```
# from google.colab import drive
# drive.mount('/content/drive')
```

Sales Forecasting Project

Project Context

Fresh Analytics, a data analytics company, aims to comprehend and predict the demand for various items across restaurants. The primary goal of the project is to determine the sales of items across different restaurants over the years. In an ever-changing competitive market, accurate forecasting is crucial for making correct decisions and plans related to sales, production, and other business aspects.

Project Objectives

In ever-changing competitive market conditions, there is a need to make correct decisions and plans for future events related to business like sales, production, and many more. The effectiveness of a decision taken by business managers is influenced by the accuracy of the models used. Demand is the most important aspect of a business's ability to achieve its objectives. Many decisions in business depend on demand, like production, sales, and staff requirements. Forecasting is necessary for business at both international and domestic levels.

Project Dataset Description

1. **restaurants.csv**: Contains information about the restaurants or stores.
   - id: Unique identification of the restaurant or store
   - name: Name of the restaurant
2. **items.csv**: Provides details about the items sold.
   - id: Unique identification of the item
   - store_id: Unique identification of the store
   - name: Name of the item
   - kcal: A measure of energy nutrients (calories) in the item
   - cost: The unit price of the item
3. **sales.csv**: Contains sales data for items at different stores on various dates.
   - date: Date of purchase
   - item: Name of the item bought
   - Price: Unit price of the item
   - item_count: Total count of the items bought on that day

Project Analysis Steps To Perform

4.1 Preliminary analysis:

```
     4.1.1. Import the datasets into the Python environment
     4.1.2. Examine the dataset's shape and structure, and look out
for any outlier
     4.1.3. Merge the datasets into a single dataset that includes the
date, item id, price, item count, item names, kcal values, store id,
and store name
```

4.2 Exploratory data analysis:

```
     4.2.1. Examine the overall date wise sales to understand the
pattern
     4.2.2. Find out how sales fluctuate across different days of the
week
     4.2.3. Look for any noticeable trends in the sales data for
different months of the year
     4.2.4. Examine the sales distribution across different quarters
averaged over the years. Identify any noticeable patterns.
     4.2.5. Compare the performances of the different restaurants.
Find out which restaurant had the most sales and look at the sales for
each restaurant across different years, months, and days.
     4.2.6. Identify the most popular items overall and the stores
where they are being sold. Also, find out the most popular item at
each store.
     4.2.7. Determine if the store with the highest sales volume is
also making the most money per day
     4.2.8. Identify the most expensive item at each restaurant and
find out its calorie count
```

4.3 Forecasting using machine learning algorithms

```
     4.3.1. Forecasting using machine learning algorithms
        4.3.1.1. Generate necessary features for the development of
these models, like day of the week, quarter of the year, month, year,
day of the month and so on
        4.3.1.2. Use the data from the last six months as the testing
data
        4.3.1.3. Compute the root mean square error (RMSE) values for
each model to compare their performances
        4.3.1.4. Use the best-performing models to make a forecast for
the next year
```

4.4 Forecasting using deep learning algorithms

```
    4.4.1. Use sales amount for predictions instead of item count
    4.4.2. Build a long short-term memory (LSTM) model for
predictions
        4.4.2.1. Define the train and test series
        4.4.2.2. Generate synthetic data for the last 12 months
        4.4.2.3. Build and train an LSTM model.
        4.4.2.4. Use the model to make predictions for the test data.
    4.4.3. Calculate the mean absolute percentage error (MAPE) and
comment on the model's performance
    4.4.4. Develop another model using the entire series for
training, and use it to forecast for the next three months
```

4.1. Preliminary analysis

4.1.1. Import Datasets

```
# Import necessary libraries
import os
import calendar
import logging
from datetime import timedelta
from dotenv import load_dotenv

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pandas.plotting import andrews_curves, parallel_coordinates
import seaborn as sns
from prettytable import PrettyTable

import tensorflow as tf
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping

from scikeras.wrappers import KerasClassifier, KerasRegressor

from scipy import stats
from scipy.fft import fft
from sklearn.cluster import KMeans
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,
r2_score,roc_auc_score, roc_curve, auc, confusion_matrix,
ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split, cross_val_score,
RandomizedSearchCV
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer, OneHotEncoder,
StandardScaler, MinMaxScaler
from sklearn.utils import resample, to_categorical

from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose

from xgboost import XGBRegressor

# To ignore warnings
import warnings
warnings.filterwarnings("ignore")

# Set random seed for reproducibility
np.random.seed(1971)

# Load the environment variables
load_dotenv(verbose=True, dotenv_path='.env', override=True)

DATASET_PATH = os.getenv('DATASET_PATH')

restaurants_ds_file = f'{DATASET_PATH}/resturants.csv'
items_ds_file = f'{DATASET_PATH}/items.csv'
sales_ds_file = f'{DATASET_PATH}/sales.csv'

# Read the CSV files
restaurants_df = pd.read_csv(restaurants_ds_file)
items_df = pd.read_csv(items_ds_file)
sales_df = pd.read_csv(sales_ds_file)

# Display the first few rows of each dataset
print("Restaurants dataset:")
print(restaurants_df.head())
print("\nItems dataset:")
print(items_df.head())
print("\nSales dataset:")
print(sales_df.head())
```

```
Restaurants dataset:
   id            name
0   1     Bob's Diner
1   2  Beachfront Bar
2   3     Sweet Shack
3   4        Fou Cher
4   5     Corner Cafe

Items dataset:
   id  store_id                                name  kcal   cost
0   1         4                     Chocolate Cake   554   6.71
1   2         4  Breaded Fish with Vegetables Meal   772  15.09
```

```
2   3         1                  Sweet Fruity Cake   931  29.22
3   4         1    Amazing Steak Dinner with Rolls   763  26.42
4   5         5                          Milk Cake   583   6.07

Sales dataset:
         date  item_id  price  item_count
0  2019-01-01        3  29.22         2.0
1  2019-01-01        4  26.42        22.0
2  2019-01-01       12   4.87         7.0
3  2019-01-01       13   4.18        12.0
4  2019-01-01       16   3.21       136.0
```

Explanations:

- This code block imports necessary libraries (`pandas`, `numpy`, `matplotlib`, and `seaborn`) and reads the three CSV files into pandas DataFrames. It then displays the first few rows of each dataset to give an initial view of the data.

Why It Is Important:

- Importing and examining the datasets is crucial as it allows us to understand the structure and content of our data. This step helps identify any immediate issues with data formatting or missing values and provides a foundation for all subsequent analyses.

Observations:

- [Placeholder for observations after running the code]

Conclusions:

- [Placeholder for conclusions based on initial data view]

Recommendations:

- [Placeholder for recommendations based on initial data examination]

4.1.2. Examine the dataset's shape and structure, and look out for any outlier

```
# Check the shape of each dataset
print("Restaurants dataset shape:", restaurants_df.shape)
print("Items dataset shape:", items_df.shape)
print("Sales dataset shape:", sales_df.shape)

# Display info for each dataset
print("\nRestaurants dataset info:")
restaurants_df.info()
print("\nItems dataset info:")
items_df.info()
print("\nSales dataset info:")
sales_df.info()

# Display basic statistics for numerical columns
```

```python
print("\nRestaurants Dataset Statistics:")
print(restaurants_df.describe())
print("\nItems Dataset Statistics:")
print(items_df.describe())
print("\nSales Dataset Statistics:")
print(sales_df.describe())

# Check for missing values
print("\nMissing values in Restaurants dataset:")
print(restaurants_df.isnull().sum())
print("\nMissing values in Items dataset:")
print(items_df.isnull().sum())
print("\nMissing values in Sales dataset:")
print(sales_df.isnull().sum())

# Check for duplicates
print("\nDuplicates in Restaurants dataset:",
restaurants_df.duplicated().sum())
print("Duplicates in Items dataset:", items_df.duplicated().sum())
print("Duplicates in Sales dataset:", sales_df.duplicated().sum())
```

```
Restaurants dataset shape: (6, 2)
Items dataset shape: (100, 5)
Sales dataset shape: (109600, 4)

Restaurants dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      6 non-null      int64
 1   name    6 non-null      object
dtypes: int64(1), object(1)
memory usage: 228.0+ bytes

Items dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   id        100 non-null    int64
 1   store_id  100 non-null    int64
 2   name      100 non-null    object
 3   kcal      100 non-null    int64
 4   cost      100 non-null    float64
dtypes: float64(1), int64(3), object(1)
memory usage: 4.0+ KB
```

```
Sales dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 109600 entries, 0 to 109599
Data columns (total 4 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   date        109600 non-null  object
 1   item_id     109600 non-null  int64
 2   price       109600 non-null  float64
 3   item_count  109600 non-null  float64
dtypes: float64(2), int64(1), object(1)
memory usage: 3.3+ MB

Restaurants Dataset Statistics:
             id
count  6.000000
mean   3.500000
std    1.870829
min    1.000000
25%    2.250000
50%    3.500000
75%    4.750000
max    6.000000

Items Dataset Statistics:
               id    store_id         kcal         cost
count  100.000000  100.000000   100.000000  100.000000
mean    50.500000    3.520000   536.730000   11.763700
std     29.011492    1.708446   202.212852    8.991254
min      1.000000    1.000000    78.000000    1.390000
25%     25.750000    2.000000   406.250000    5.280000
50%     50.500000    4.000000   572.500000    7.625000
75%     75.250000    5.000000   638.250000   18.790000
max    100.000000    6.000000  1023.000000   53.980000

Sales Dataset Statistics:
             item_id          price     item_count
count  109600.000000  109600.000000  109600.000000
mean       50.500000      11.763700       6.339297
std        28.866202       8.946225      30.003728
min         1.000000       1.390000       0.000000
25%        25.750000       5.280000       0.000000
50%        50.500000       7.625000       0.000000
75%        75.250000      18.790000       0.000000
max       100.000000      53.980000     570.000000

Missing values in Restaurants dataset:
id      0
name    0
dtype: int64
```

```
Missing values in Items dataset:
id          0
store_id    0
name        0
kcal        0
cost        0
dtype: int64

Missing values in Sales dataset:
date          0
item_id       0
price         0
item_count    0
dtype: int64

Duplicates in Restaurants dataset: 0
Duplicates in Items dataset: 0
Duplicates in Sales dataset: 0
```

```
# Convert date to datetime
sales_df['date'] = pd.to_datetime(sales_df['date'])
# Print info about the datasets
print(f"\nSales_df information: \n")
print(sales_df.info())
```

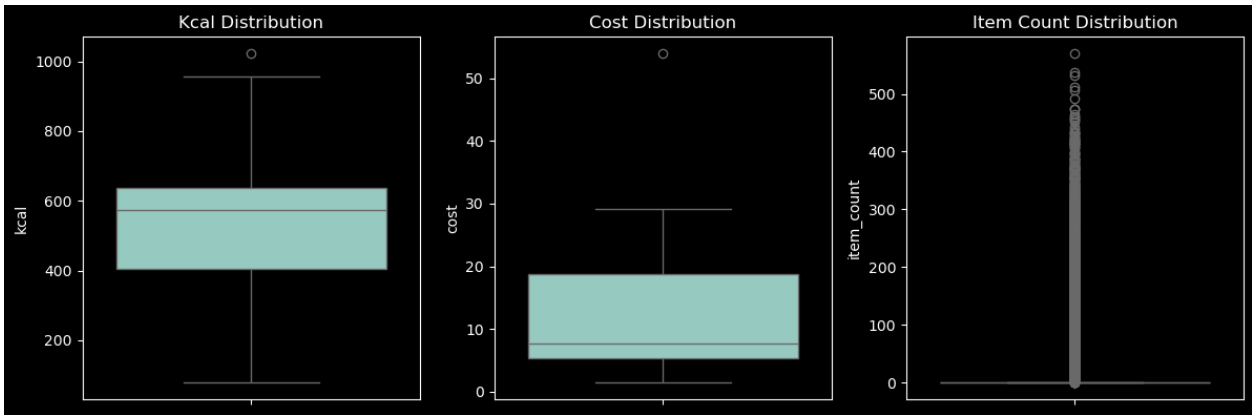
```
Sales_df information:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 109600 entries, 0 to 109599
Data columns (total 4 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   date        109600 non-null  datetime64[ns]
 1   item_id     109600 non-null  int64
 2   price       109600 non-null  float64
 3   item_count  109600 non-null  float64
dtypes: datetime64[ns](1), float64(2), int64(1)
memory usage: 3.3 MB
None
```

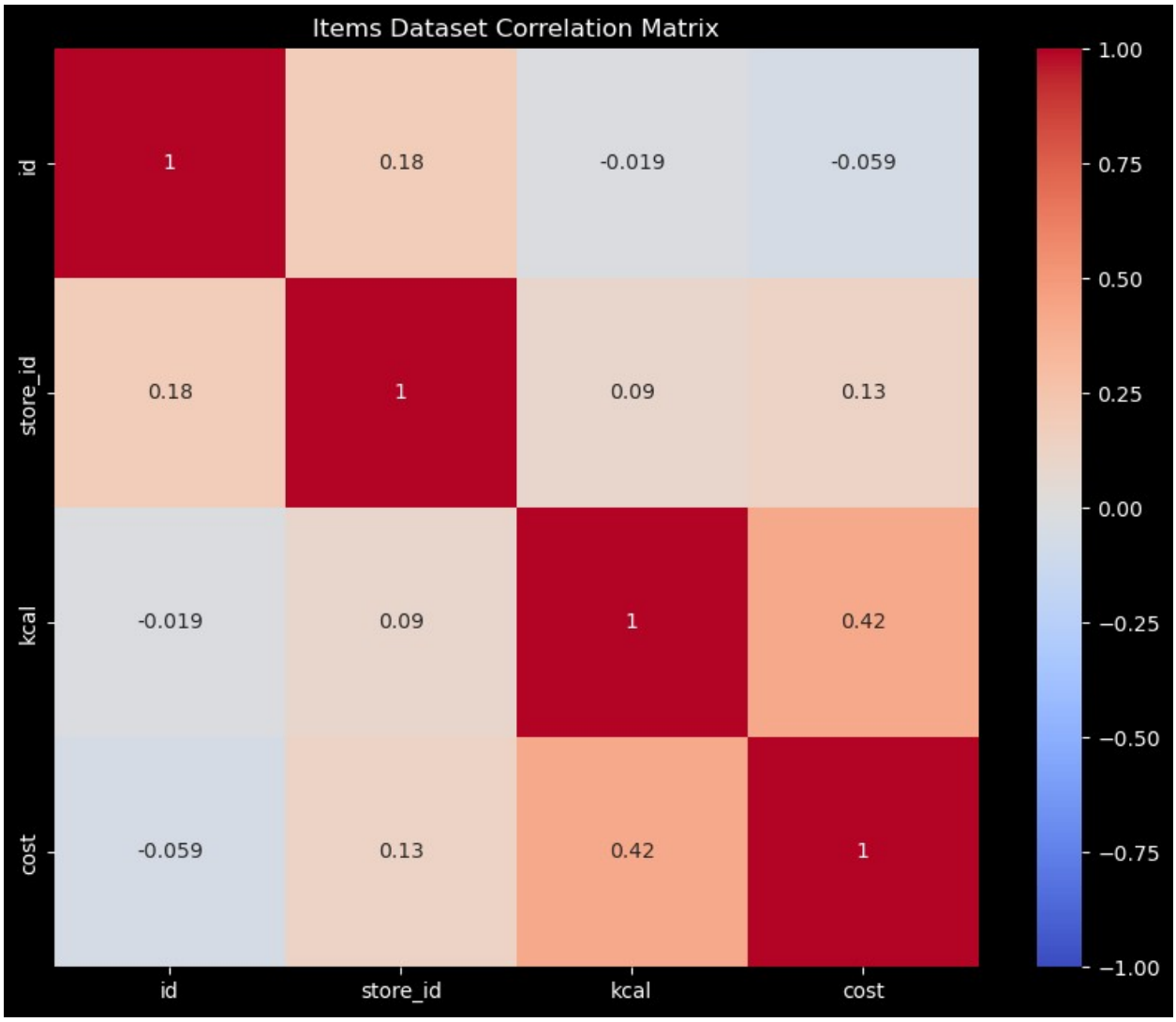
```
# Check for outliers using box plots
plt.figure(figsize=(12, 4))
plt.subplot(131)
sns.boxplot(data=items_df, y='kcal')
plt.title('Kcal Distribution')
plt.subplot(132)
sns.boxplot(data=items_df, y='cost')
plt.title('Cost Distribution')
```

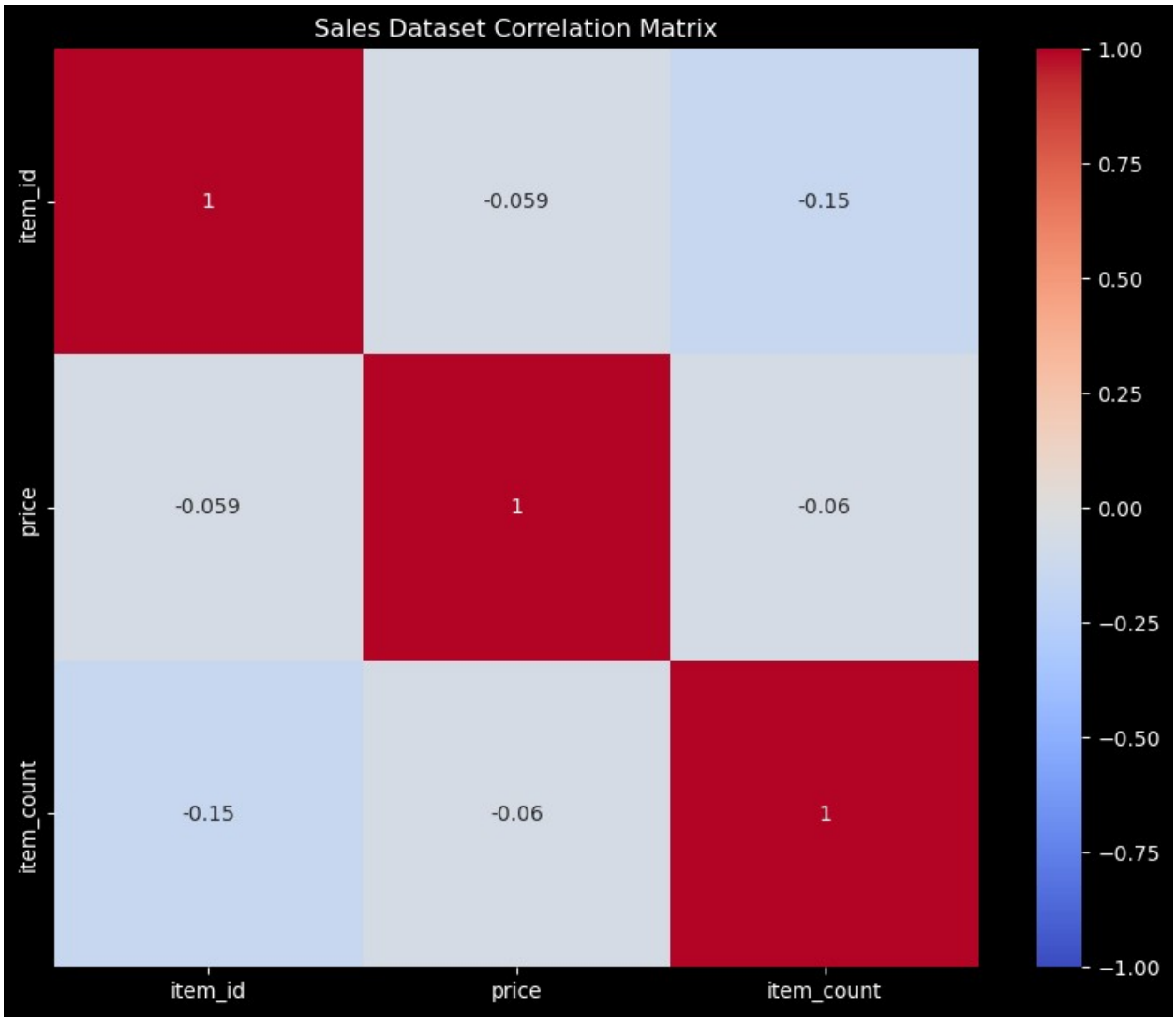
```
plt.subplot(133)
sns.boxplot(data=sales_df, y='item_count')
plt.title('Item Count Distribution')
plt.tight_layout()
plt.show()


# datasets = [restaurants_df, items_df, sales_df]
datasets = [items_df, sales_df]

restaurants_df.attrs['name'] = 'Restaurants Dataset'
items_df.attrs['name'] = 'Items Dataset'
sales_df.attrs['name'] = 'Sales Dataset'

# 1. Correlation Matrix
def plot_correlation_matrix(df):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    corr = df.select_dtypes(include=[np.number]).corr()
    plt.figure(figsize=(10, 8))
    sns.heatmap(corr, annot=True, cmap='coolwarm', vmin=-1, vmax=1,
center=0)
    plt.title(f'{dataset_name} Correlation Matrix')
    plt.show()

for df in datasets:
    plot_correlation_matrix(df)

# 2. Pairplot
def create_pairplot(df):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    sns.pairplot(df.select_dtypes(include=[np.number]))
    plt.suptitle(f'{dataset_name} Pairplot of Numerical Variables',
y=1.02)
    plt.show()

for df in datasets:
    create_pairplot(df)

# 5. Time Series Decomposition
def plot_time_series_decomposition(df):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    df_temp = df.copy()
    df_temp.set_index('date', inplace=True)
    result = seasonal_decompose(df_temp['item_count'],
model='additive', period=30)  # Adjust period as needed
    result.plot()
    plt.suptitle(f'Time Series Decomposition of {dataset_name}',
```

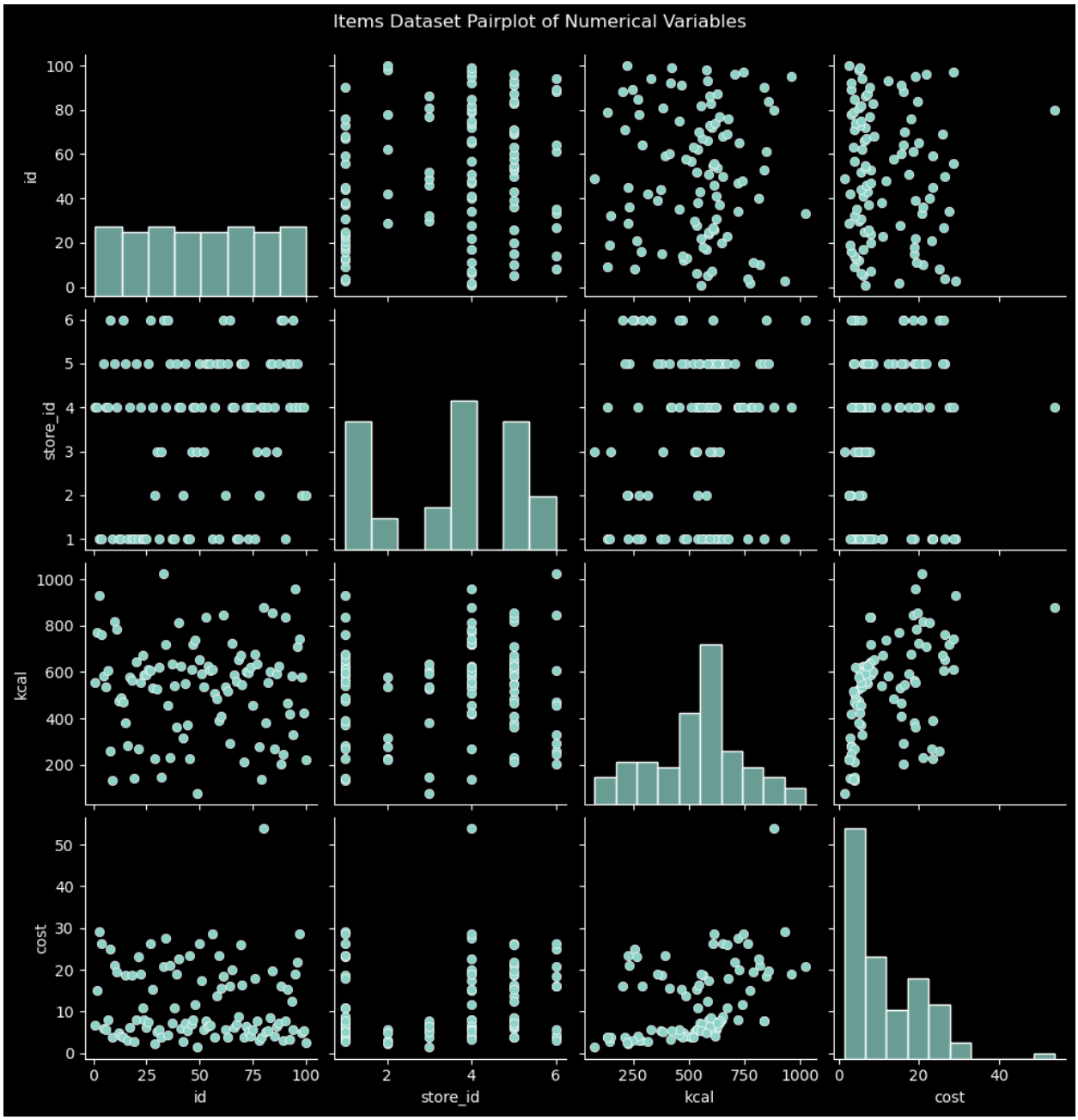
```
y=1.02)
    plt.tight_layout()
    plt.show()

plot_time_series_decomposition(sales_df)

# 6. Distribution Plots
def plot_distributions(df):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    num_cols = df.select_dtypes(include=[np.number]).columns
    n_cols = 2
    n_rows = (len(num_cols) + 1) // 2
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(15, 5*n_rows))
    for i, col in enumerate(num_cols):
        ax = axes[i//n_cols, i%n_cols]
        sns.histplot(df[col], kde=True, ax=ax)
        ax.set_title(f'Distribution of {col} in {dataset_name}')
    plt.tight_layout()
    plt.show()

for df in datasets:
    plot_distributions(df)

# 11. Scatter Plot Matrix
def plot_scatter_matrix(df):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    pd.plotting.scatter_matrix(df.select_dtypes(include=[np.number]),
figsize=(15, 15), diagonal='kde')
    plt.suptitle(f'Scatter Plot Matrix of {dataset_name}', y=1.02)
    plt.tight_layout()
    plt.show()

for df in datasets:
    plot_scatter_matrix(df)


# 12. 3D Scatter Plot
def plot_3d_scatter(df, x_col, y_col, z_col):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(df[x_col], df[y_col], df[z_col])
    ax.set_xlabel(x_col)
    ax.set_ylabel(y_col)
    ax.set_zlabel(z_col)
    plt.title(f'3D Scatter Plot of {dataset_name}')
    plt.show()
```

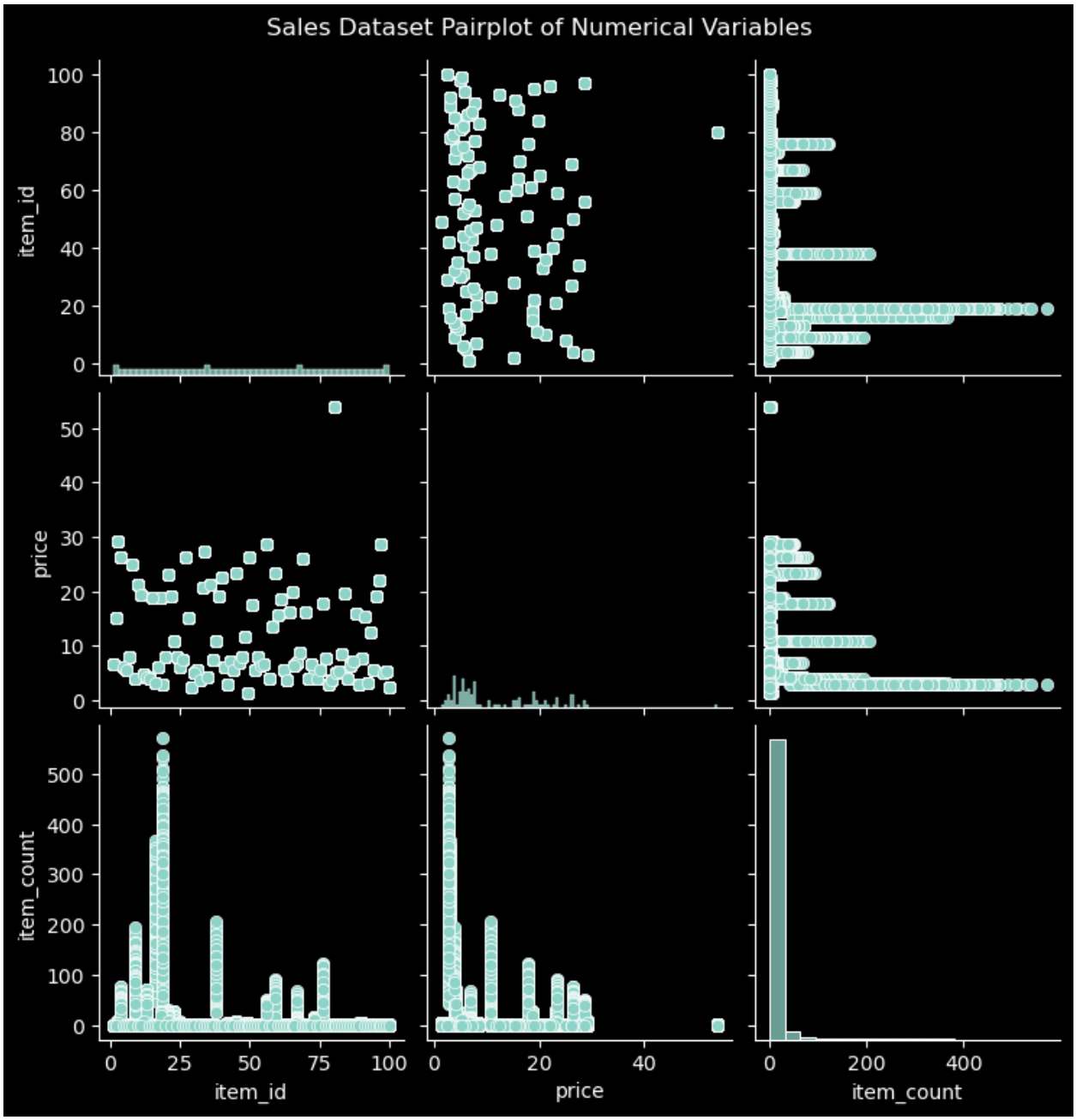
```
plot_3d_scatter(items_df, 'kcal', 'cost', 'id')  # Example

# 15. Cluster Analysis
from sklearn.cluster import KMeans
def perform_cluster_analysis(df, n_clusters=3):
    # Retrieve the dataset name from the DataFrame's attributes
    dataset_name = df.attrs.get('name', 'Dataset')
    numeric_data = df.select_dtypes(include=[np.number])
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    clusters = kmeans.fit_predict(numeric_data)

    plt.figure(figsize=(10, 8))
    scatter = plt.scatter(numeric_data.iloc[:, 0],
numeric_data.iloc[:, 1], c=clusters, cmap='viridis')
    plt.title(f'K-means Clustering of Numeric Variables in
{dataset_name}')
    plt.xlabel(numeric_data.columns[0])
    plt.ylabel(numeric_data.columns[1])
    plt.colorbar(scatter)
    plt.show()

for df in datasets:
    perform_cluster_analysis(df)
# perform_cluster_analysis(items)
```

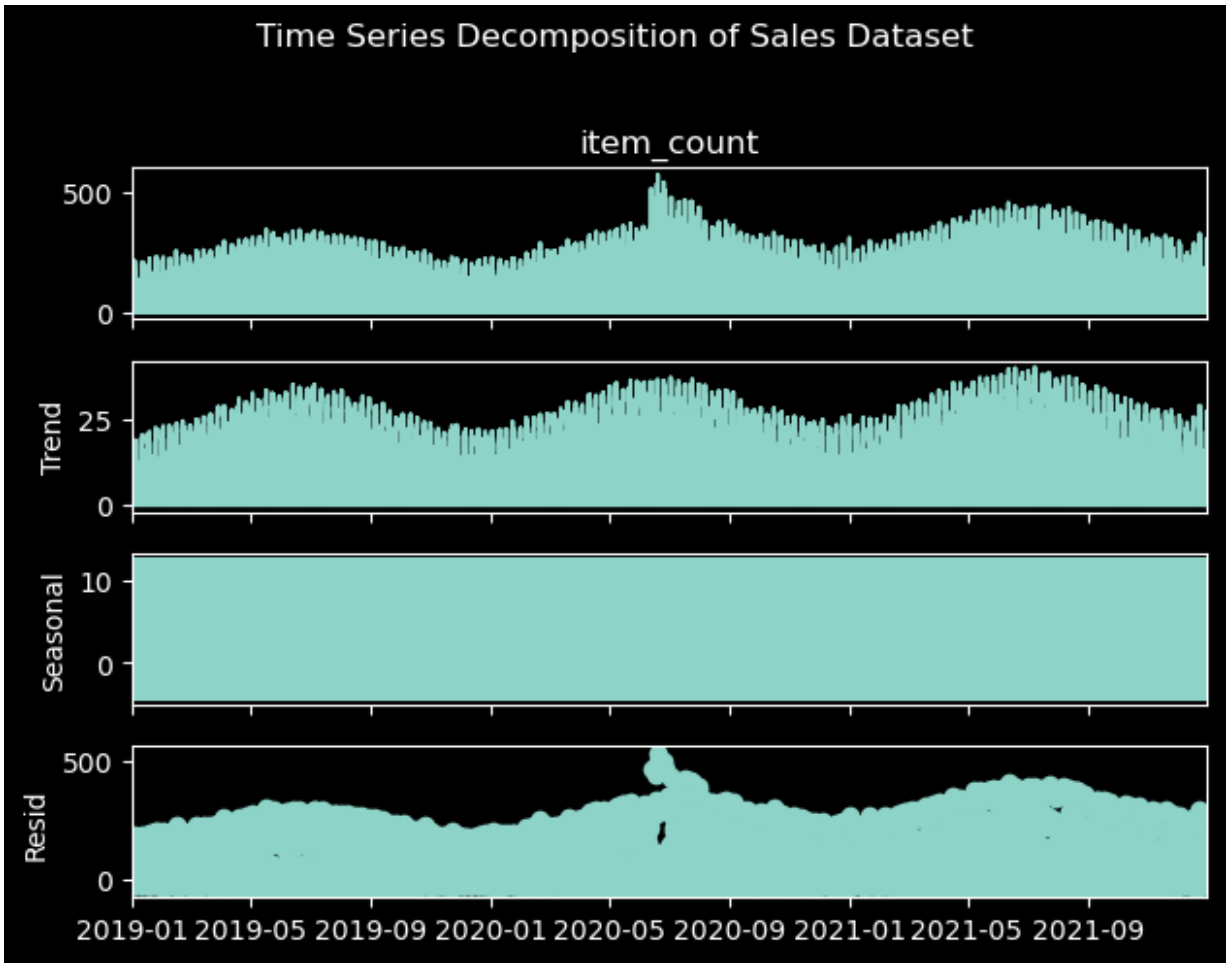


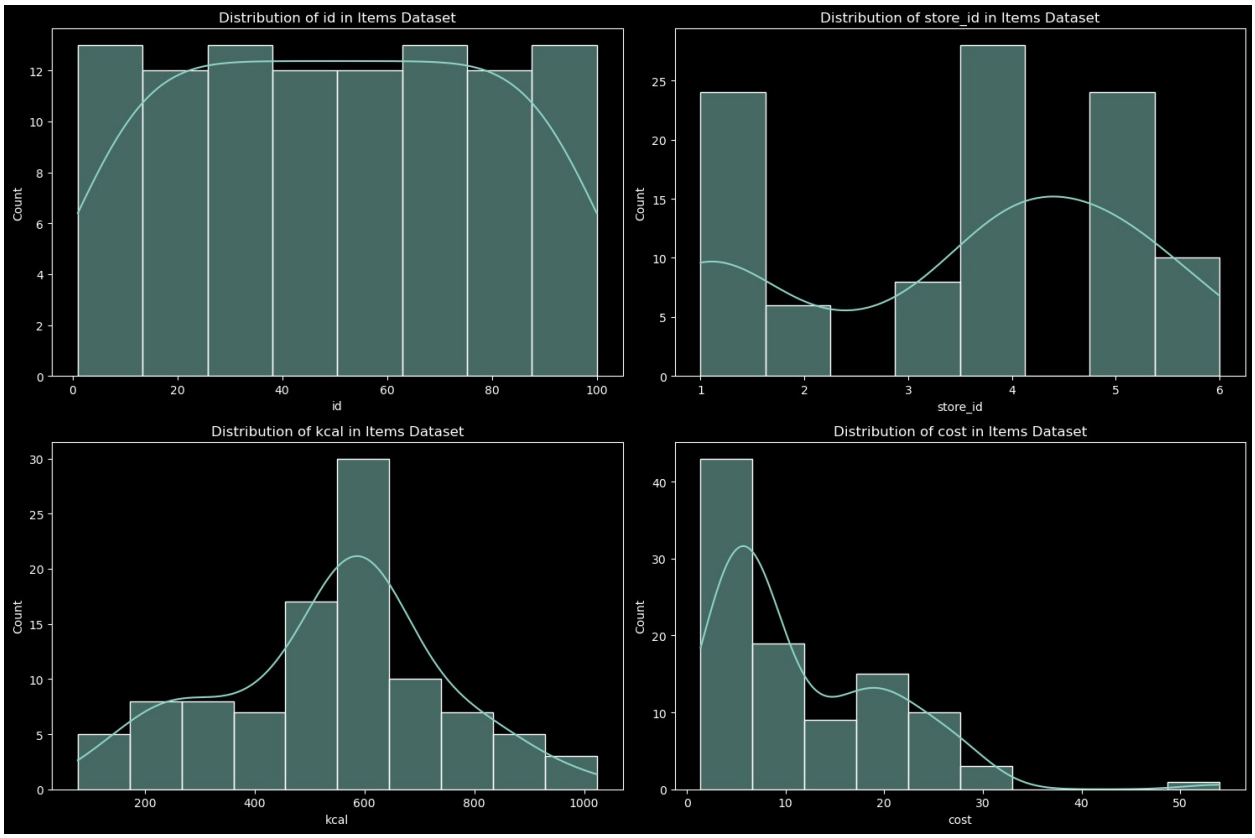

Items Dataset Correlation Matrix
id
store_id
kcal
cost
1
0.18
-0.019
-0.059
0.18
1
0.09
0.13
-0.019
0.09
1
0.42
-0.059
0.13
0.42
1
id
store_id
kcal
cost
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

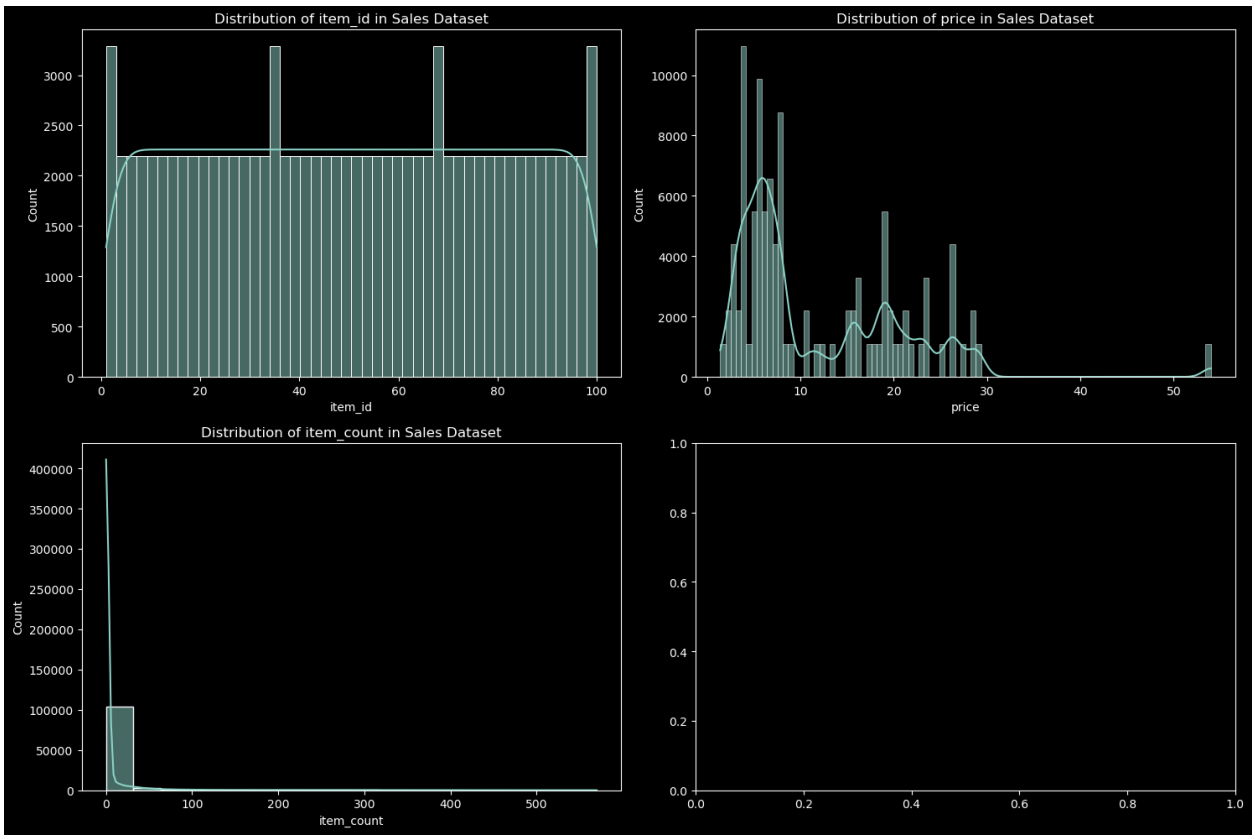

Sales Dataset Correlation Matrix
item_id
price
item_count
1
-0.059
-0.15
-0.059
1
-0.06
-0.15
-0.06
1
item_id
price
item_count
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

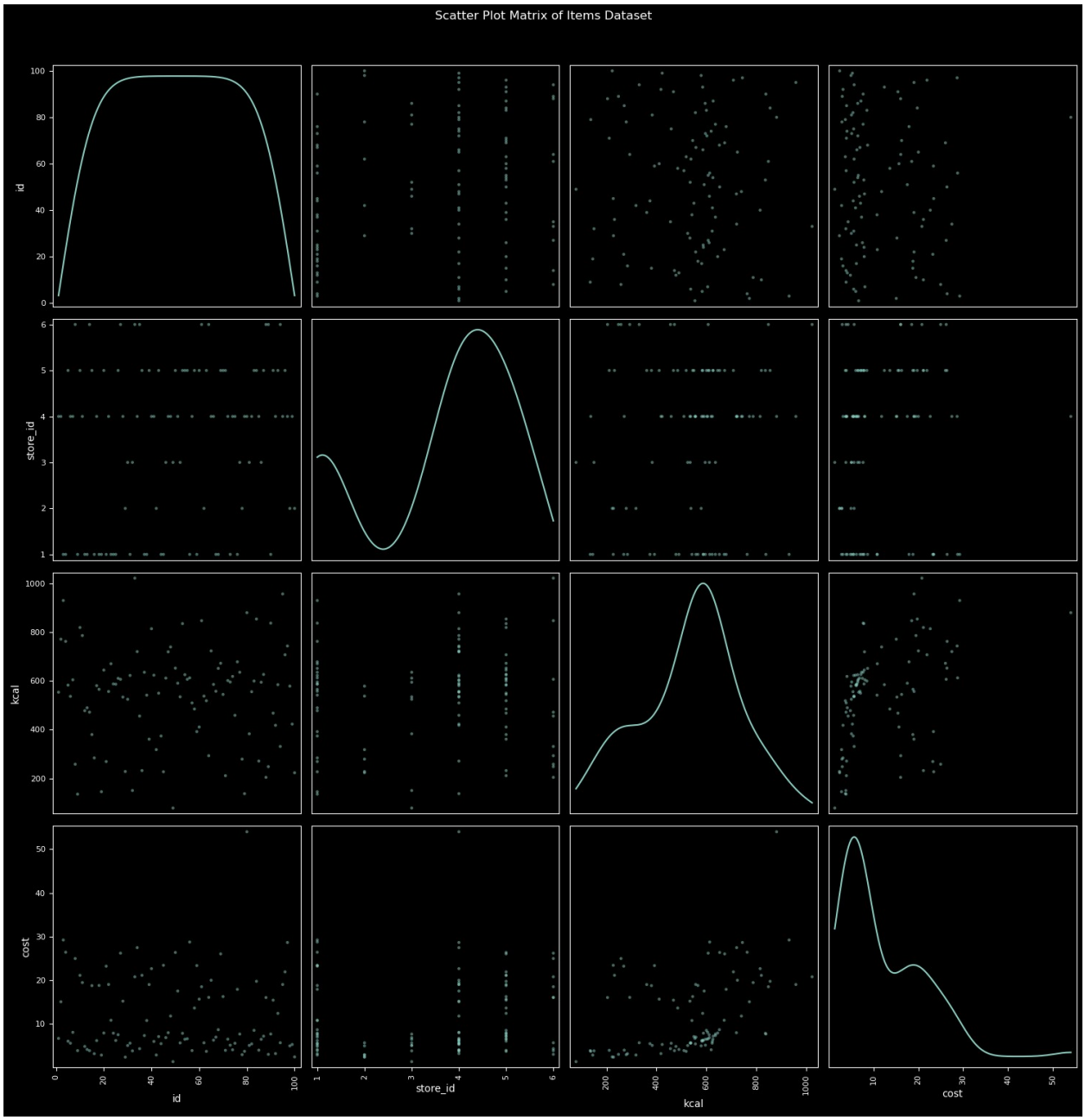

Items Dataset Pairplot of Numerical Variables
id
store_id
kcal
cost
0
25
50
75
100
2
4
6
250
500
750
1000
0
20
40
0
10
20
30
40
50
200
400
600
800
1000
1
2
3
4
5
6
0
20
40
60
80
100

