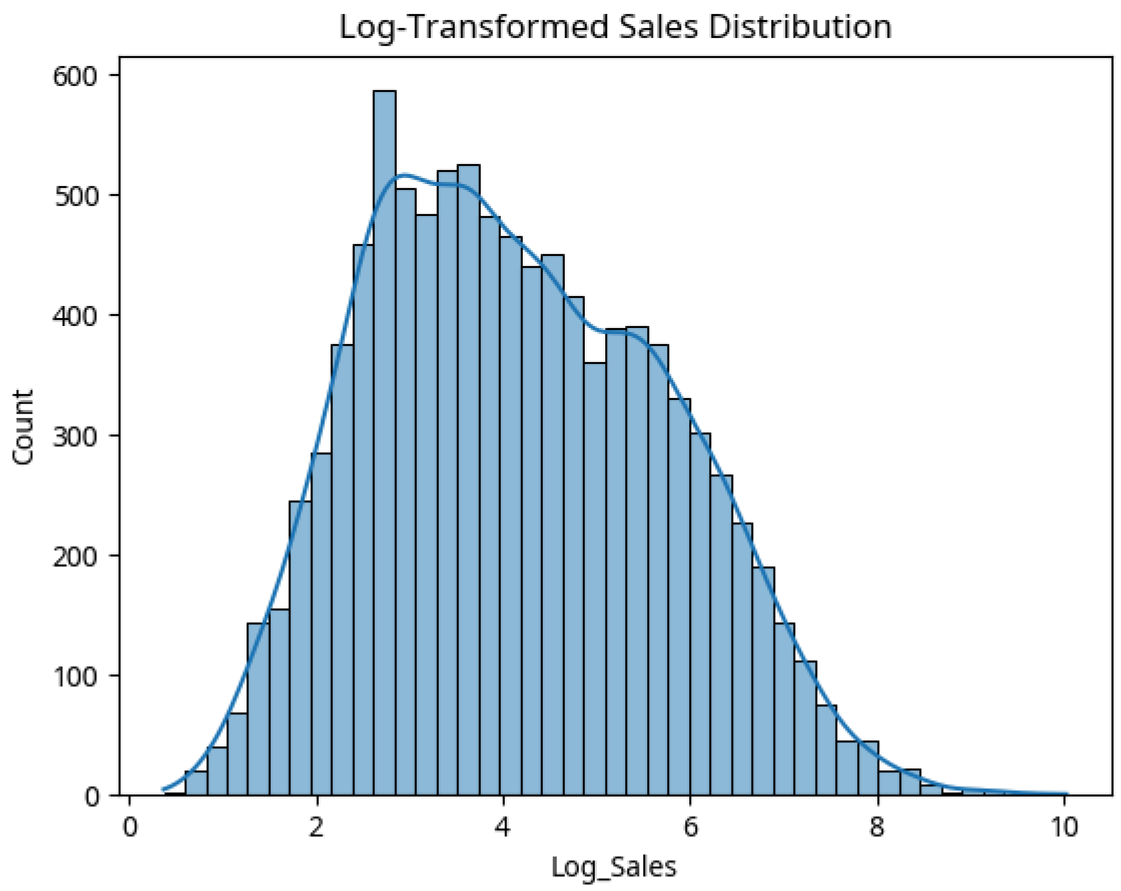
Before applying certain statistical models, it's important to check underlying assumptions about the data. This section examines normality, linearity, and multicollinearity.