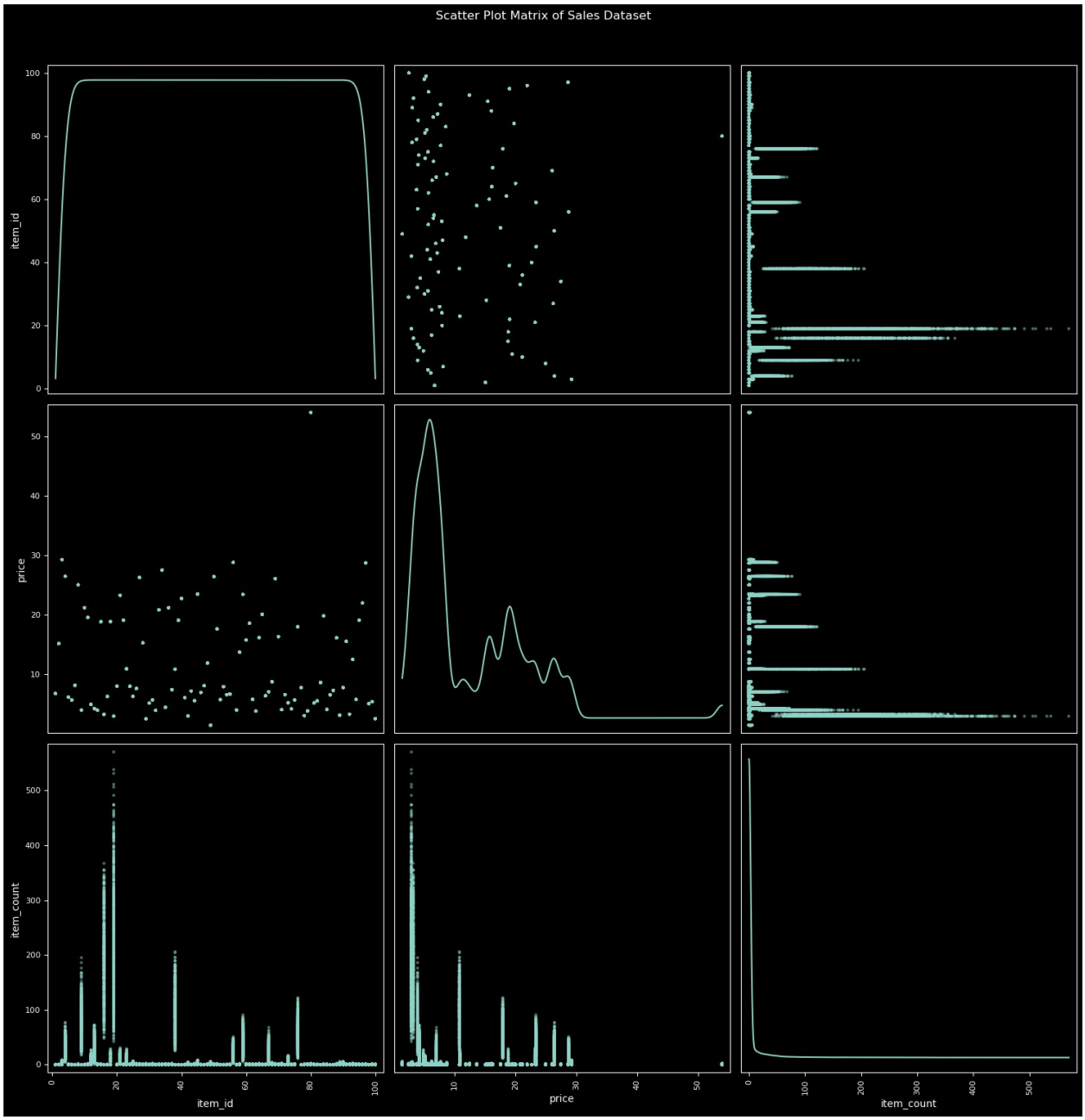

Sales Dataset Pairplot of Numerical Variables
item_id
price
item_count
0
20
40
60
80
100
0
10
20
30
40
50
0
100
200
300
400
500
0
25
50
75
100
0
20
40
0
200
400

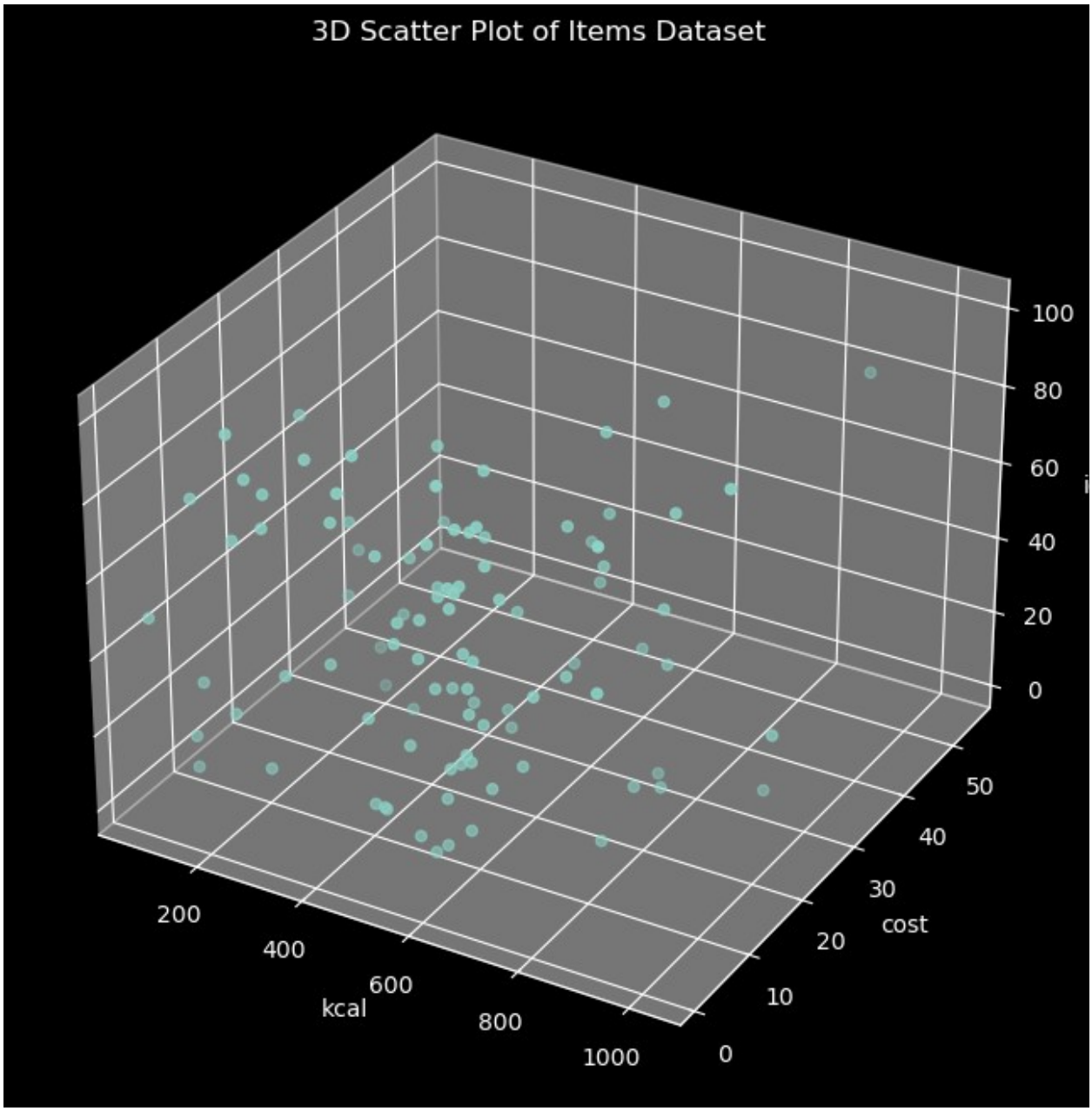

Time Series Decomposition of Sales Dataset
item_count
500
0
Trend
25
0
Seasonal
10
0
Resid
500
0
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09

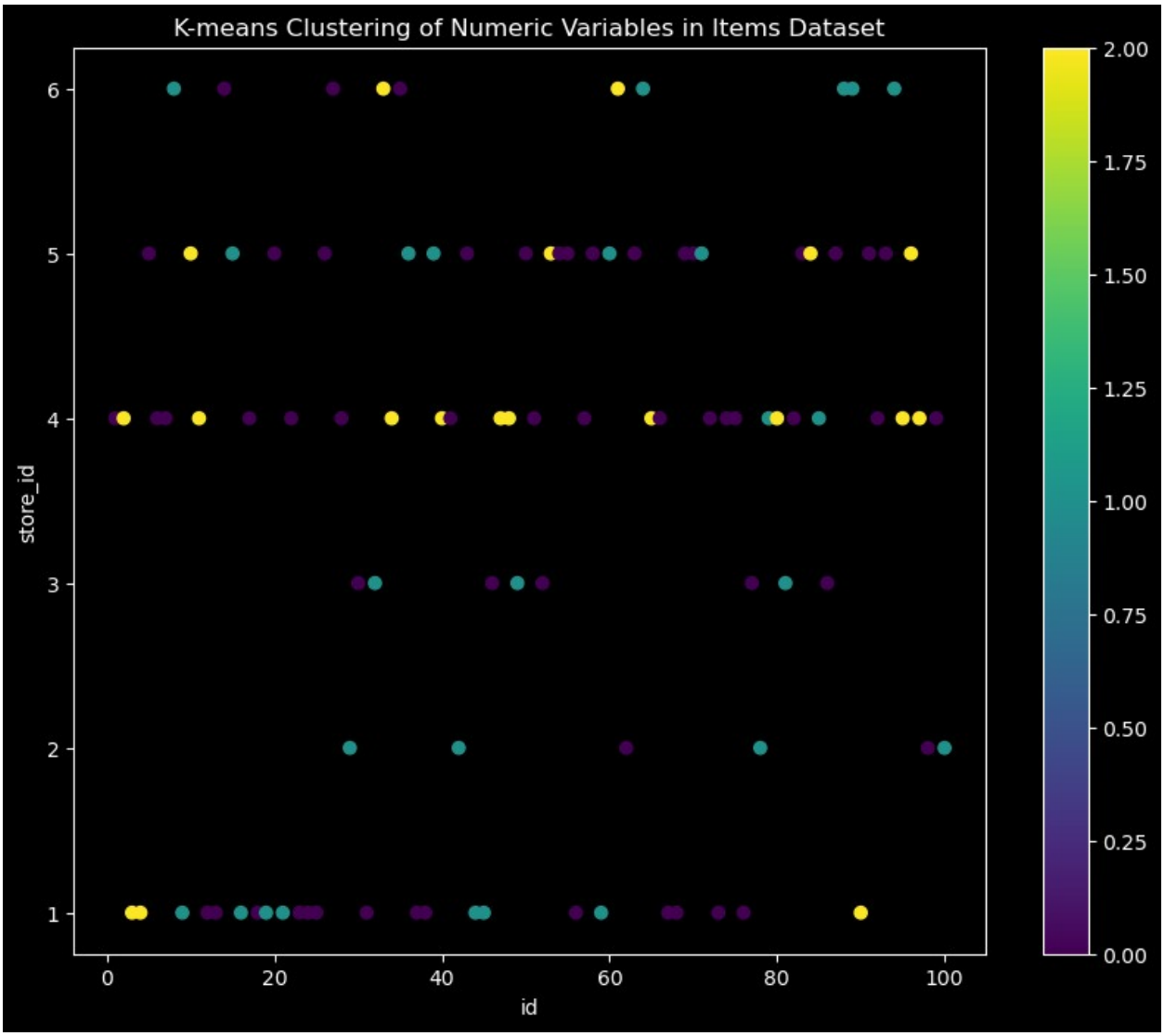

Distribution of id in Items Dataset
Count
id
Distribution of store_id in Items Dataset
Count
store_id
Distribution of kcal in Items Dataset
Count
kcal
Distribution of cost in Items Dataset
Count
cost

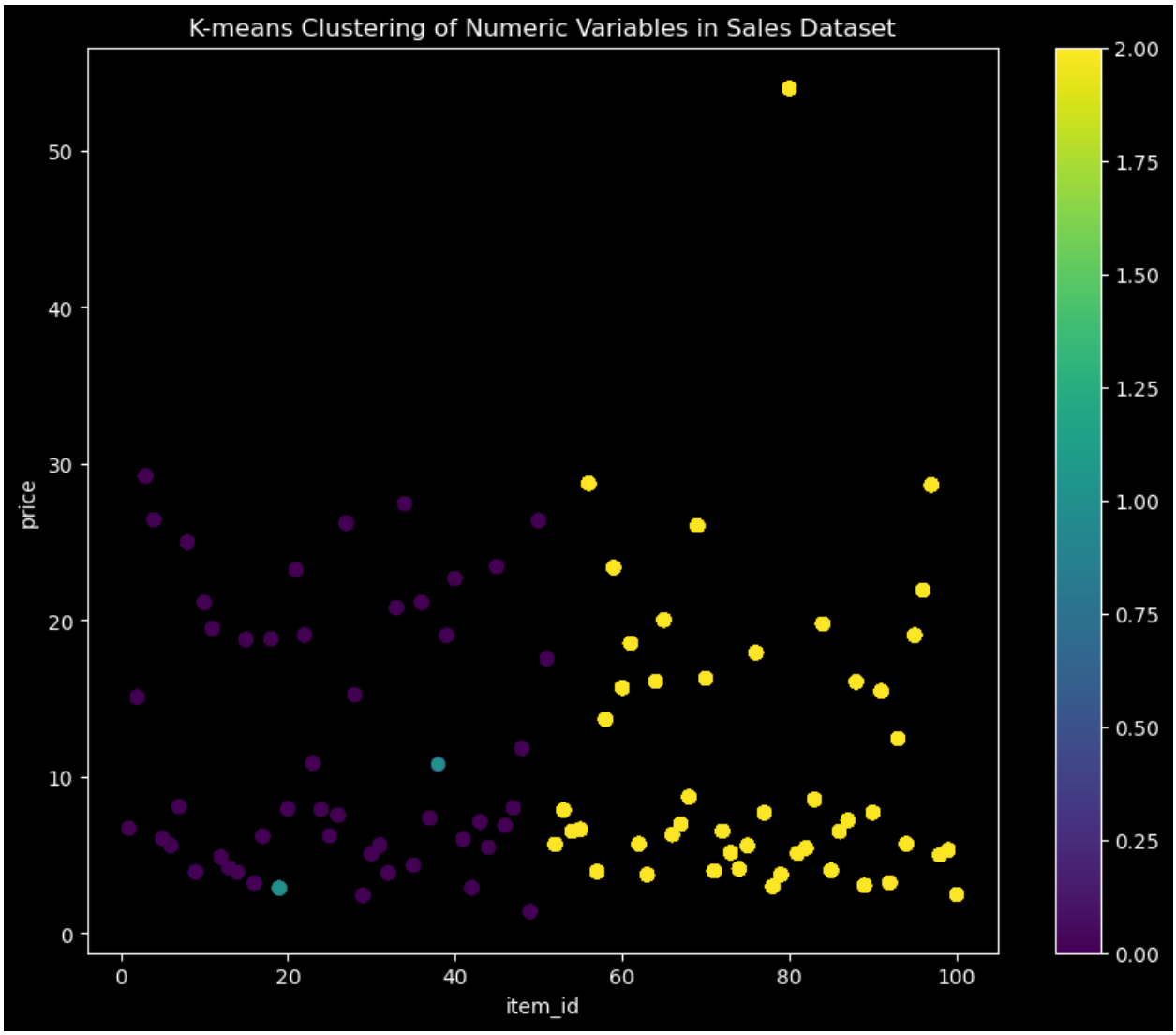

Distribution of item_id in Sales Dataset
Count
item_id
Distribution of price in Sales Dataset
Count
price
Distribution of item_count in Sales Dataset
Count
item_count


Scatter Plot Matrix of Items Dataset
id
store_id
kcal
cost
id
store_id
kcal
cost


Scatter Plot Matrix of Sales Dataset
item_id
price
item_count
item_id
price
item_count


3D Scatter Plot of Items Dataset
100
80
60
40
20
0
50
40
30
20
10
0
cost
200
400
600
800
1000
kcal


K-means Clustering of Numeric Variables in Items Dataset
store_id
id
0
20
40
60
80
100
1
2
3
4
5
6
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00



Explanations:

- This code examines the shape, structure, and quality of each dataset. It checks the number of rows and columns, data types of each column, presence of missing values, and existence of duplicate entries.

Why It Is Important:

- Understanding the dataset's structure and quality is crucial for data preprocessing and analysis. It helps identify potential issues like missing data or duplicates that need to be addressed before proceeding with the analysis.

Observations:

1. **Dataset Sizes:**
   - Restaurants: 6 entries, 2 columns
   - Items: 100 entries, 5 columns
   - Sales: 109,600 entries, 4 columns

2. **Data Quality:**
   - No missing values in any dataset
   - No duplicates in any dataset
3. **Correlations:**
   - Items dataset: Moderate positive correlation (0.42) between kcal and cost
   - Sales dataset: Weak negative correlation (-0.15) between item_id and item_count
4. **Distributions:**
   - Item costs are right-skewed, with most items clustered at lower cost levels
   - Calorie (kcal) distribution is relatively normal but with some high-calorie outliers
   - Sales prices show multiple peaks, suggesting different price tiers
   - Item counts in sales are heavily right-skewed, with many low-count transactions
5. **Time Series:**
   - Sales data shows clear seasonal patterns with regular peaks and troughs
   - There's an overall increasing trend in sales over time

Conclusions:

1. **Menu Composition:**
   - The restaurant chain offers a diverse menu with varying nutritional values and prices
   - There's a slight tendency for higher-calorie items to cost more
2. **Pricing Strategy:**
   - The multi-modal price distribution suggests tiered pricing or distinct categories of items
3. **Sales Patterns:**
   - Strong seasonality in sales, possibly reflecting weekly or monthly patterns
   - Overall positive trend in sales, indicating business growth
4. **Customer Behavior:**
   - Most transactions involve a small number of items, with some large orders as outliers
5. **Restaurant Chain Size:**
   - With only 6 restaurants, this appears to be a small to medium-sized chain

Recommendations:

1. **Menu Optimization:**
   - Analyze the relationship between item cost, price, and sales volume to optimize menu offerings
   - Consider introducing more items in the mid-range of calories and cost to balance the menu
2. **Pricing Strategy Review:**
   - Evaluate the effectiveness of the current tiered pricing structure
   - Consider dynamic pricing based on identified seasonal patterns to maximize revenue
3. **Seasonal Promotions:**

- Develop targeted marketing campaigns aligned with the observed seasonal sales patterns
- Prepare inventory and staffing based on predicted busy periods

4. **Upselling Initiatives:**
   - Given the prevalence of small transactions, implement strategies to increase average order size
5. **Expansion Consideration:**
   - With positive sales trends, explore opportunities for opening new locations
6. **Data Collection Enhancement:**
   - Include more detailed time data (hour of day) to analyze intra-day sales patterns
   - Collect customer demographic data to enable more targeted marketing and menu planning
7. **Outlier Analysis:**
   - Investigate high-calorie and high-cost menu items to ensure they are contributing positively to overall sales and profitability
8. **Health-Conscious Options:**
   - Given the wide range of calorie contents, ensure there are sufficient low-calorie options to cater to health-conscious customers
9. **Loyalty Program:**
   - Consider implementing a loyalty program to encourage repeat business and gather more customer-specific data
10. **Operational Efficiency:**
    - Use the sales pattern data to optimize staff scheduling and inventory management

4.1.2. Merge the datasets into a single dataset that includes the date, item id, price, item count, item names, kcal values, store id, and store name

```
# Merge sales with items
merged_df = pd.merge(sales_df, items_df, left_on='item_id',
right_on='id', how='left')

# Merge with restaurants
merged_df = pd.merge(merged_df, restaurants_df, left_on='store_id',
right_on='id', how='left')

# Rename columns to avoid confusion
merged_df = merged_df.rename(columns={'name_x': 'item_name', 'name_y':
'restaurant_name'})

# Select relevant columns
final_df = merged_df[['date', 'item_id', 'price', 'item_count',
'item_name', 'kcal', 'store_id', 'restaurant_name']]

# Convert date to datetime
# final_df['date'] = pd.to_datetime(final_df['date'])

# Display the first few rows of the merged dataset
```

```
print(final_df.head())

# Display info of the merged dataset
final_df.info()
```

```
        date  item_id  price  item_count
item_name  \
0 2019-01-01        3  29.22         2.0                Sweet Fruity
Cake
1 2019-01-01        4  26.42        22.0  Amazing Steak Dinner with
Rolls
2 2019-01-01       12   4.87         7.0             Fantastic Sweet
Cola
3 2019-01-01       13   4.18        12.0          Sweet Frozen Soft
Drink
4 2019-01-01       16   3.21       136.0             Frozen Milky
Smoothy

   kcal  store_id restaurant_name
0   931         1     Bob's Diner
1   763         1     Bob's Diner
2   478         1     Bob's Diner
3   490         1     Bob's Diner
4   284         1     Bob's Diner
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 109600 entries, 0 to 109599
Data columns (total 8 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   date             109600 non-null  datetime64[ns]
 1   item_id          109600 non-null  int64
 2   price            109600 non-null  float64
 3   item_count       109600 non-null  float64
 4   item_name        109600 non-null  object
 5   kcal             109600 non-null  int64
 6   store_id         109600 non-null  int64
 7   restaurant_name  109600 non-null  object
dtypes: datetime64[ns](1), float64(2), int64(3), object(2)
memory usage: 6.7+ MB
```

Explanations:

- This code merges the three datasets based on common identifiers (item_id and store_id). It then selects relevant columns, renames them for clarity, and converts the date column to datetime format.

Why It Is Important:

- Merging the datasets is crucial for conducting comprehensive analyses that involve information from all three sources. It allows us to examine relationships between sales, item characteristics, and restaurant information in a single DataFrame.

4.2. Exploratory data analysis

4.2.1. Examine the overall date wise sales to understand the pattern

```
# Count the number of entries for each restaurant_name
restaurant_counts = final_df['restaurant_name'].value_counts()

# Display the counts
print(f"Restaurant Sales Transaction Counts From Combined Dataset: \
n")
print(restaurant_counts)
print()

restaurant_items = final_df.groupby('restaurant_name')
['item_name'].apply(set)

# Find the common item_names across all restaurants
common_items = set.intersection(*restaurant_items)

# Check if there are common items and print the appropriate message
if common_items:
    print("Common item_names between restaurants:")
    print(sorted(common_items))
else:
    print(f"No shared items between restaurants.\n")

# Group the DataFrame by restaurant_name and get unique item_names for
each restaurant
restaurant_items = final_df.groupby('restaurant_name')
['item_name'].apply(set)

# Print the unique item_names for each restaurant in alphabetical
order
for restaurant, items in restaurant_items.items():
    sorted_items = sorted(items)
    print(f"Restaurant: {restaurant}")
    print(f"Unique Items: {sorted_items}")
    print()

# Group the DataFrame by restaurant_name and item_name, and aggregate
the prices
restaurant_item_prices = final_df.groupby(['restaurant_name',
'item_name'])['price'].mean().reset_index()

# Iterate over each restaurant and print the table of unique items and
their prices
for restaurant, group in
restaurant_item_prices.groupby('restaurant_name'):
    print(f"Restaurant: {restaurant}")
    print(group[['item_name', 'price']].to_string(index=False))
    print()
```

```
# Group the DataFrame by item_name and calculate the standard
deviation of prices for each item
price_variability = final_df.groupby('item_name')
['price'].std().reset_index()

# Filter items with non-zero standard deviation
variable_price_items = price_variability[price_variability['price'] >
0]

# Check if there are any items with price variability and print the
appropriate message
if not variable_price_items.empty:
    print("Items with price variability:")
    for _, row in variable_price_items.iterrows():
        item_name = row['item_name']
        std_dev = row['price']
        restaurants_selling_item = final_df[final_df['item_name'] ==
item_name]['restaurant_name'].unique()
        print(f"Item: {item_name}, Std Dev: {std_dev:.2f},
Restaurants: {', '.join(restaurants_selling_item)}")
else:
    print("No price variability found for any items.")

print()
print()

# Group by date and calculate total sales
daily_sales = final_df.groupby('date').agg({'price': 'sum',
'item_count': 'sum'})
daily_sales['total_sales'] = daily_sales['price'] *
daily_sales['item_count']

# Plot daily sales
plt.figure(figsize=(15, 6))
plt.plot(daily_sales.index, daily_sales['total_sales'])
plt.title('Daily Sales Over Time')
plt.xlabel('Date')
plt.ylabel('Total Sales')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Calculate and plot 7-day moving average
daily_sales['7_day_ma'] =
daily_sales['total_sales'].rolling(window=7).mean()

plt.figure(figsize=(15, 6))
plt.plot(daily_sales.index, daily_sales['total_sales'], alpha=0.5,
label='Daily Sales')
```

```
plt.plot(daily_sales.index, daily_sales['7_day_ma'], color='red',
label='7-day Moving Average')
plt.title('Daily Sales and 7-day Moving Average')
plt.xlabel('Date')
plt.ylabel('Total Sales')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# First, let's check the structure of your DataFrame
# print(daily_sales.index)
# print(daily_sales.columns)

# If the index is not already a DatetimeIndex, we'll try to convert it
if not isinstance(daily_sales.index, pd.DatetimeIndex):
    # Check if there's a column that could be a date
    date_columns =
daily_sales.select_dtypes(include=[np.datetime64]).columns
    if len(date_columns) > 0:
        date_column = date_columns[0]
        daily_sales.set_index(date_column, inplace=True)
    else:
        # If no date column is found, we'll create a date range
        daily_sales.index = pd.date_range(start='2019-01-01',
periods=len(daily_sales), freq='D')

# Ensure the index is sorted
daily_sales.sort_index(inplace=True)

# 1. Decomposition Plot
def plot_decomposition():
    result = seasonal_decompose(daily_sales['total_sales'],
model='additive', period=365)
    fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 16))
    result.observed.plot(ax=ax1)
    ax1.set_title('Observed')
    result.trend.plot(ax=ax2)
    ax2.set_title('Trend')
    result.seasonal.plot(ax=ax3)
    ax3.set_title('Seasonal')
    result.resid.plot(ax=ax4)
    ax4.set_title('Residual')
    plt.tight_layout()
    plt.show()

# 2. Heatmap of Daily Sales
def plot_daily_heatmap():
    daily_data = daily_sales['total_sales'].resample('D').sum()
    heatmap_data = pd.DataFrame({
```

```
        'year': daily_data.index.year,
        'day_of_year': daily_data.index.dayofyear,
        'sales': daily_data.values
    })
    heatmap_pivot = heatmap_data.pivot(index='year',
columns='day_of_year', values='sales')
    plt.figure(figsize=(16, 8))
    sns.heatmap(heatmap_pivot, cmap='YlOrRd')
    plt.title('Daily Sales Heatmap')
    plt.xlabel('Day of Year')
    plt.ylabel('Year')
    plt.show()

# 3. Year-over-Year Comparison
def plot_year_over_year():
    daily_sales['year'] = daily_sales.index.year
    daily_sales['day_of_year'] = daily_sales.index.dayofyear
    plt.figure(figsize=(12, 6))
    for year in daily_sales['year'].unique():
        year_data = daily_sales[daily_sales['year'] == year]
        plt.plot(year_data['day_of_year'], year_data['total_sales'],
label=str(year))
    plt.title('Year-over-Year Sales Comparison')
    plt.xlabel('Day of Year')
    plt.ylabel('Total Sales')
    plt.legend()
    plt.show()

# 5. Autocorrelation and Partial Autocorrelation Plots
def plot_acf_pacf():
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))
    plot_acf(daily_sales['total_sales'], ax=ax1, lags=50)
    plot_pacf(daily_sales['total_sales'], ax=ax2, lags=50)
    plt.tight_layout()
    plt.show()

# 6. Cumulative Sales Plot
def plot_cumulative_sales():
    cumulative_sales = daily_sales['total_sales'].cumsum()
    plt.figure(figsize=(12, 6))
    plt.plot(cumulative_sales.index, cumulative_sales)
    plt.title('Cumulative Sales Over Time')
    plt.xlabel('Date')
    plt.ylabel('Cumulative Sales')
    plt.show()

# 7. Seasonal Subseries Plot
def plot_seasonal_subseries():
    daily_sales['quarter'] = daily_sales.index.quarter
    fig, axes = plt.subplots(2, 2, figsize=(16, 12))
```

```
    for i, quarter in enumerate([1, 2, 3, 4]):
        ax = axes[i // 2, i % 2]
        quarter_data = daily_sales[daily_sales['quarter'] == quarter]
        sns.lineplot(x=quarter_data.index.dayofyear, y='total_sales',
hue=quarter_data.index.year, data=quarter_data, ax=ax)
        ax.set_title(f'Q{quarter} Sales')
    plt.tight_layout()
    plt.show()

# 8. Rolling Statistics
def plot_rolling_stats():
    rolling_mean = daily_sales['total_sales'].rolling(window=7).mean()
    rolling_std = daily_sales['total_sales'].rolling(window=7).std()
    rolling_median =
daily_sales['total_sales'].rolling(window=7).median()

    plt.figure(figsize=(12, 6))
    plt.plot(daily_sales.index, daily_sales['total_sales'],
label='Daily Sales')
    plt.plot(rolling_mean.index, rolling_mean, label='7-day Moving
Average')
    plt.plot(rolling_std.index, rolling_std, label='7-day Moving Std')
    plt.plot(rolling_median.index, rolling_median, label='7-day Moving
Median')
    plt.title('Rolling Statistics of Daily Sales')
    plt.xlabel('Date')
    plt.ylabel('Sales')
    plt.legend()
    plt.show()

# 9. Fourier Transform
def plot_fourier_transform():
    sales_fft = fft(daily_sales['total_sales'].values)
    frequencies = np.fft.fftfreq(len(sales_fft))
    plt.figure(figsize=(12, 6))
    plt.plot(frequencies, np.abs(sales_fft))
    plt.title('Fourier Transform of Sales Data')
    plt.xlabel('Frequency')
    plt.ylabel('Magnitude')
    plt.xlim(0, 0.5)  # Only show positive frequencies
    plt.show()

# 10. Anomaly Detection
def detect_anomalies():
    rolling_mean = daily_sales['total_sales'].rolling(window=7).mean()
    rolling_std = daily_sales['total_sales'].rolling(window=7).std()
    anomalies = daily_sales[(daily_sales['total_sales'] > rolling_mean
+ 2*rolling_std) |
                            (daily_sales['total_sales'] < rolling_mean
- 2*rolling_std)]
```

```
    plt.figure(figsize=(12, 6))
    plt.plot(daily_sales.index, daily_sales['total_sales'],
label='Daily Sales')
    plt.scatter(anomalies.index, anomalies['total_sales'],
color='red', label='Anomalies')
    plt.title('Daily Sales with Anomalies')
    plt.xlabel('Date')
    plt.ylabel('Sales')
    plt.legend()
    plt.show()

# Run all the functions
plot_decomposition()
plot_daily_heatmap()
plot_year_over_year()
# plot_boxplots()
plot_acf_pacf()
plot_cumulative_sales()
plot_seasonal_subseries()
plot_rolling_stats()
plot_fourier_transform()
detect_anomalies()
```

```
Restaurant Sales Transaction Counts From Combined Dataset:

restaurant_name
Fou Cher          30688
Bob's Diner       26304
Corner Cafe       26304
Surfs Up          10960
Sweet Shack        8768
Beachfront Bar     6576
Name: count, dtype: int64

No shared items between restaurants.

Restaurant: Beachfront Bar
Unique Items: ['Awesome Vodka Cocktail', 'Fantastic Milky Smoothy',
'Original Crazy Cocktail', 'Original Gin Cocktail', 'Original Sweet
Milky Soft Drink', 'Sweet Vegi Soft Drink']

Restaurant: Bob's Diner
Unique Items: ['Amazing Fish with Vegetables Meal', 'Amazing Frozen
Milky Cake', 'Amazing Steak Dinner with Rolls', 'Amazing Sweet Fruity
Vegetable Cake', 'Amazing pork lunch', 'Awesome Fish with Vegetables
Entree', 'Awesome Milky Cake', 'Awesome Sweet Lamb Cake', 'Blue Ribbon
Beef Entree', 'Blue Ribbon Fruity Milky Cake', 'Fantastic Frozen Milky
Cake', 'Fantastic Sweet Cola', 'Frozen Milky Smoothy', 'Milky Cake',
'Milky Vegi Smoothy', 'Mutton Dinner', 'Mutton Lunch Plate', 'Orange
```

```
Juice', 'Sea Bass with Vegetables Dinner', 'Strawberry Smoothy',
'Sweet Breaded Zucchini Cake', 'Sweet Frozen Soft Drink', 'Sweet
Fruity Cake', 'Sweet Lamb Cake']

Restaurant: Corner Cafe
Unique Items: ['Amazing Sweet Breaded Carrot Cake', 'Anglefood Cake',
'Awesome Fruity Lamb with Vegetables Dinner', 'Awesome Hamburger with
Fries', 'BBQ Pork Steak', 'Blue Ribbon Fish with Bread Lunch', 'Blue
Ribbon Lamb with Rolls Lunch', 'Fantastic Fish with Vegetables
Entree', 'Frozen Chocolate Cake', 'Frozen Milky Cake', 'Frozen Milky
Smoothy', 'Fruity Milky Soft Drink', 'Halibut with Bread Dinner',
'Lamb with Bread Entree', 'Lamb with Vegetables Meal', 'Milk Cake',
'Milky Cake', 'Mutton with Roles and Vegetables Plate', 'Original
Fruity Carrot Cake', 'Original Lamb with Vegetables Meal', 'Original
Pork Meal', 'Original Vegetable with Bread Meal', 'Pike Lunch', 'Pork
with Bread Lunch']

Restaurant: Fou Cher
Unique Items: ['Amazing Lamb Dinner', 'Amazing Vegetable and Bread
Dinner', 'Awesome Vegetable with Bread Meal', 'Blue Ribbon Fruity Vegi
Lunch', 'Blue Ribbon Milky Smoothy', 'Blue Ribbon Pork Chop with Rolls
Dinner', 'Breaded Fish with Vegetables Meal', 'Carrot Cake', 'Cherry
Cake', 'Chocolate Cake', 'Fantastic Fruity Salmon with Bread meal',
'Fantastic Sweet Fish Cake', 'Fish with Dread and Vegetables Dinner',
'Frozen Tomato Soft Drink', 'Fruity Frozen Cocktail', 'Fruity Frozen
Slushy', 'Fruity Milky Cake', 'Lamb with Bread Dinner', 'Lamb with
Bread and Vegetables Meal', 'Margarita', 'Martini', 'Milky Cake',
'Original Breaded Lamb Dinner', 'Original Fruity Cod with Bread and
Vegetables Entree', 'Original Milky Cake', 'Super Sweet Cocktail',
'Sweet Savory Cake', 'Vegetarian Plate with Bread entree']

Restaurant: Surfs Up
Unique Items: ['Amazing Cocktail', 'Amazing Trout with Vegetables
Dinner', 'Awesome Pork with Vegetables Lunch', 'Awesome Soft Drink',
'Blue Ribbon Cocktail', 'Fruity Milky Smoothy', 'Original Breaded Pork
with Vegetables Dinner', 'Oysters Rockefeller', 'Roast Mutton Entree',
'Steak Meal']

Restaurant: Sweet Shack
Unique Items: ['Awesome Smoothy', 'Blue Ribbon Frozen Milky Cake',
'Blue Ribbon Milky Cake', 'Fantastic Cake', 'Fantastic Milky Smoothy',
'Milky vegi Smoothy', 'Original Milky Cake', 'Original Sweet Milky
Soft Drink']

Restaurant: Beachfront Bar
                     item_name  price
         Awesome Vodka Cocktail   2.48
        Fantastic Milky Smoothy   2.91
        Original Crazy Cocktail   2.43
          Original Gin Cocktail   2.99
```

```
Original Sweet Milky Soft Drink   5.00
          Sweet Vegi Soft Drink   5.70
```

```
Restaurant: Bob's Diner
```

item_name	price
Amazing Fish with Vegetables Meal	23.23
Amazing Frozen Milky Cake	5.62
Amazing Steak Dinner with Rolls	26.42
Amazing Sweet Fruity Vegetable Cake	7.91
Amazing pork lunch	17.93
Awesome Fish with Vegetables Entree	23.43
Awesome Milky Cake	7.36
Awesome Sweet Lamb Cake	10.86
Blue Ribbon Beef Entree	23.37
Blue Ribbon Fruity Milky Cake	8.70
Fantastic Frozen Milky Cake	6.23
Fantastic Sweet Cola	4.87
Frozen Milky Smoothy	3.21
Milky Cake	5.16
Milky Vegi Smoothy	5.50
Mutton Dinner	10.80
Mutton Lunch Plate	18.82
Orange Juice	3.91
Sea Bass with Vegetables Dinner	28.75
Strawberry Smoothy	2.89
Sweet Breaded Zucchini Cake	7.71
Sweet Frozen Soft Drink	4.18
Sweet Fruity Cake	29.22
Sweet Lamb Cake	7.00

```
Restaurant: Corner Cafe
```

item_name	price
Amazing Sweet Breaded Carrot Cake	7.87
Anglefood Cake	7.13
Awesome Fruity Lamb with Vegetables Dinner	19.03
Awesome Hamburger with Fries	26.04
BBQ Pork Steak	19.77
Blue Ribbon Fish with Bread Lunch	21.93
Blue Ribbon Lamb with Rolls Lunch	7.55
Fantastic Fish with Vegetables Entree	21.14
Frozen Chocolate Cake	3.74
Frozen Milky Cake	6.63
Frozen Milky Smoothy	3.98
Fruity Milky Soft Drink	7.95
Halibut with Bread Dinner	15.46
Lamb with Bread Entree	12.44
Lamb with Vegetables Meal	13.66
Milk Cake	6.07
Milky Cake	7.22

```
    Mutton with Roles and Vegetables Plate  21.13
               Original Fruity Carrot Cake   8.55
        Original Lamb with Vegetables Meal  18.78
                        Original Pork Meal  15.69
        Original Vegetable with Bread Meal   6.51
                                Pike Lunch  26.37
                     Pork with Bread Lunch  16.28
```

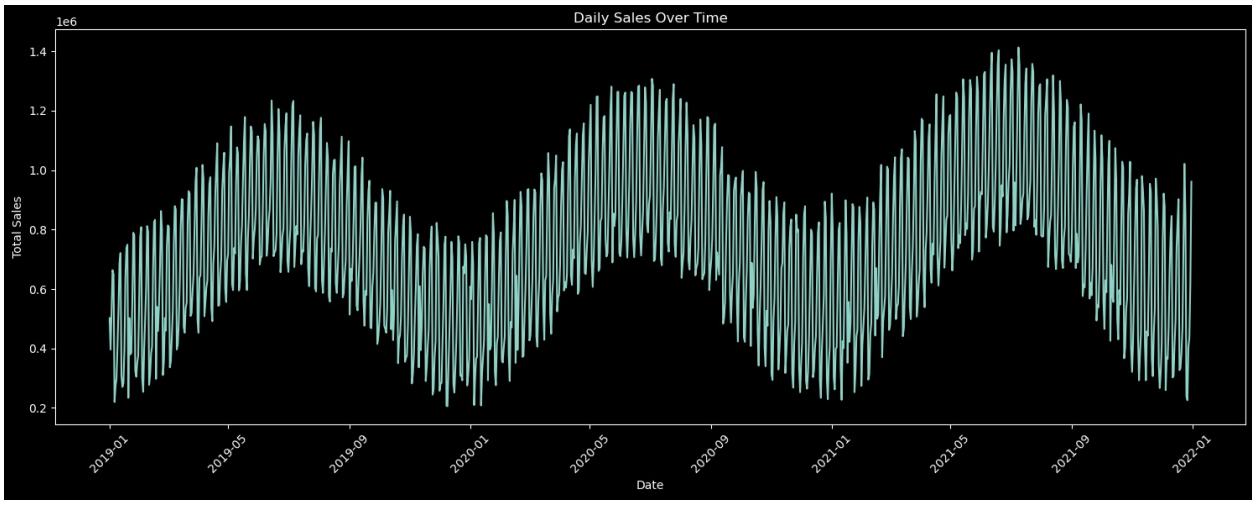
Restaurant: Fou Cher

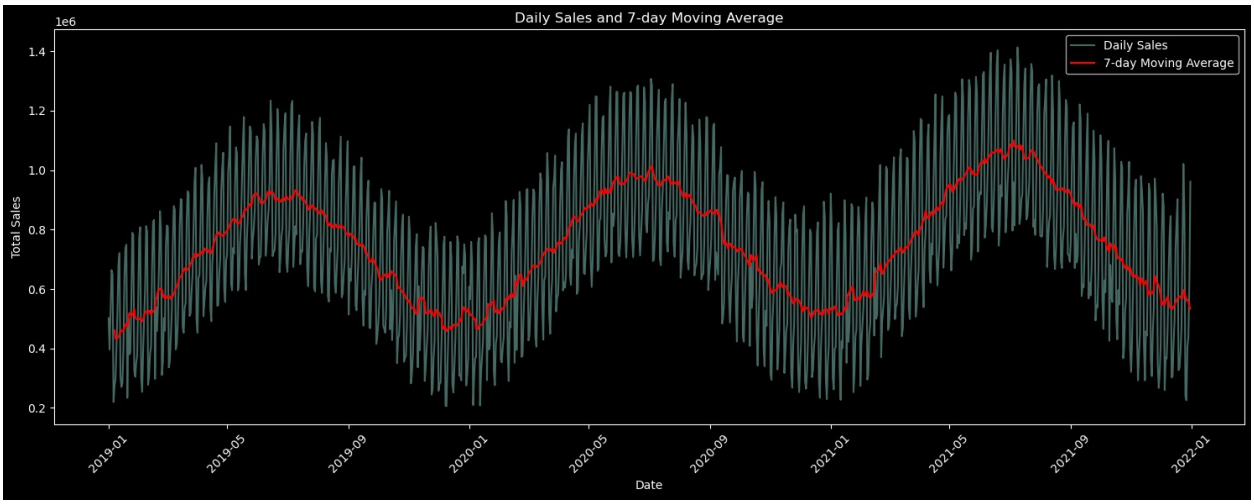
```
                                          item_name  price
                                  Amazing Lamb Dinner  17.55
                    Amazing Vegetable and Bread Dinner   8.01
                     Awesome Vegetable with Bread Meal   6.21
                         Blue Ribbon Fruity Vegi Lunch  53.98
                             Blue Ribbon Milky Smoothy   5.60
               Blue Ribbon Pork Chop with Rolls Dinner  19.04
                     Breaded Fish with Vegetables Meal  15.09
                                         Carrot Cake   3.93
                                         Cherry Cake   6.31
                                       Chocolate Cake   6.71
               Fantastic Fruity Salmon with Bread meal  22.67
                              Fantastic Sweet Fish Cake  19.48
                Fish with Dread and Vegetables Dinner  15.23
                              Frozen Tomato Soft Drink   5.32
                                Fruity Frozen Cocktail   5.61
                                  Fruity Frozen Slushy   3.75
                                     Fruity Milky Cake   8.10
                                  Lamb with Bread Dinner  19.05
                   Lamb with Bread and Vegetables Meal  20.02
                                           Margarita   3.23
                                             Martini   5.45
                                           Milky Cake   6.01
                           Original Breaded Lamb Dinner  11.82
Original Fruity Cod with Bread and Vegetables Entree  28.65
                                    Original Milky Cake   6.53
                                   Super Sweet Cocktail   4.11
                                      Sweet Savory Cake  27.47
                     Vegetarian Plate with Bread entree   4.02
```

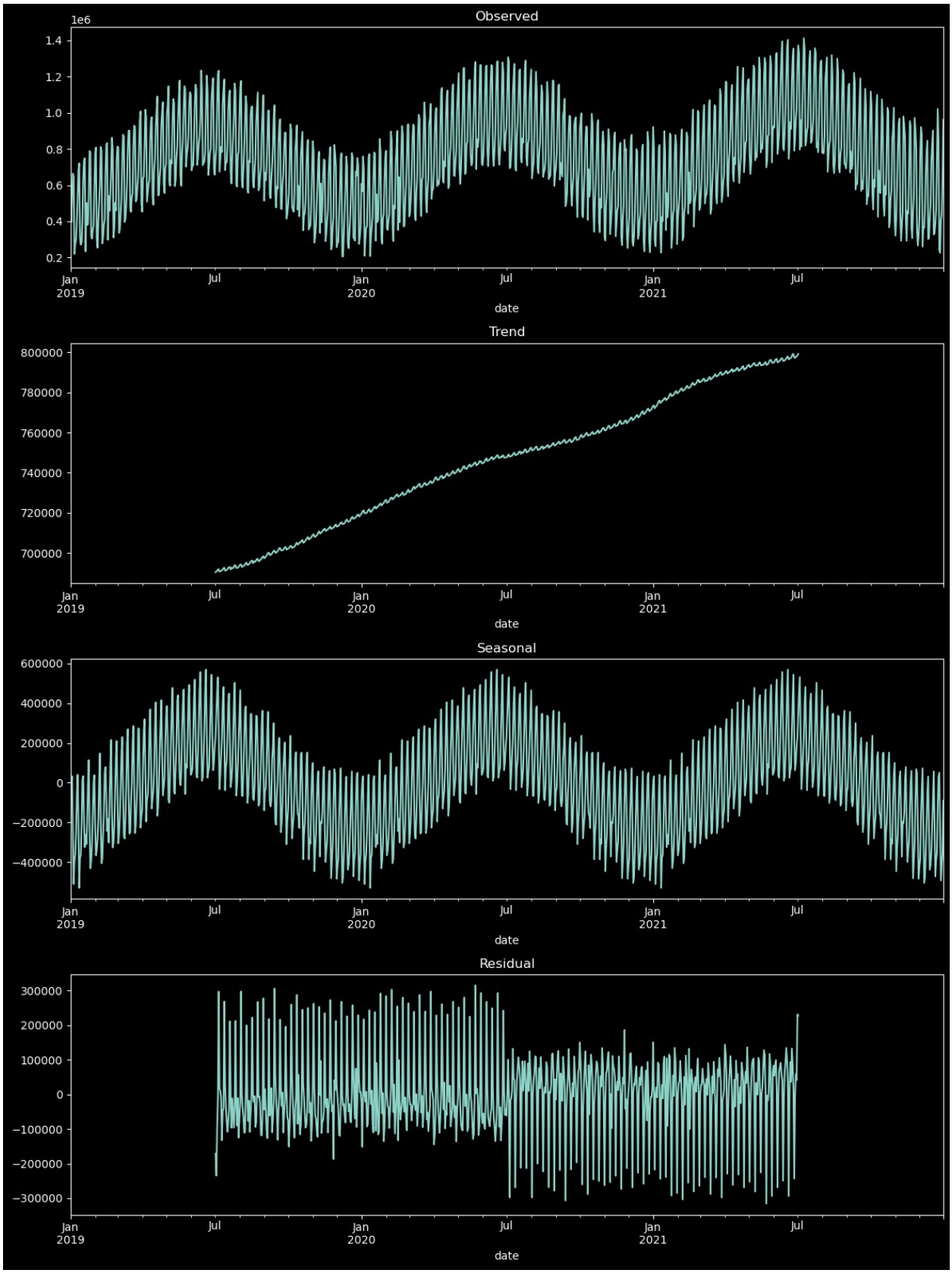
Restaurant: Surfs Up

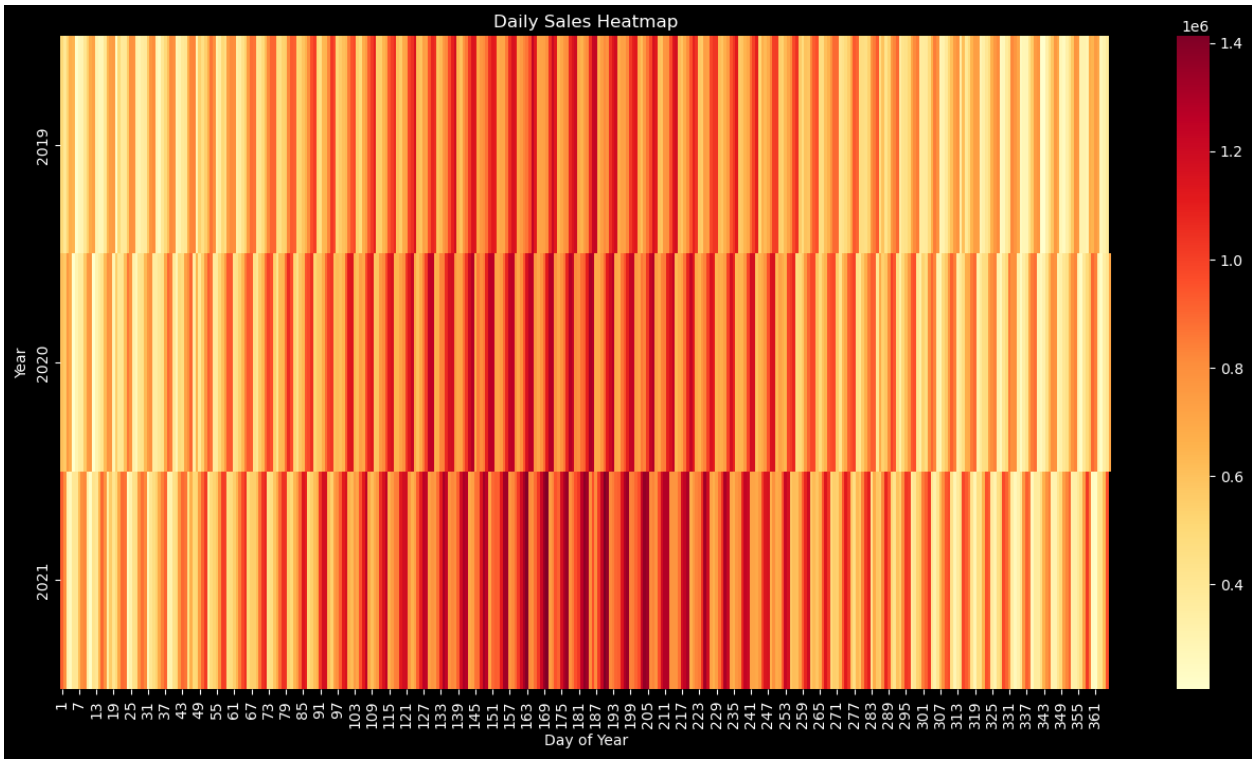
```
                                    item_name  price
                             Amazing Cocktail   3.90
          Amazing Trout with Vegetables Dinner  24.98
            Awesome Pork with Vegetables Lunch  18.53
                           Awesome Soft Drink   3.06
                         Blue Ribbon Cocktail   4.36
                         Fruity Milky Smoothy   5.71
Original Breaded Pork with Vegetables Dinner  20.80
                          Oysters Rockefeller  16.06
```

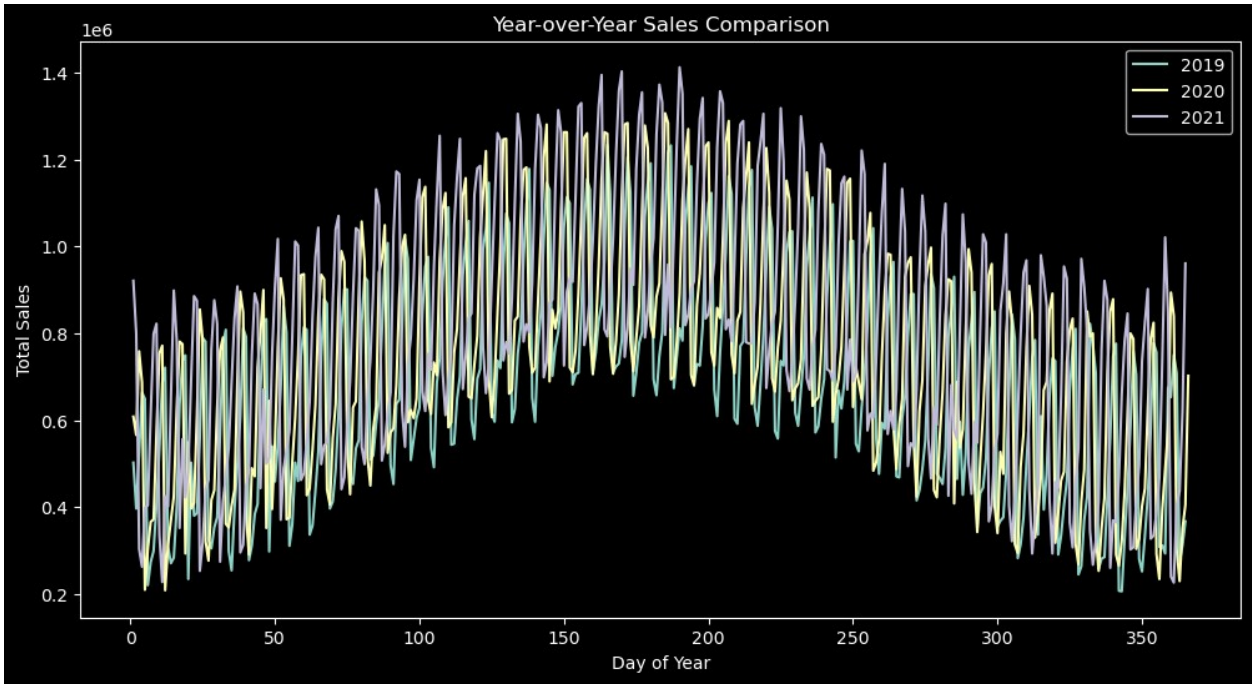
```
                        Roast Mutton Entree  16.09
                                  Steak Meal  26.21

Restaurant: Sweet Shack
                      item_name  price
                 Awesome Smoothy   1.39
  Blue Ribbon Frozen Milky Cake   7.70
         Blue Ribbon Milky Cake   6.89
                 Fantastic Cake   5.08
        Fantastic Milky Smoothy   5.11
             Milky vegi Smoothy   3.86
            Original Milky Cake   6.50
Original Sweet Milky Soft Drink   5.68

Items with price variability:
Item: Fantastic Milky Smoothy, Std Dev: 1.10, Restaurants: Beachfront
Bar, Sweet Shack
Item: Frozen Milky Smoothy, Std Dev: 0.39, Restaurants: Bob's Diner,
Corner Cafe
Item: Milky Cake, Std Dev: 0.85, Restaurants: Bob's Diner, Fou Cher,
Corner Cafe
Item: Original Milky Cake, Std Dev: 0.02, Restaurants: Fou Cher, Sweet
Shack
Item: Original Sweet Milky Soft Drink, Std Dev: 0.34, Restaurants:
Sweet Shack, Beachfront Bar
```

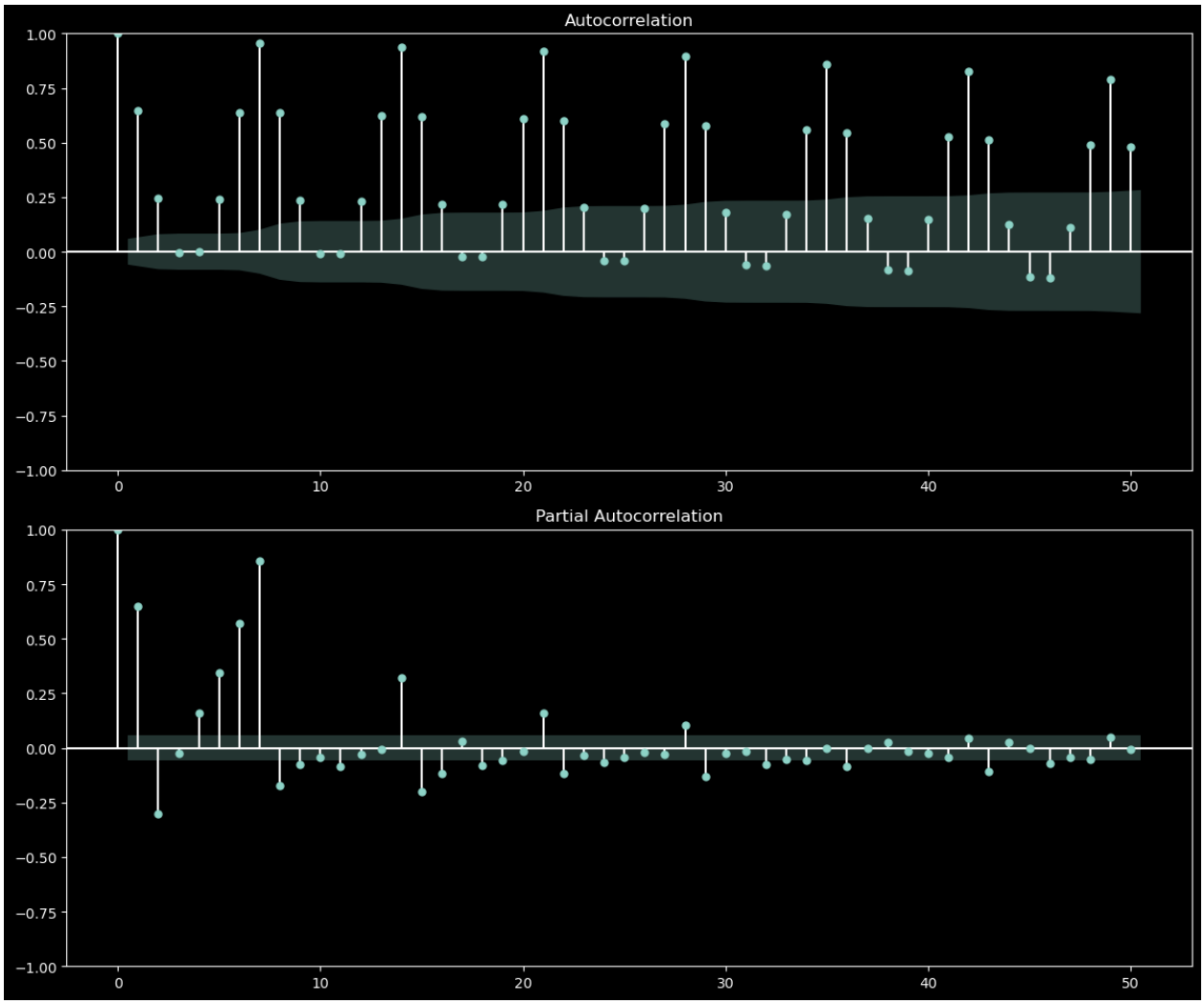


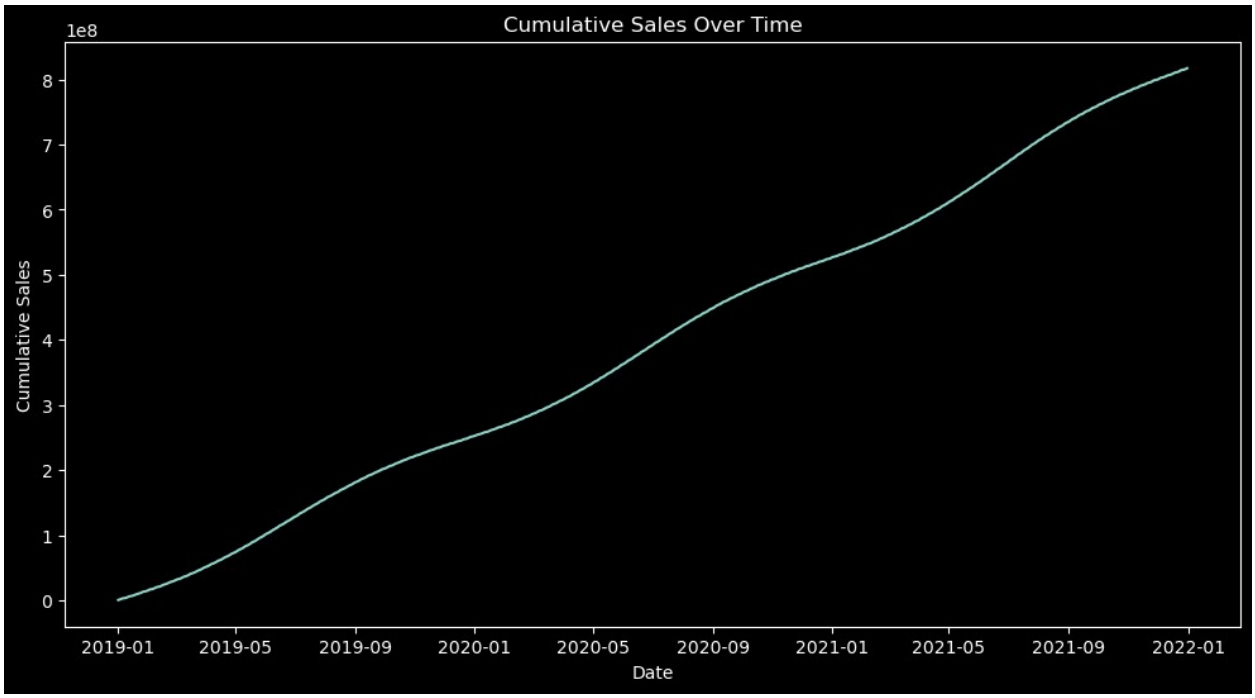

Daily Sales and 7-day Moving Average
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
Total Sales
Daily Sales
7-day Moving Average
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01
Date

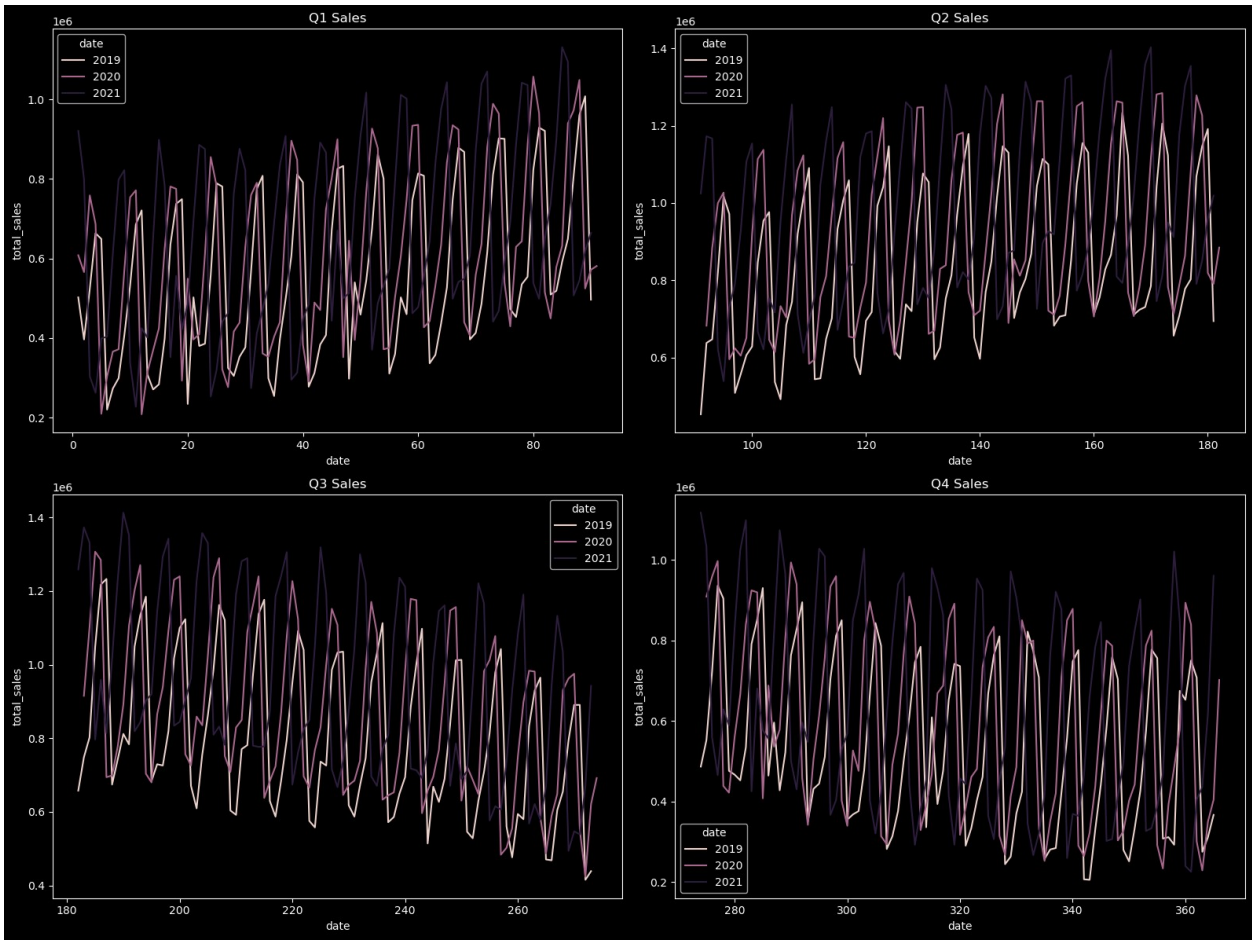

Observed
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
Jan
2019
Jul
Jan
2020
Jul
Jan
2021
Jul
date
Trend
800000
780000
760000
740000
720000
700000
Jan
2019
Jul
Jan
2020
Jul
Jan
2021
Jul
date
Seasonal
600000
400000
200000
0
−200000
−400000
Jan
2019
Jul
Jan
2020
Jul
Jan
2021
Jul
date
Residual
300000
200000
100000
0
−100000
−200000
−300000
Jan
2019
Jul
Jan
2020
Jul
Jan
2021
Jul
date

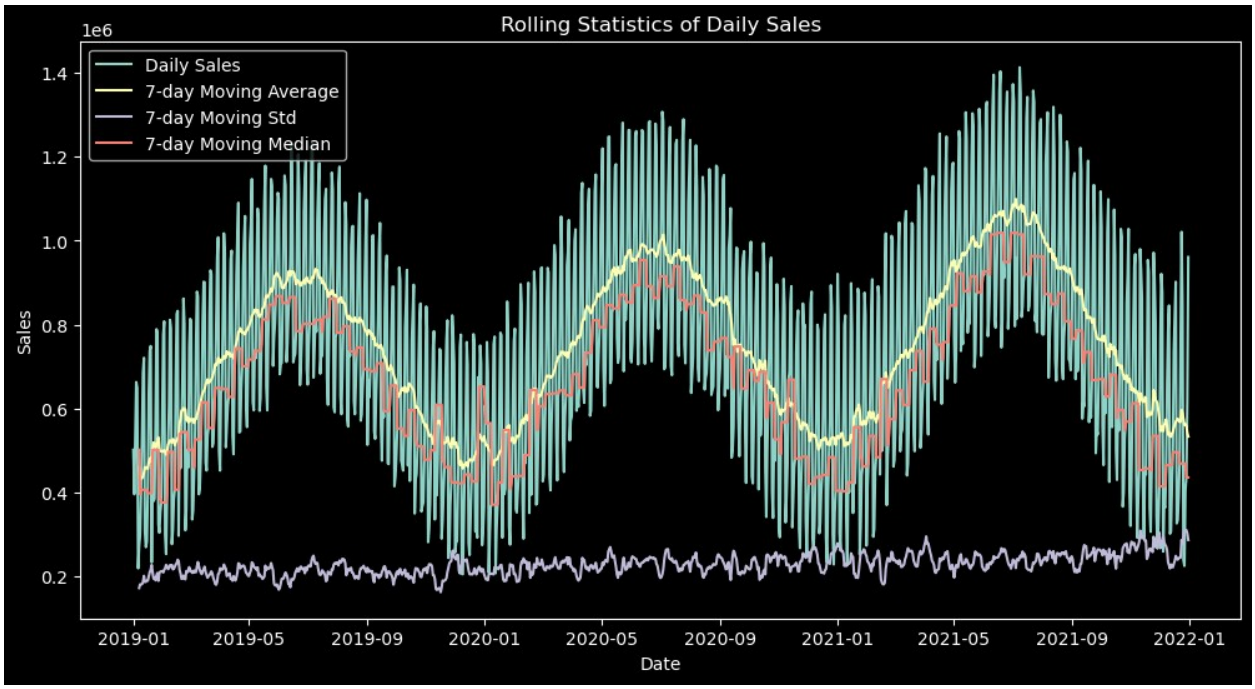

Daily Sales Heatmap
Year
2019
2020
2021
Day of Year
1e6
1.4
1.2
1.0
0.8
0.6
0.4

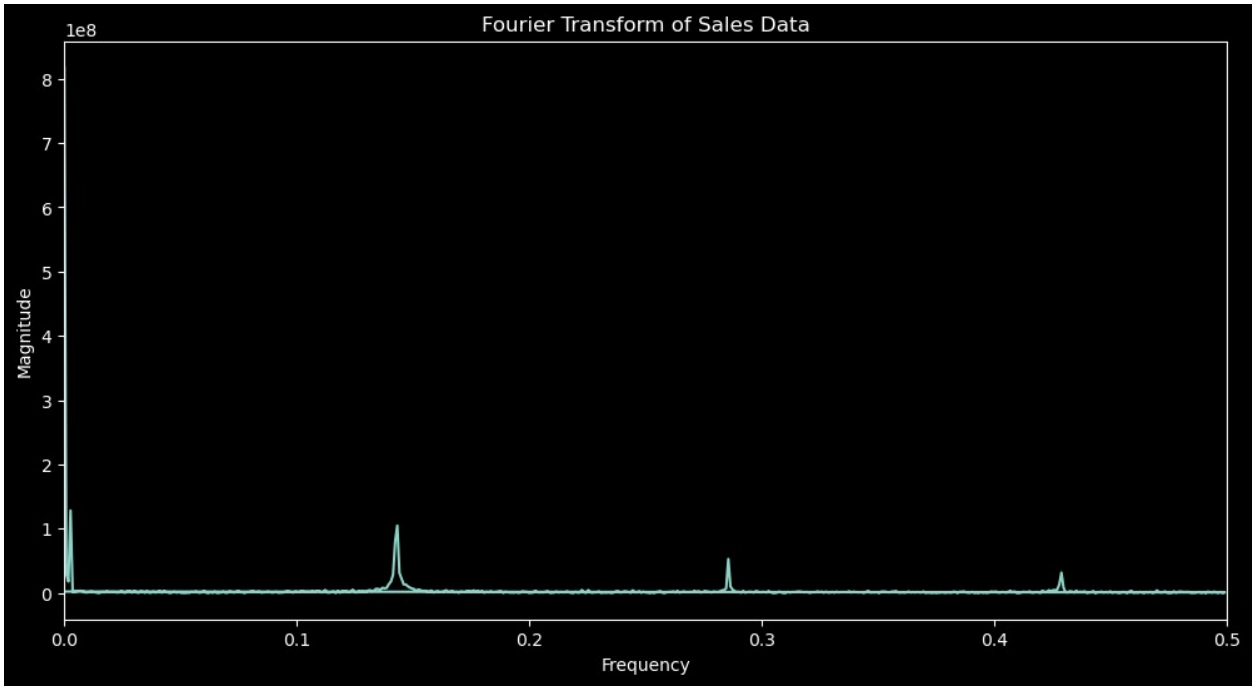

Year-over-Year Sales Comparison
1e6
Total Sales
Day of Year
2019
2020
2021
0
50
100
150
200
250
300
350
0.2
0.4
0.6
0.8
1.0
1.2
1.4

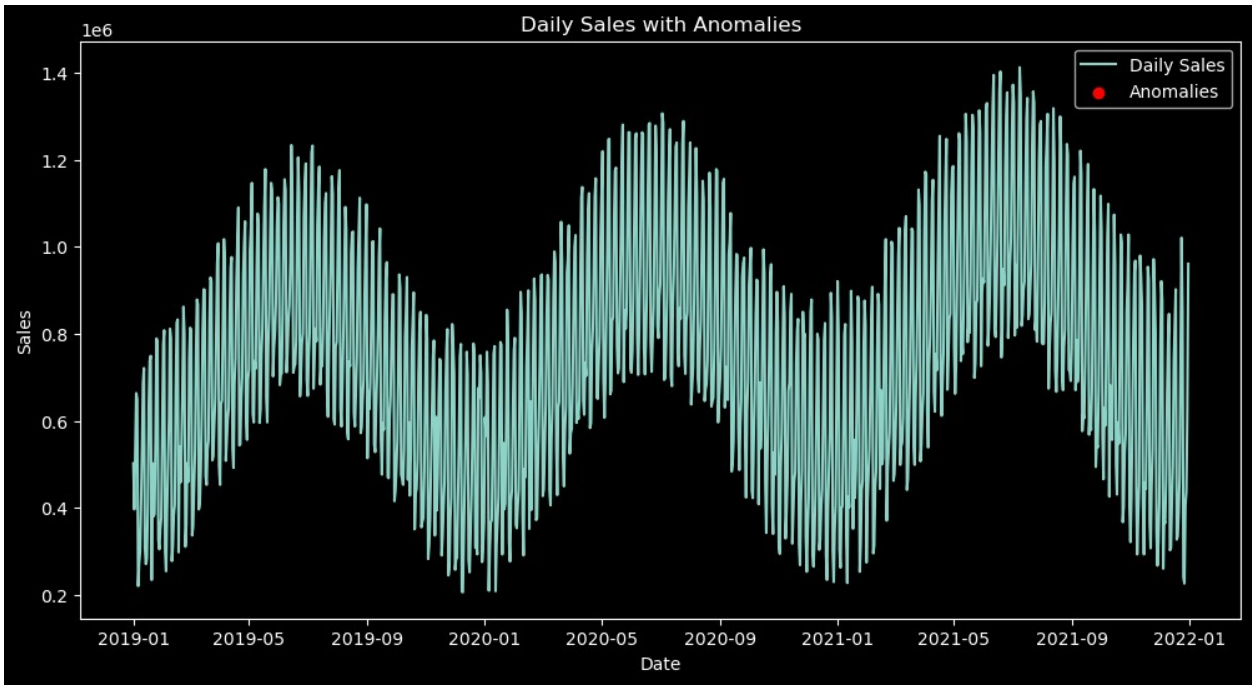

Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50
Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50


Cumulative Sales Over Time
1e8
Cumulative Sales
Date
0
1
2
3
4
5
6
7
8
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01


Q1 Sales
Q2 Sales
Q3 Sales
Q4 Sales
1e6
total_sales
date
2019
2020
2021


Rolling Statistics of Daily Sales
1e6
Daily Sales
7-day Moving Average
7-day Moving Std
7-day Moving Median
Sales
Date
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01


Fourier Transform of Sales Data
1e8
Magnitude
Frequency



Explanations:

1. Data Overview and Aggregation
   - Counts sales transactions per restaurant
   - Identifies common items across restaurants
   - Lists unique items for each restaurant
   - Calculates average prices for items at each restaurant
   - Analyzes price variability across restaurants
2. Time Series Analysis
   - Creates daily sales totals
   - Plots daily sales over time
   - Calculates and plots a 7-day moving average
3. Advanced Time Series Analysis
   - **Seasonal Decomposition:** Separates the time series into trend, seasonality, and residual components
   - **Heatmap of Daily Sales:** Visualizes sales patterns across years and days
   - **Year-over-Year Comparison:** Compares sales patterns across different years
   - **Box Plots by Month and Day of Week:** Shows sales distribution patterns
   - **Autocorrelation and Partial Autocorrelation Plots:** Identifies time-dependent patterns and potential forecasting models
   - **Cumulative Sales Plot:** Visualizes overall growth trends
   - **Seasonal Subseries Plot:** Compares seasonal patterns across years
   - **Rolling Statistics:** Provides smoothed trends and variability measures
   - **Fourier Transform:** Identifies dominant frequencies in the sales data

- **Anomaly Detection:** Highlights unusual sales days

Why It Is Important:

In the data science process, this comprehensive EDA is crucial because it:

1. Helps understand the underlying structure and patterns in the data
2. Guides feature engineering for predictive modeling
3. Informs the selection of appropriate forecasting models
4. Identifies potential issues or anomalies in the data
5. Provides business insights for strategic decision-making
6. Helps in formulating hypotheses for further analysis
7. Aids in communicating findings to stakeholders through visualizations

- Provides a high-level understanding of the dataset structure, product offerings, and pricing strategies across different restaurants.
- These visualizations help identify overall trends and smooth out daily fluctuations for clearer pattern recognition.
- These analyses provide deep insights into various aspects of the sales data:
  - Seasonal patterns and trends
  - Weekly and monthly variations
  - Year-over-year growth
  - Potential forecasting model structures
  - Unusual events or outliers

Observations:

1. **Seasonal Patterns:**
   - Strong yearly seasonality is evident, with peaks occurring around the same time each year.
   - Weekly patterns are visible, with sales generally higher on weekends (Friday and Saturday) and lower on Sundays.
2. **Long-term Trend:**
   - There's a clear upward trend in sales over the three-year period from 2019 to 2021.
3. **Daily Variations:**
   - Significant day-to-day fluctuations in sales are present.
4. **Yearly Comparison:**
   - Sales patterns are similar across years, but with noticeable growth year-over-year.
5. **Monthly Distribution:**
   - Sales tend to be higher in mid-year months (June-August) and lower in early and late months of the year.
6. **Autocorrelation:**
   - Strong positive autocorrelation at lag 7, indicating weekly patterns.
   - Decreasing but significant autocorrelation at larger lags, suggesting longer-term trends.

7. **Fourier Transform:**
   - Clear peaks at certain frequencies, confirming the presence of regular cyclical patterns.
8. **Anomalies:**
   - Few anomalies detected, mostly during peak sales periods.

Conclusions:

1. **Seasonality:** The business has strong seasonal components, both on a yearly and weekly basis.
2. **Growth:** The company is experiencing consistent growth over the observed period.
3. **Day of Week Impact:** Weekends are crucial for sales, while Sundays see a significant drop.
4. **Yearly Consistency:** The overall sales pattern remains consistent year to year, despite growth.
5. **Summer Peak:** Mid-year (summer) months generally outperform other seasons.
6. **Predictable Patterns:** The strong autocorrelation and clear Fourier transform peaks indicate highly predictable sales patterns.
7. **Stability:** Few anomalies suggest a relatively stable business model with predictable fluctuations.

Recommendations:

1. **Seasonal Staffing:** Adjust staffing levels to match the observed seasonal patterns, ensuring adequate coverage during peak periods.
2. **Inventory Management:** Optimize inventory based on the predictable seasonal and weekly patterns to minimize waste and stockouts.
3. **Marketing Campaigns:** Time marketing efforts to coincide with typically slower periods to boost sales during these times.
4. **Weekend Focus:** Develop strategies to maximize revenue during peak weekend days (Friday and Saturday).
5. **Sunday Specials:** Create promotions or events to increase Sunday sales and smooth out the weekly pattern.
6. **Year-Round Growth Strategies:** While maintaining focus on peak seasons, develop strategies to boost sales during typically slower months.
7. **Capacity Planning:** Use the observed growth trend to plan for increased capacity in the coming years.

8. **Anomaly Investigation:** Analyze the few detected anomalies to understand their causes and potentially replicate positive outliers.
9. **Forecasting Model:** Develop a sales forecasting model incorporating the observed seasonality and trends for better business planning.
10. **Customer Behavior Analysis:** Conduct deeper analysis into customer behavior during different seasons and days of the week to tailor offerings.
11. **Menu Optimization:** Adjust menu items seasonally based on observed sales patterns to maximize profitability.
12. **Expansion Consideration:** Given the consistent growth, consider expanding to new locations or markets.

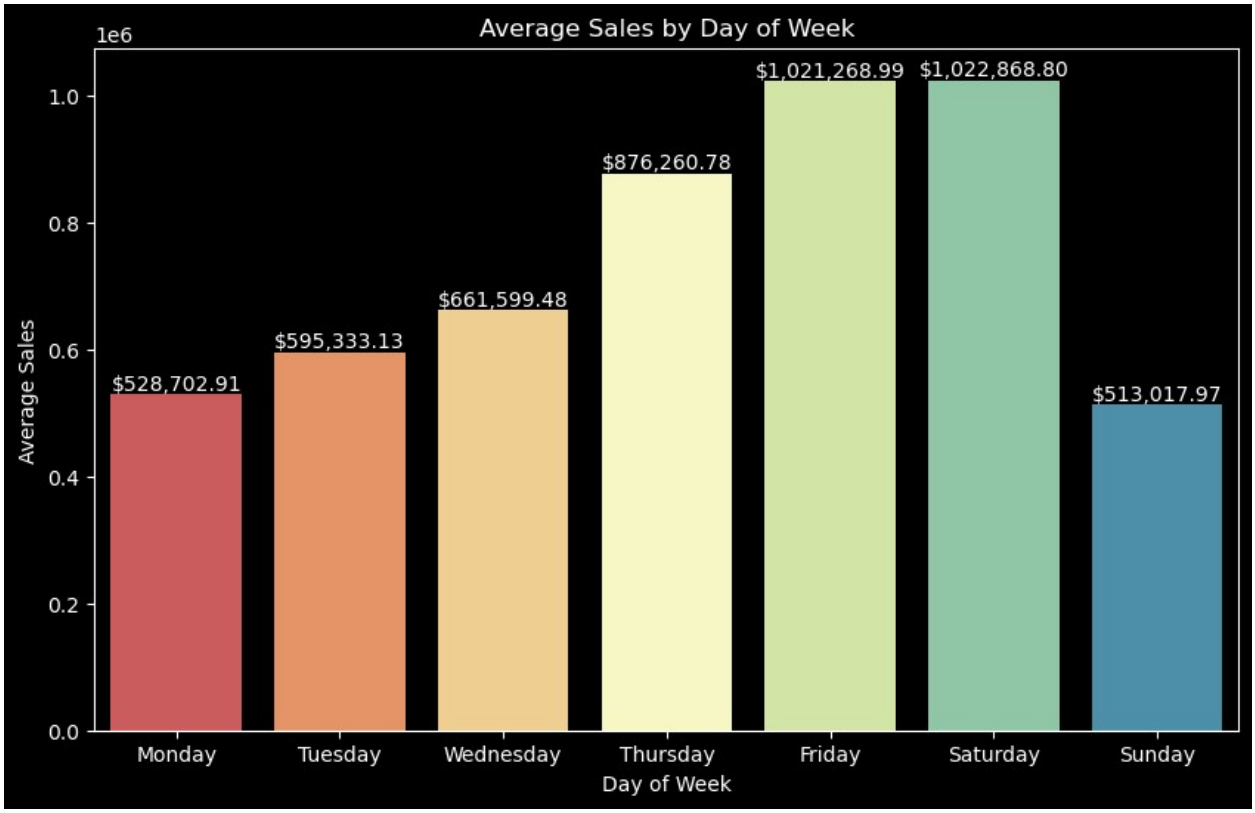
4.2.2. Find out how sales fluctuate across different days of the week

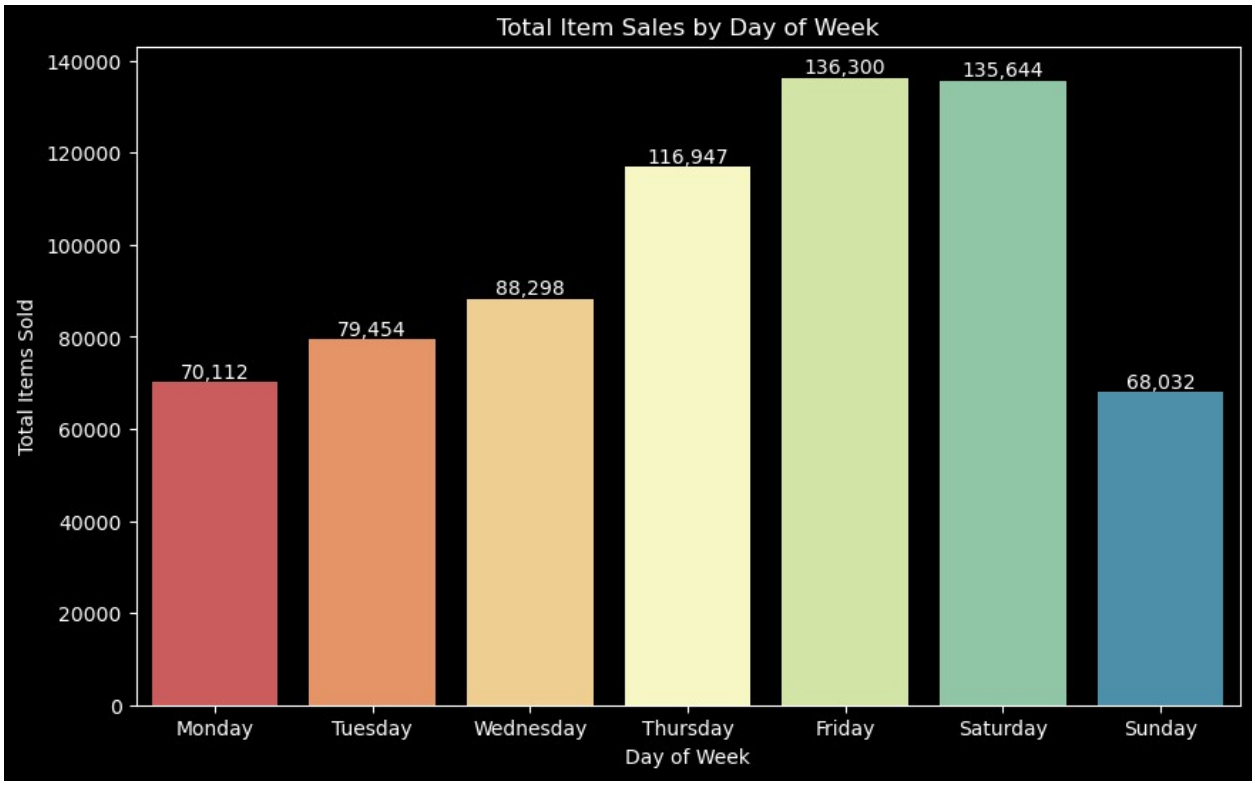
```
# Add day of week column
daily_sales['day_of_week'] = daily_sales.index.dayofweek
daily_sales['day_of_week_name'] = daily_sales.index.day_name()

# Calculate average sales by day of week
avg_sales_by_day = daily_sales.groupby('day_of_week_name')
['total_sales'].mean().reindex(['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday', 'Sunday'])
total_sales_by_day = daily_sales.groupby('day_of_week_name')
['item_count'].sum().reindex(['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday', 'Sunday'])

# Plot average sales by day of week
plt.figure(figsize=(10, 6))
sns.barplot(x=avg_sales_by_day.index, y=avg_sales_by_day.values,
palette='Spectral')
plt.title('Average Sales by Day of Week')
plt.xlabel('Day of Week')
plt.ylabel('Average Sales')

# Annotate each bar with the average sales value formatted with commas
and dollar signs
for index, value in enumerate(avg_sales_by_day.values):
    plt.text(index, value, f'${value:,.2f}', ha='center', va='bottom')

plt.show()

plt.figure(figsize=(10, 6))
sns.barplot(x=total_sales_by_day.index, y=total_sales_by_day.values,
palette='Spectral')
plt.title('Total Item Sales by Day of Week')
plt.xlabel('Day of Week')
plt.ylabel('Total Items Sold')
```

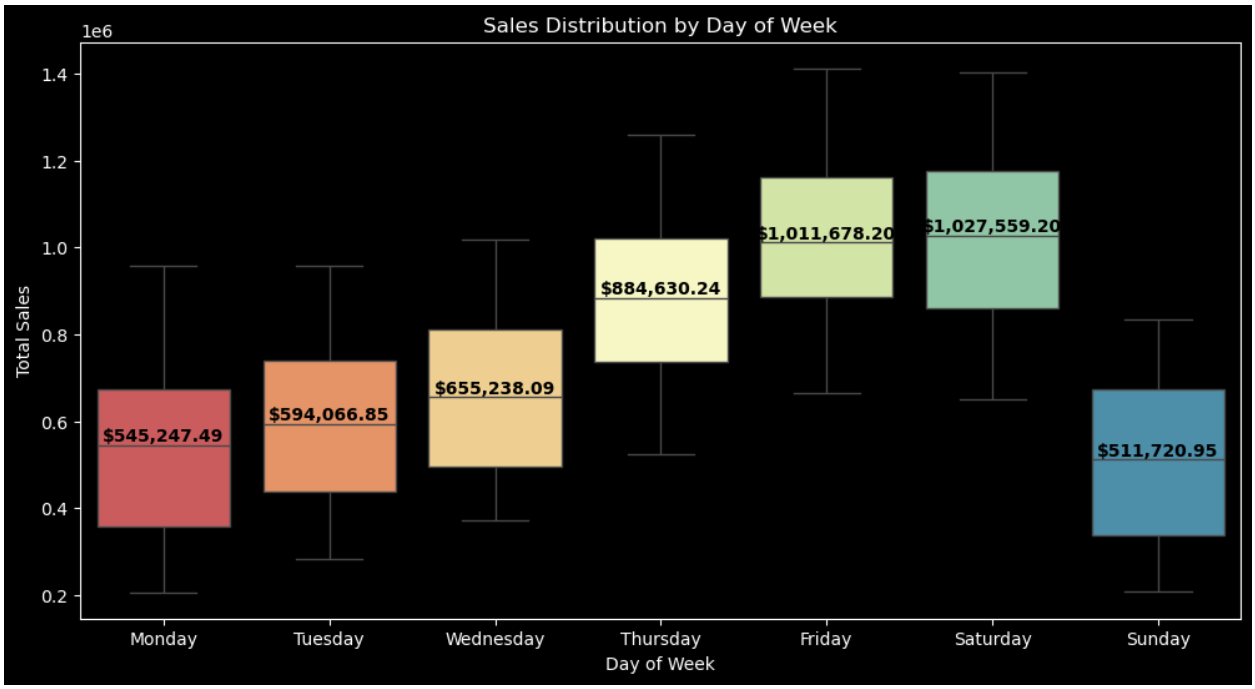
```
# Annotate each bar with the total items sold value formatted with
commas
for index, value in enumerate(total_sales_by_day.values):
    plt.text(index, value, f'{value:,.0f}', ha='center', va='bottom')
plt.show()

# Define the order of the days
days_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']

# Boxplot of sales by day of week
plt.figure(figsize=(12, 6))
sns.boxplot(x='day_of_week_name', y='total_sales', data=daily_sales,
order=days_order, palette='Spectral')
plt.title('Sales Distribution by Day of Week')
plt.xlabel('Day of Week')
plt.ylabel('Total Sales')

# Annotate the median values
medians = daily_sales.groupby('day_of_week_name')
['total_sales'].median().reindex(days_order)
for index, median in enumerate(medians):
    plt.text(index, median, f'${median:,.2f}', ha='center',
va='bottom', color='black', weight='bold')

plt.show()

# 3. Day of Week Trend Over Time
def plot_day_of_week_trend():
    daily_sales['year_month'] = daily_sales.index.to_period('M')
    trend_data = daily_sales.groupby(['year_month',
'day_of_week_name'])['total_sales'].mean().unstack()
    ax = trend_data.plot(figsize=(12, 6), colormap='Spectral')
    plt.title('Day of Week Sales Trend Over Time')
    plt.xlabel('Date')
    plt.ylabel('Average Sales')
    plt.legend(title='Day of Week', bbox_to_anchor=(1.05, 1),
loc='upper left')
    plt.tight_layout()
    plt.show()

# 4. Average Order Value by Day
def plot_avg_order_value():
    avg_order = daily_sales.groupby('day_of_week_name')
['total_sales'].mean() / daily_sales.groupby('day_of_week_name')
['item_count'].mean()
    ax = avg_order.plot(kind='bar', figsize=(10, 6),
color=sns.color_palette('Spectral', len(avg_order)))
    plt.title('Average Order Value by Day of Week')
    plt.xlabel('Day of Week')
```

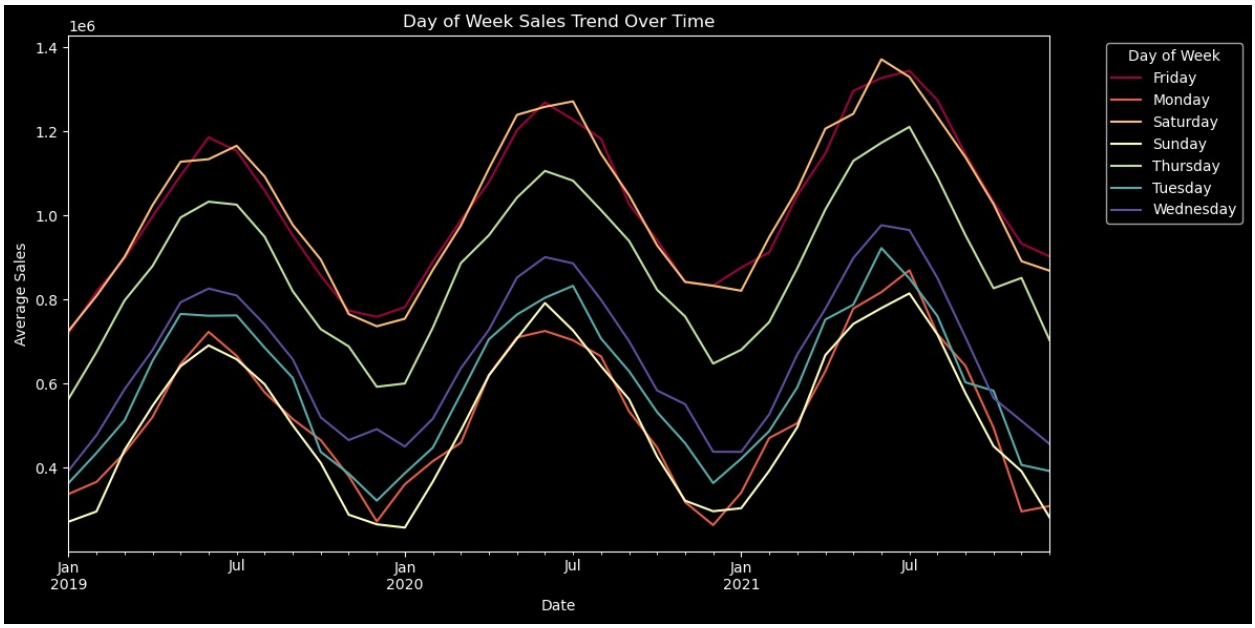
```
    plt.ylabel('Average Order Value')

    # Annotate each bar with the average order value formatted with
commas and dollar signs
    for index, value in enumerate(avg_order.values):
        plt.text(index, value, f'${value:,.2f}', ha='center',
va='bottom')

    plt.show()

# 6. Violin Plot of Sales Distribution
def plot_sales_violin():
    plt.figure(figsize=(12, 6))
    sns.violinplot(x='day_of_week_name', y='total_sales',
data=daily_sales, order=days_order, palette='Spectral')
    plt.title('Sales Distribution by Day of Week (Violin Plot)')
    plt.xlabel('Day of Week')
    plt.ylabel('Total Sales')

    # Annotate the median values
    medians = daily_sales.groupby('day_of_week_name')
['total_sales'].median().reindex(days_order)
    for index, median in enumerate(medians):
        plt.text(index, median, f'${median:,.2f}', ha='center',
va='bottom', color='black', weight='bold')

    plt.show()

# 9. Sales Variance Plot
def plot_sales_variance():
    cv = daily_sales.groupby('day_of_week_name')
['total_sales'].agg(lambda x: x.std() / x.mean())
    ax = cv.plot(kind='bar', figsize=(10, 6),
color=sns.color_palette('Spectral', len(cv)))
    plt.title('Coefficient of Variation of Sales by Day of Week')
    plt.xlabel('Day of Week')
    plt.ylabel('Coefficient of Variation')

    # Annotate each bar with the coefficient of variation value
formatted with commas
    for index, value in enumerate(cv.values):
        plt.text(index, value, f'{value:,.2f}', ha='center',
va='bottom')

    plt.show()

# 10. Day-to-Day Sales Transition Heatmap
def plot_day_to_day_transition():
    daily_sales['next_day_sales'] = daily_sales['total_sales'].shift(-
1)
```