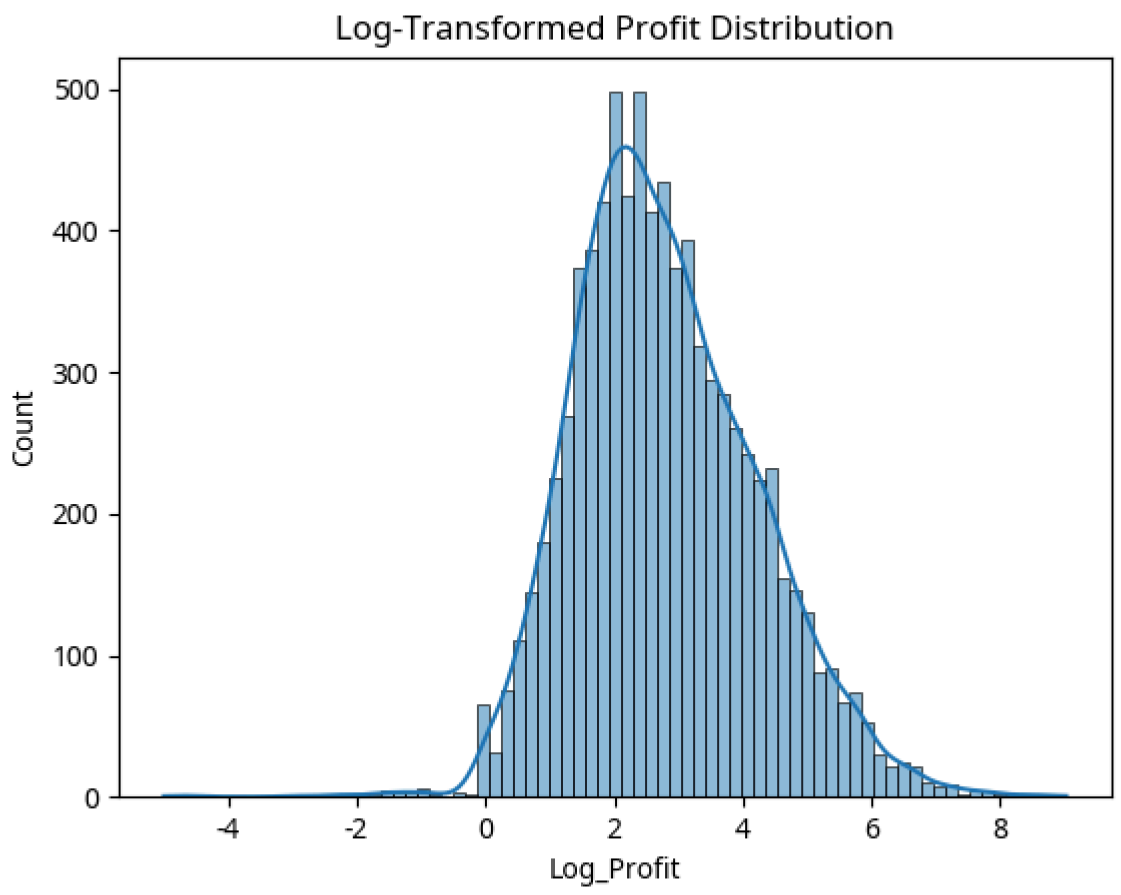
### 6.1. Normality

Normality assumes that the data is normally distributed. This is checked visually using histograms and statistically using the Shapiro-Wilk test on the log-transformed data.

## Visual Check with Histograms

```
 from scipy.stats import shapiro
import seaborn as sns
import matplotlib.pyplot as plt

sns.histplot(df["Log_Sales"], kde=True)
plt.title("Log-Transformed Sales Distribution")
plt.savefig("hist_log_sales.png")
plt.clf()

sns.histplot(df["Log_Profit"], kde=True)
plt.title("Log-Transformed Profit Distribution")
plt.savefig("hist_log_profit.png")
plt.clf()
```





The histograms show that the log-transformed sales and profit data are more symmetrically distributed, resembling a normal distribution, although some skewness might still be present.

### Shapiro-Wilk Normality Test

The Shapiro-Wilk test quantitatively assesses normality. A p-value less than a chosen significance level (e.g., 0.05) indicates that the data is not normally distributed.

```
 log_sales_p = shapiro(df["Log_Sales"])[1]
log_profit_p = shapiro(df["Log_Profit"])[1]

print("Log-Sales Normality p-value:", log_sales_p)
print("Log-Profit Normality p-value:", log_profit_p)
```