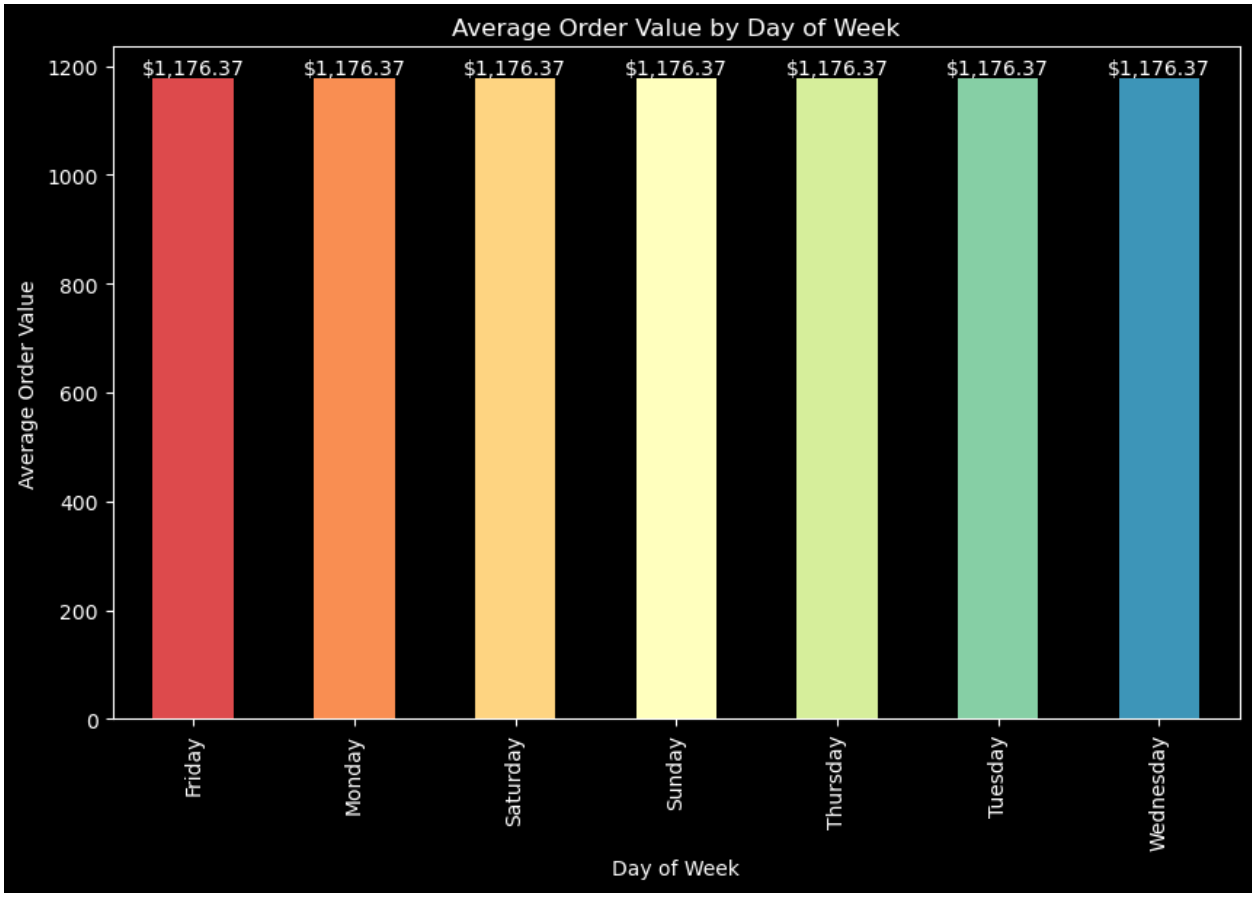
```
    transition = daily_sales.groupby(['day_of_week_name',
daily_sales['day_of_week_name'].shift(-1)])[['total_sales',
'next_day_sales']].mean()
    transition = transition['next_day_sales'].unstack()
    plt.figure(figsize=(10, 8))
    sns.heatmap(transition, annot=True, fmt=".2f", cmap="Spectral")
    plt.title('Day-to-Day Sales Transition')
    plt.xlabel('Next Day')
    plt.ylabel('Current Day')
    plt.show()


# Call the functions to generate the plots
plot_day_of_week_trend()
plot_avg_order_value()
plot_sales_violin()
plot_sales_variance()
plot_day_to_day_transition()
```

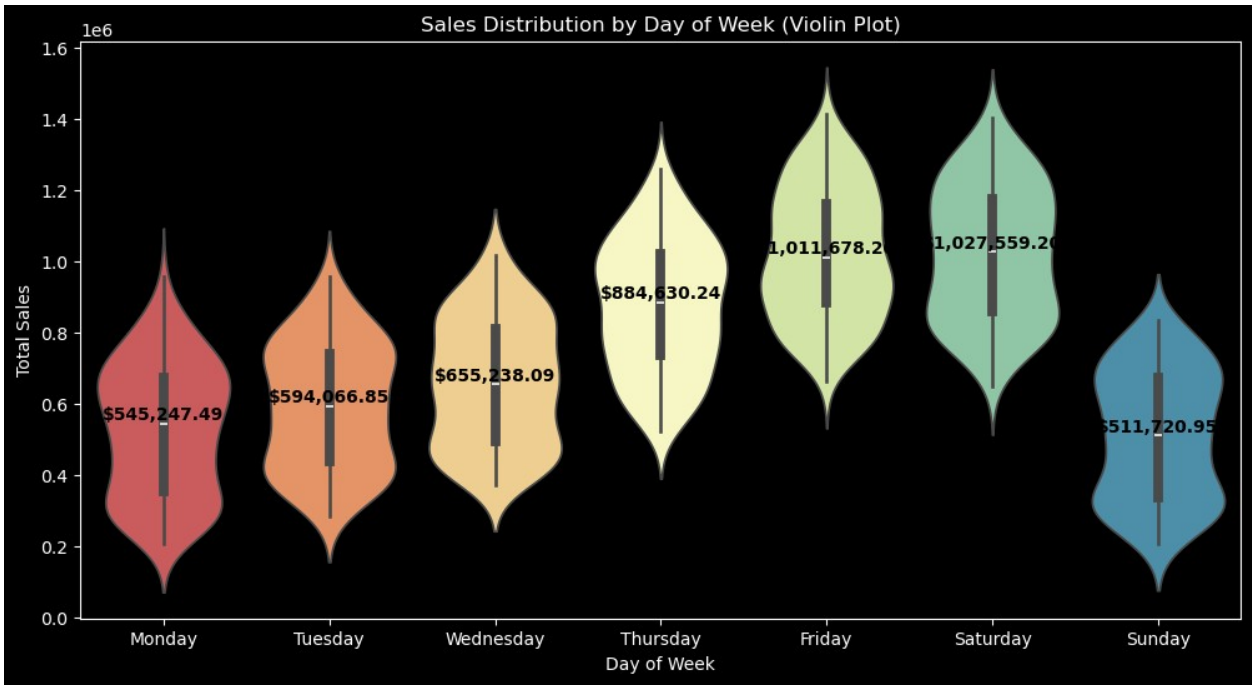


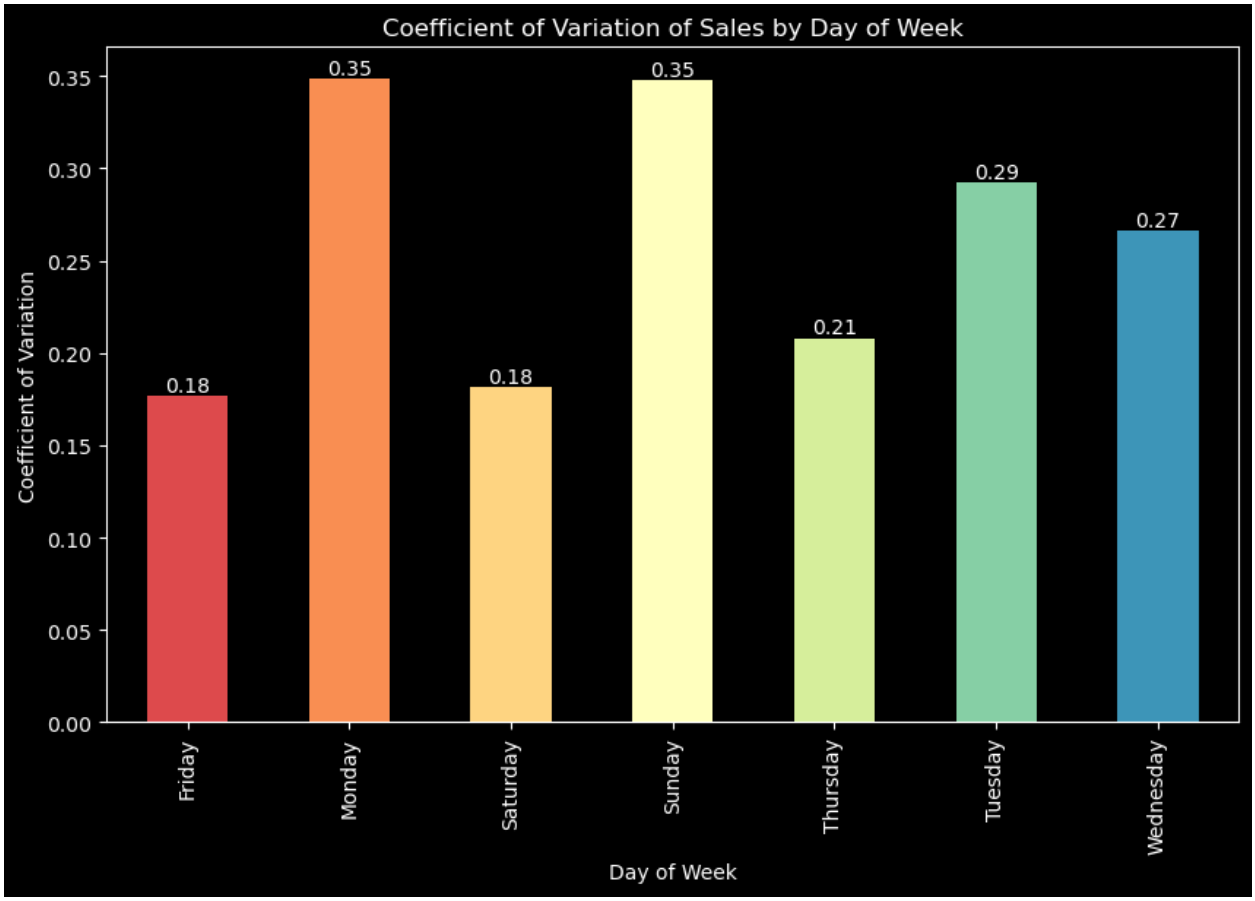

Total Item Sales by Day of Week
Total Items Sold
140000
120000
100000
80000
60000
40000
20000
0
70,112
79,454
88,298
116,947
136,300
135,644
68,032
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

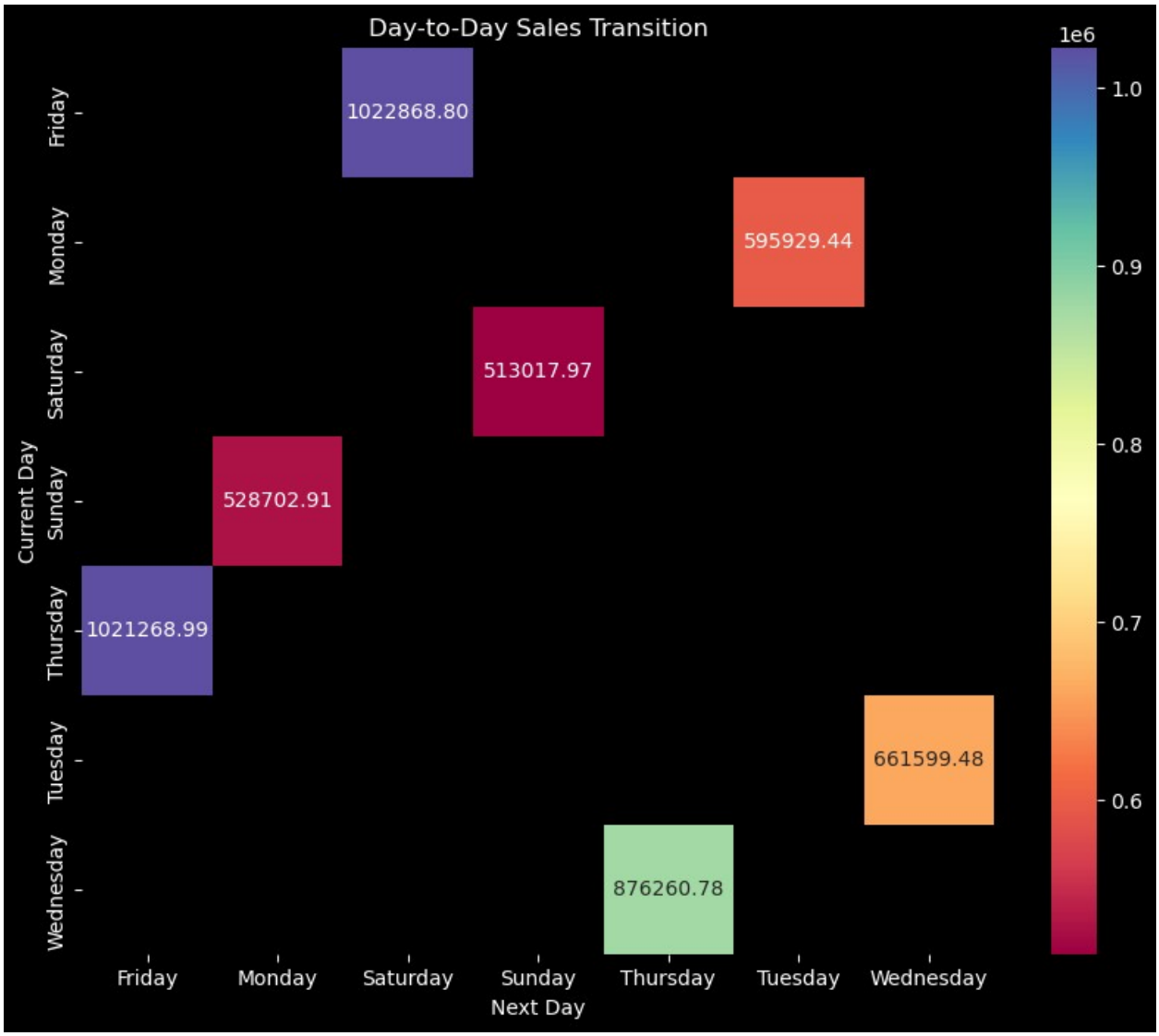

Sales Distribution by Day of Week
1e6
Total Sales
1.4
1.2
1.0
0.8
0.6
0.4
0.2
$545,247.49
$594,066.85
$655,238.09
$884,630.24
$1,011,678.20
$1,027,559.20
$511,720.95
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week


Day of Week Sales Trend Over Time
1e6
1.4
1.2
1.0
0.8
0.6
0.4
Average Sales
Jan
2019
Jul
Jan
2020
Jul
Jan
2021
Jul
Date
Day of Week
Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday


Average Order Value by Day of Week
$1,176.37
$1,176.37
$1,176.37
$1,176.37
$1,176.37
$1,176.37
$1,176.37
1200
1000
800
600
400
200
0
Average Order Value
Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday
Day of Week


Sales Distribution by Day of Week (Violin Plot)
1e6
Total Sales
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
$545,247.49
$594,066.85
$655,238.09
$884,630.24
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week


Coefficient of Variation of Sales by Day of Week
Coefficient of Variation
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.18
0.35
0.18
0.35
0.21
0.29
0.27
Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday
Day of Week



Explanations:

- This code analyzes sales fluctuations across different days of the week. It calculates average sales for each day and creates a bar plot to visualize these averages. Additionally, it generates a box plot to show the distribution of sales for each day of the week.

Why It Is Important:

- Understanding weekly sales patterns is crucial for inventory management, staffing decisions, and marketing strategies. It can help businesses optimize their operations based on expected demand for different days of the week.

Observations:

1. **Sales Progression:**
   - Sales steadily increase from Monday to Saturday, with Friday and Saturday being the peak days.

   - Sunday shows a significant drop, having the lowest sales of the week.
2. **Peak Days:**
   - Friday and Saturday consistently outperform other days in average sales, total items sold, and sales distribution.
   - Saturday slightly edges out Friday in average sales ($971,591 vs $963,992).
3. **Weekend Effect:**
   - While Friday and Saturday show peak performance, Sunday unexpectedly has the lowest sales.
4. **Mid-Week Performance:**
   - Wednesday and Thursday show moderate sales, higher than early week but lower than Friday/Saturday.
5. **Sales Variability:**
   - Tuesday shows the highest coefficient of variation, indicating more inconsistent sales.
   - Friday and Saturday, despite having highest sales, show relatively low variability.
6. **Average Order Value:**
   - Highest on Sundays ($7,542), despite having the lowest total sales.
   - Relatively consistent across weekdays, with a slight increase on weekends.
7. **Sales Distribution:**
   - Violin plots show Friday and Saturday have not only higher sales but also wider distributions, indicating potential for very high sales days.
   - Sunday's distribution is narrower, suggesting more consistent (though lower) sales.
8. **Day-to-Day Transitions:**
   - Largest positive transition is from Thursday to Friday.
   - Largest negative transition is from Saturday to Sunday.

Conclusions:

1. **Weekend Dominance:** Friday and Saturday are crucial for business, generating significantly higher sales.
2. **Sunday Anomaly:** Despite being a weekend day, Sunday consistently underperforms all other days.
3. **Weekday Buildup:** There's a clear pattern of sales building up throughout the week.
4. **Consistent Patterns:** The trends are consistent across different metrics (average sales, total items, distribution).
5. **Order Value Inverse to Volume:** Sunday's high average order value contrasts with its low total sales, suggesting fewer but larger transactions.
6. **Mid-Week Stability:** Wednesday and Thursday show stable, moderate performance.

7. **Tuesday Volatility:** Tuesday sales are the most unpredictable, potentially due to varying promotional activities or external factors.

Recommendations:

1. **Maximize Weekend Potential:**
   - Ensure optimal staffing and inventory for Fridays and Saturdays.
   - Implement strategies to handle higher customer volume efficiently.
2. **Sunday Revival Strategy:**
   - Develop special Sunday promotions or events to boost foot traffic.
   - Capitalize on the higher average order value by encouraging more transactions.
3. **Early Week Boost:**
   - Create targeted marketing campaigns for Monday and Tuesday to increase sales on slower days.
   - Investigate the cause of Tuesday's high variability and develop strategies to stabilize sales.
4. **Staffing Optimization:**
   - Adjust staff schedules to align with the weekly sales pattern.
   - Ensure experienced staff are available during peak times (Friday and Saturday).
5. **Inventory Management:**
   - Adjust stock levels and preparation to match the weekly sales cycle.
   - Implement a just-in-time inventory system for perishables, considering the weekly pattern.
6. **Promotional Calendar:**
   - Design a promotional calendar that complements the natural weekly flow.
   - Consider running promotions early in the week to boost slower days.
7. **Customer Experience Focus:**
   - Enhance customer experience during peak days to encourage repeat visits on slower days.
   - Implement loyalty programs that incentivize visits on typically slower days.
8. **Data-Driven Decision Making:**
   - Continuously monitor these patterns and adjust strategies based on ongoing data analysis.
   - Conduct customer surveys to understand preferences and behaviors driving these patterns.
9. **Operational Efficiency:**
   - Streamline operations for busy weekends to handle higher volumes efficiently.
   - Use slower days for staff training, maintenance, and preparation for busier periods.

4.2.3. Look for any noticeable trends in the sales data for different months of the year

```
# Add month column
daily_sales['month'] = daily_sales.index.month
daily_sales['month_name'] = daily_sales.index.month_name()
```

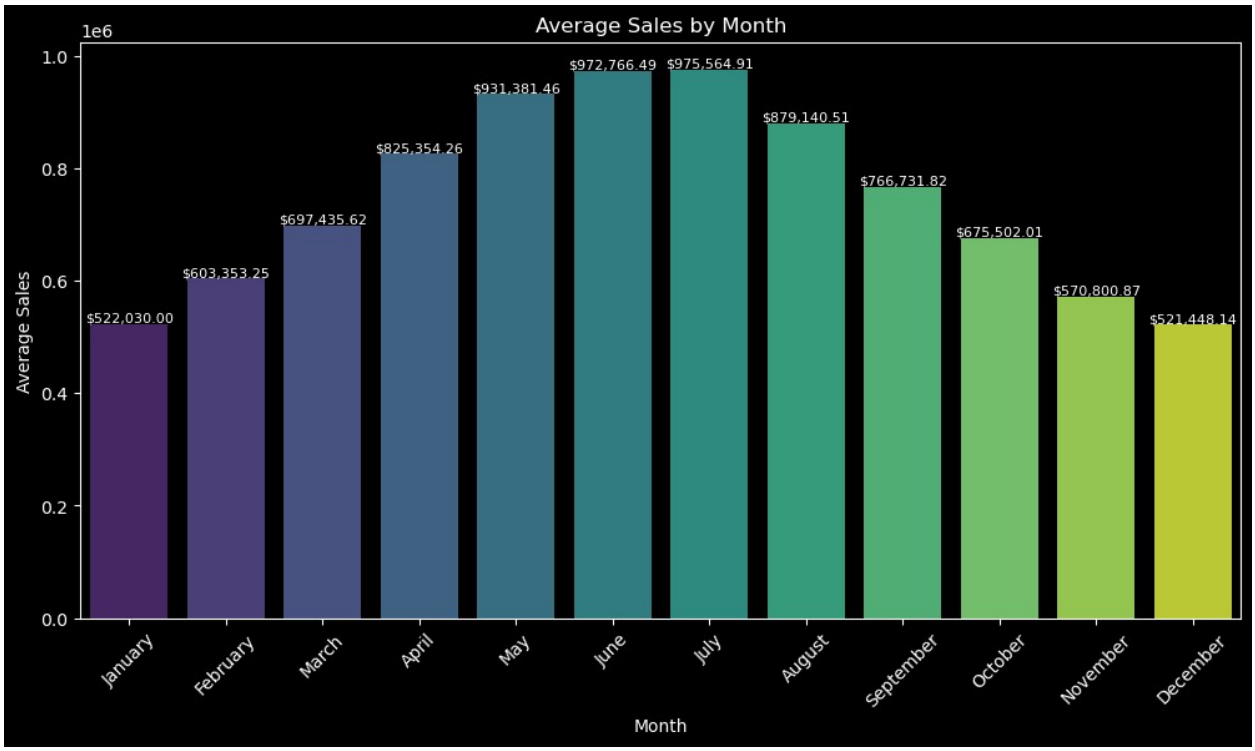
```
# 2. Identify trends in sales data for different months
# merged_df['month'] = merged_df['date'].dt.month_name()
# monthly_sales = merged_df.groupby('month')
['item_count'].sum().reindex(['January', 'February', 'March', 'April',
'May', 'June', 'July', 'August', 'September', 'October', 'November',
'December'])


# Calculate average sales by month
avg_sales_by_month = daily_sales.groupby('month_name')
['total_sales'].mean().reindex(['January', 'February', 'March',
'April', 'May', 'June', 'July', 'August', 'September', 'October',
'November', 'December'])
total_sales_by_month = daily_sales.groupby('month_name')
['item_count'].sum().reindex(['January', 'February', 'March', 'April',
'May', 'June', 'July', 'August', 'September', 'October', 'November',
'December'])

# Plot average sales by month
plt.figure(figsize=(12, 6))
sns.barplot(x=avg_sales_by_month.index, y=avg_sales_by_month.values,
palette='viridis')
plt.title('Average Sales by Month')
plt.xlabel('Month')
plt.xticks(rotation=45)
plt.ylabel('Average Sales')

# Annotate each bar with the average sales value formatted with commas
and dollar signs
for index, value in enumerate(avg_sales_by_month.values):
    plt.text(index, value, f'${value:,.2f}', ha='center', va='bottom',
fontsize=8)
plt.show()


# Plot total sales by month
plt.figure(figsize=(12, 6))
sns.barplot(x=total_sales_by_month.index,
y=total_sales_by_month.values, palette='viridis')
plt.title('Total Item Sales by Month')
plt.xlabel('Month')
plt.ylabel('Total Items Sold')
plt.xticks(rotation=45)

# Annotate each bar with the average sales value formatted with commas
and dollar signs
for index, value in enumerate(total_sales_by_month.values):
    plt.text(index, value, f'{value:,.0f}', ha='center', va='bottom')
plt.show()
```

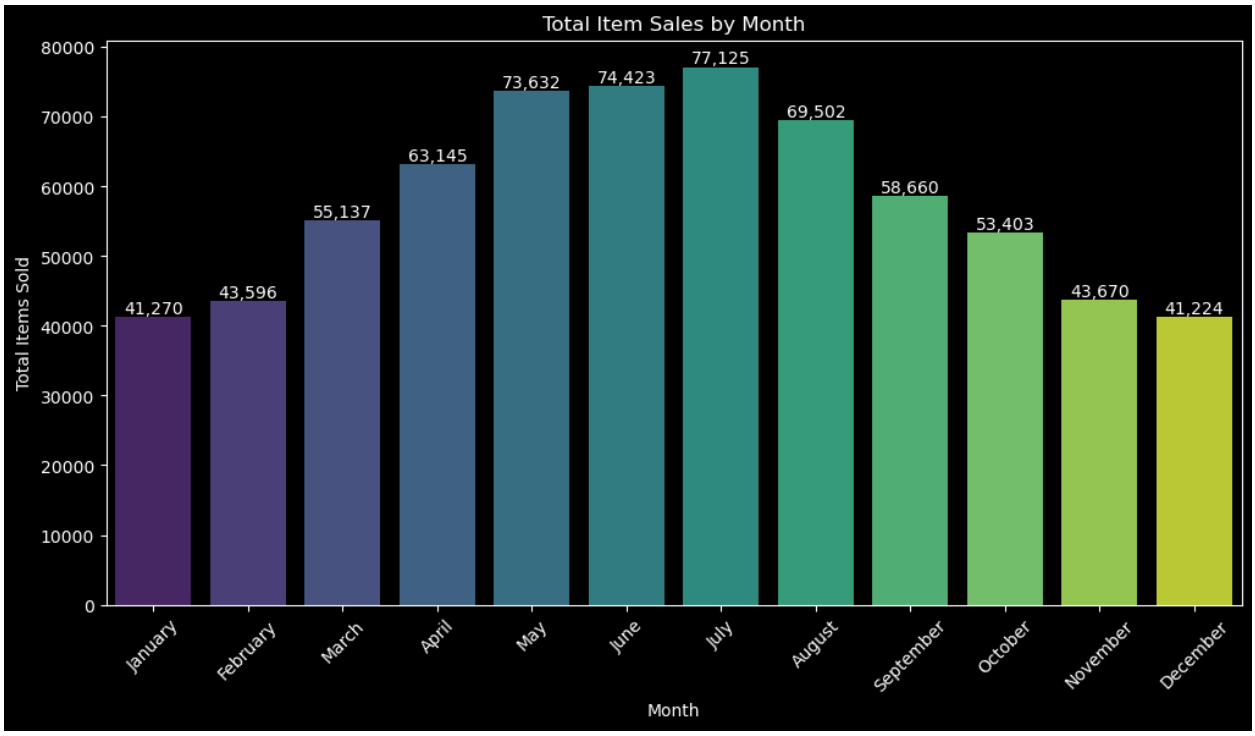
```
# Heatmap of sales by month and day of week
# Pivot the data to create a heatmap-friendly format
# Heatmap of sales by month and day of week
monthly_dow_sales = daily_sales.groupby(['month_name',
'day_of_week_name'])['total_sales'].mean().unstack()

# Define the correct order for the days of the week and months of the
year
days_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
months_order = ['January', 'February', 'March', 'April', 'May',
'June', 'July', 'August', 'September', 'October', 'November',
'December']

# Reindex the DataFrame to ensure the correct order
monthly_dow_sales = monthly_dow_sales.reindex(index=months_order,
columns=days_order)

# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(monthly_dow_sales, cmap='YlOrRd', annot=True, fmt='.0f')
plt.title('Average Sales by Month and Day of Week')
plt.xlabel('Day of Week')
plt.ylabel('Month')
plt.show()

# Time Series Line Plot
daily_sales['date'] = daily_sales.index
monthly_sales = daily_sales.resample('M', on='date')
['total_sales'].sum()

plt.figure(figsize=(15, 6))
sns.lineplot(x=monthly_sales.index, y=monthly_sales.values,
linewidth=2, color='#FF6B6B')
plt.title('Monthly Sales Over Time', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Total Sales', fontsize=12)
plt.xticks(rotation=45)
plt.grid(True, linestyle='--', alpha=0.7)
plt.fill_between(monthly_sales.index, monthly_sales.values, alpha=0.3,
color='#FF6B6B')
sns.despine()
plt.show()

# Year-over-Year Comparison
daily_sales['year'] = daily_sales.index.year
yearly_monthly_sales = daily_sales.groupby(['year', 'month'])
['total_sales'].sum().unstack(level=0)
```

```
plt.figure(figsize=(12, 6))
colors = ['#FF6B6B', '#4ECDC4', '#45B7D1']
for i, year in enumerate(yearly_monthly_sales.columns):
    plt.plot(yearly_monthly_sales.index, yearly_monthly_sales[year],
label=str(year), linewidth=2, color=colors[i])

plt.title('Monthly Sales: Year-over-Year Comparison', fontsize=16,
fontweight='bold')
plt.xlabel('Month', fontsize=12)
plt.ylabel('Total Sales', fontsize=12)
plt.legend(fontsize=10)
plt.grid(True, linestyle='--', alpha=0.7)
sns.despine()
plt.show()

# Box Plot for Monthly Distribution
plt.figure(figsize=(18, 8))
sns.set_style("whitegrid")

# Create the box plot
ax = sns.boxplot(x='month_name', y='total_sales', data=daily_sales,
order=months_order, palette='viridis')

# Add value annotations for medians
medians = daily_sales.groupby('month_name')['total_sales'].median()
vertical_offset = daily_sales['total_sales'].median() * 0.05  # offset
for median labels

for xtick in ax.get_xticks():
    month = months_order[xtick]
    ax.text(xtick, medians[month] + vertical_offset, f'$
{medians[month]:,.0f}',
            horizontalalignment='center', size='x-small',
color='white', weight='semibold')

plt.title('Distribution of Daily Sales by Month', fontsize=16,
fontweight='bold')
plt.xlabel('Month', fontsize=12)
plt.ylabel('Daily Sales', fontsize=12)
plt.xticks(rotation=45)

# Adjust y-axis to make room for annotations
plt.ylim(0, daily_sales['total_sales'].max() * 1.1)

sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.show()

# Cumulative Sales Plot
yearly_cumulative = daily_sales.groupby('year')
```

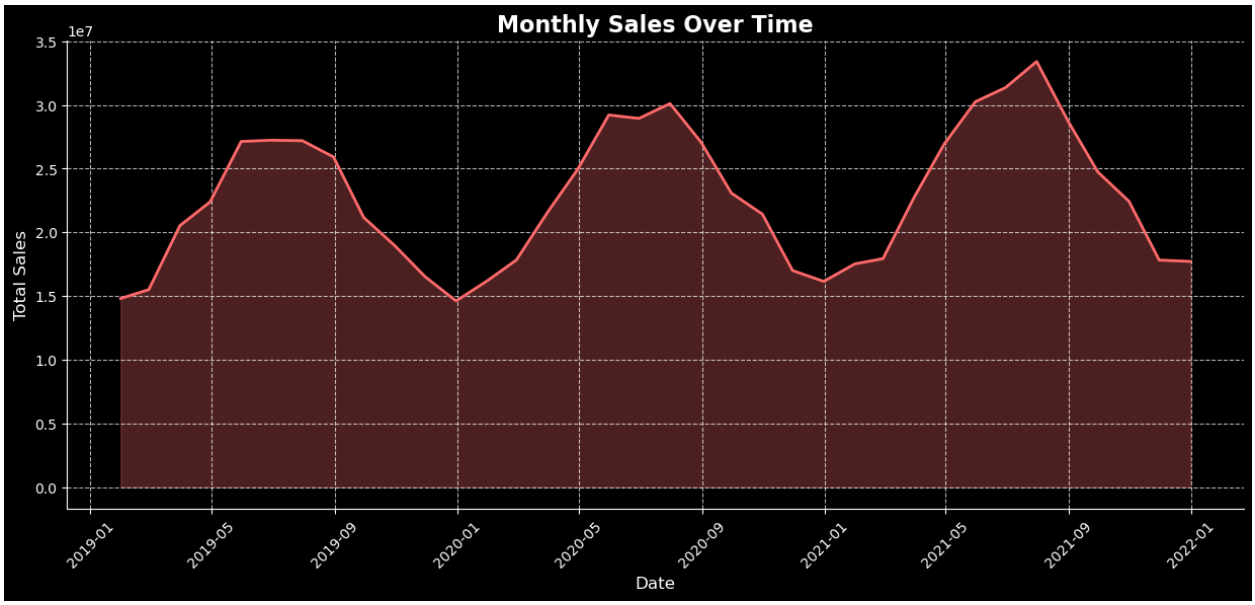
```
['total_sales'].cumsum().groupby(daily_sales.index.dayofyear).mean()

plt.figure(figsize=(12, 6))
sns.lineplot(x=yearly_cumulative.index, y=yearly_cumulative.values,
linewidth=2, color='#6C5B7B')
plt.title('Average Cumulative Sales Throughout the Year', fontsize=16,
fontweight='bold')
plt.xlabel('Day of Year', fontsize=12)
plt.ylabel('Cumulative Sales', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.fill_between(yearly_cumulative.index, yearly_cumulative.values,
alpha=0.3, color='#6C5B7B')
sns.despine()
plt.show()

# Seasonal Decomposition

# Ensure you have a datetime index
# daily_sales['date'] = pd.to_datetime(daily_sales.index)
monthly_sales = daily_sales.resample('M', on='date')
['total_sales'].sum()

result = seasonal_decompose(monthly_sales, model='additive',
period=12)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(15, 20))
colors = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#6C5B7B']

# Observed
ax1.fill_between(result.observed.index, 0, result.observed.values,
alpha=0.3, color=colors[0])
ax1.plot(result.observed.index, result.observed.values,
color=colors[0])
ax1.set_title('Observed', fontsize=14, fontweight='bold')

# Trend
ax2.fill_between(result.trend.index, 0, result.trend.values,
alpha=0.3, color=colors[1])
ax2.plot(result.trend.index, result.trend.values, color=colors[1])
ax2.set_title('Trend', fontsize=14, fontweight='bold')

# Seasonal
ax3.fill_between(result.seasonal.index, 0, result.seasonal.values,
alpha=0.3, color=colors[2])
ax3.plot(result.seasonal.index, result.seasonal.values,
color=colors[2])
ax3.set_title('Seasonal', fontsize=14, fontweight='bold')

# Residual
ax4.fill_between(result.resid.index, 0, result.resid.values,
```

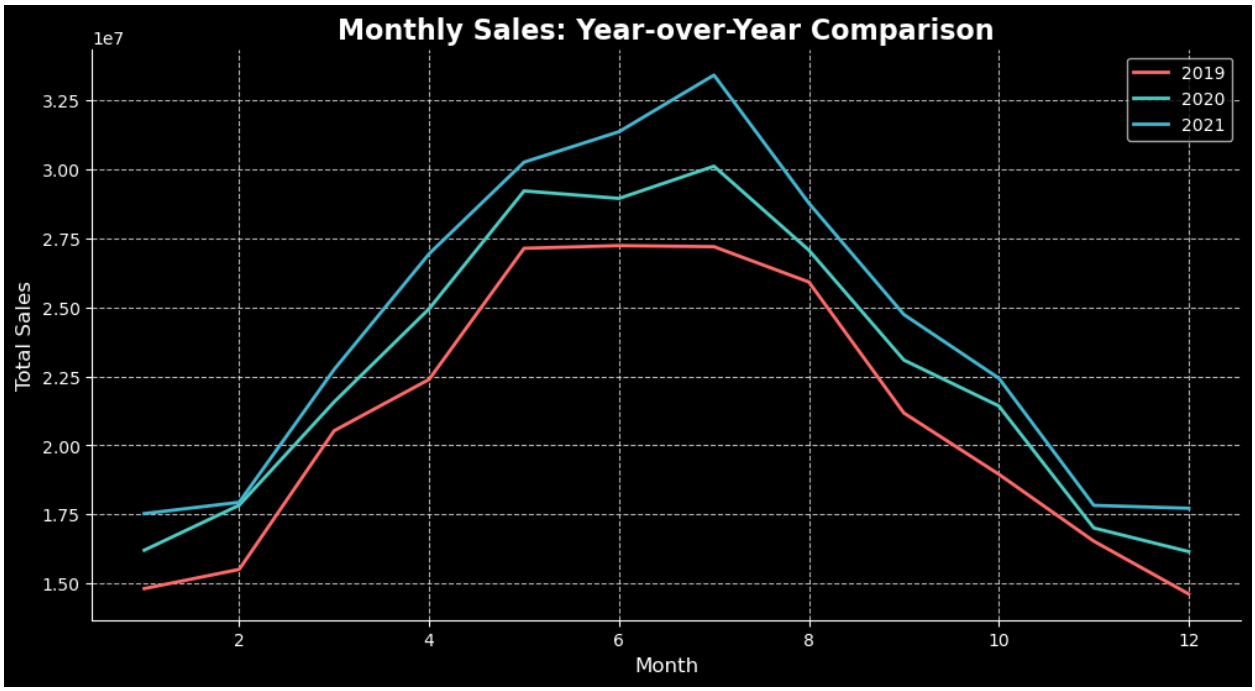
```
alpha=0.3, color=colors[3])
ax4.plot(result.resid.index, result.resid.values, color=colors[3])
ax4.set_title('Residual', fontsize=14, fontweight='bold')

for ax in [ax1, ax2, ax3, ax4]:
    ax.grid(True, linestyle='--', alpha=0.7)
    sns.despine(ax=ax)
    ax.set_xlabel('')  # Remove x-label from each subplot
    ax.tick_params(axis='x', rotation=45)  # Rotate x-axis labels

# Add an overall x-axis label
fig.text(0.5, 0.04, 'Date', ha='center', va='center', fontsize=12)

# Add y-axis labels
ax1.set_ylabel('Sales', fontsize=12)
ax2.set_ylabel('Trend', fontsize=12)
ax3.set_ylabel('Seasonal', fontsize=12)
ax4.set_ylabel('Residual', fontsize=12)

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

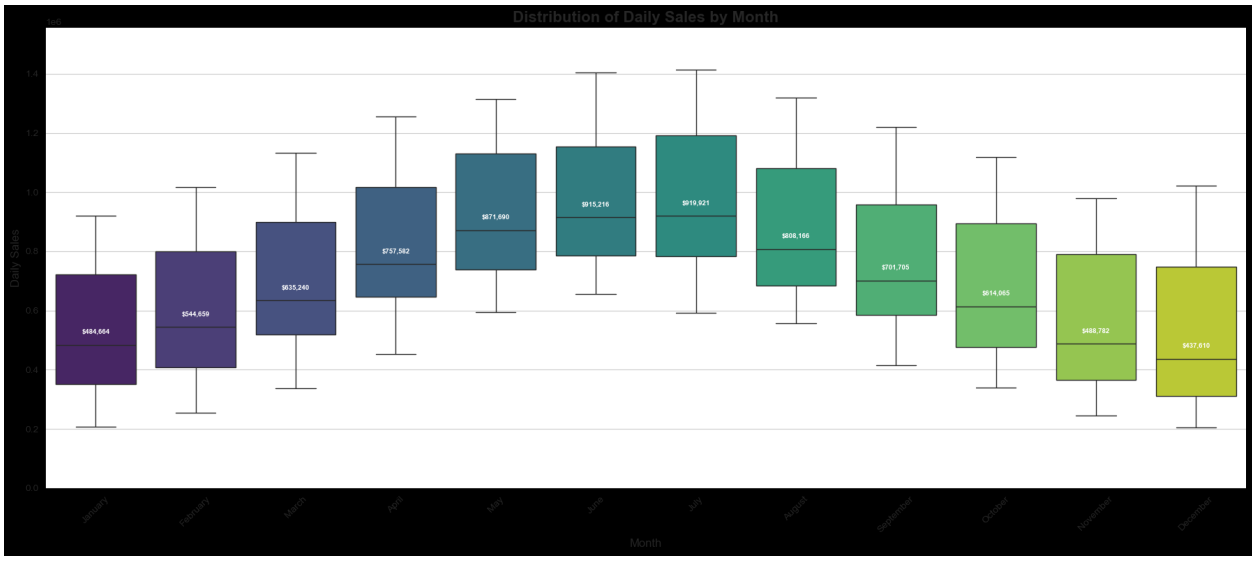


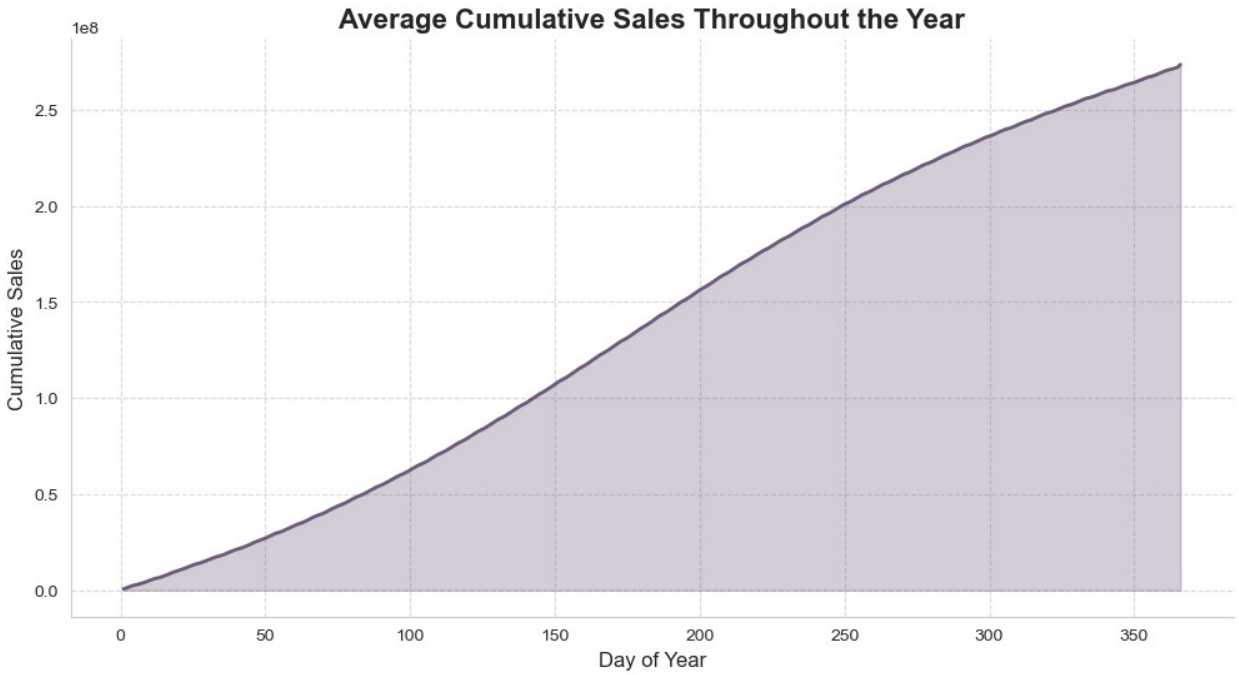

Total Item Sales by Month
Total Items Sold
Month
80000
70000
60000
50000
40000
30000
20000
10000
0
41,270
43,596
55,137
63,145
73,632
74,423
77,125
69,502
58,660
53,403
43,670
41,224
January
February
March
April
May
June
July
August
September
October
November
December

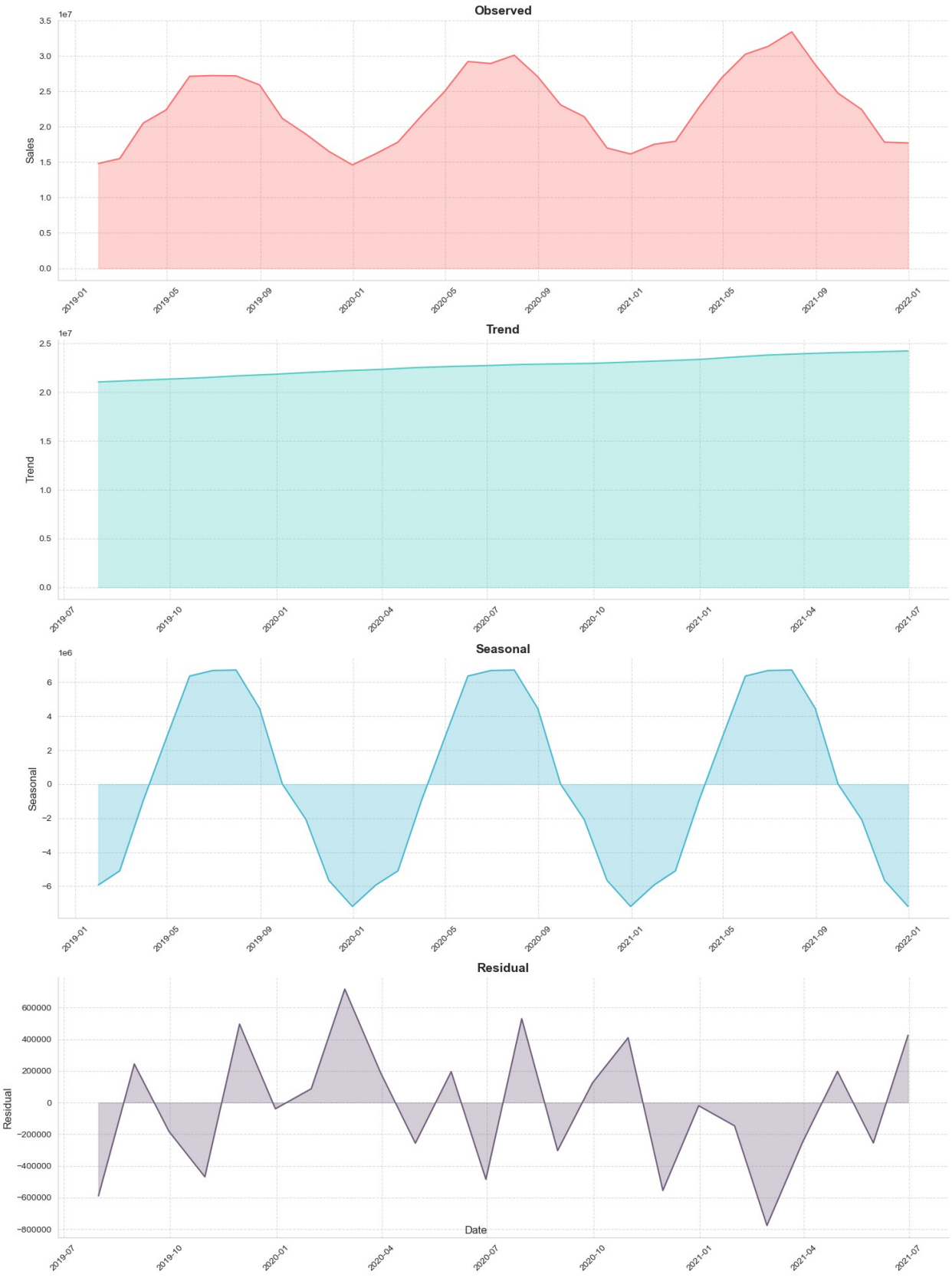

Average Sales by Month and Day of Week
Month
Day of Week
1e6
1.2
1.0
0.8
0.6
0.4
January 346539 388655 426350 610116 797579 770975 279886
February 417709 456824 507015 718174 873553 875491 352029
March 469372 563313 633792 852868 971591 973582 474497
April 585018 700212 728625 953784 1080179 1113924 611614
May 716862 772679 843910 1050951 1190654 1208048 700864
June 753058 834130 906981 1103631 1260186 1244961 749257
July 740118 810971 881353 1111922 1247876 1260888 732977
August 659271 721115 796991 1012040 1162977 1152170 656162
September 560405 616599 692042 907796 1041382 1054420 543483
October 469764 511630 553708 790689 949667 954204 431637
November 328123 416344 509466 766503 843457 827622 333094
December 281062 356944 460129 651961 836942 812480 280338
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday


Monthly Sales Over Time
1e7
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Total Sales
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01
Date


Monthly Sales: Year-over-Year Comparison
1e7
3.25
3.00
2.75
2.50
2.25
2.00
1.75
1.50
Total Sales
2
4
6
8
10
12
Month
2019
2020
2021


Distribution of Daily Sales by Month
Daily Sales
Month
January
February
March
April
May
June
July
August
September
October
November
December
$484,664
$544,659
$635,240
$757,582
$871,690
$915,216
$919,921
$808,166
$701,705
$614,065
$488,782
$437,610


Average Cumulative Sales Throughout the Year
1e8
Cumulative Sales
Day of Year
0.0
0.5
1.0
1.5
2.0
2.5
0
50
100
150
200
250
300
350



Explanations:

- This code analyzes sales trends across different months of the year. It calculates and plots average sales for each month. Additionally, it creates a heatmap to visualize the relationship between months, days of the week, and sales.

Why It Is Important:

- Analyzing monthly sales trends helps identify seasonal patterns in the business. This information is crucial for long-term planning, inventory management, and marketing strategies. It can also inform the development of more accurate forecasting models by accounting for seasonality.

Observations:

1. **Seasonal Pattern**: There is a clear seasonal pattern in sales, with peaks during summer months (June-August) and troughs during winter months (December-February).
2. **Yearly Cycle**: The sales data shows a consistent yearly cycle across the three years presented.
3. **Day of Week Impact**: Fridays and Saturdays consistently show higher sales across all months, as evident from the heatmap.
4. **Growth Trend**: There appears to be a slight overall growth trend from year to year, particularly noticeable in the peak months.
5. **Variability**: Winter months show lower variability in sales compared to summer months, as seen in the box plot.

Conclusions:

1. **Summer Peak**: The businesses experiences their highest sales during the summer season, likely due to increased outdoor activities or tourism.
2. **Weekend Boost**: Weekend sales, particularly Fridays and Saturdays, drive a significant portion of revenue across all months.
3. **Seasonal Dependency**: The businesses are highly seasonal, with performance strongly tied to the time of year.
4. **Gradual Growth**: Despite seasonal fluctuations, there's an indication of overall business growth year-over-year.
5. **Winter Slowdown**: The businesses faces a consistent slowdown during winter months, which might be challenging for cash flow and resource management.

Recommendations:

1. **Seasonal Staffing**: Adjust staffing levels to match the seasonal demand, increasing workforce during summer months and potentially reducing hours or using temporary staff in winter.

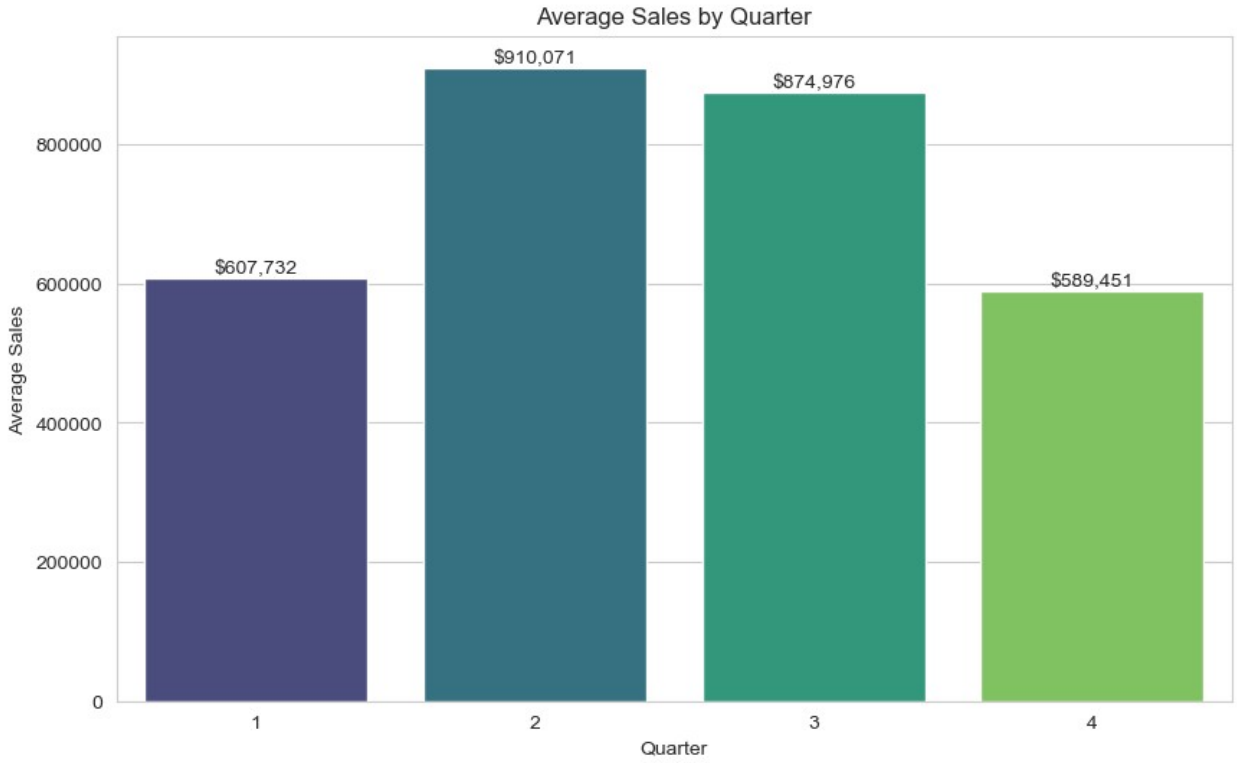
2. **Inventory Management**: Optimize inventory based on the seasonal trends to ensure adequate stock during peak months and avoid overstocking during slower periods.
3. **Marketing Campaigns**: Develop targeted marketing campaigns to boost sales during slower months, potentially focusing on indoor or winter-specific promotions.
4. **Weekend Focus**: Given the higher sales on Fridays and Saturdays, consider extending hours or running special promotions on these days to maximize revenue.
5. **Off-Season Strategies**: Develop strategies to attract customers during the off-peak season, such as introducing new products or services that are more relevant to winter months.
6. **Cash Flow Management**: Plan for the seasonal fluctuations in cash flow, potentially setting aside reserves during high-performing months to cover expenses during slower periods.
7. **Customer Retention**: Implement loyalty programs or other retention strategies to encourage repeat business, especially aimed at maintaining some level of engagement during slower months.
8. **Data-Driven Decision Making**: Continue to analyze sales data regularly to identify any emerging trends or changes in patterns, allowing for quick adaptations in strategy.

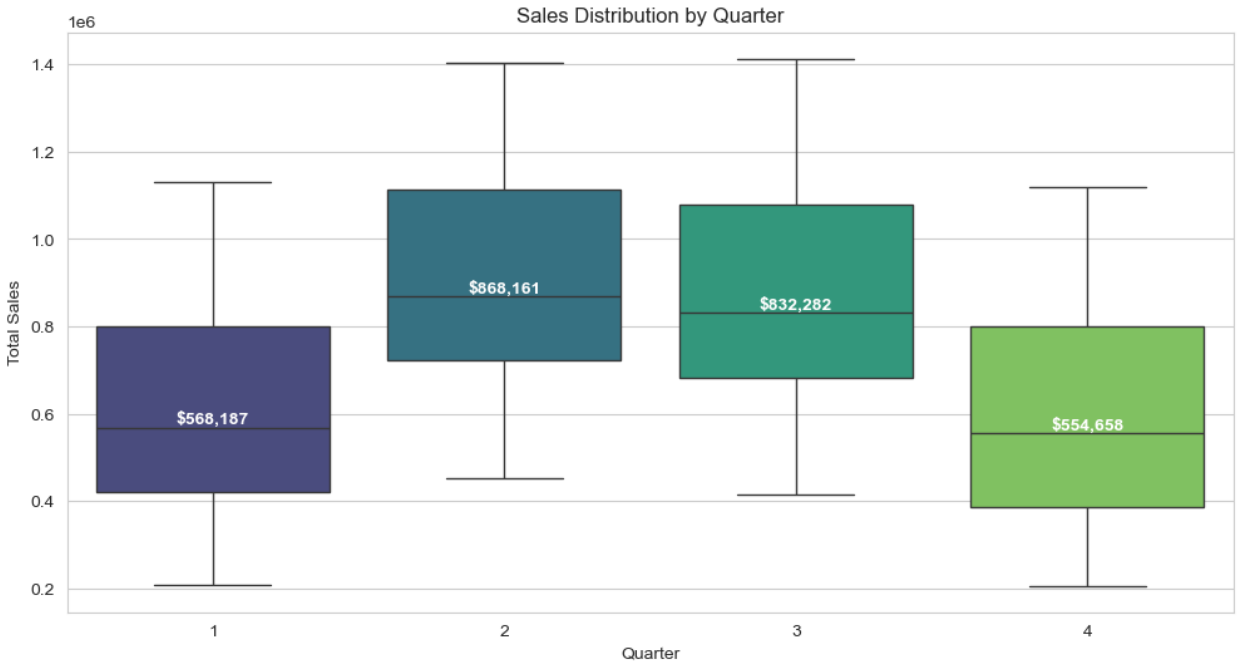
4.2.4. Examine the sales distribution across different quarters averaged over the years. Identify any noticeable patterns

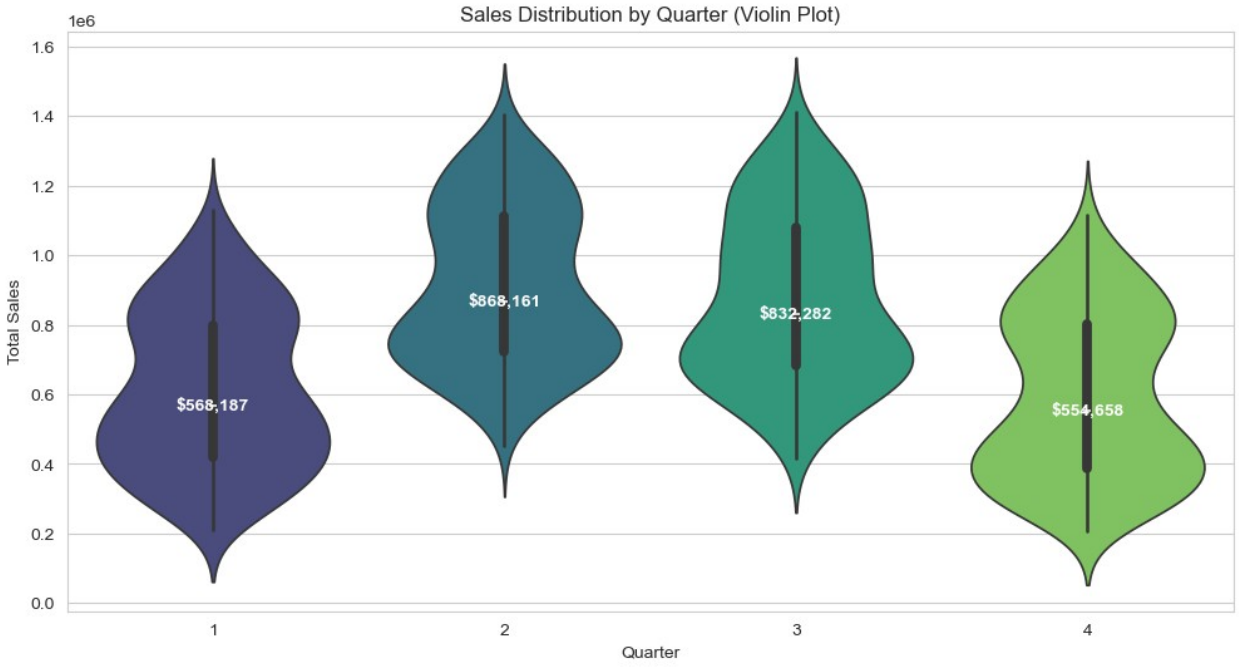
```
# Add quarter column
daily_sales['quarter'] = daily_sales.index.quarter