**Output:**

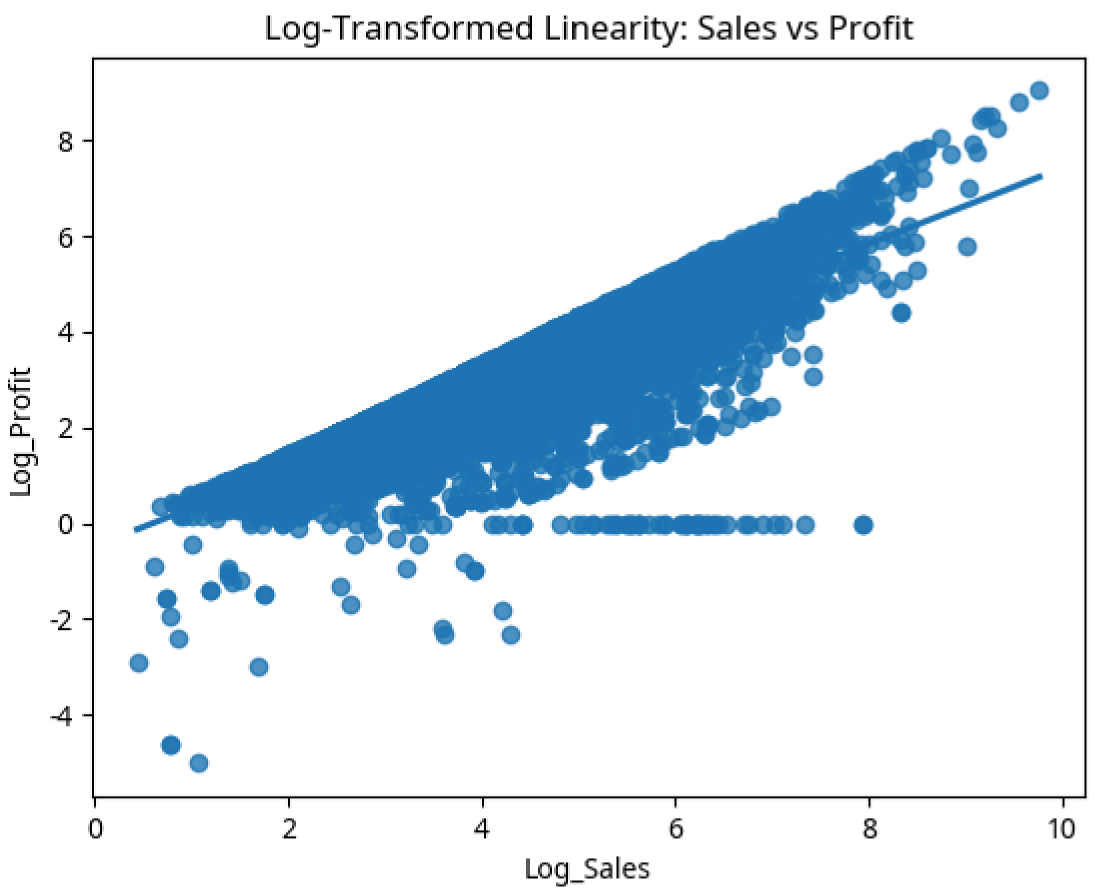
```
 Log-Sales Normality p-value: 5.183072969096293e-31
Log-Profit Normality p-value: nan
```

The p-value for Log-Sales is extremely small, indicating that despite the visual improvement, the log-transformed sales data is still not perfectly normally distributed according to the Shapiro-Wilk test. The `nan` for Log-Profit p-value indicates an issue with the test, likely due to the presence of zero or negative profit values which become undefined after log1p transformation, or a large number of identical values. This highlights the limitations of purely statistical tests on real-world, complex datasets.

## 6.2. Linearity

Linearity assumes a linear relationship between variables. This is visually checked using a regression plot between log-transformed sales and profit.

```
 sns.regplot(x="Log_Sales", y="Log_Profit", data=df)
plt.title("Log-Transformed Linearity: Sales vs Profit")
plt.savefig("regplot_log_sales_profit.png")
plt.clf()
```