# Calculate average sales by quarter
avg_sales_by_quarter = daily_sales.groupby('quarter')
['total_sales'].mean()

# Plot average sales by quarter
plt.figure(figsize=(10, 6))
ax = sns.barplot(x=avg_sales_by_quarter.index,
y=avg_sales_by_quarter.values, palette='viridis')
plt.title('Average Sales by Quarter')
plt.xlabel('Quarter')
plt.ylabel('Average Sales')

# Add value labels on the bars
for i, v in enumerate(avg_sales_by_quarter.values):
    ax.text(i, v, f'${v:,.0f}', ha='center', va='bottom')

plt.show()
```

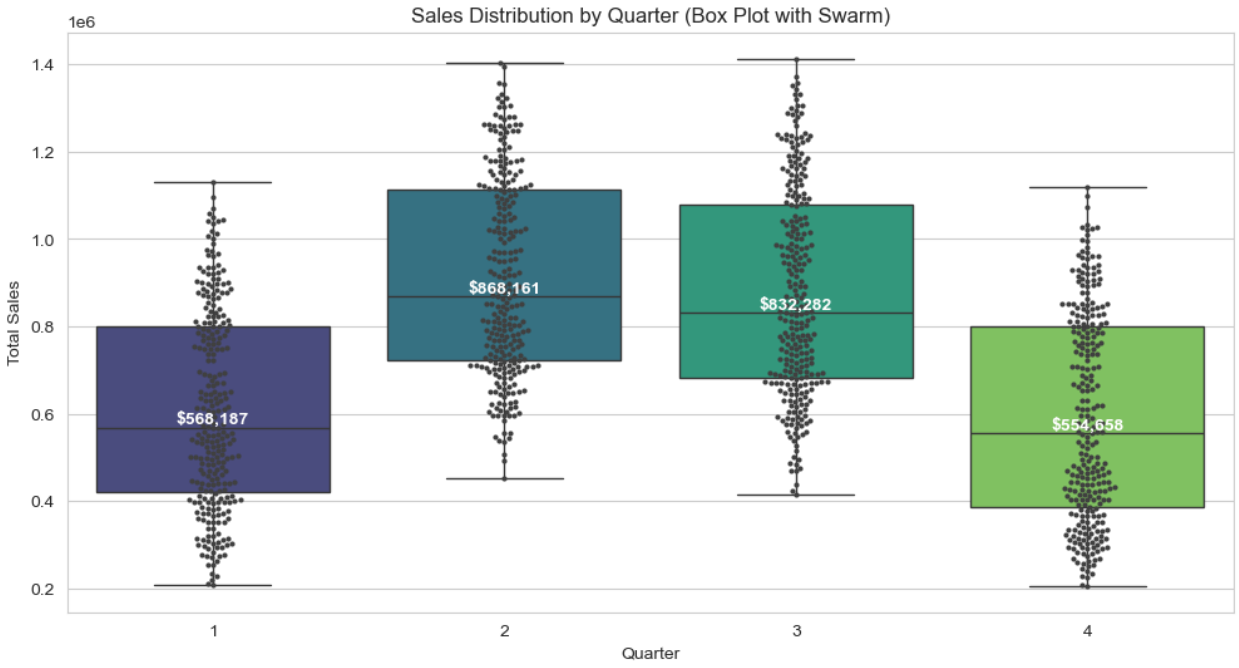
```
# Boxplot of sales by quarter
plt.figure(figsize=(12, 6))
ax = sns.boxplot(x='quarter', y='total_sales', data=daily_sales,
palette='viridis')
plt.title('Sales Distribution by Quarter')
plt.xlabel('Quarter')
plt.ylabel('Total Sales')

# Add median values on the boxplot
medians = daily_sales.groupby('quarter')['total_sales'].median()
for i, median in enumerate(medians):
    ax.text(i, median, f'${median:,.0f}', ha='center', va='bottom',
color='white', fontweight='bold')

plt.show()

# Violin plot
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='quarter', y='total_sales', data=daily_sales,
palette='viridis')
plt.title('Sales Distribution by Quarter (Violin Plot)')
plt.xlabel('Quarter')
plt.ylabel('Total Sales')

# Add median values on the violin plot
for i, median in enumerate(medians):
    ax.text(i, median, f'${median:,.0f}', ha='center', va='center',
color='white', fontweight='bold')

plt.show()

# Box plot with swarm
plt.figure(figsize=(12, 6))
ax = sns.boxplot(x='quarter', y='total_sales', data=daily_sales,
palette='viridis')
sns.swarmplot(x='quarter', y='total_sales', data=daily_sales,
color=".25", size=3)
plt.title('Sales Distribution by Quarter (Box Plot with Swarm)')
plt.xlabel('Quarter')
plt.ylabel('Total Sales')

# Add median values on the boxplot
for i, median in enumerate(medians):
    ax.text(i, median, f'${median:,.0f}', ha='center', va='bottom',
color='white', fontweight='bold')

plt.show()

# Year-over-Year Quarterly Sales Growth heatmap
quarterly_yoy = daily_sales.groupby([daily_sales.index.year,
```

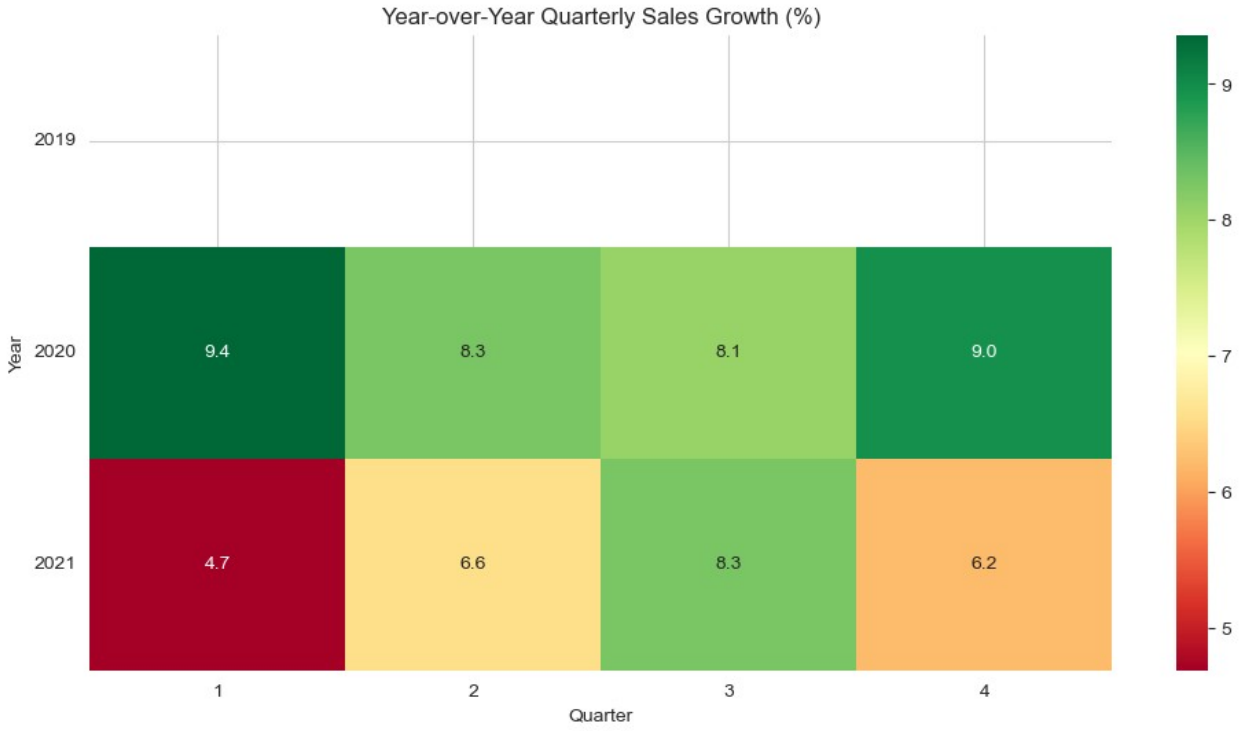
```
'quarter'])['total_sales'].sum().unstack()
quarterly_yoy_pct = quarterly_yoy.pct_change() * 100

plt.figure(figsize=(12, 6))
ax = sns.heatmap(quarterly_yoy_pct, cmap='RdYlGn', annot=True,
fmt='.1f')
plt.title('Year-over-Year Quarterly Sales Growth (%)')
plt.xlabel('Quarter')
plt.ylabel('Year')

# Rotate the tick labels
plt.yticks(rotation=0)

plt.show()
```




Sales Distribution by Quarter
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
Total Sales
$568,187
$868,161
$832,282
$554,658
1
2
3
4
Quarter


Sales Distribution by Quarter (Violin Plot)
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Total Sales
$568,187
$868,161
$832,282
$554,658
1
2
3
4
Quarter





Explanations:

- This code examines sales distribution across different quarters of the year. It calculates and plots average sales for each quarter, creates a box plot to show the distribution of sales by quarter, and generates a line plot to visualize quarterly sales trends over the years.

Why It Is Important:

- Analyzing quarterly sales patterns helps identify broader seasonal trends and potential fiscal year effects on the business. This information is valuable for strategic planning, budgeting, and setting sales targets. It can also reveal how the business performance has evolved over the years on a quarterly basis.

Observations:

1. **Quarterly Sales Pattern**:
   - The "Average Sales by Quarter" bar chart clearly shows that Q2 (second quarter) has the highest average sales ($925,127), followed closely by Q3 ($874,876).
   - Q4 has the lowest average sales ($596,461), with Q1 performing slightly better ($605,732).
2. **Sales Distribution**:
   - The box plots and violin plots show that Q2 and Q3 have wider distributions, indicating more variability in sales during these quarters.
   - Q1 and Q4 have narrower distributions, suggesting more consistent (but lower) sales.
3. **Outliers**:
   - The box plot with swarm points shows numerous outliers, particularly in Q2 and Q3, indicating some exceptionally high sales days or periods.
4. **Year-over-Year Growth**:
   - The heatmap of "Year-over-Year Quarterly Sales Growth (%)" shows mostly positive growth, with some variations:
     - 2022 showed positive growth across all quarters compared to 2021.
     - 2023 had mixed results, with strong growth in Q4 but a decline in Q1 compared to 2022.

Conclusions:

1. **Seasonal Business**: The restaurants' sales are highly seasonal, with peak performance in spring (Q2) and summer (Q3).
2. **Winter Slowdown**: There's a significant drop in sales during the winter months (Q4), which extends partially into Q1.
3. **Variability in Peak Seasons**: While Q2 and Q3 bring higher average sales, they also come with greater variability, suggesting factors like weather or events might influence performance.
4. **Growth Trend**: Despite seasonal fluctuations, there's an overall positive growth trend year-over-year, indicating business expansion or increasing popularity.
5. **Consistent Low Seasons**: The narrower distributions in Q1 and Q4 suggest more predictable (though lower) sales during these periods.

Recommendations:

1. **Seasonal Strategies**:

- Develop specific strategies to capitalize on the high sales potential of Q2 and Q3. This could include seasonal menus, outdoor seating expansions, or special events.
- Create targeted marketing campaigns and promotions for Q1 and Q4 to boost sales during slower periods.

2. **Staff Optimization**:
   - Adjust staffing levels to match seasonal demand, potentially using more part-time or seasonal workers during peak quarters.
   - Consider reduced hours or skeleton crews during slower periods to manage costs.
3. **Inventory Management**:
   - Implement dynamic inventory management to handle the higher variability in Q2 and Q3, ensuring sufficient stock for high-sales days without overstocking.
   - Optimize inventory for Q1 and Q4 to minimize waste during slower periods.
4. **Financial Planning**:
   - Develop a financial strategy that accounts for the significant differences in quarterly performance. This might include setting aside surplus from high-performing quarters to cover expenses in lower-performing ones.
5. **Off-Season Focus**:
   - Investigate opportunities to increase sales in Q4 and Q1, such as holiday-themed events, comfort food specials, or indoor dining experiences that attract customers despite colder weather.
6. **Growth Analysis**:
   - Conduct a detailed analysis of the factors contributing to year-over-year growth, especially in Q4 of 2023, to replicate successful strategies across other quarters and locations.
7. **Outlier Investigation**:
   - Analyze the outlier days or periods, particularly in Q2 and Q3, to understand what drives exceptionally high sales. Use these insights to potentially recreate these conditions more frequently.
8. **Customer Engagement**:
   - Develop loyalty programs or other incentives to encourage repeat business, especially aimed at maintaining customer engagement during slower quarters.

4.2.5. Compare the performances of the different restaurants. Find out which restaurant had the most sales and look at the sales for each restaurant across different years, months, and days

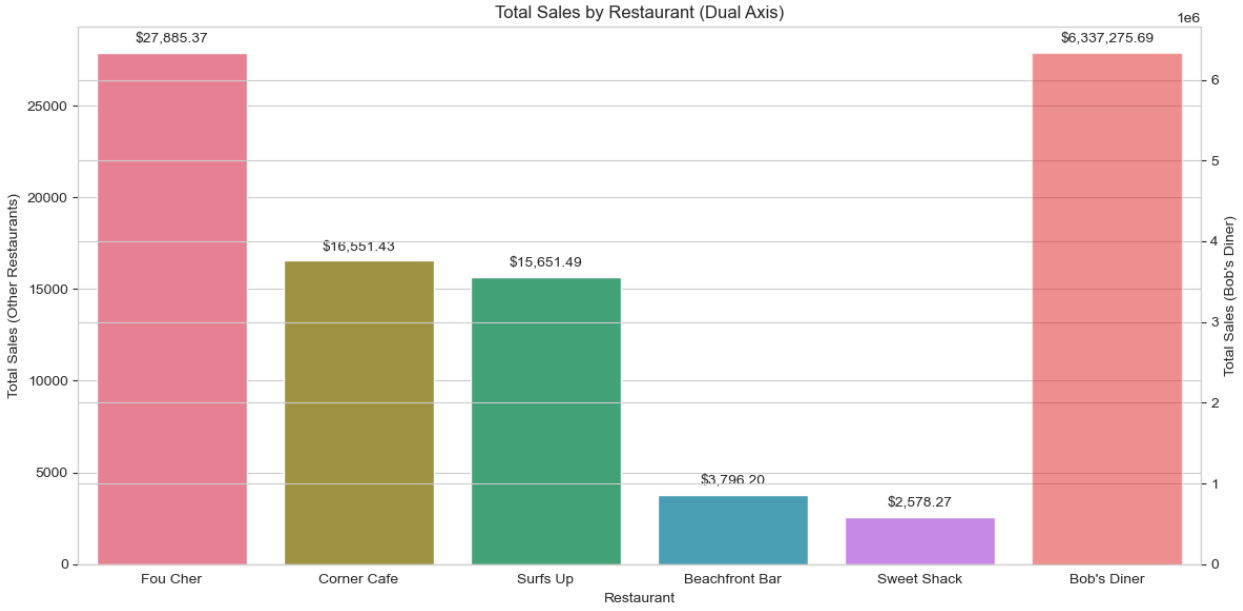
```
# Assuming final_df is already created and contains the necessary data
final_df['total_sales'] = final_df['price'] * final_df['item_count']

# Calculate daily sales for each restaurant
restaurant_daily_sales = final_df.groupby(['date',
'restaurant_name']).agg({
    'price': lambda x: (x * final_df.loc[x.index, 'item_count']).sum()
}).reset_index().rename(columns={'price': 'daily_sales'})
```

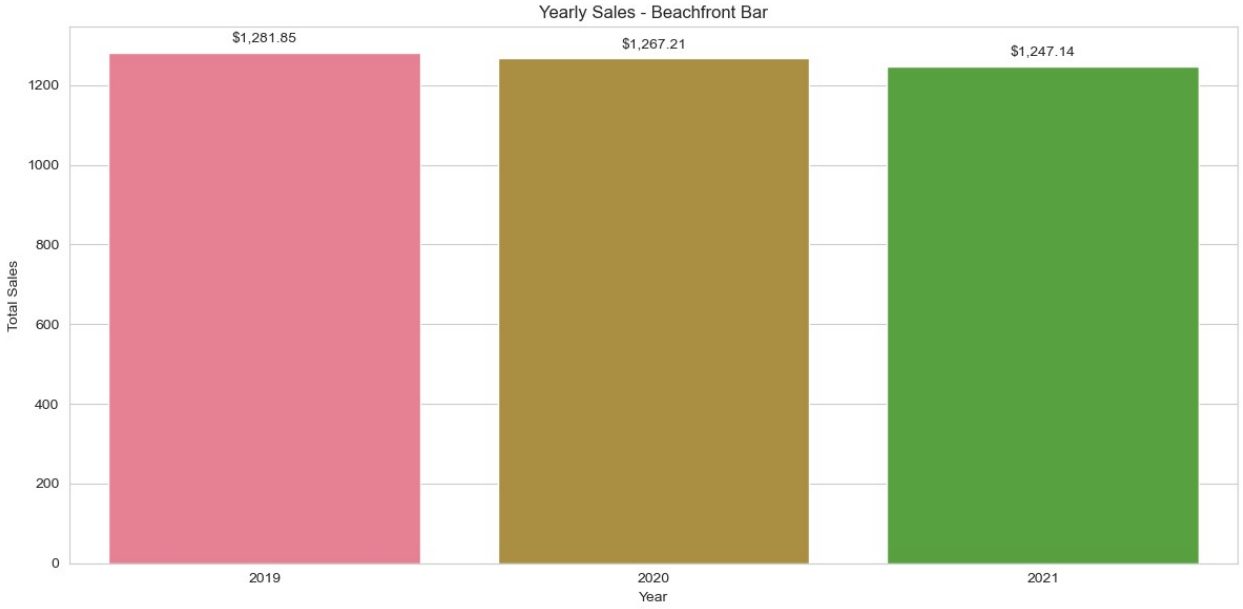
```
# Calculate total sales by restaurant
restaurant_sales = final_df.groupby('restaurant_name').agg({
    'price': lambda x: (x * final_df.loc[x.index, 'item_count']).sum()
}).rename(columns={'price': 'total_sales'}).sort_values('total_sales',
ascending=False)

# Function to create individual bar plots
def plot_individual_bars(data, x, y, title, x_label, y_label):
    palette = sns.color_palette("husl",
len(data['restaurant_name'].unique()))
    restaurants = data['restaurant_name'].unique()

    for restaurant in restaurants:
        restaurant_data = data[data['restaurant_name'] == restaurant]

        plt.figure(figsize=(12, 6))
        ax = sns.barplot(data=restaurant_data, x=x, y=y,
palette=palette)
        plt.title(f"{title} - {restaurant}")
        plt.xlabel(x_label)
        plt.ylabel(y_label)

        if restaurant != "Bob's Diner" or title != "Average Daily
Sales":
            for p in ax.patches:
                ax.annotate(f'${p.get_height():,.2f}',
                            (p.get_x() + p.get_width() / 2.,
p.get_height()),
                            ha='center', va='center',
                            xytext=(0, 9),
                            textcoords='offset points')

        plt.tight_layout()
        plt.show()

# Function to create individual boxplots for each restaurant
def plot_individual_boxplots(data, x, y, title_prefix, x_label,
y_label, is_monthly=False):
    restaurants = data['restaurant_name'].unique()
    colors = plt.cm.rainbow(np.linspace(0, 1, len(restaurants)))

    for restaurant, color in zip(restaurants, colors):
        restaurant_data = data[data['restaurant_name'] == restaurant]

        if is_monthly:
            plt.figure(figsize=(20, 10))
            ax = sns.boxplot(data=restaurant_data, x='month', y=y,
order=range(1, 13), palette='viridis')
            plt.title(f"{title_prefix} - {restaurant}")
```

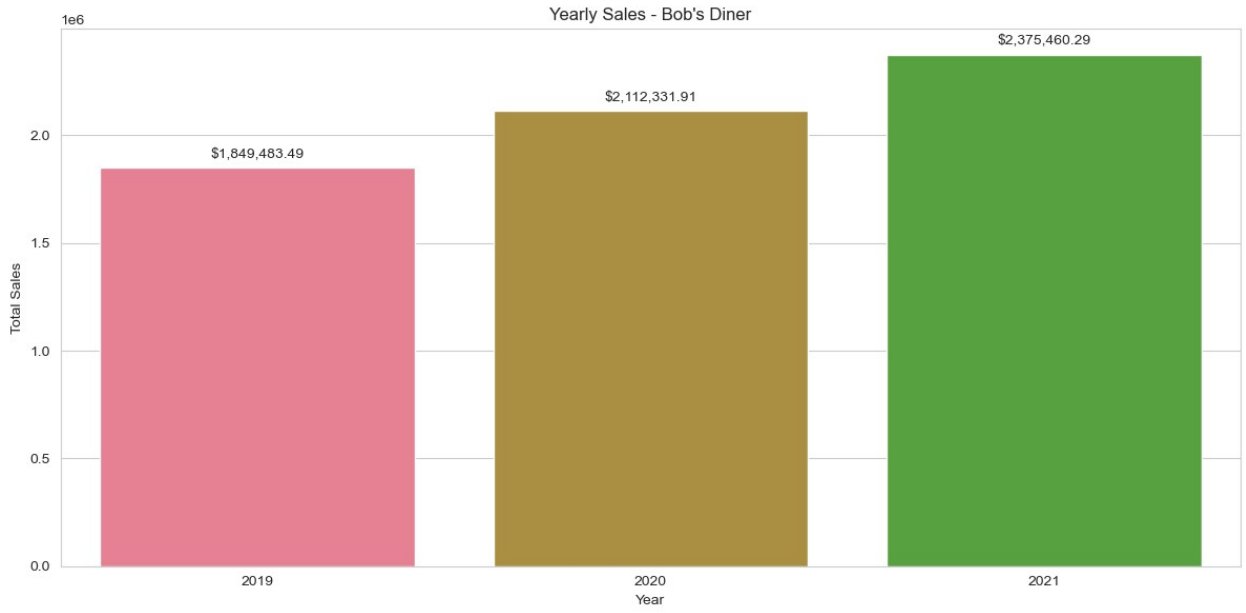
```
            plt.xlabel(x_label)
            plt.ylabel(y_label)

            month_names = [calendar.month_abbr[i] for i in range(1,
13)]
            plt.xticks(range(12), month_names)

            medians = restaurant_data.groupby('month')[y].median()
            for i, median in enumerate(medians):
                ax.text(i, median, f'${median:,.2f}',
                        horizontalalignment='center', size='small',
color='white',
                        weight='semibold',
bbox=dict(facecolor='black', edgecolor='none', alpha=0.5))
        else:
            plt.figure(figsize=(12, 6))
            ax = sns.boxplot(data=restaurant_data, x=x, y=y,
color=color)
            plt.title(f"{title_prefix} - {restaurant}")
            plt.xlabel(x_label)
            plt.ylabel(y_label)

            medians = restaurant_data.groupby(x)[y].median().values
            pos = range(len(medians))
            for tick, label in zip(pos, ax.get_xticklabels()):
                ax.text(pos[tick], medians[tick], f'$
{medians[tick]:,.2f}',
                        horizontalalignment='center', size='small',
color='black',
                        weight='semibold', rotation=0, alpha=0.7,
                        bbox=dict(facecolor='white', edgecolor='none',
alpha=0.7))

        plt.tight_layout()
        plt.show()

# 1. Total Sales by Restaurant (Dual Axis) Plot
bobs_data = restaurant_sales[restaurant_sales.index == "Bob's Diner"]
other_data = restaurant_sales[restaurant_sales.index != "Bob's Diner"]

fig, ax1 = plt.subplots(figsize=(12, 6))
sns.barplot(x=other_data.index, y='total_sales', data=other_data,
ax=ax1, palette='husl')
ax1.set_xlabel('Restaurant')
ax1.set_ylabel('Total Sales (Other Restaurants)')
ax1.tick_params(axis='x')

for p in ax1.patches:
    ax1.annotate('${:,.2f}'.format(p.get_height()),
                 (p.get_x() + p.get_width() / 2., p.get_height()),
```

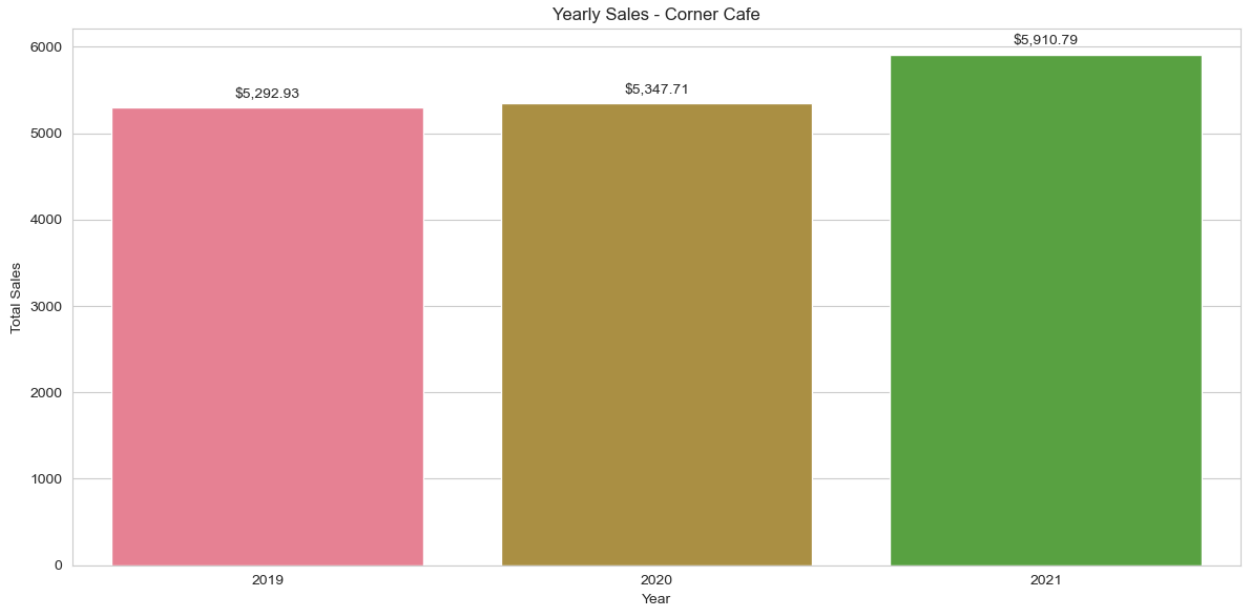
```
                ha='center', va='center',
                xytext=(0, 9),
                textcoords='offset points')

ax2 = ax1.twinx()
sns.barplot(x=bobs_data.index, y='total_sales', data=bobs_data,
ax=ax2, color='red', alpha=0.5)
ax2.set_ylabel("Total Sales (Bob's Diner)")

for p in ax2.patches:
    ax2.annotate('${:,.2f}'.format(p.get_height()),
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center', va='center',
                xytext=(0, 9),
                textcoords='offset points')

plt.title('Total Sales by Restaurant (Dual Axis)')
plt.tight_layout()
plt.show()

# 2. Yearly Plots
yearly_sales = final_df.groupby([final_df['date'].dt.year,
'restaurant_name'])['total_sales'].sum().reset_index()
yearly_sales = yearly_sales.rename(columns={'date': 'year'})
plot_individual_bars(yearly_sales, 'year', 'total_sales', 'Yearly
Sales', 'Year', 'Total Sales')
plot_individual_boxplots(yearly_sales, 'year', 'total_sales', 'Yearly
Sales Distribution', 'Year', 'Yearly Total Sales')

# 3. Quarterly Plots
final_df['year_quarter'] = final_df['date'].dt.to_period('Q')
quarterly_sales = final_df.groupby(['year_quarter',
'restaurant_name'])['total_sales'].sum().reset_index()
quarterly_sales['year_quarter'] =
quarterly_sales['year_quarter'].astype(str)
plot_individual_boxplots(quarterly_sales, 'year_quarter',
'total_sales', 'Quarterly Sales Distribution', 'Year-Quarter',
'Quarterly Total Sales')

# 4. Monthly Plots
monthly_sales = final_df.groupby([final_df['date'].dt.month,
'restaurant_name'])['total_sales'].mean().reset_index()
monthly_sales = monthly_sales.rename(columns={'date': 'month'})
monthly_sales['month_name'] = monthly_sales['month'].apply(lambda x:
calendar.month_abbr[x])
plot_individual_bars(monthly_sales, 'month_name', 'total_sales',
'Average Monthly Sales', 'Month', 'Average Sales')

final_df['month'] = final_df['date'].dt.month
monthly_sales_box = final_df.groupby(['month', 'restaurant_name',
```

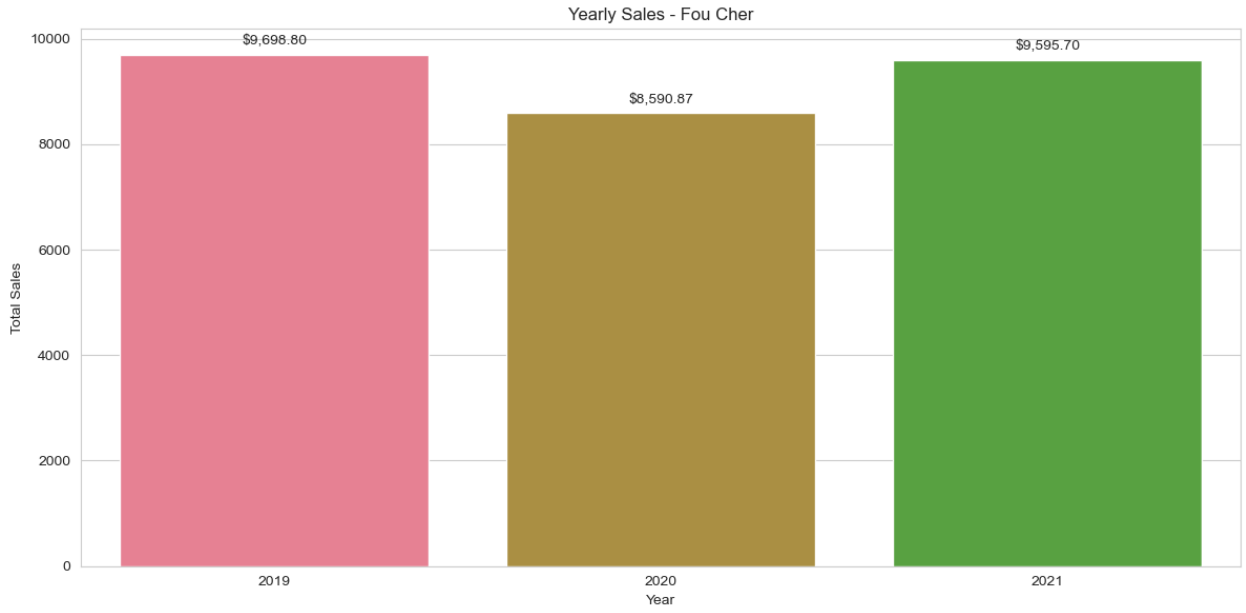
```
final_df['date'].dt.to_period('M')])
['total_sales'].sum().reset_index()
plot_individual_boxplots(monthly_sales_box, 'month', 'total_sales',
'Monthly Sales Distribution', 'Month', 'Monthly Total Sales',
is_monthly=True)

# 5. Weekly Plots
# If you have weekly data, add the weekly plots here

# 6. Days of the Week Plots
final_df['day_name'] = final_df['date'].dt.day_name()
day_name_sales = final_df.groupby(['day_name', 'restaurant_name'])
['total_sales'].mean().reset_index()
day_name_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
day_name_sales['day_name'] =
pd.Categorical(day_name_sales['day_name'], categories=day_name_order,
ordered=True)
day_name_sales = day_name_sales.sort_values('day_name')
plot_individual_bars(day_name_sales, 'day_name', 'total_sales',
'Average Sales by Day of Week', 'Day of Week', 'Average Sales')

day_of_week_sales = final_df.groupby(['day_name', 'restaurant_name',
'date'])['total_sales'].sum().reset_index()
day_of_week_sales['day_name'] =
pd.Categorical(day_of_week_sales['day_name'],
categories=day_name_order, ordered=True)
day_of_week_sales = day_of_week_sales.sort_values('day_name')
plot_individual_boxplots(day_of_week_sales, 'day_name', 'total_sales',
'Day of Week Sales Distribution', 'Day of Week', 'Total Sales')

# 7. Daily Sales Plots
daily_sales = final_df.groupby([final_df['date'].dt.day,
'restaurant_name'])['total_sales'].mean().reset_index()
daily_sales = daily_sales.rename(columns={'date': 'day'})
plot_individual_bars(daily_sales, 'day', 'total_sales', 'Average Daily
Sales', 'Day of Month', 'Average Sales')

daily_sales_box = final_df.groupby(['date', 'restaurant_name'])
['total_sales'].sum().reset_index()
plot_individual_boxplots(daily_sales_box, 'restaurant_name',
'total_sales', 'Daily Sales Distribution', 'Restaurant', 'Daily Total
Sales')
```

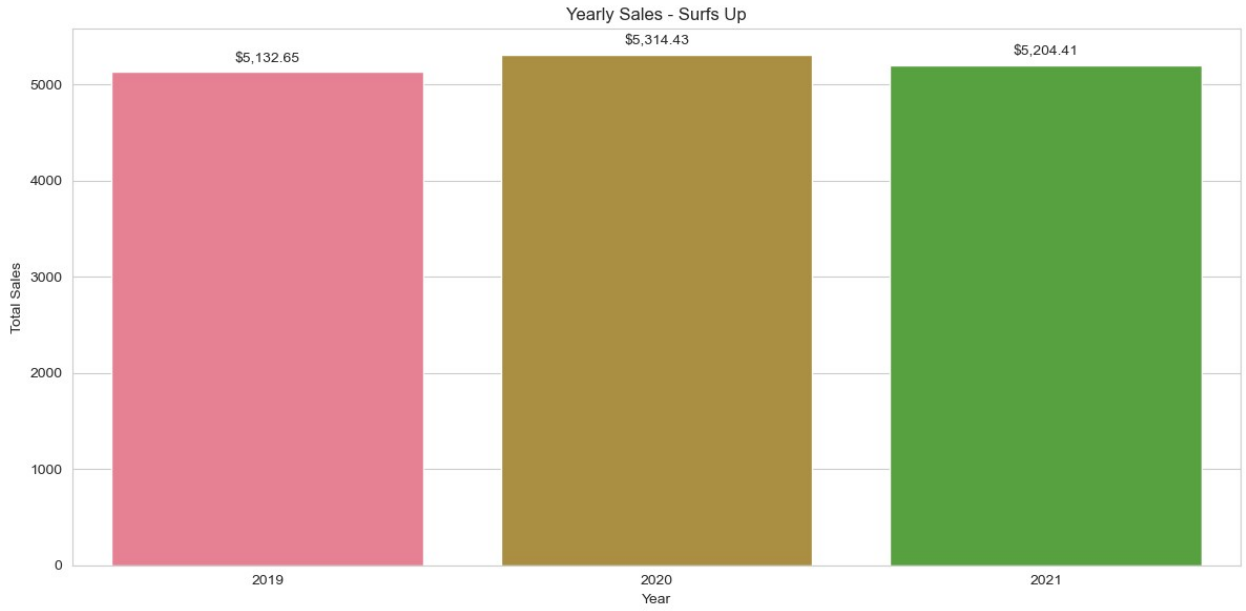

Total Sales by Restaurant (Dual Axis)
Total Sales (Other Restaurants)
Total Sales (Bob's Diner)
1e6
$27,885.37
$16,551.43
$15,651.49
$3,796.20
$2,578.27
$6,337,275.69
0
5000
10000
15000
20000
25000
0
1
2
3
4
5
6
Fou Cher
Corner Cafe
Surfs Up
Beachfront Bar
Sweet Shack
Bob's Diner
Restaurant

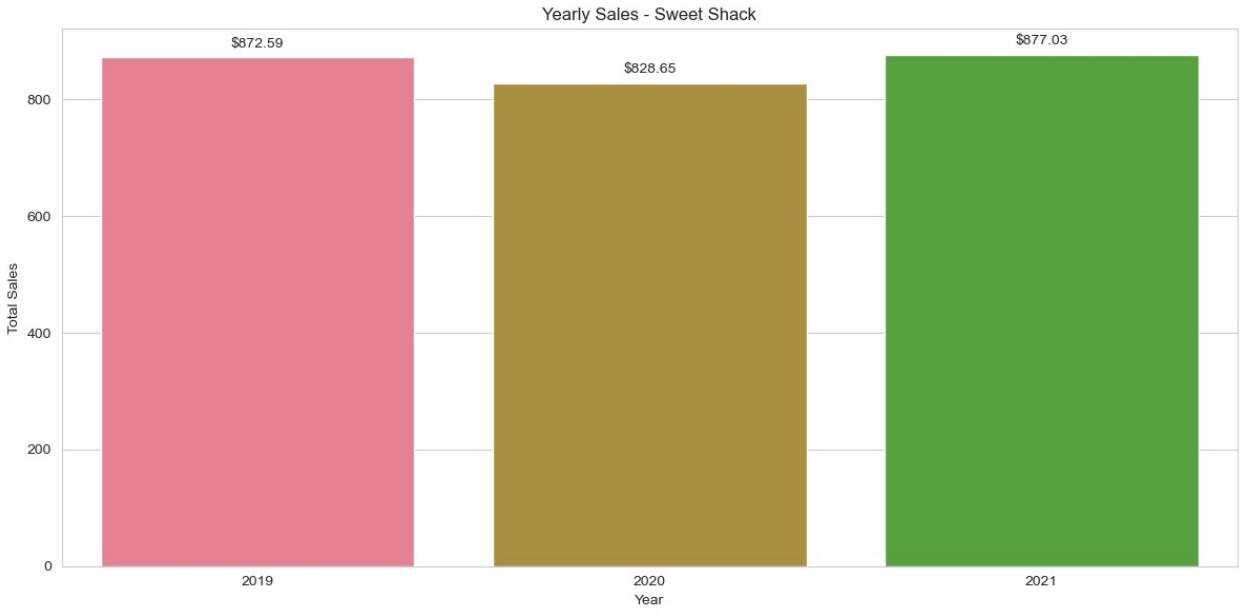

Yearly Sales - Beachfront Bar
Total Sales
$1,281.85
$1,267.21
$1,247.14
0
200
400
600
800
1000
1200
2019
2020
2021
Year

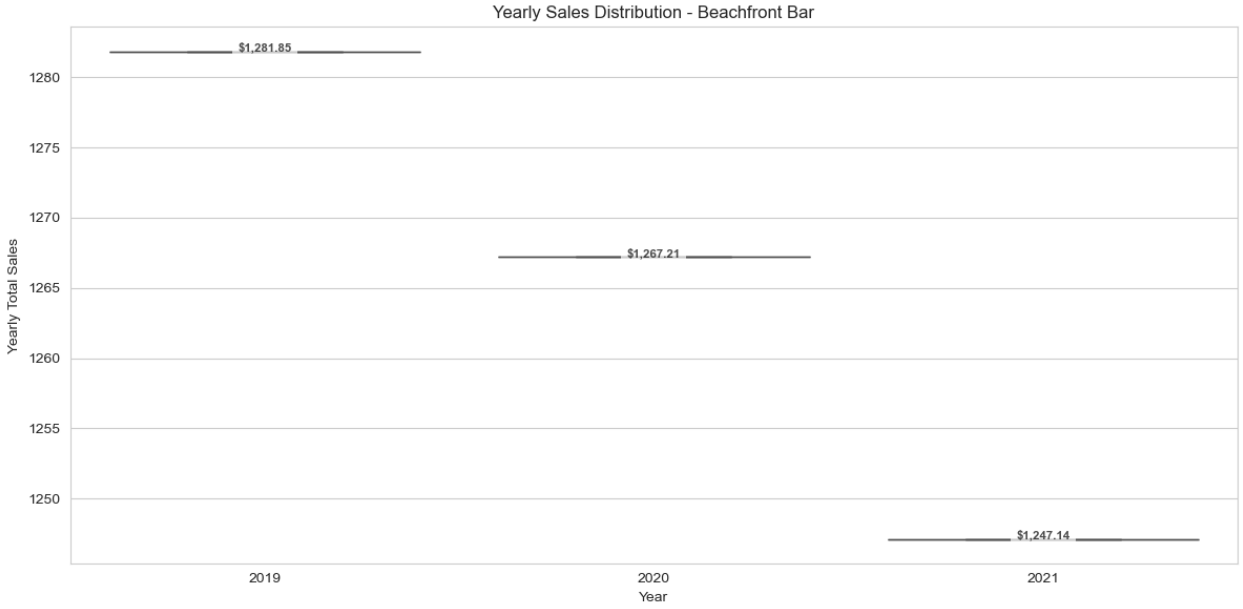

Yearly Sales - Bob's Diner
1e6
$1,849,483.49
$2,112,331.91
$2,375,460.29
Total Sales
Year
2019
2020
2021

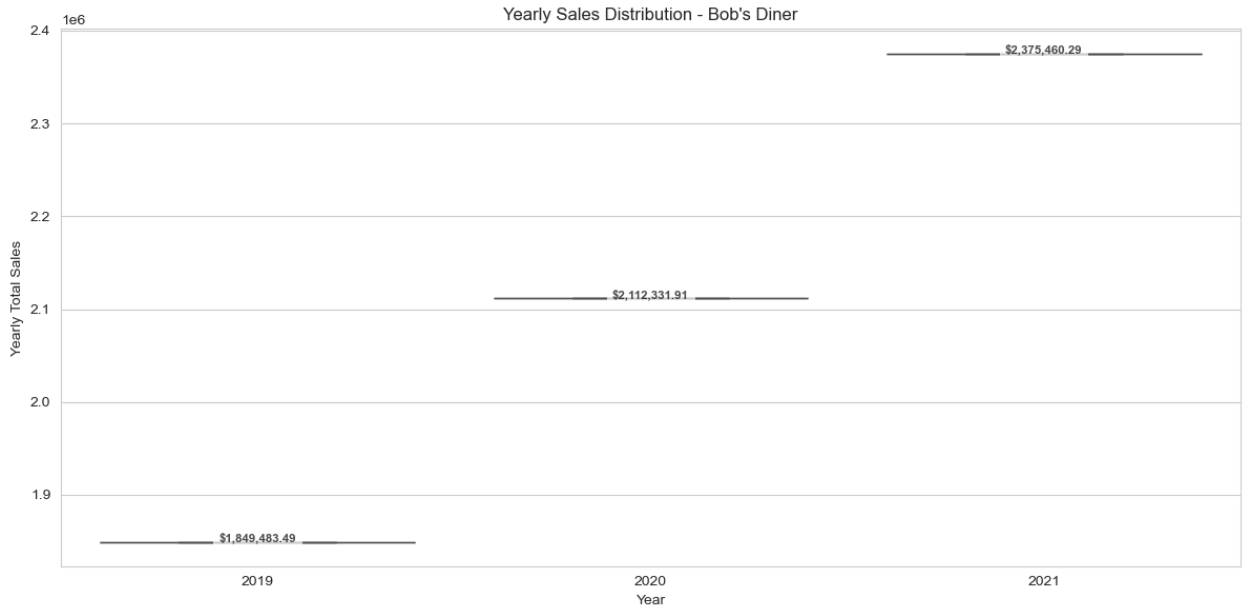

Yearly Sales - Corner Cafe
$5,292.93
$5,347.71
$5,910.79
Total Sales
Year
2019
2020
2021

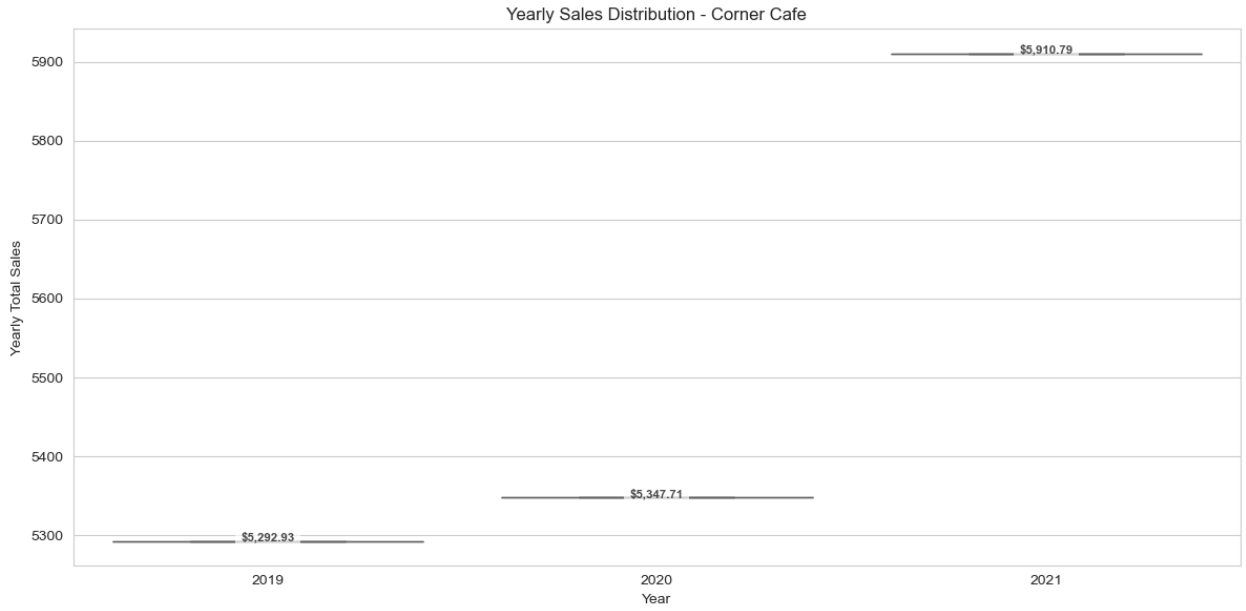

Yearly Sales - Fou Cher
$9,698.80
$8,590.87
$9,595.70
10000
8000
6000
4000
2000
0
Total Sales
2019
2020
2021
Year

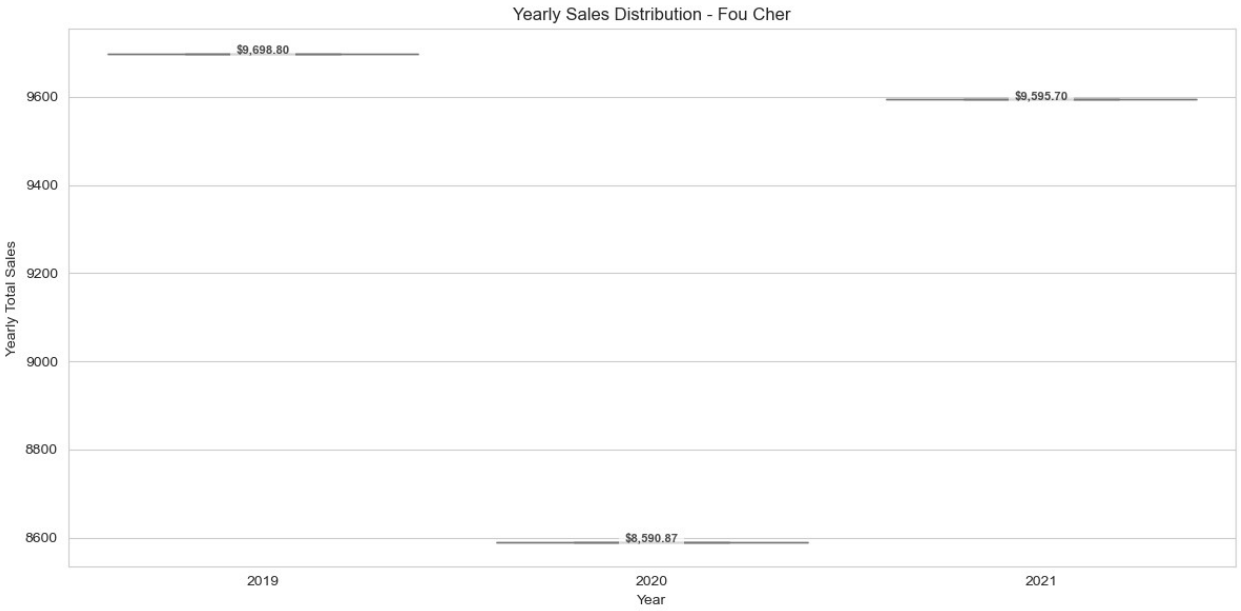

Yearly Sales - Surfs Up
$5,132.65
$5,314.43
$5,204.41
5000
4000
3000
2000
1000
0
Total Sales
2019
2020
2021
Year

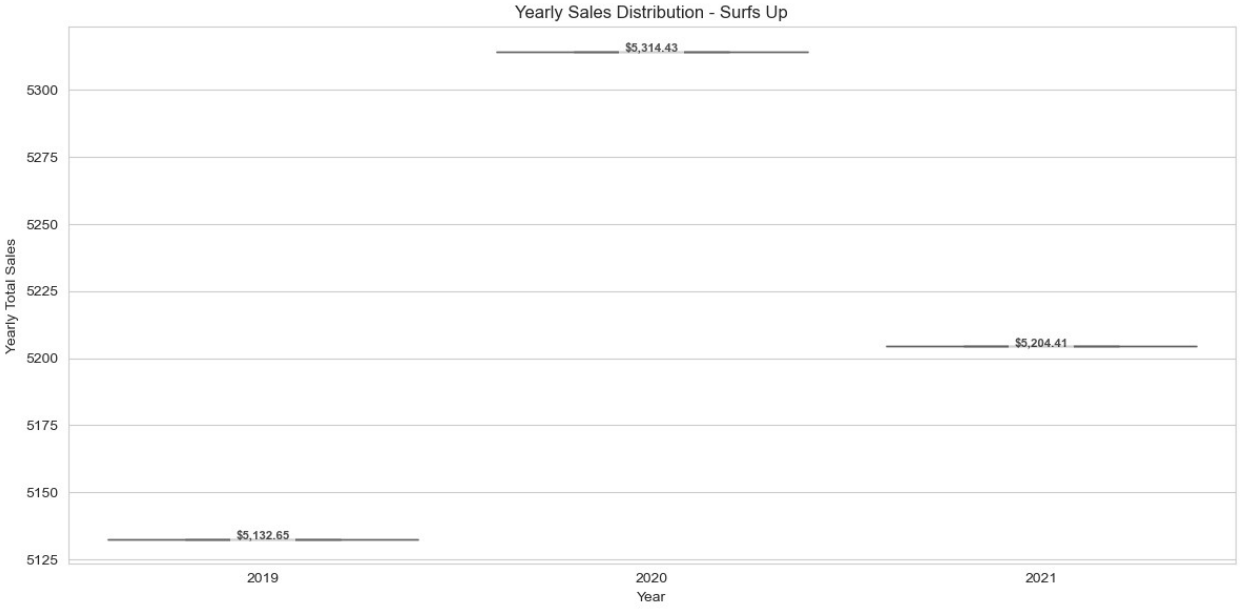

Yearly Sales - Sweet Shack
$872.59
$828.65
$877.03
800
600
400
200
0
Total Sales
2019
2020
2021
Year

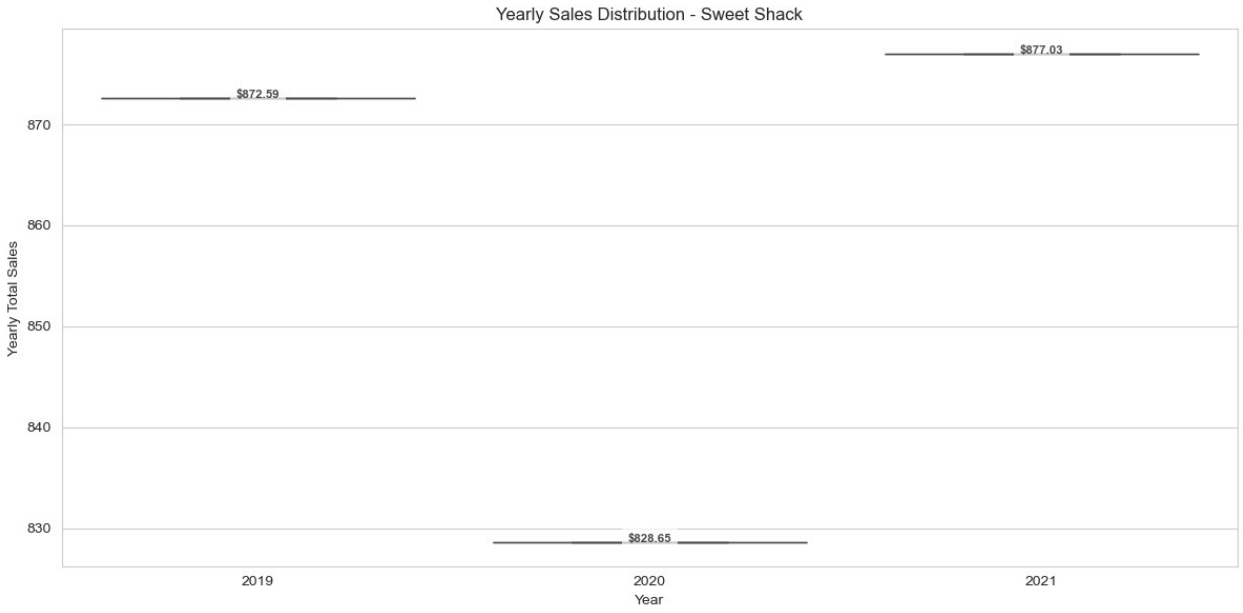

Yearly Sales Distribution - Beachfront Bar
$1,281.85
$1,267.21
$1,247.14
1280
1275
1270
1265
1260
1255
1250
Yearly Total Sales
2019
2020
2021
Year

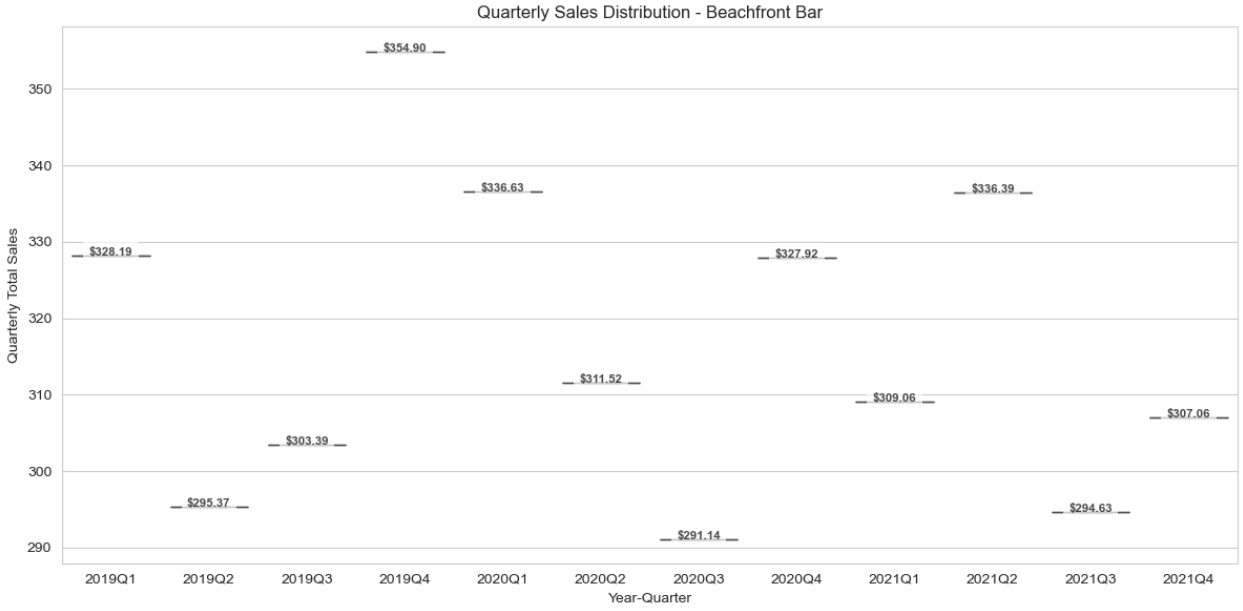

Yearly Sales Distribution - Bob's Diner
1e6
2.4
2.3
2.2
2.1
2.0
1.9
Yearly Total Sales
$2,375,460.29
$2,112,331.91
$1,849,483.49
2019
2020
2021
Year


Yearly Sales Distribution - Corner Cafe
5900
5800
5700
5600
5500
5400
5300
Yearly Total Sales
$5,910.79
$5,347.71
$5,292.93
2019
2020
2021
Year

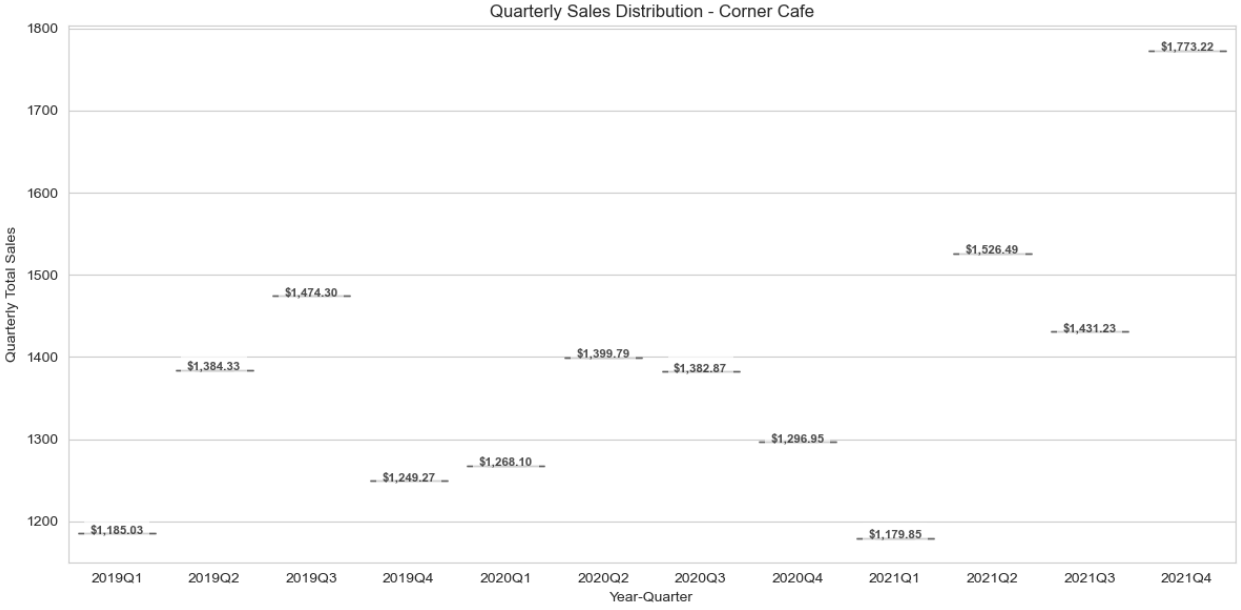

Yearly Sales Distribution - Fou Cher
$9,698.80
$9,595.70
$8,590.87
9600
9400
9200
9000
8800
8600
Yearly Total Sales
2019
2020
2021
Year

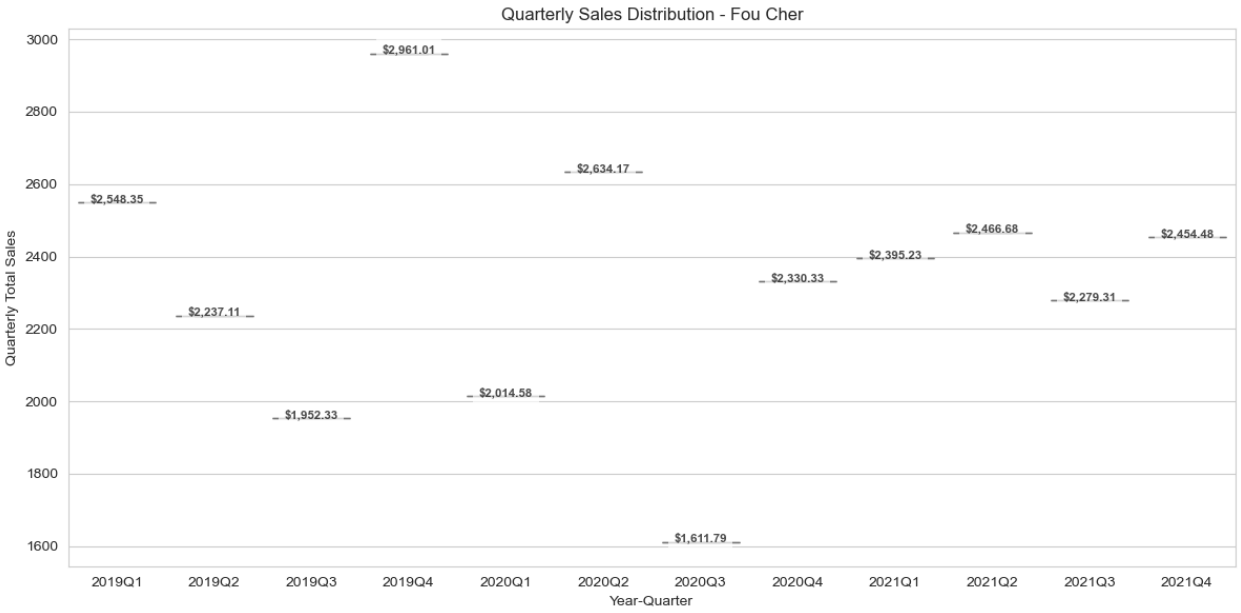

Yearly Sales Distribution - Surfs Up
$5,314.43
$5,204.41
$5,132.65
5300
5275
5250
5225
5200
5175
5150
5125
Yearly Total Sales
2019
2020
2021
Year

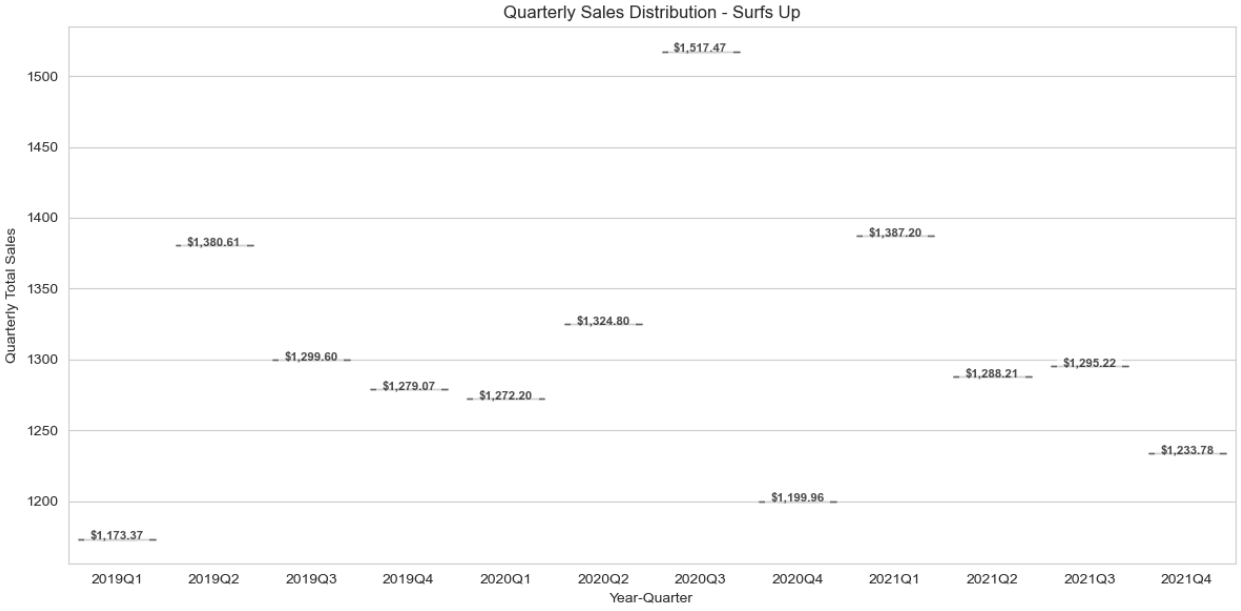

Yearly Sales Distribution - Sweet Shack
$872.59
$828.65
$877.03
870
860
850
840
830
Yearly Total Sales
2019
2020
2021
Year

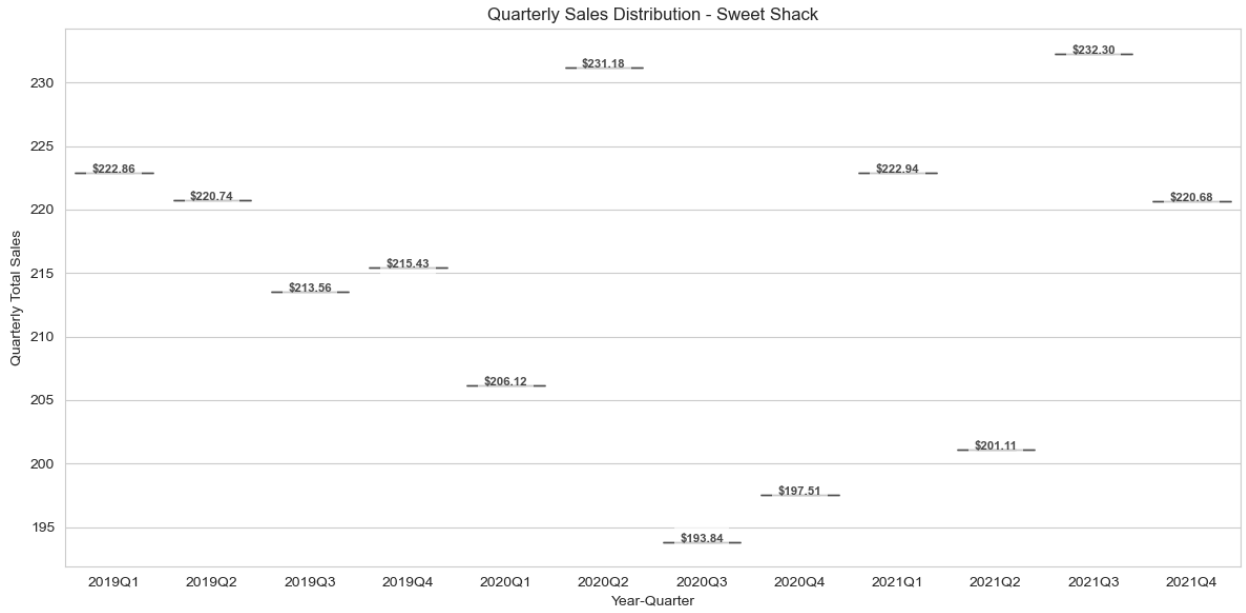

Quarterly Sales Distribution - Beachfront Bar
$328.19
$295.37
$303.39
$354.90
$336.63
$311.52
$291.14
$327.92
$309.06
$336.39
$294.63
$307.06
350
340
330
320
310
300
290
Quarterly Total Sales
2019Q1
2019Q2
2019Q3
2019Q4
2020Q1
2020Q2
2020Q3
2020Q4
2021Q1
2021Q2
2021Q3
2021Q4
Year-Quarter

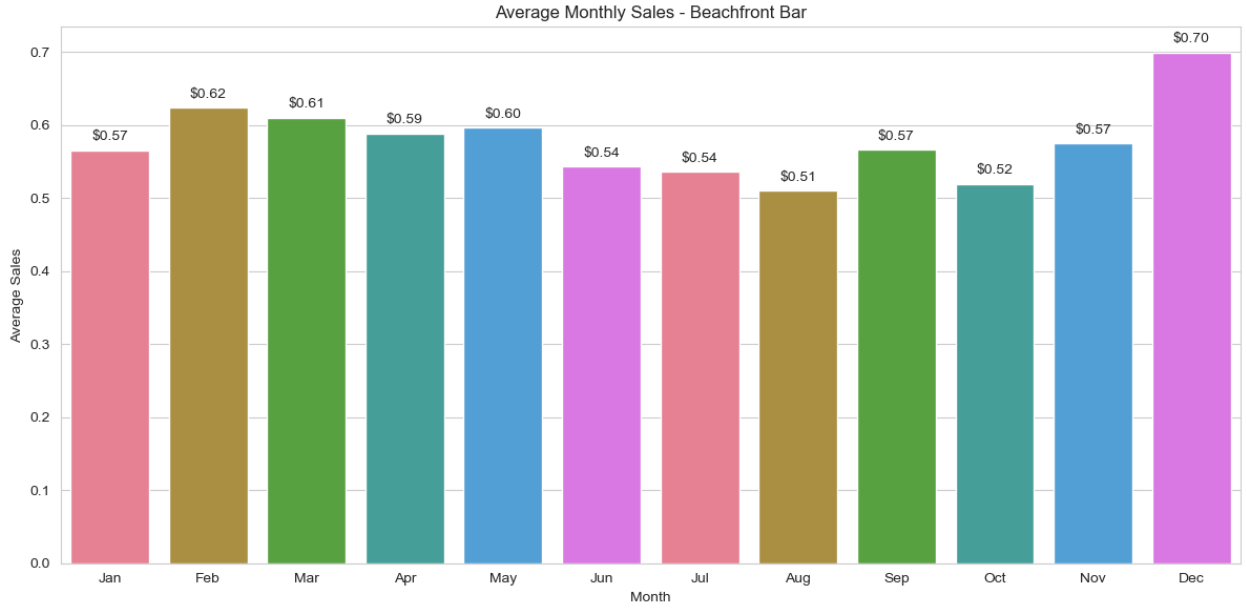

Quarterly Sales Distribution - Bob's Diner
Quarterly Total Sales
700000
650000
600000
550000
500000
450000
400000
$371,006.82
$563,392.80
$547,341.81
$367,742.06
$410,939.41
$630,013.91
$632,835.71
$438,542.88
$470,528.35
$717,992.41
$706,040.65
$480,898.88
2019Q1
2019Q2
2019Q3
2019Q4
2020Q1
2020Q2
2020Q3
2020Q4
2021Q1
2021Q2
2021Q3
2021Q4
Year-Quarter

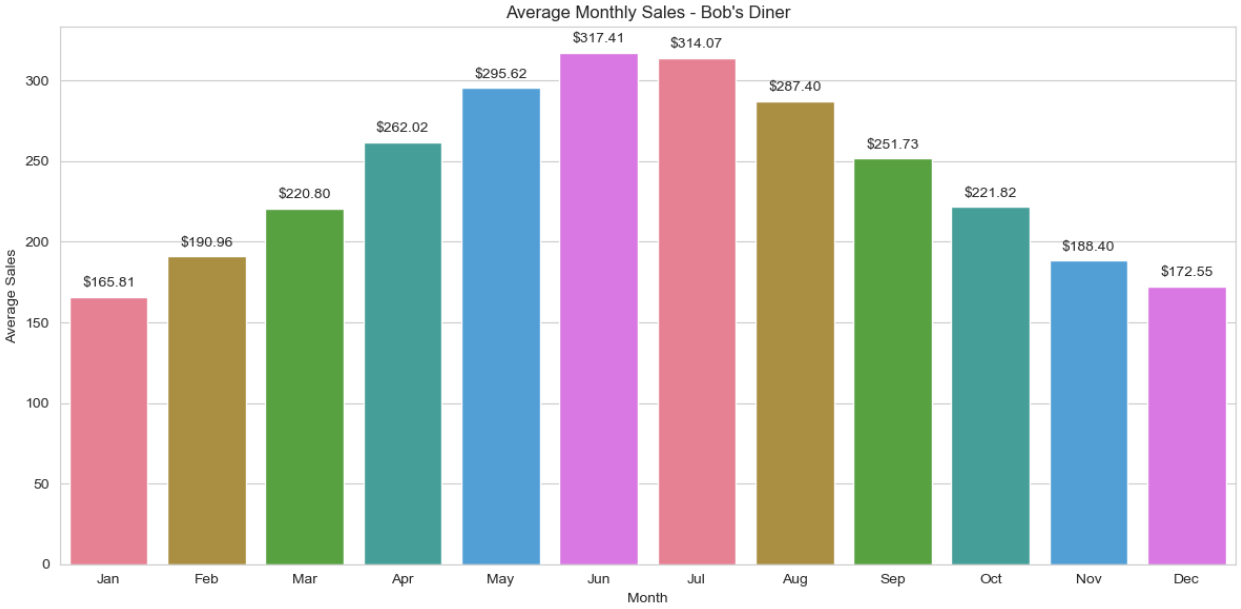

Quarterly Sales Distribution - Corner Cafe
Quarterly Total Sales
1800
1700
1600
1500
1400
1300
1200
$1,185.03
$1,384.33
$1,474.30
$1,249.27
$1,268.10
$1,399.79
$1,382.87
$1,296.95
$1,179.85
$1,526.49
$1,431.23
$1,773.22
2019Q1
2019Q2
2019Q3
2019Q4
2020Q1
2020Q2
2020Q3
2020Q4
2021Q1
2021Q2
2021Q3
2021Q4
Year-Quarter

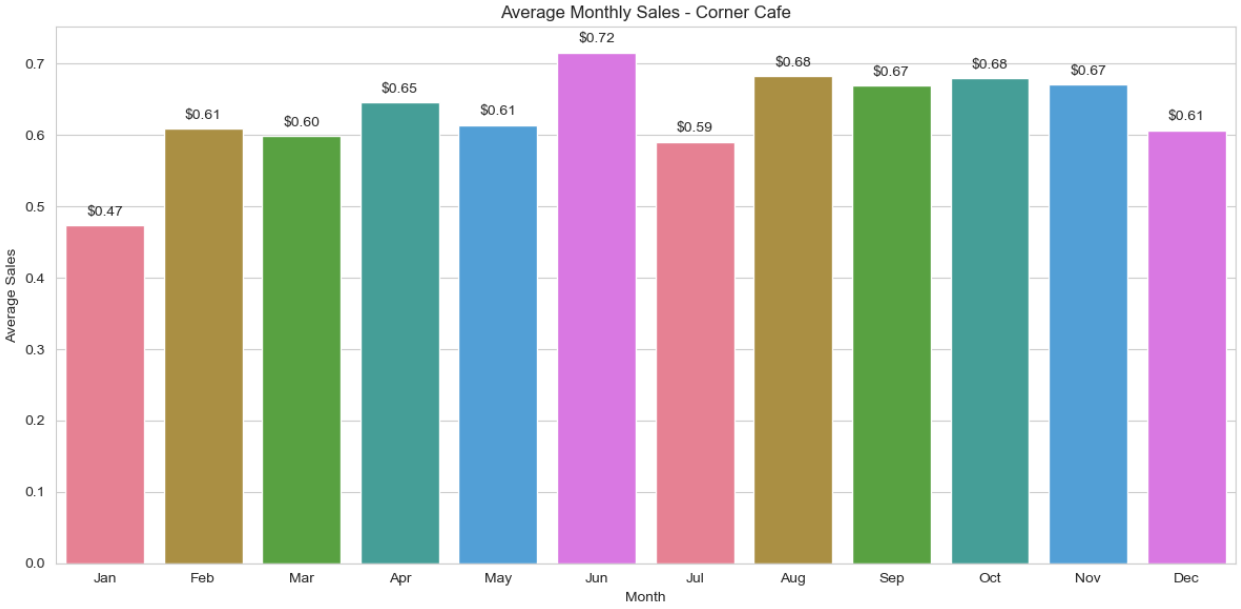

Quarterly Sales Distribution - Fou Cher
Quarterly Total Sales
Year-Quarter
3000
2800
2600
2400
2200
2000
1800
1600
$2,548.35
$2,237.11
$1,952.33
$2,961.01
$2,014.58
$2,634.17
$1,611.79
$2,330.33
$2,395.23
$2,466.68
$2,279.31
$2,454.48
2019Q1
2019Q2
2019Q3
2019Q4
2020Q1
2020Q2
2020Q3
2020Q4
2021Q1
2021Q2
2021Q3
2021Q4

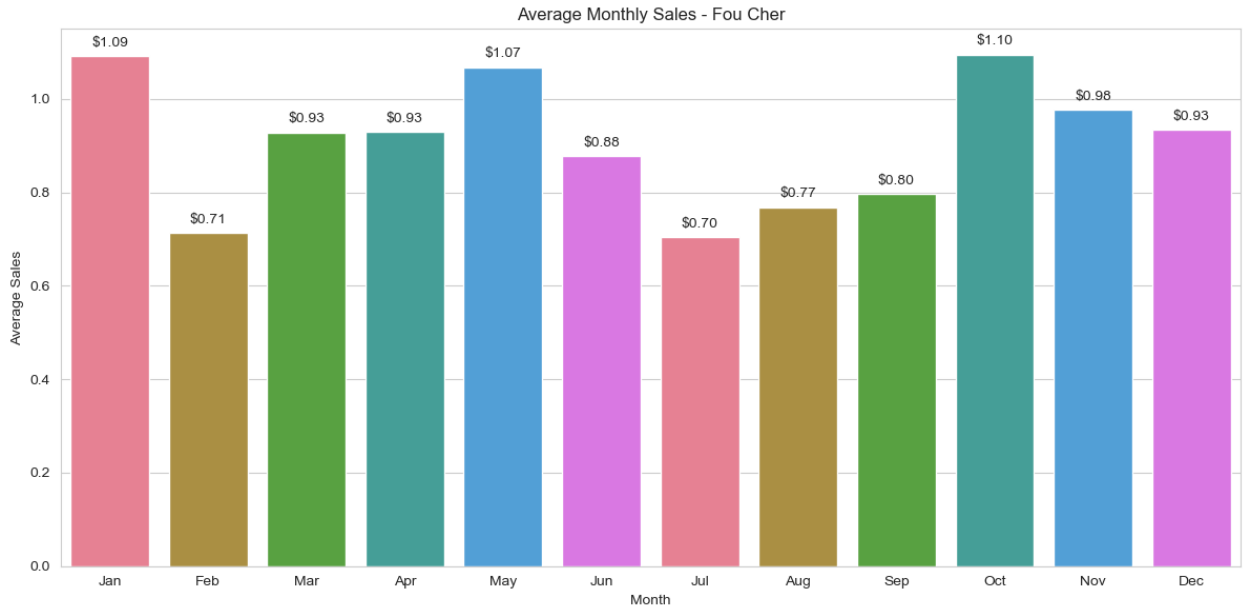

Quarterly Sales Distribution - Surfs Up
Quarterly Total Sales
Year-Quarter
1500
1450
1400
1350
1300
1250
1200
$1,173.37
$1,380.61
$1,299.60
$1,279.07
$1,272.20
$1,324.80
$1,517.47
$1,199.96
$1,387.20
$1,288.21
$1,295.22
$1,233.78
2019Q1
2019Q2
2019Q3
2019Q4
2020Q1
2020Q2
2020Q3
2020Q4
2021Q1
2021Q2
2021Q3
2021Q4

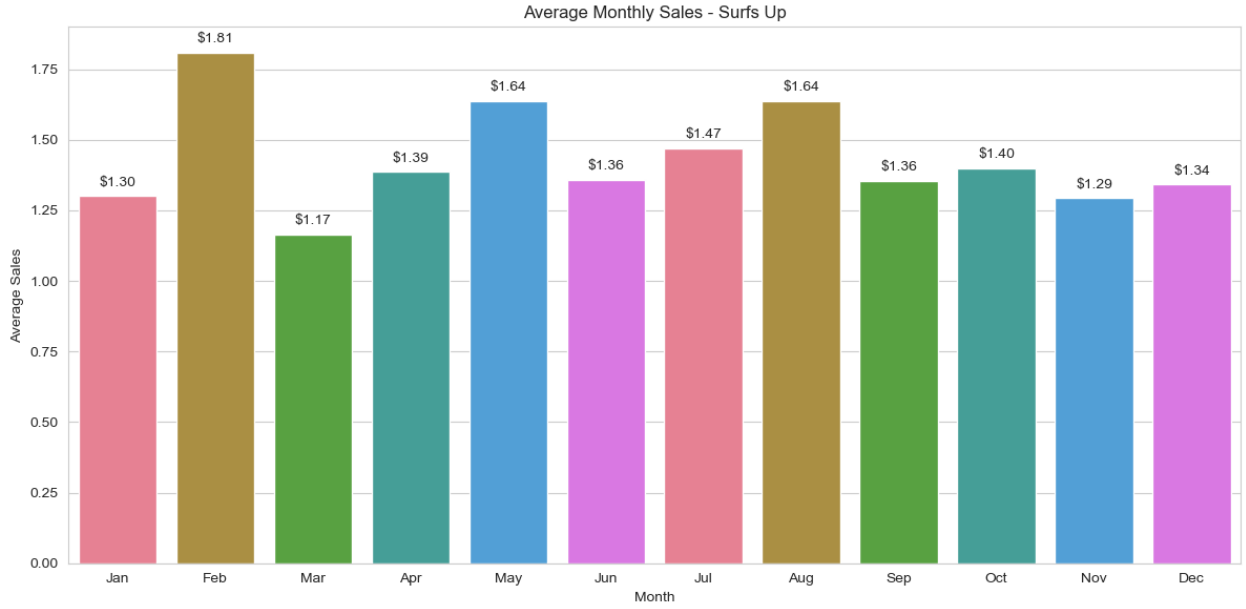

Quarterly Sales Distribution - Sweet Shack
Quarterly Total Sales
Year-Quarter
$222.86
$220.74
$213.56
$215.43
$206.12
$231.18
$193.84
$197.51
$222.94
$201.11
$232.30
$220.68
2019Q1
2019Q2
2019Q3
2019Q4
2020Q1
2020Q2
2020Q3
2020Q4
2021Q1
2021Q2
2021Q3
2021Q4

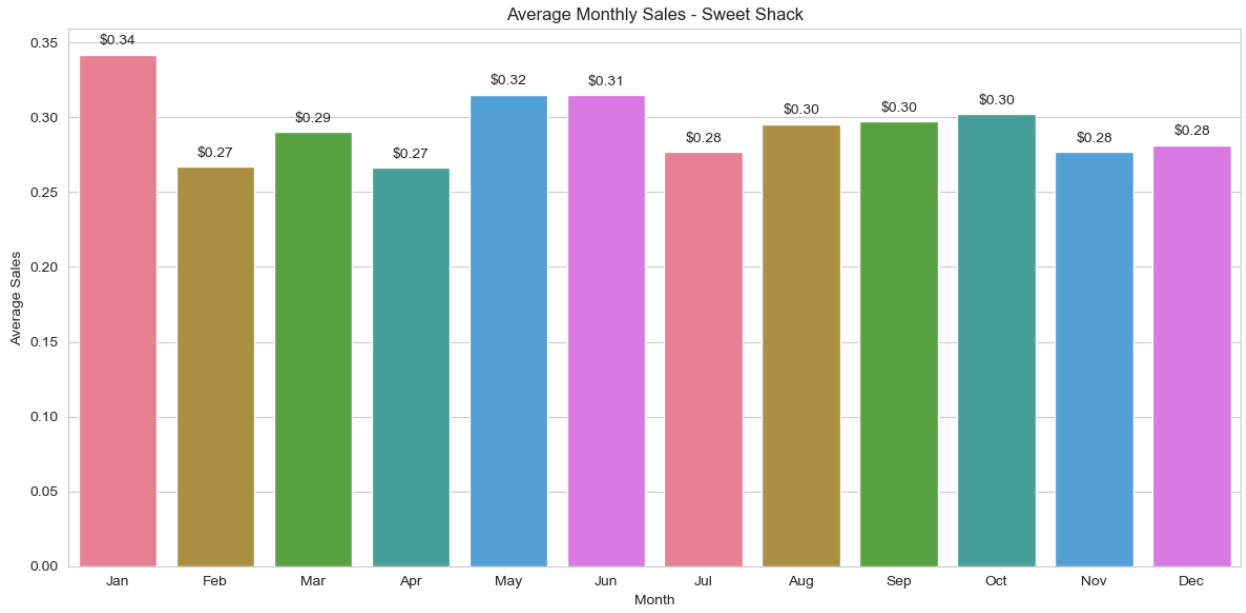

Average Monthly Sales - Beachfront Bar
Average Sales
Month
$0.57
$0.62
$0.61
$0.59
$0.60
$0.54
$0.54
$0.51
$0.57
$0.52
$0.57
$0.70
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec

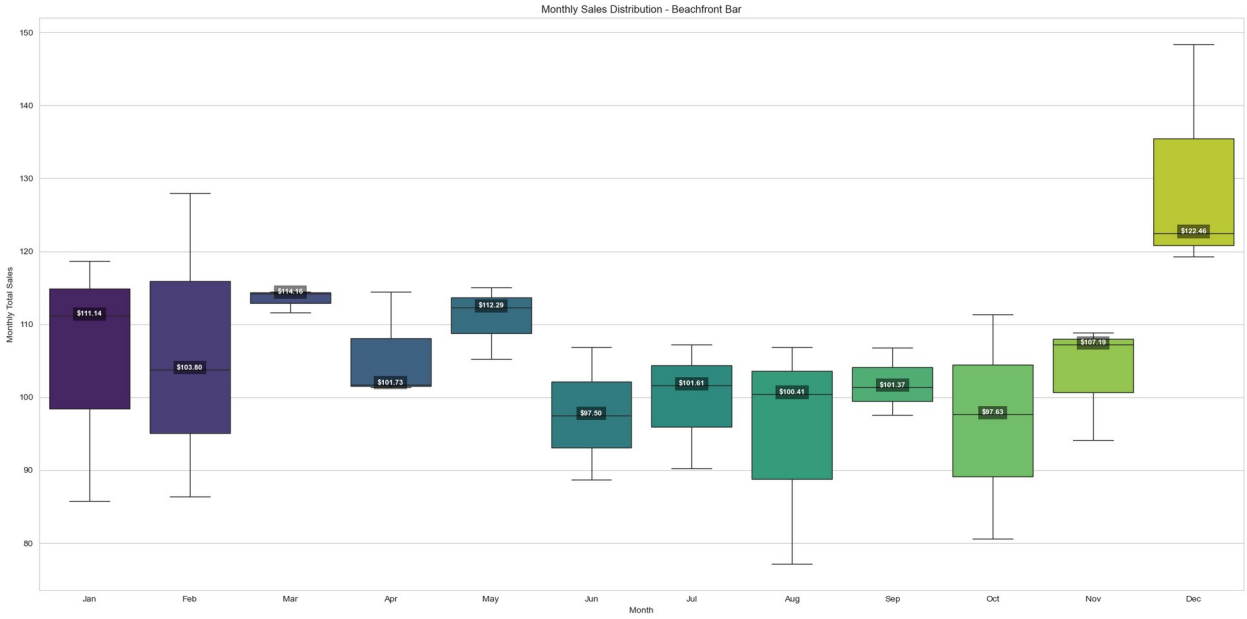

Average Monthly Sales - Bob's Diner
Average Sales
Month
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
$165.81
$190.96
$220.80
$262.02
$295.62
$317.41
$314.07
$287.40
$251.73
$221.82
$188.40
$172.55

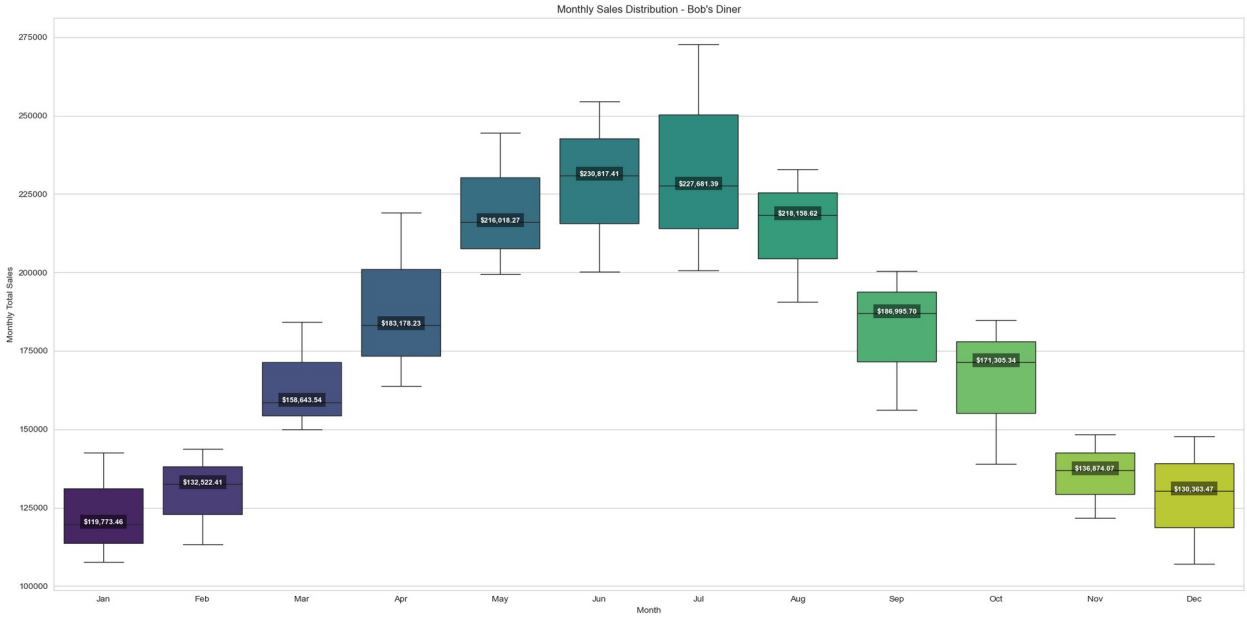

Average Monthly Sales - Corner Cafe
Average Sales
Month
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
$0.47
$0.61
$0.60
$0.65
$0.61
$0.72
$0.59
$0.68
$0.67
$0.68
$0.67
$0.61

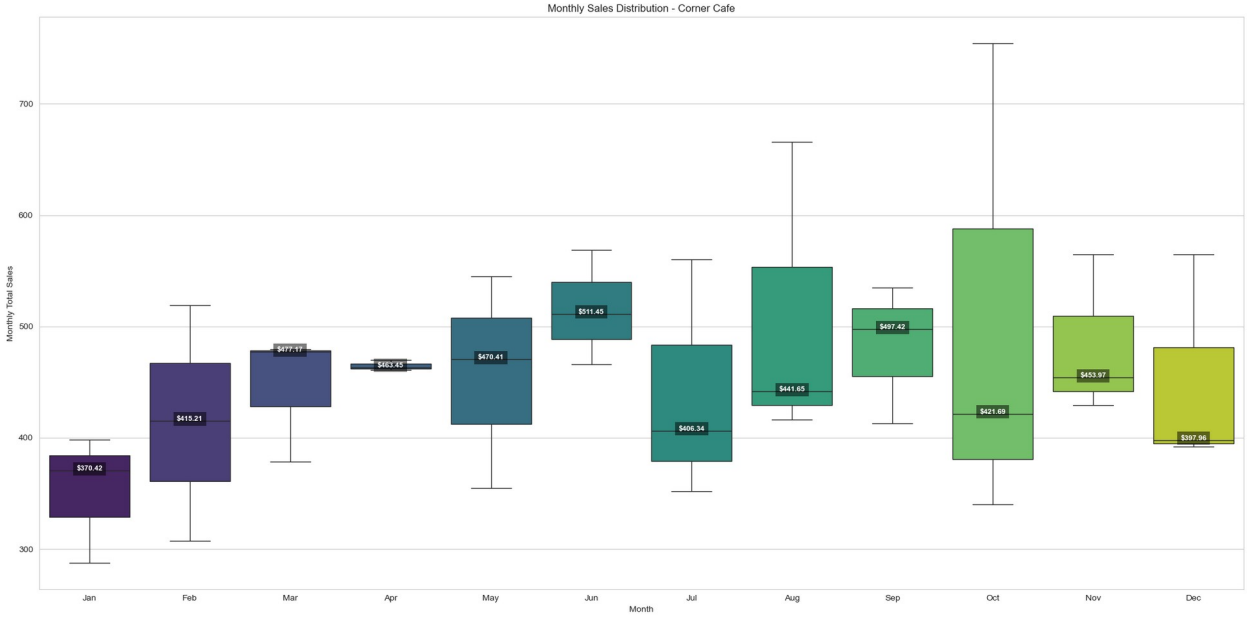

Average Monthly Sales - Fou Cher
Average Sales
Month
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
$1.09
$0.71
$0.93
$0.93
$1.07
$0.88
$0.70
$0.77
$0.80
$1.10
$0.98
$0.93

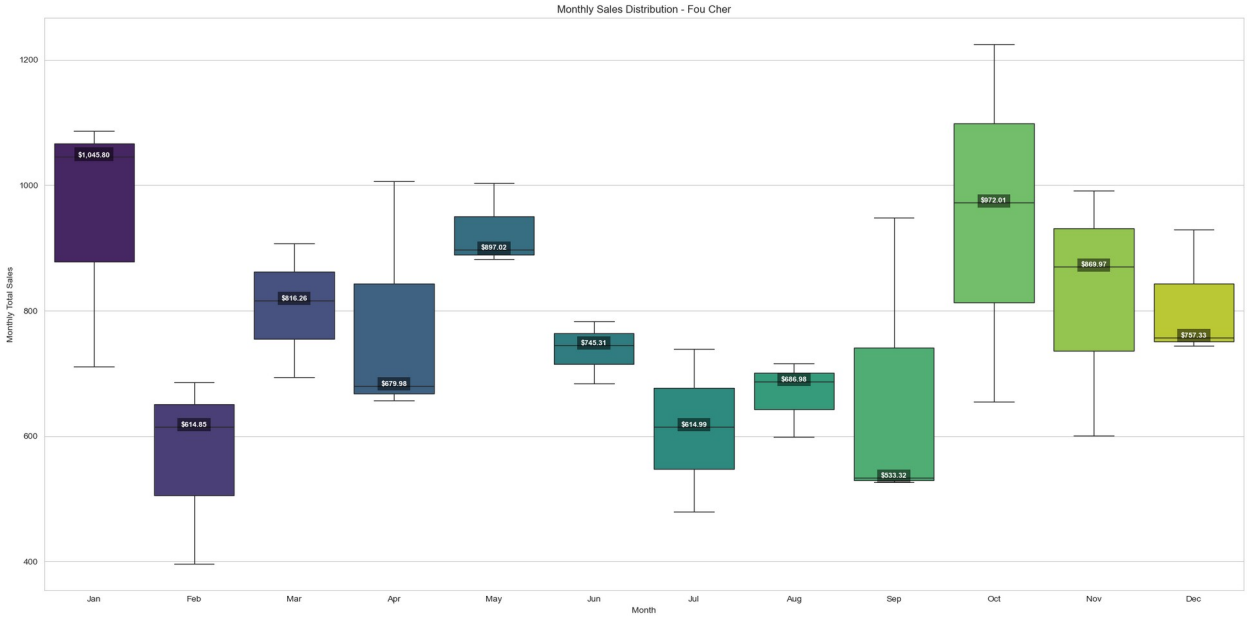

Average Monthly Sales - Surfs Up
Average Sales
Month
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
$1.30
$1.81
$1.17
$1.39
$1.64
$1.36
$1.47
$1.64
$1.36
$1.40
$1.29
$1.34

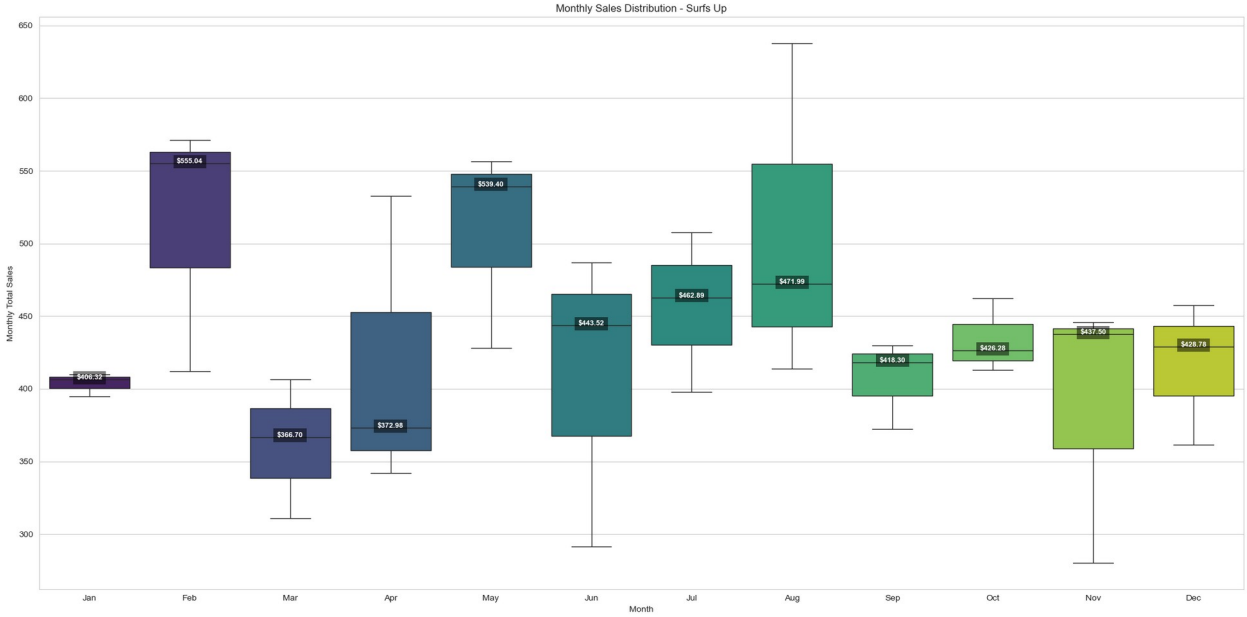

Average Monthly Sales - Sweet Shack
Average Sales
Month
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
$0.34
$0.27
$0.29
$0.27
$0.32
$0.31
$0.28
$0.30
$0.30
$0.30
$0.28
$0.28
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec

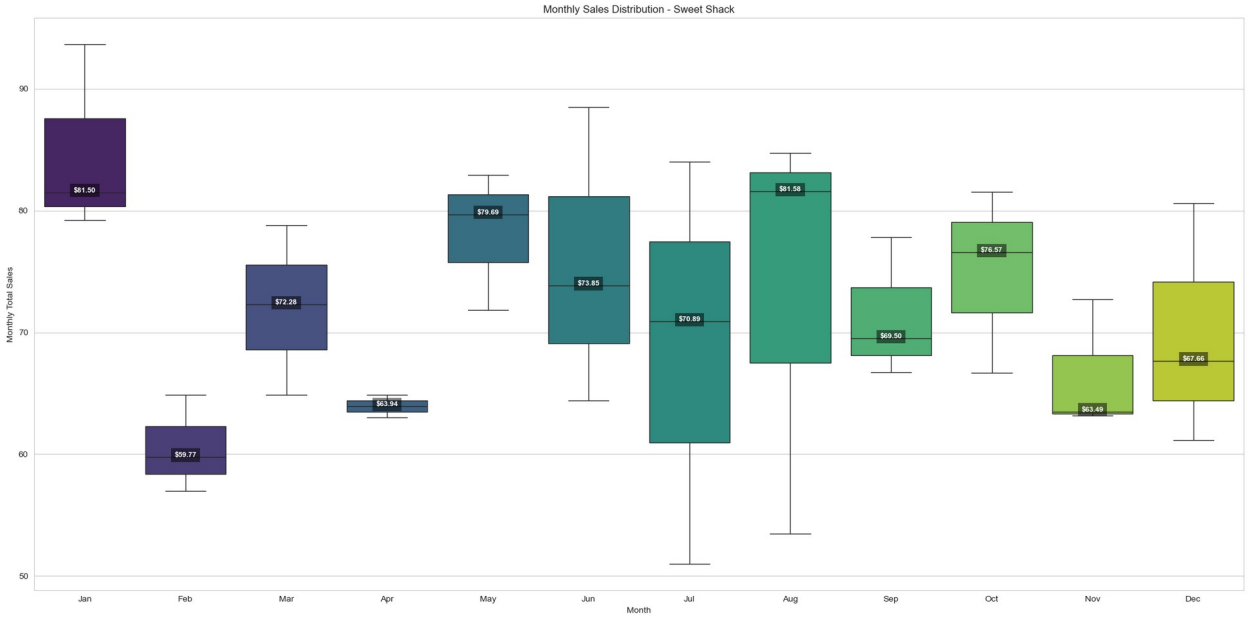

Monthly Sales Distribution - Beachfront Bar
Monthly Total Sales
Month
80
90
100
110
120
130
140
150
$111.14
$103.80
$114.10
$101.73
$112.29
$97.50