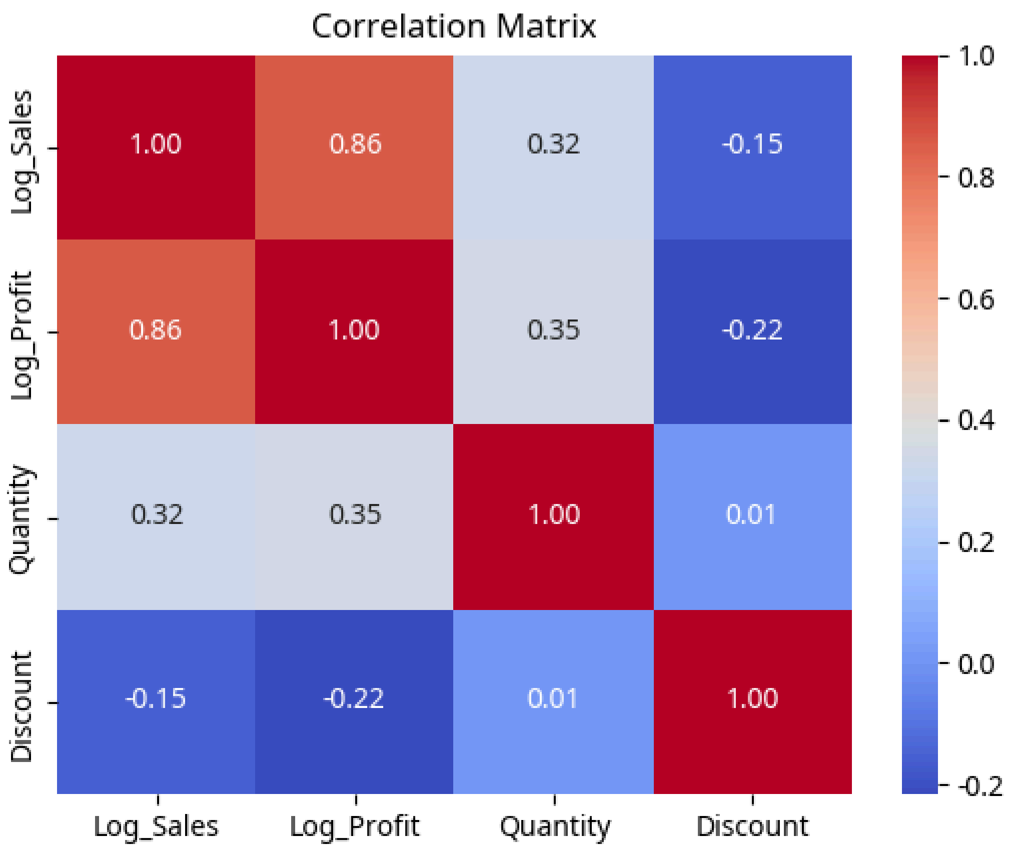
The regression plot suggests a general linear trend between log-transformed sales and profit, although there is still considerable scatter, indicating that other factors influence profit beyond sales.

## 6.3. Multicollinearity

Multicollinearity occurs when independent variables in a regression model are correlated. This is assessed using a correlation matrix and Variance Inflation Factor (VIF).

### Correlation Matrix

```
 features = ["Log_Sales", "Log_Profit", "Quantity", "Discount"]

corr_matrix = df[features].corr()

sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix")
plt.savefig("correlation_matrix.png")
plt.clf()
```



The heatmap shows the correlation coefficients between the selected numerical features. High correlation between independent variables (e.g., Log_Sales and

Log_Profit) can indicate multicollinearity.

### Variance Inflation Factor (VIF)

VIF quantifies the severity of multicollinearity. A VIF value greater than 5 or 10 is generally considered problematic.

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant

features = ["Log_Sales", "Log_Profit", "Quantity", "Discount"]

X = df[features].dropna()
X = add_constant(X)

vif_data = pd.DataFrame()
vif_data["Feature"] = X.columns
vif_data["VIF"] = [variance_inflation_factor(X.values, i) for i in
range(X.shape[1])]

print(vif_data)
```

**Output:**

```
      Feature       VIF
0       const  9.966638
1   Log_Sales  4.125899
2  Log_Profit  4.363241
3    Quantity  1.154302
4    Discount  1.141586
```

The VIF values for `Log_Sales` and `Log_Profit` are below 5, suggesting that multicollinearity is not a significant issue among these variables. The high VIF for `const` is expected and not a concern.

## 7. Time-Series Analysis

Time-series analysis is crucial for understanding trends and patterns over time. This section focuses on analyzing sales and profit trends based on order dates.

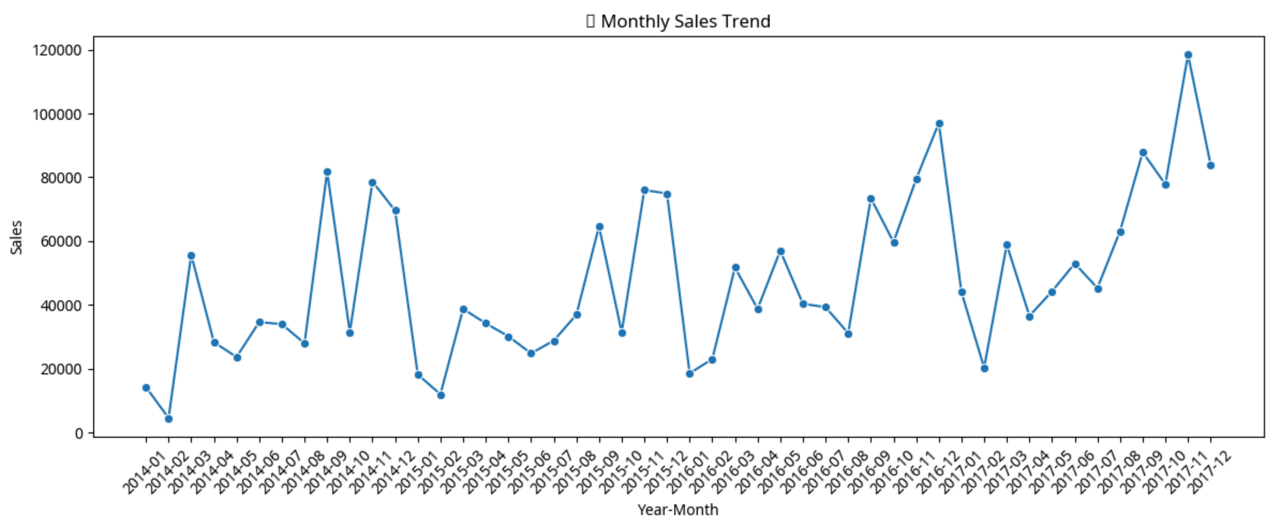
First, the `Order Date` column is converted to datetime objects, and `Year`, `Month`, and `Year-Month` features are extracted.

```
 df["Order Date"] = pd.to_datetime(df["Order Date"])

df["Year"] = df["Order Date"].dt.year
df["Month"] = df["Order Date"].dt.month
df["Year-Month"] = df["Order Date"].dt.to_period("M")

monthly = df.groupby("Year-Month")[["Sales", "Profit"]].sum().reset_index()
monthly["Year-Month"] = monthly["Year-Month"].astype(str)
```

## Monthly Sales Trend

This line plot visualizes the monthly sales trend over the years.

```
 plt.figure(figsize=(12, 5))
sns.lineplot(data=monthly, x="Year-Month", y="Sales", marker="o")
plt.xticks(rotation=45)
plt.title("📈 Monthly Sales Trend")
plt.ylabel("Sales")
plt.tight_layout()
plt.savefig("monthly_sales_trend.png")
plt.clf()
```

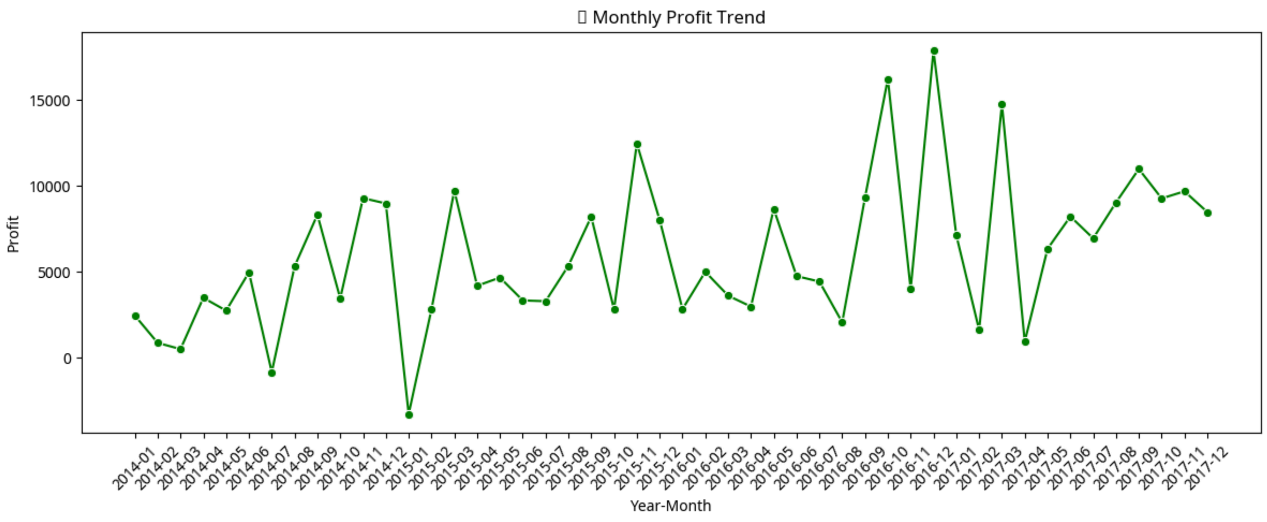


The plot shows an increasing trend in sales over the years, with some seasonal fluctuations.

## Monthly Profit Trend

This line plot visualizes the monthly profit trend over the years.