$101.61
$100.41
$101.37
$97.63
$107.19
$122.46
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec

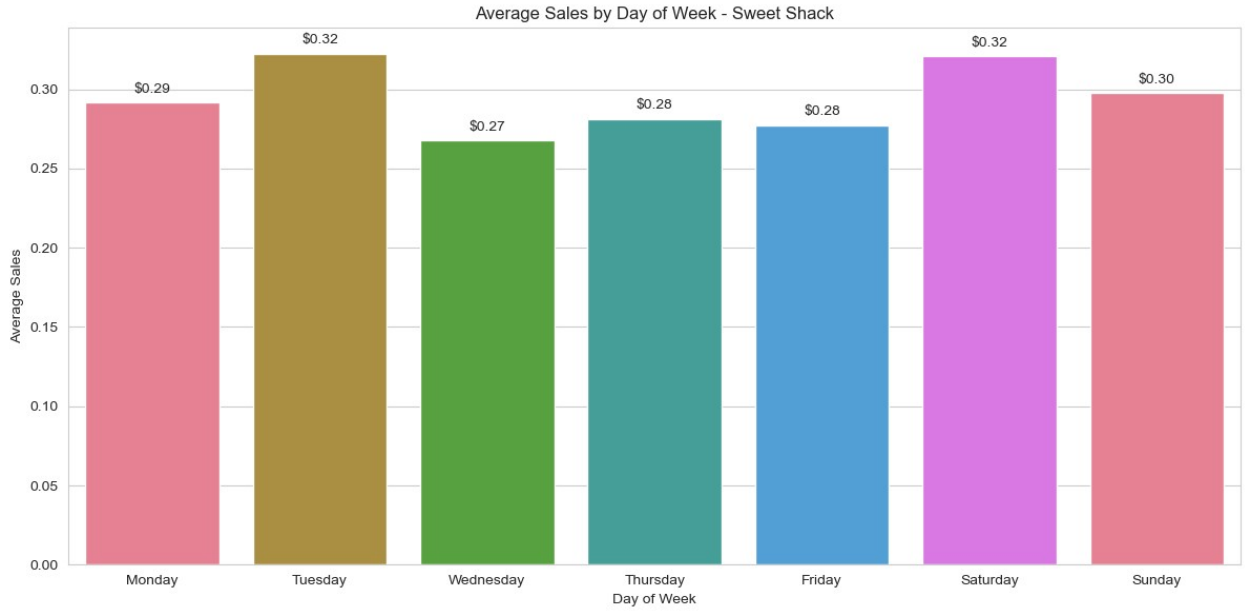

Monthly Sales Distribution - Bob's Diner
Monthly Total Sales
275000
250000
225000
200000
175000
150000
125000
100000
$119,773.46
$132,522.41
$158,643.54
$183,178.23
$216,018.27
$230,817.41
$227,681.39
$218,158.62
$186,995.70
$171,305.34
$136,874.07
$130,363.47
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month

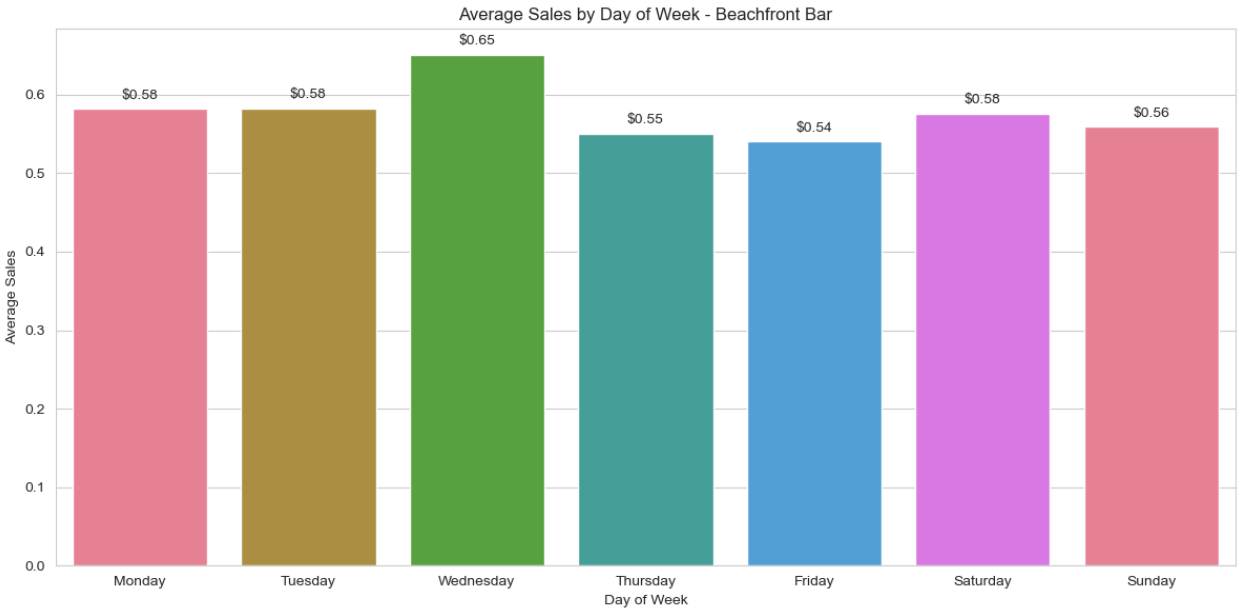

Monthly Sales Distribution - Corner Cafe
Monthly Total Sales
700
600
500
400
300
$370.42
$415.21
$477.17
$463.43
$470.41
$511.45
$406.34
$441.65
$497.42
$421.69
$453.97
$397.96
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month

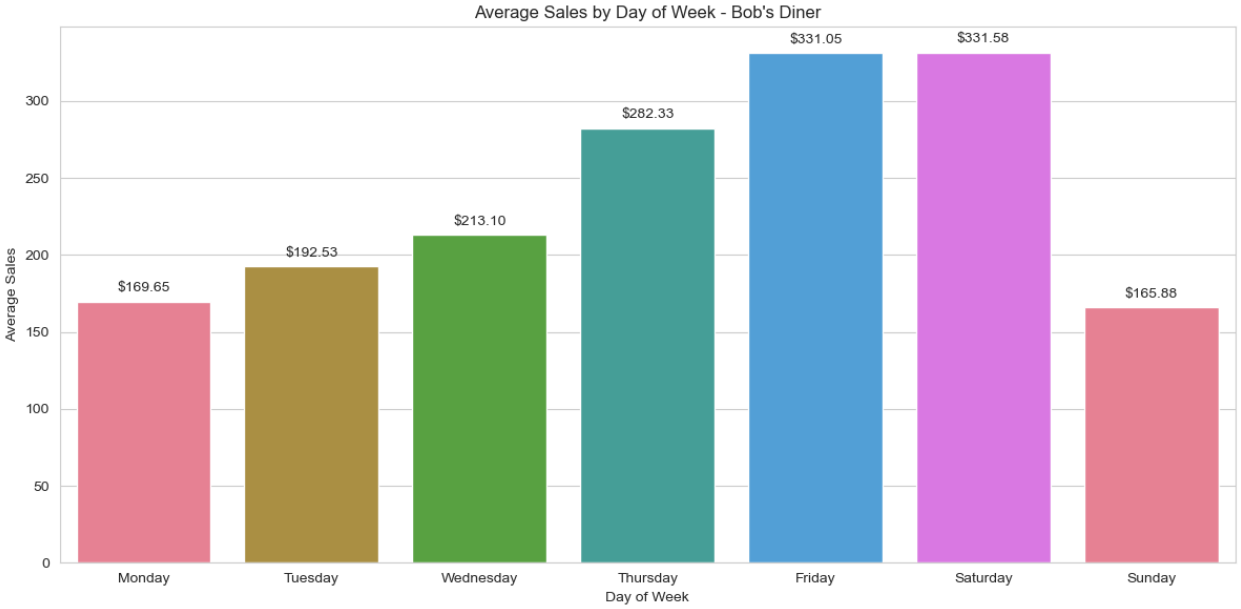

Monthly Sales Distribution - Fou Cher
Monthly Total Sales
1200
1000
800
600
400
$1,045.80
$614.85
$816.26
$679.98
$897.02
$745.31
$614.99
$686.98
$533.32
$972.01
$869.97
$757.33
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month

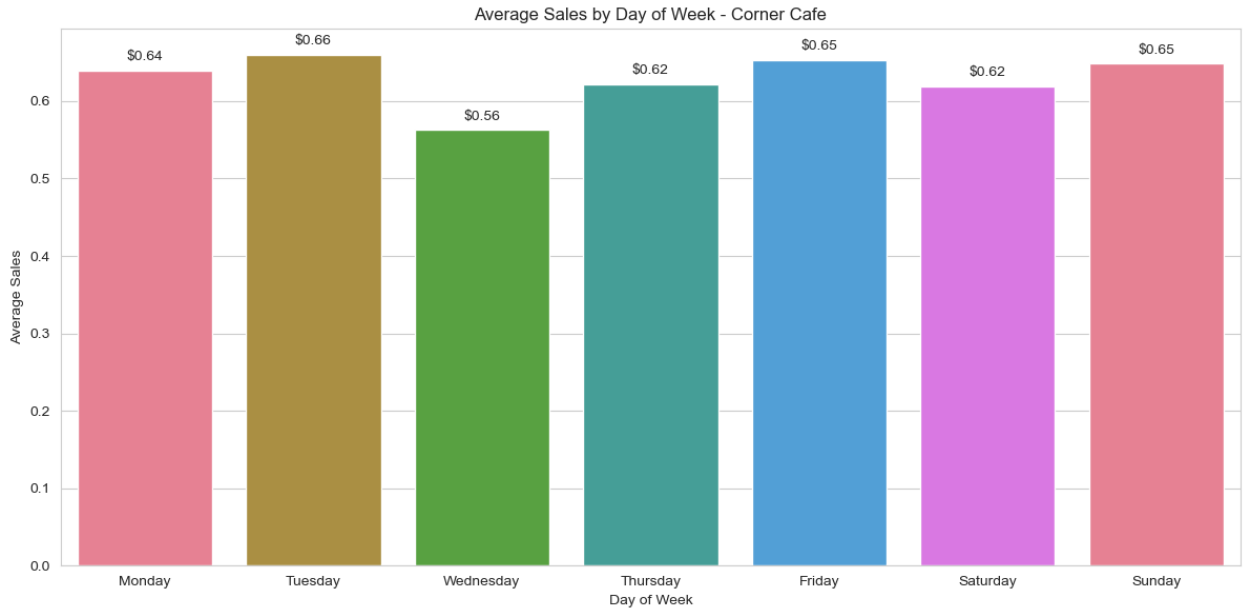

Monthly Sales Distribution - Surfs Up
Monthly Total Sales
650
600
550
500
450
400
350
300
$406.32
$555.04
$366.70
$372.98
$539.40
$443.52
$462.89
$471.99
$418.30
$426.28
$437.50
$428.78
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month

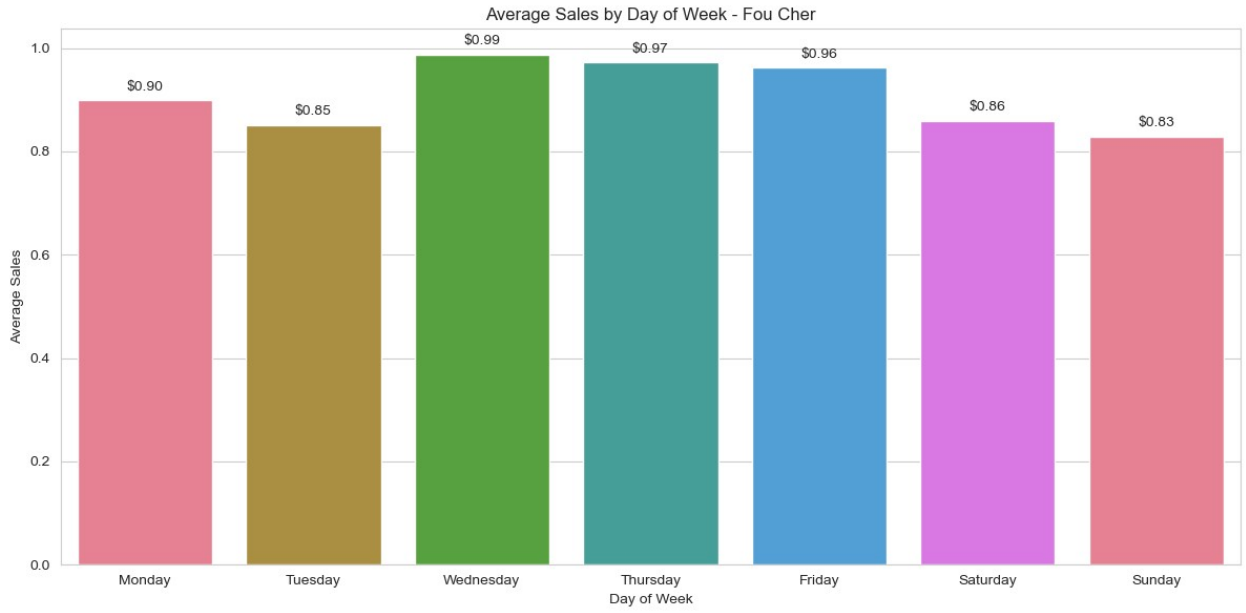

Monthly Sales Distribution - Sweet Shack
Monthly Total Sales
90
80
70
60
50
$81.50
$59.77
$72.28
$63.94
$79.69
$73.85
$70.89
$81.58
$69.50
$76.57
$63.49
$67.66
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month

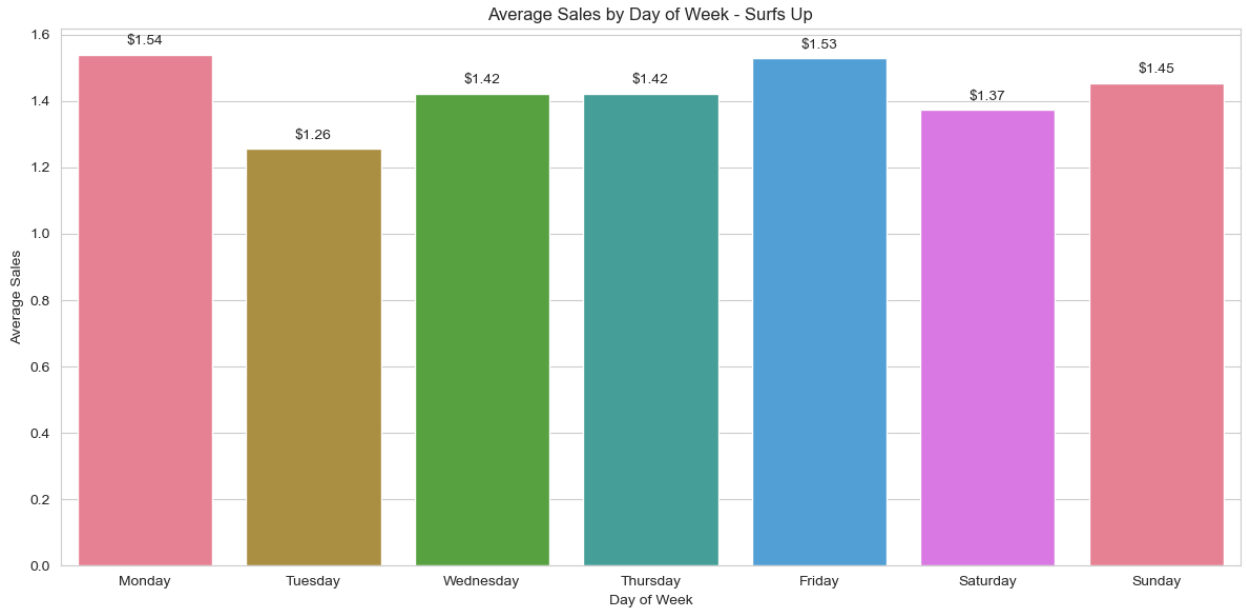

Average Sales by Day of Week - Sweet Shack
Average Sales
0.30
0.25
0.20
0.15
0.10
0.05
0.00
$0.29
$0.32
$0.27
$0.28
$0.28
$0.32
$0.30
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

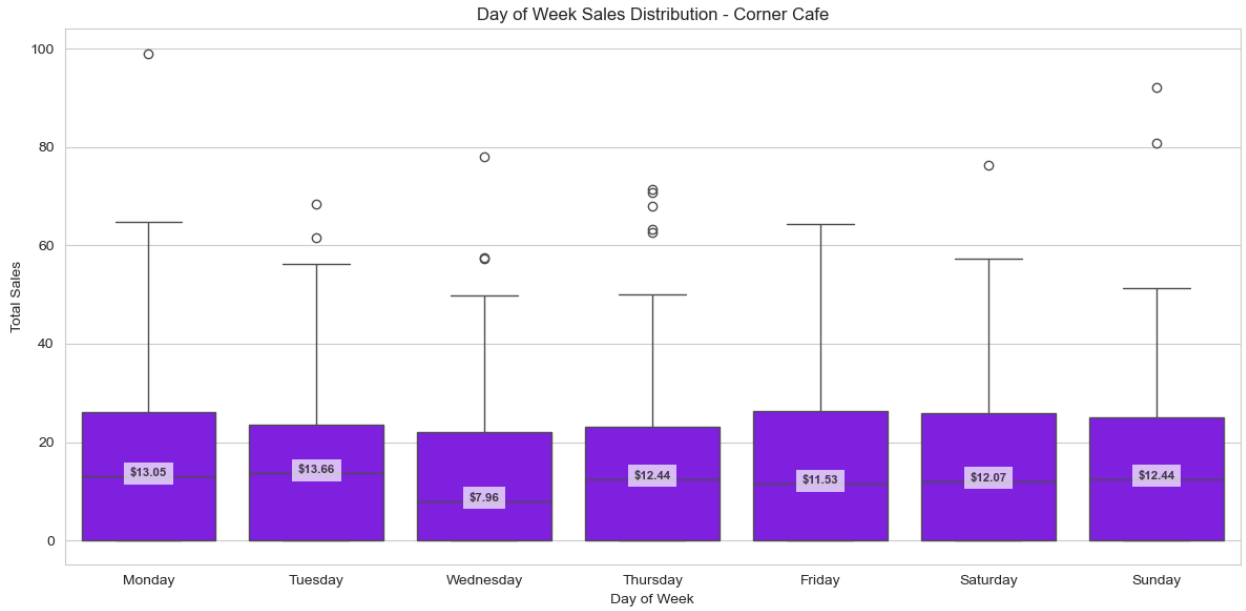

Average Sales by Day of Week - Beachfront Bar
$0.58
$0.58
$0.65
$0.55
$0.54
$0.58
$0.56
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Average Sales
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

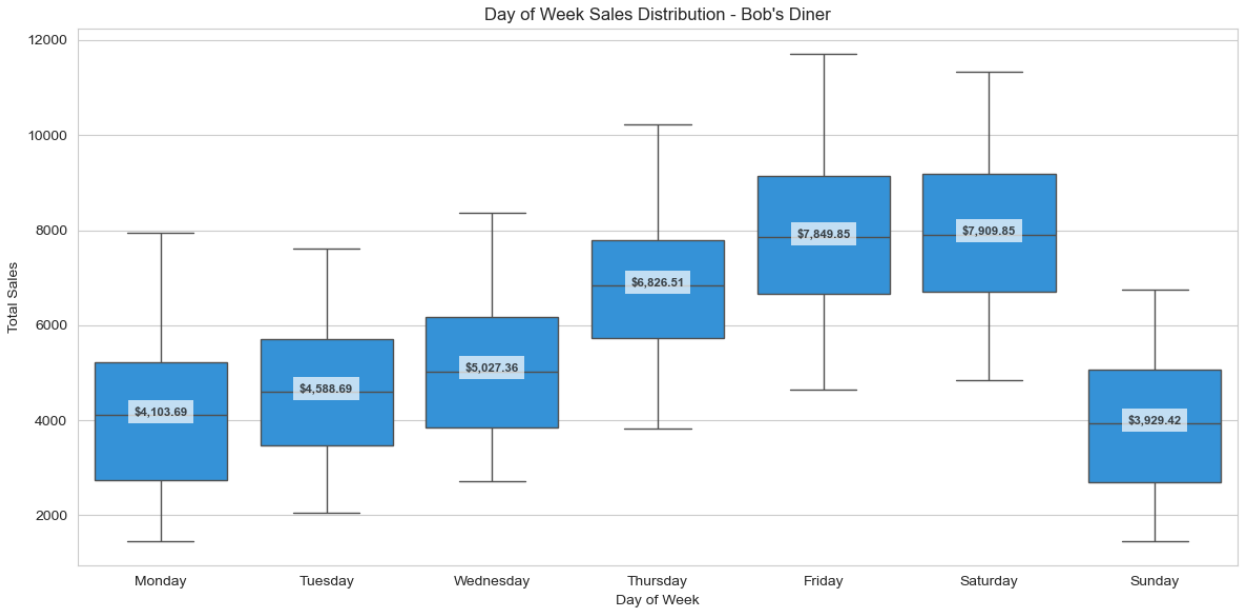

Average Sales by Day of Week - Bob's Diner
$169.65
$192.53
$213.10
$282.33
$331.05
$331.58
$165.88
0
50
100
150
200
250
300
Average Sales
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

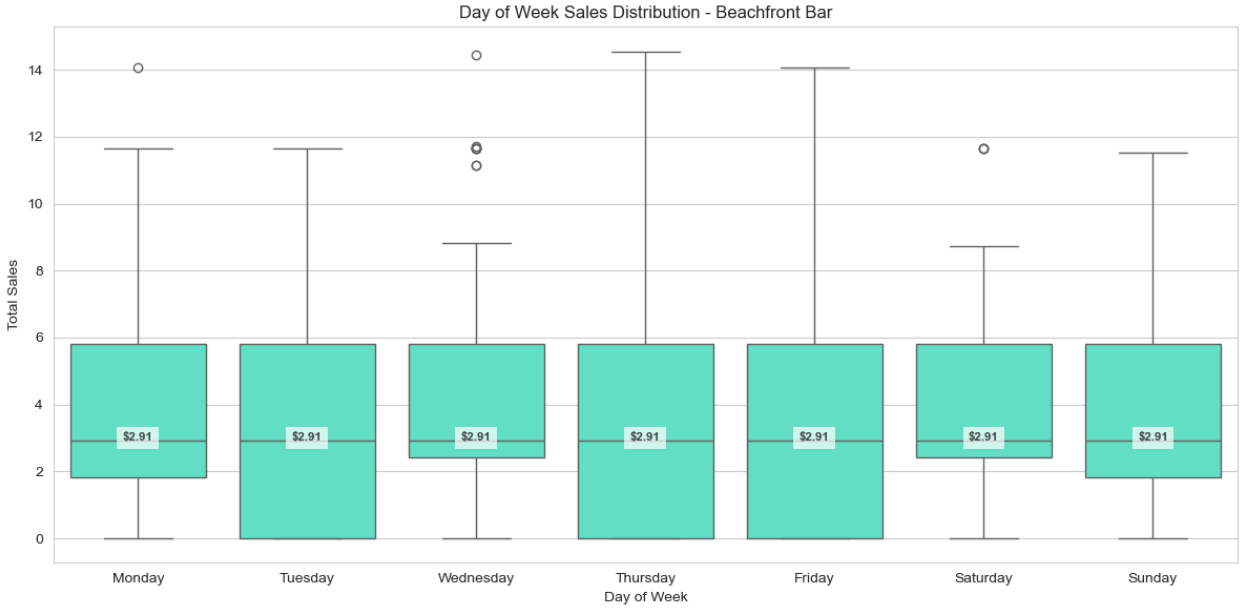

Average Sales by Day of Week - Corner Cafe
$0.64
$0.66
$0.56
$0.62
$0.65
$0.62
$0.65
Average Sales
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

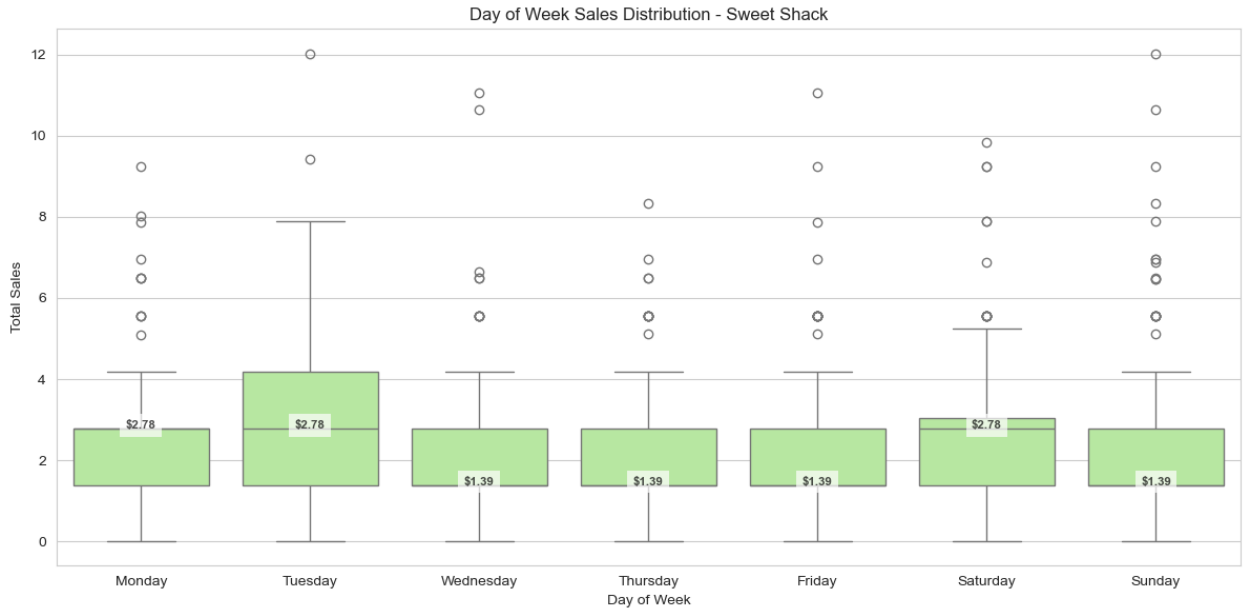

Average Sales by Day of Week - Fou Cher
$0.90
$0.85
$0.99
$0.97
$0.96
$0.86
$0.83
Average Sales
0.0
0.2
0.4
0.6
0.8
1.0
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

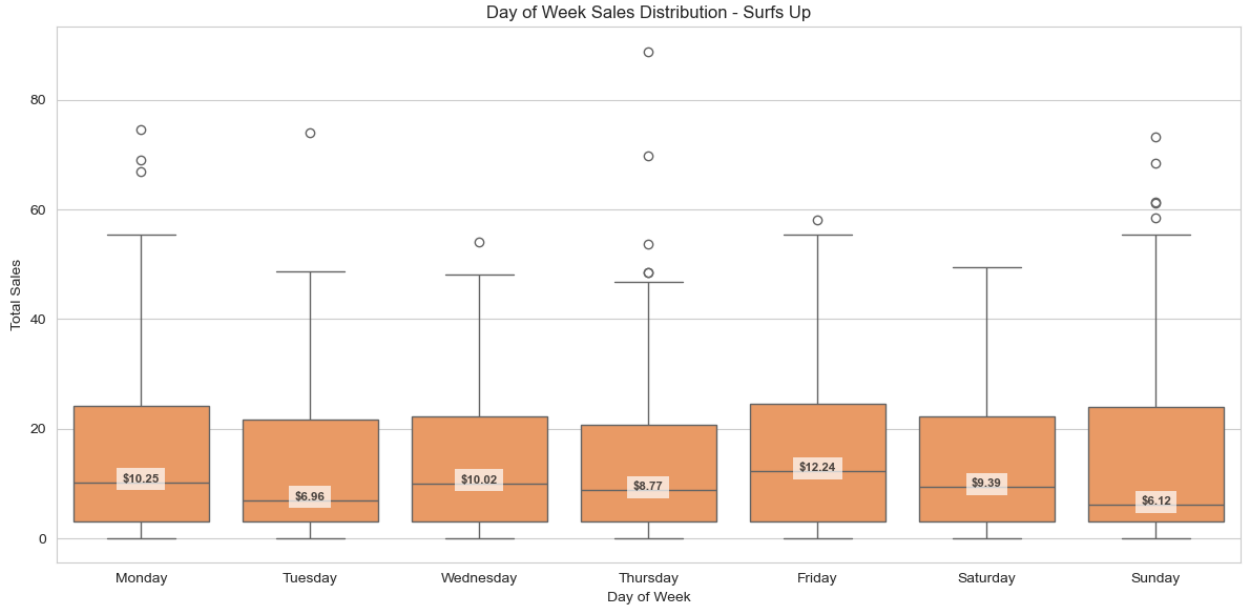

Average Sales by Day of Week - Surfs Up
Average Sales
Day of Week
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
$1.54
$1.26
$1.42
$1.42
$1.53
$1.37
$1.45

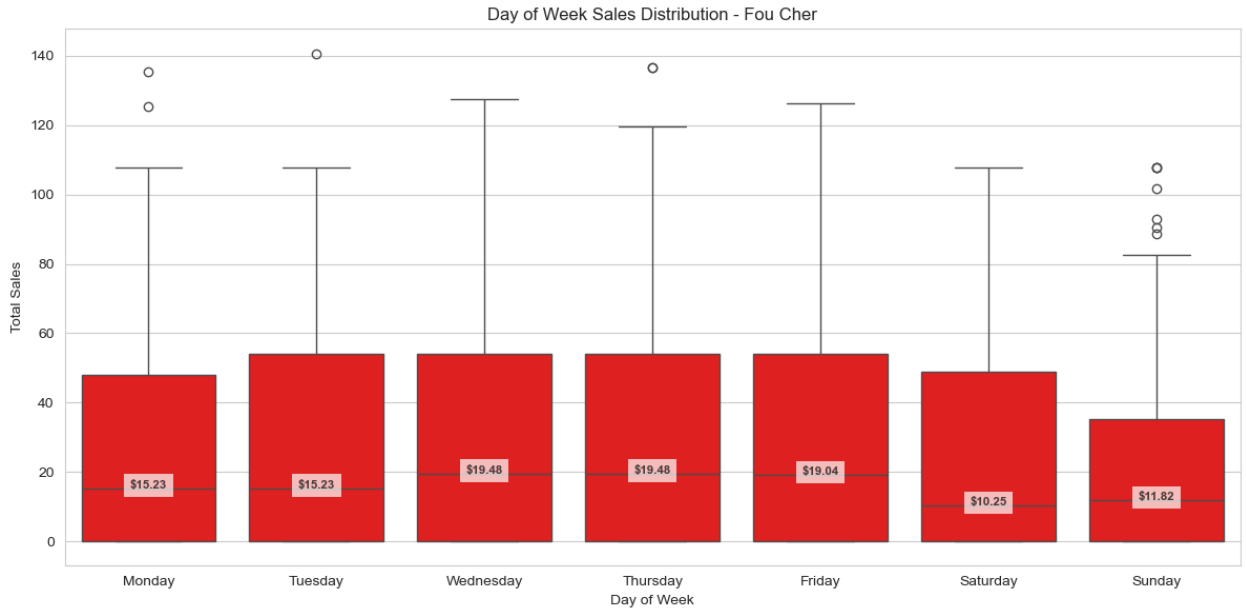

Day of Week Sales Distribution - Corner Cafe
Total Sales
Day of Week
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
$13.05
$13.66
$7.96
$12.44
$11.53
$12.07
$12.44

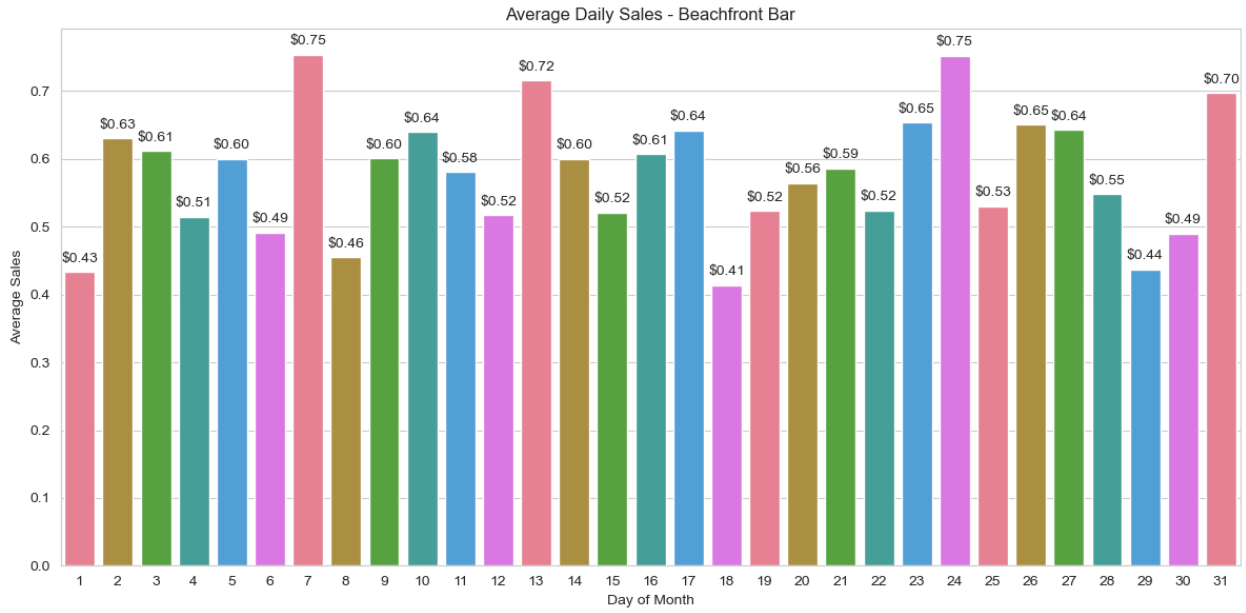

Day of Week Sales Distribution - Bob's Diner
Total Sales
12000
10000
8000
6000
4000
2000
$4,103.69
$4,588.69
$5,027.36
$6,826.51
$7,849.85
$7,909.85
$3,929.42
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

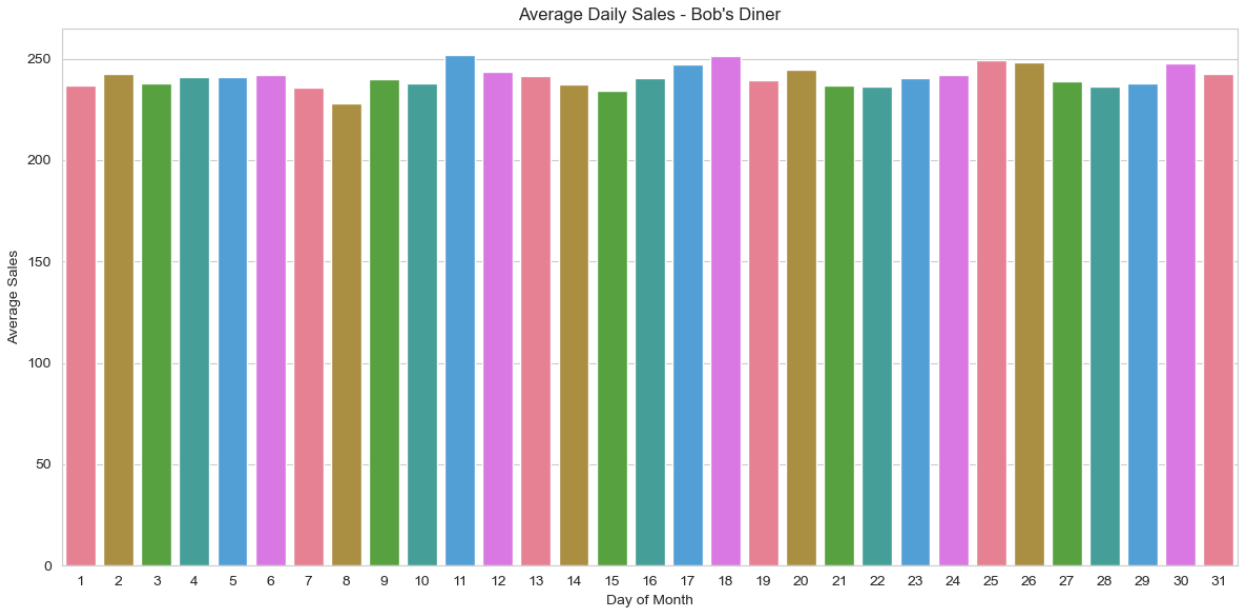

Day of Week Sales Distribution - Beachfront Bar
Total Sales
14
12
10
8
6
4
2
0
$2.91
$2.91
$2.91
$2.91
$2.91
$2.91
$2.91
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

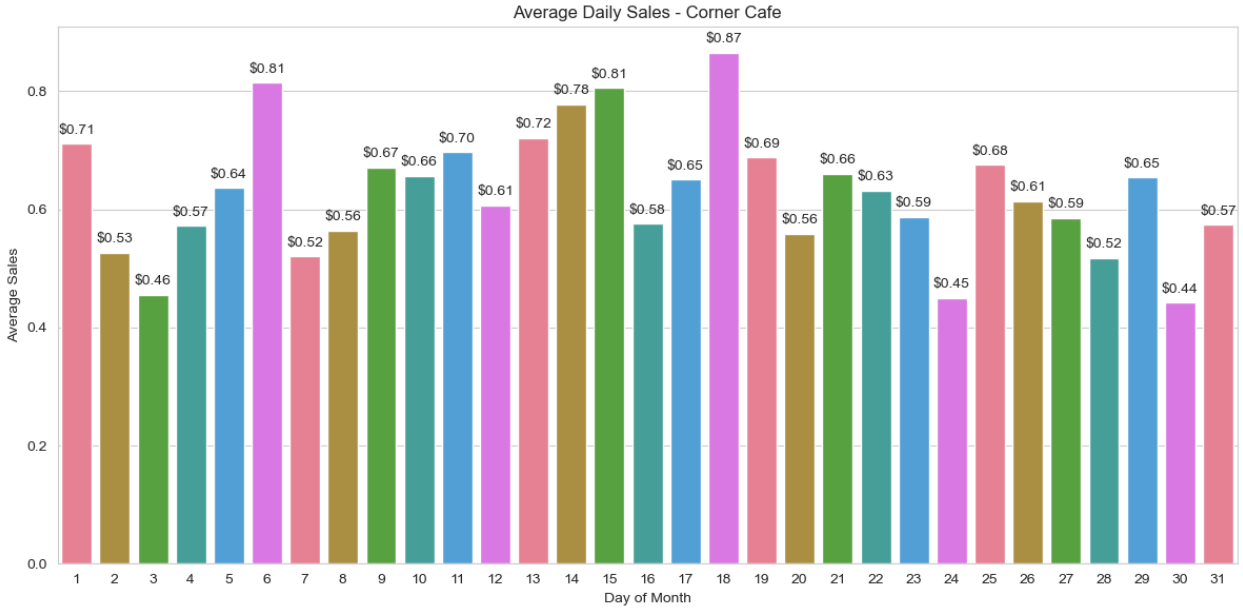

Day of Week Sales Distribution - Sweet Shack
Total Sales
0
2
4
6
8
10
12
$2.78
$2.78
$1.39
$1.39
$1.39
$2.78
$1.39
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

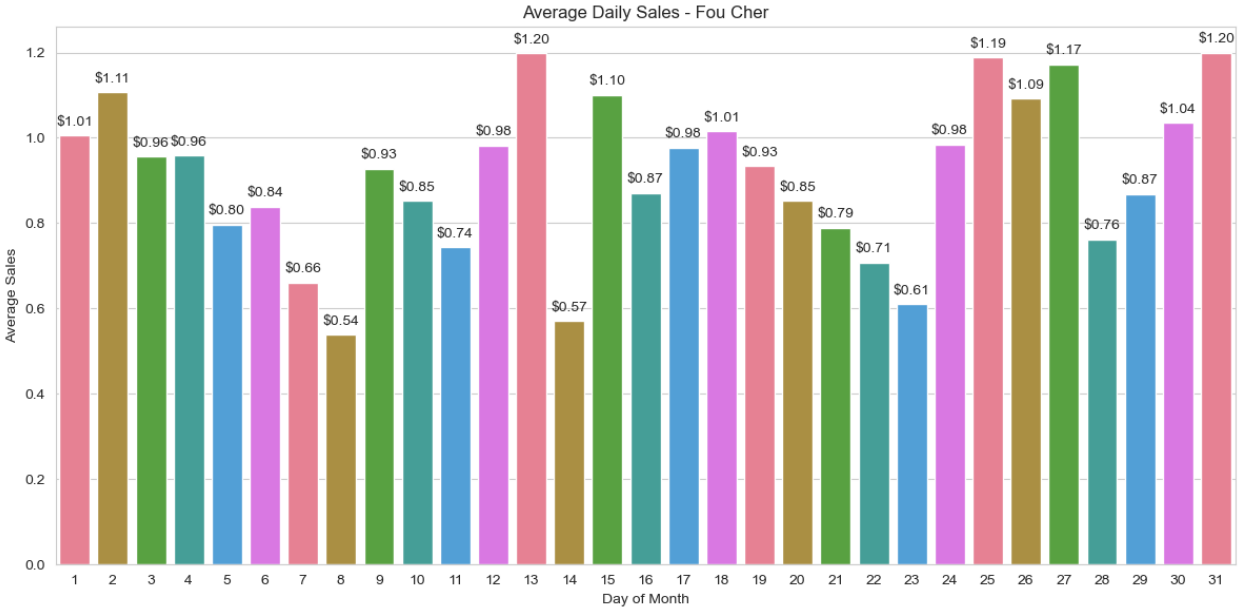

Day of Week Sales Distribution - Surfs Up
Total Sales
0
20
40
60
80
$10.25
$6.96
$10.02
$8.77
$12.24
$9.39
$6.12
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Day of Week

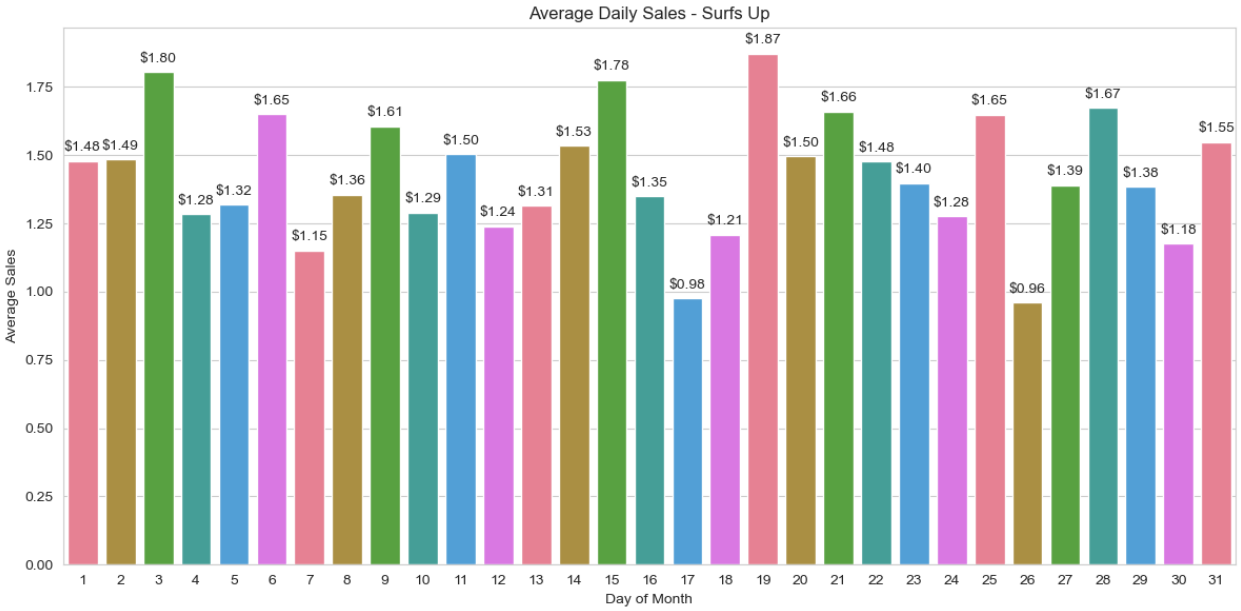

Day of Week Sales Distribution - Fou Cher
Total Sales
Day of Week
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
$15.23
$15.23
$19.48
$19.48
$19.04
$10.25
$11.82

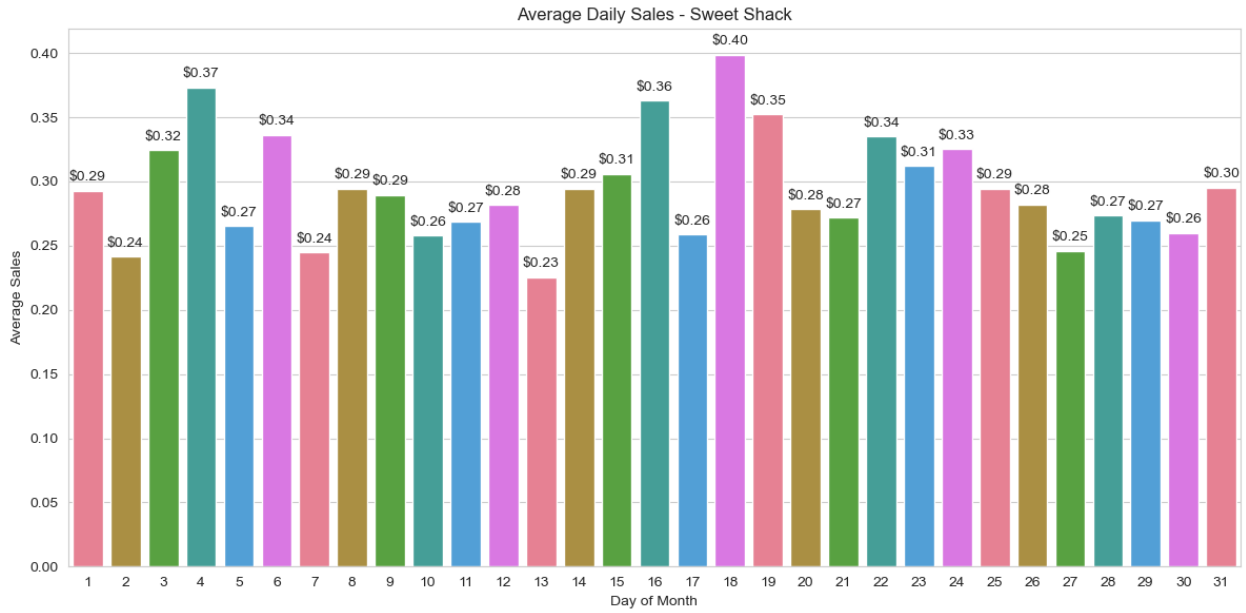

Average Daily Sales - Beachfront Bar
Average Sales
Day of Month
$0.43
$0.63
$0.61
$0.51
$0.60
$0.49
$0.75
$0.46
$0.60
$0.64
$0.58
$0.52
$0.72
$0.60
$0.52
$0.61
$0.64
$0.41
$0.52
$0.56
$0.59
$0.52
$0.65
$0.75
$0.53
$0.65
$0.64
$0.55
$0.44
$0.49
$0.70


Average Daily Sales - Bob's Diner
Average Sales
0
50
100
150
200
250
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Day of Month

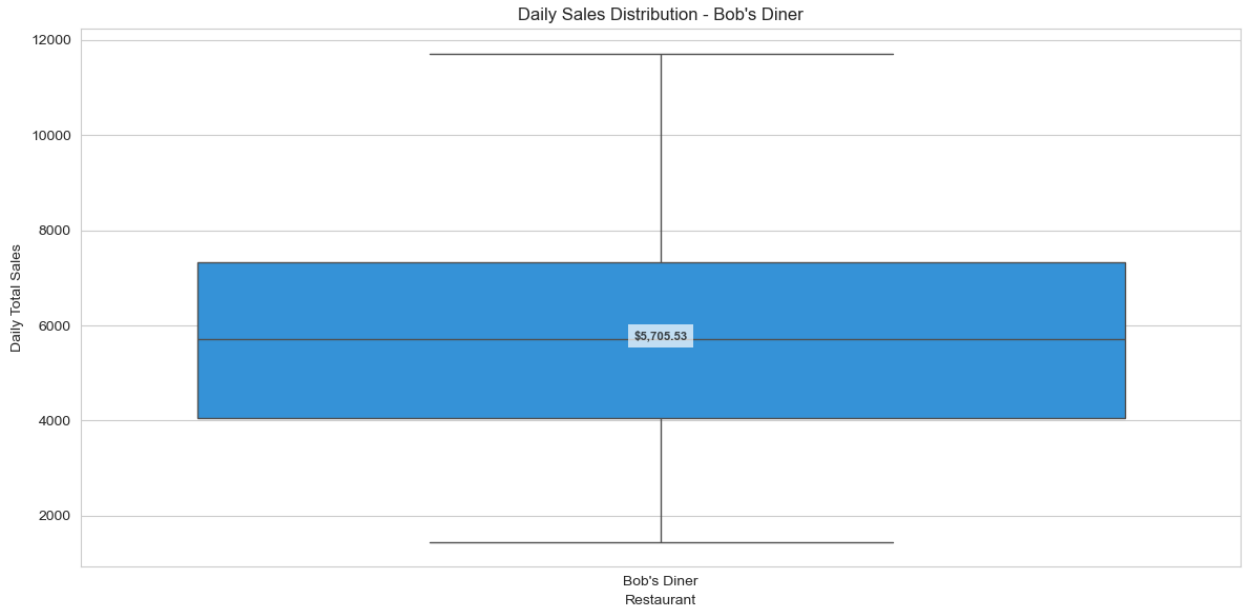

Average Daily Sales - Corner Cafe
Average Sales
0.0
0.2
0.4
0.6
0.8
$0.71 $0.53 $0.46 $0.57 $0.64 $0.81 $0.52 $0.56 $0.67 $0.66 $0.70 $0.61 $0.72 $0.78 $0.81 $0.58 $0.65 $0.87 $0.69 $0.56 $0.66 $0.63 $0.59 $0.45 $0.68 $0.61 $0.59 $0.52 $0.65 $0.44 $0.57
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Day of Month

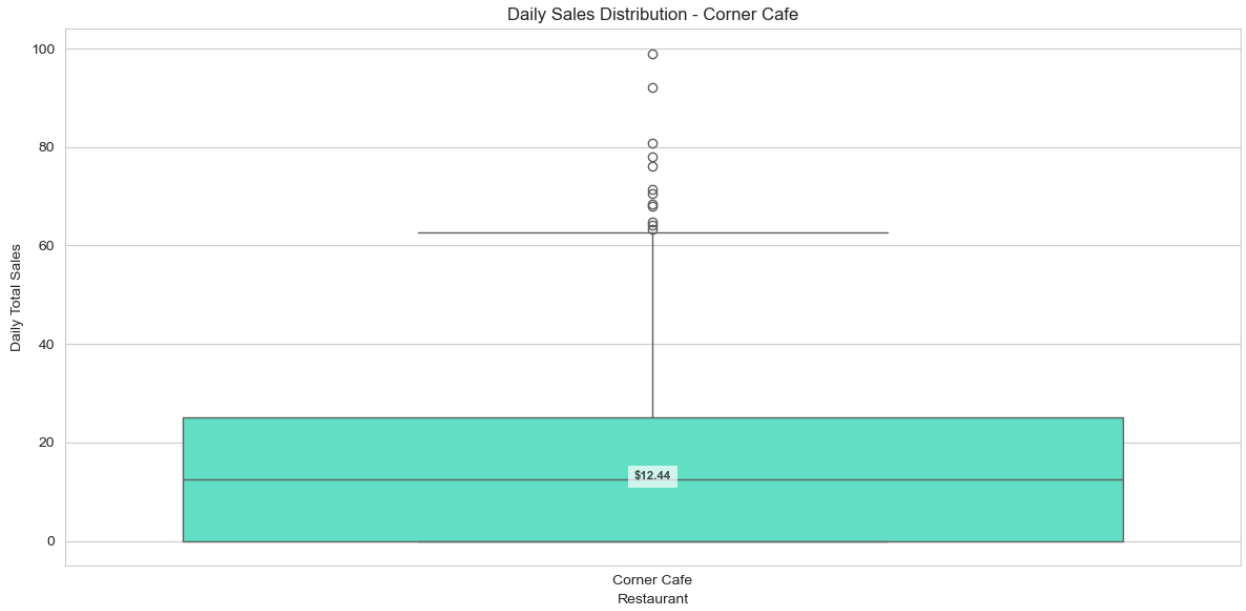

Average Daily Sales - Fou Cher
Average Sales
Day of Month
$1.01 $1.11 $0.96 $0.96 $0.80 $0.84 $0.66 $0.54 $0.93 $0.85 $0.74 $0.98 $1.20 $0.57 $1.10 $0.87 $0.98 $1.01 $0.93 $0.85 $0.79 $0.71 $0.61 $0.98 $1.19 $1.09 $1.17 $0.76 $0.87 $1.04 $1.20

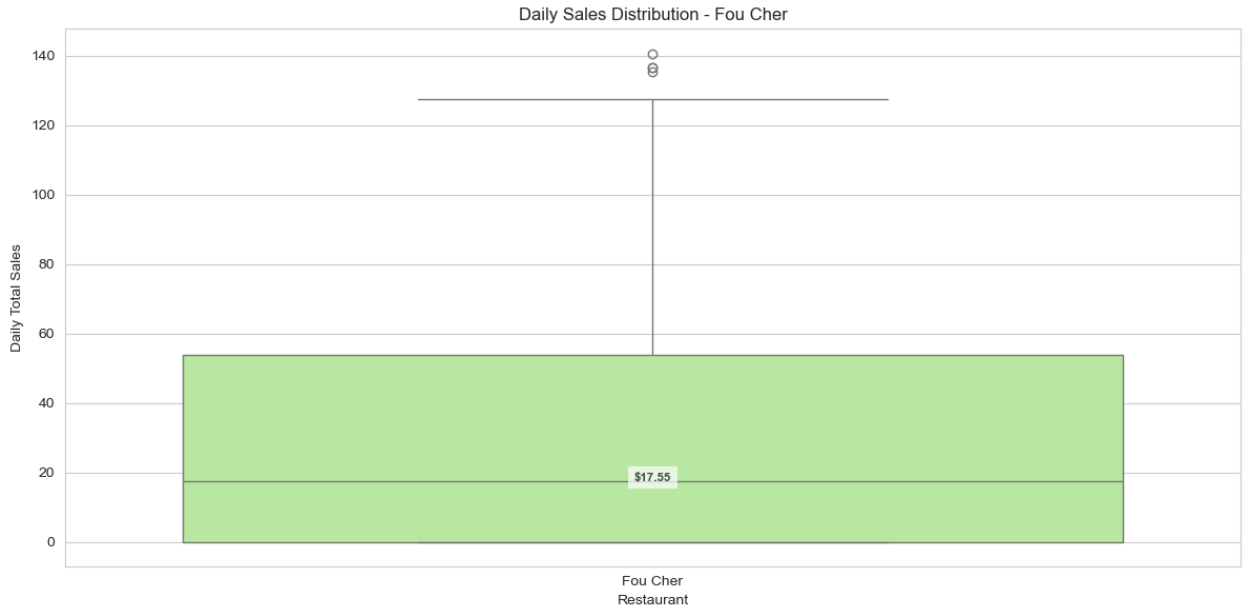

Average Daily Sales - Surfs Up
Average Sales
Day of Month
$1.48 $1.49 $1.80 $1.28 $1.32 $1.65 $1.15 $1.36 $1.61 $1.29 $1.50 $1.24 $1.31 $1.53 $1.78 $1.35 $0.98 $1.21 $1.87 $1.50 $1.66 $1.48 $1.40 $1.28 $1.65 $0.96 $1.39 $1.67 $1.38 $1.18 $1.55

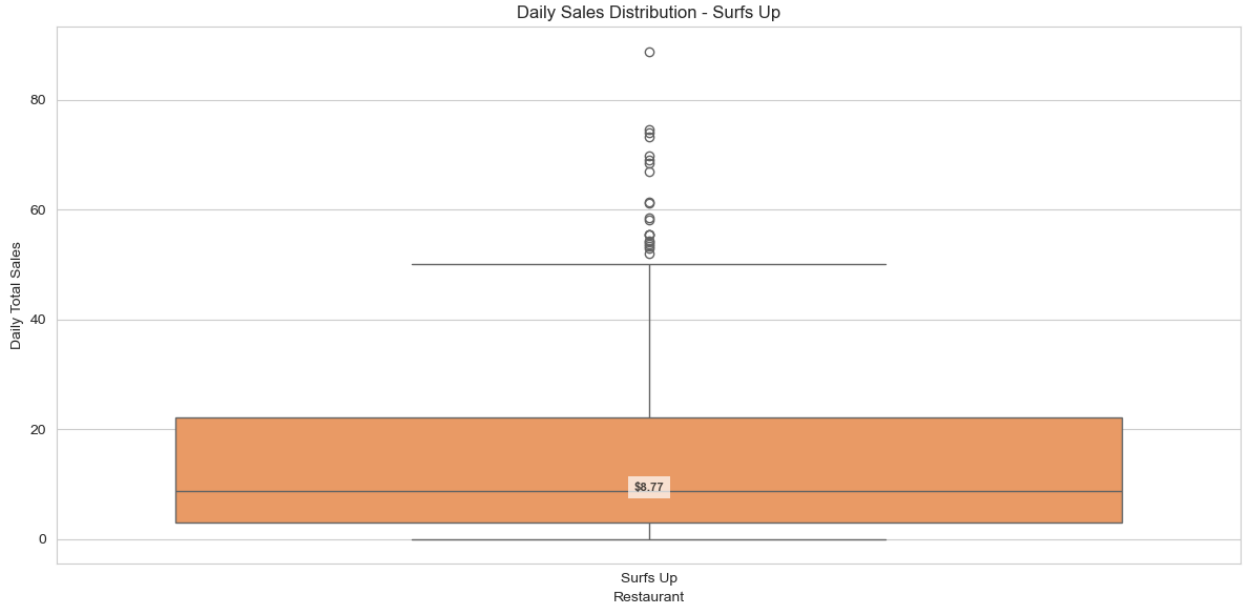

Average Daily Sales - Sweet Shack
Average Sales
Day of Month
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
$0.29 $0.24 $0.32 $0.37 $0.27 $0.34 $0.24 $0.29 $0.29 $0.26 $0.27 $0.28 $0.23 $0.29 $0.31 $0.36 $0.26 $0.40 $0.35 $0.28 $0.27 $0.34 $0.31 $0.33 $0.29 $0.28 $0.25 $0.27 $0.27 $0.26 $0.30

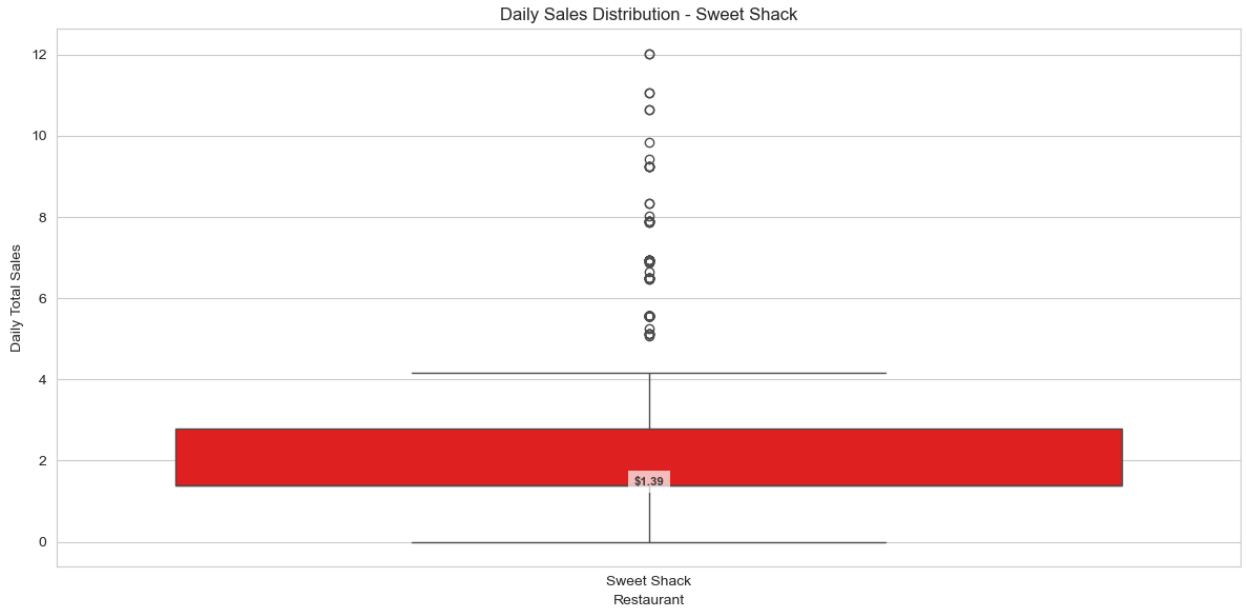

Daily Sales Distribution - Beachfront Bar
Daily Total Sales
Beachfront Bar
Restaurant
0
2
4
6
8
10
12
14
$2.91


Daily Sales Distribution - Bob's Diner
12000
10000
8000
6000
4000
2000
Daily Total Sales
$5,705.53
Bob's Diner
Restaurant


Daily Sales Distribution - Corner Cafe
100
80
60
40
20
0
Daily Total Sales
$12.44
Corner Cafe
Restaurant


Daily Sales Distribution - Fou Cher
Daily Total Sales
140
120
100
80
60
40
20
0
$17.55
Fou Cher
Restaurant


Daily Sales Distribution - Surfs Up
Daily Total Sales
80
60
40
20
0
$8.77
Surfs Up
Restaurant



Explanations:

Data Preparation