```
plt.figure(figsize=(12, 5))
sns.lineplot(data=monthly, x="Year-Month", y="Profit", marker="o",
color="green")
plt.xticks(rotation=45)
plt.title("💼 Monthly Profit Trend")
plt.ylabel("Profit")
plt.tight_layout()
plt.savefig("monthly_profit_trend.png")
plt.clf()
```

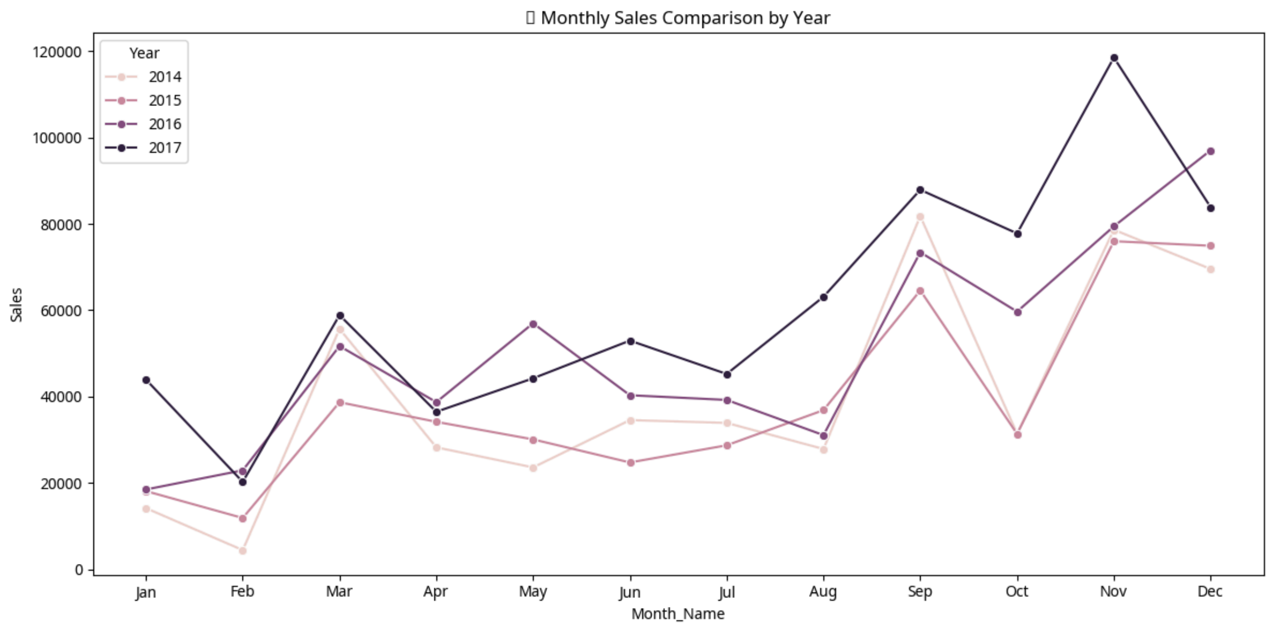


Similar to sales, profit also shows an upward trend, but with more volatility and occasional dips into negative territory.

## Monthly Sales Comparison by Year

This plot compares monthly sales patterns across different years, providing insights into seasonality.

```
df["Month_Name"] = df["Order Date"].dt.strftime("%b")
month_order = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
               "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

monthly_by_year = df.groupby(["Year", "Month_Name"])
["Sales"].sum().reset_index()
monthly_by_year["Month_Name"] = pd.Categorical(monthly_by_year["Month_Name"],
categories=month_order, ordered=True)
monthly_by_year = monthly_by_year.sort_values(["Year", "Month_Name"])

plt.figure(figsize=(12, 6))
sns.lineplot(data=monthly_by_year, x="Month_Name", y="Sales", hue="Year",
marker="o")
plt.title("📆 Monthly Sales Comparison by Year")
plt.ylabel("Sales")
plt.tight_layout()
plt.savefig("monthly_sales_comparison_by_year.png")
plt.clf()
```



The comparison by year highlights consistent seasonal peaks, typically towards the end of the year, and troughs in the earlier months.