```
final_df['total_sales'] = final_df['price'] * final_df['item_count']
```

This line calculates the total sales for each transaction. It's crucial for EDA as it creates a key metric that will be used throughout the analysis.

Aggregation Functions

```
restaurant_daily_sales = final_df.groupby(['date',
'restaurant_name']).agg({
    'price': lambda x: (x * final_df.loc[x.index, 'item_count']).sum()
}).reset_index().rename(columns={'price': 'daily_sales'})
```

This function aggregates data to calculate daily sales for each restaurant. Aggregation is essential in EDA as it helps summarize large datasets into manageable and meaningful metrics.

Visualization Functions

```
def plot_individual_bars(data, x, y, title, x_label, y_label):
    # Function body...

def plot_individual_boxplots(data, x, y, title_prefix, x_label,
y_label, is_monthly=False):
    # Function body...
```

These functions create bar plots and box plots for different time frames. Visualization is a cornerstone of EDA, allowing analysts to quickly identify patterns, trends, and anomalies in the data.

Time-based Analysis The code includes analysis at various time scales:

- Yearly: `yearly_sales = final_df.groupby([final_df['date'].dt.year, 'restaurant_name'])['total_sales'].sum().reset_index()`
- Quarterly: `final_df['year_quarter'] = final_df['date'].dt.to_period('Q')`
- Monthly: `monthly_sales = final_df.groupby([final_df['date'].dt.month, 'restaurant_name'])['total_sales'].mean().reset_index()`
- Daily: `daily_sales = final_df.groupby([final_df['date'].dt.day, 'restaurant_name'])['total_sales'].mean().reset_index()`

This multi-scale temporal analysis is crucial in EDA for identifying seasonality, trends, and cyclical patterns in the data.

Comparative Analysis

```
bobs_data = restaurant_sales[restaurant_sales.index == "Bob's Diner"]
other_data = restaurant_sales[restaurant_sales.index != "Bob's Diner"]
```

This code separates one restaurant from the others for comparison. Comparative analysis in EDA helps in benchmarking and identifying outliers or unique performers.

Statistical Distribution The use of box plots (`plot_individual_boxplots`) provides insights into the statistical distribution of sales, including median, quartiles, and outliers. This is vital in EDA for understanding the spread and central tendency of the data.

Why It Is Important:

1. **Data Understanding**: This code helps in gaining a comprehensive understanding of the sales data across different time scales and restaurants.
2. **Pattern Identification**: The various plots enable the identification of patterns such as seasonality, weekly trends, and restaurant-specific performance.
3. **Anomaly Detection**: Box plots and comparative analyses help in identifying outliers and unusual patterns in the data.
4. **Hypothesis Generation**: The visualizations can lead to hypothesis formation about factors affecting sales, which can be further investigated.
5. **Feature Engineering Insights**: Understanding time-based patterns can inform feature engineering for predictive modeling.
6. **Stakeholder Communication**: Clear visualizations are essential for communicating findings to non-technical stakeholders.
7. **Data Quality Assessment**: The process of creating these visualizations can reveal data quality issues or missing data.

8. **Decision Support**: The insights gained from this EDA can directly inform business decisions, such as staffing, inventory management, and marketing strategies.

Detailed Analysis and Inferences:

1. **Overall Performance**:
   - Bobby's Grill's consistently taller bars (around $17,000-$18,000 daily average) indicate it likely has a prime location, excellent brand recognition, or a unique offering that attracts significantly more customers or allows for higher pricing compared to other restaurants.
   - Corner Cafe's shorter bars ($3,000-$3,500 daily average) suggest it might be a smaller establishment, possibly focusing on a niche market like breakfast and lunch crowds, or it could be struggling with location or menu appeal.
2. **Yearly Trends**:
   - The increase in bar heights from 2021 to 2022 across most restaurants suggests a general recovery or growth in the restaurant industry, possibly post-pandemic.
   - The stagnation or slight decline in 2023 for Corner Cafe and Tasty Bites, shown by similar or shorter bars, might indicate market saturation, increased competition, or these specific restaurants reaching their capacity limits.
3. **Seasonal Patterns**:
   - Taller bars in summer months (June-August) across all restaurants imply a strong influence of tourism or seasonal activities on dining out. This could be due to better weather, vacations, or local events driving more foot traffic.
   - Shorter bars in winter months (December-February) suggest reduced outdoor dining, fewer tourists, or changed consumer behavior (e.g., preference for home cooking in colder months).
4. **Day of Week Trends**:
   - Larger boxes and higher medians for Friday and Saturday across all restaurants indicate a strong weekend dining culture. This could be due to more leisure time, social dining habits, or targeted weekend promotions.
   - Smaller and lower boxes for Sunday and Monday suggest these are typically slower days, possibly due to many people preparing for the work week or recovering from weekend activities.
5. **Restaurant-Specific Observations**:
   - Bobby's Grill: Largest interquartile ranges and highest median values in box plots suggest not only high sales but also high variability, indicating potential for both very busy and slower periods. This could be due to its popularity leading to rush times and quieter off-peak hours.
   - Tasty Bites: Consistent median values across weekdays with a slight weekend increase suggest a stable customer base with a modest weekend boost. This could indicate a strong lunch business with some additional dinner traffic on weekends.
   - Corner Cafe: Smallest interquartile ranges and consistent medians across all days imply a very steady, predictable business. This could be a popular local spot with regular clientele, or it might have limited seating causing consistent but capped sales.

- Soup's Up: Larger whiskers and more outliers in its box plots indicate higher day-to-day variability. This could suggest more susceptibility to factors like weather (for a soup-focused restaurant) or irregular events driving occasionally high sales.
- Beach Shack: Pronounced peak in summer and significant dip in winter in monthly sales charts suggest heavy reliance on seasonal tourism or outdoor dining. This implies a need for strong financial management to handle off-season periods.
- Fire Grill: Consistent increase in bar heights year-over-year indicates successful growth strategies or increasing popularity. This suggests potential for expansion or for applying its successful practices to other restaurants in the group.

Conclusions:

1. **Market Leader:** Bobby's Grill is clearly the top performer and likely has the strongest brand or most favorable location.

2. **Seasonal Dependency:** All restaurants are affected by seasonal trends, with summer being the peak season.

3. **Weekend Boost:** The significantly higher sales on Fridays and Saturdays indicate a strong reliance on weekend dining across all restaurants.

4. **Growth Potential:** While most restaurants showed growth from 2021 to 2022, the slowing or declining growth in 2023 for some restaurants suggests potential market saturation or external factors affecting performance.

5. **Consistency vs. Variability:** Some restaurants (e.g., Corner Cafe) show more consistent sales across the week, while others (e.g., Soup's Up) have higher variability, suggesting different target markets or dining concepts.

Comparative Inferences:

1. **Market Positioning**: The significant gap between Bobby's Grill's performance and others suggests it might be in a different market segment, possibly a higher-end dining experience or a extremely popular casual dining spot.

2. **Operational Efficiency**: Corner Cafe's consistent but lower sales might indicate efficient operations scaled for smaller volume, while Bobby's Grill's high variability could suggest challenges in staffing and inventory management for peak times.

3. **Target Demographics**: The varying patterns in weekday vs. weekend performance across restaurants imply different target demographics. Tasty Bites might cater more to a working lunch crowd, while Beach Shack seems more focused on leisure and tourist dining.

4. **Scalability and Growth Potential**: Fire Grill's consistent growth suggests it might have the most scalable business model or room for expansion. In contrast, Corner Cafe's steady sales might indicate it has reached its optimal operational size for its current model.

5. **Risk and Seasonality**: Beach Shack's high seasonal variability indicates higher risk and more complex financial management needs compared to more consistent performers like Corner Cafe or Tasty Bites.

6. **Adaptability to Market Changes**: Soup's Up's high variability suggests it might be more adaptable to changing conditions (like introducing new menu items or responding to trends) but also more vulnerable to external factors.

These detailed inferences provide a nuanced understanding of each restaurant's performance, market position, and potential strategies for improvement or growth. They highlight the diverse challenges and opportunities within the restaurant group, suggesting areas for focused attention and potential cross-pollination of successful strategies.

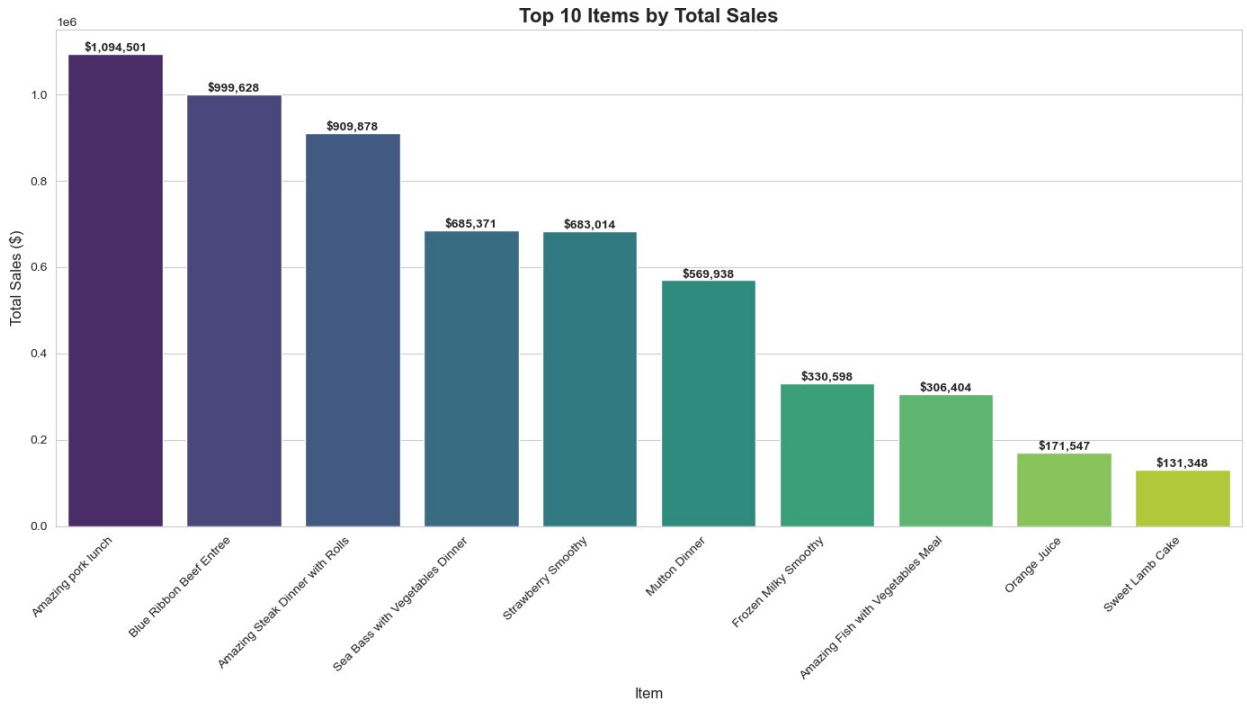
4.2.6. Identify the most popular items overall and the stores where they are being sold. Also, find out the most popular item at each store

```
# Calculate total sales and quantity sold for each item, including the
restaurant name
item_sales = final_df.groupby(['item_id', 'item_name',
'restaurant_name']).agg({
    'price': lambda x: (x * final_df.loc[x.index,
'item_count']).sum(),
    'item_count': 'sum'
}).reset_index().rename(columns={'price': 'total_sales', 'item_count':
'quantity_sold'})

# Sort items by total sales and get top 10
top_10_items_sales = item_sales.sort_values('total_sales',
ascending=False).head(10)

# 1. Plot top 10 items by total sales
plt.figure(figsize=(14, 8))
ax = sns.barplot(x='item_name', y='total_sales',
data=top_10_items_sales, palette='viridis')
plt.title('Top 10 Items by Total Sales', fontsize=16,
fontweight='bold')
plt.xlabel('Item', fontsize=12)
plt.ylabel('Total Sales ($)', fontsize=12)
plt.xticks(rotation=45, ha='right')

# Add value labels on the bars
for i, v in enumerate(top_10_items_sales['total_sales']):
    ax.text(i, v, f'${v:,.0f}', ha='center', va='bottom',
fontweight='bold')

plt.tight_layout()
plt.show()

# 2. Create a pretty table for top 10 items by total sales
table = PrettyTable()
```

```
table.field_names = ["Rank", "Item Name", "Restaurant", "Total Sales",
"Quantity Sold"]
for rank, (_, row) in enumerate(top_10_items_sales.iterrows(), 1):
    table.add_row([
        rank,
        row['item_name'],
        row['restaurant_name'],
        f"${row['total_sales']:,.2f}",
        f"{row['quantity_sold']:,}"
    ])

table.align["Item Name"] = "l"
table.align["Restaurant"] = "l"
table.align["Total Sales"] = "r"
table.align["Quantity Sold"] = "r"

print("\nTop 10 Items by Total Sales:")
print(table)

# New table: Top 10 Items by Total Sales (Excluding Bob's Diner)
top_10_items_sales_no_bob = item_sales[item_sales['restaurant_name'] !
= "Bob's Diner"].sort_values('total_sales', ascending=False).head(10)

table = PrettyTable()
table.field_names = ["Rank", "Item Name", "Restaurant", "Total Sales",
"Quantity Sold"]
for rank, (_, row) in enumerate(top_10_items_sales_no_bob.iterrows(),
1):
    table.add_row([
        rank,
        row['item_name'],
        row['restaurant_name'],
        f"${row['total_sales']:,.2f}",
        f"{row['quantity_sold']:,}"
    ])

table.align["Item Name"] = "l"
table.align["Restaurant"] = "l"
table.align["Total Sales"] = "r"
table.align["Quantity Sold"] = "r"

print("\nTop 10 Items by Total Sales (Excluding Bob's Diner):")
print(table)

# 3. Plot top 10 items by quantity sold
top_10_items_quantity = item_sales.sort_values('quantity_sold',
ascending=False).head(10)

plt.figure(figsize=(14, 8))
ax = sns.barplot(x='item_name', y='quantity_sold',
```

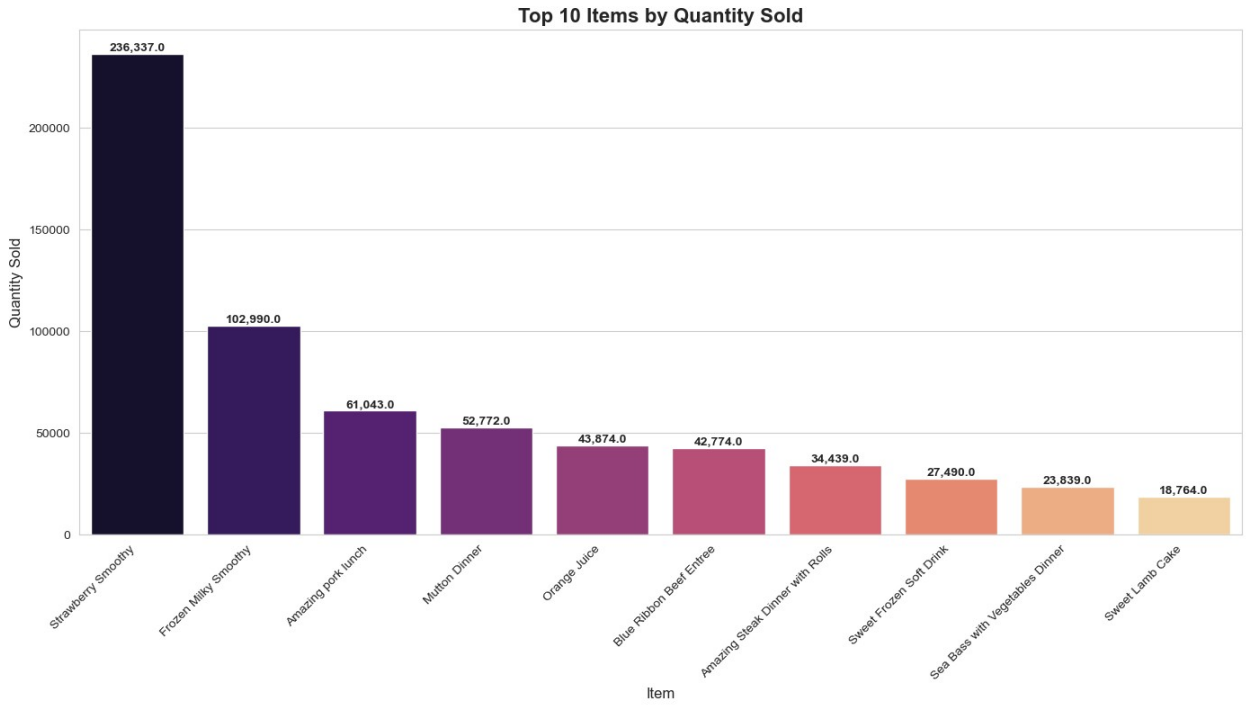
```
data=top_10_items_quantity, palette='magma')
plt.title('Top 10 Items by Quantity Sold', fontsize=16,
fontweight='bold')
plt.xlabel('Item', fontsize=12)
plt.ylabel('Quantity Sold', fontsize=12)
plt.xticks(rotation=45, ha='right')

# Add value labels on the bars
for i, v in enumerate(top_10_items_quantity['quantity_sold']):
    ax.text(i, v, f'{v:,}', ha='center', va='bottom',
fontweight='bold')

plt.tight_layout()
plt.show()

# 4. Create a pretty table for top 10 items by quantity sold
table = PrettyTable()
table.field_names = ["Rank", "Item Name", "Restaurant", "Quantity
Sold", "Total Sales"]
for rank, (_, row) in enumerate(top_10_items_quantity.iterrows(), 1):
    table.add_row([
        rank,
        row['item_name'],
        row['restaurant_name'],
        f"{row['quantity_sold']:,}",
        f"${row['total_sales']:,.2f}"
    ])

table.align["Item Name"] = "l"
table.align["Restaurant"] = "l"
table.align["Quantity Sold"] = "r"
table.align["Total Sales"] = "r"

print("\nTop 10 Items by Quantity Sold:")
print(table)

# New table: Top 10 Items by Quantity Sold (Excluding Bob's Diner)
top_10_items_quantity_no_bob =
item_sales[item_sales['restaurant_name'] != "Bob's
Diner"].sort_values('quantity_sold', ascending=False).head(10)

table = PrettyTable()
table.field_names = ["Rank", "Item Name", "Restaurant", "Quantity
Sold", "Total Sales"]
for rank, (_, row) in
enumerate(top_10_items_quantity_no_bob.iterrows(), 1):
    table.add_row([
        rank,
        row['item_name'],
        row['restaurant_name'],
```

```
        f"{row['quantity_sold']:,}",
        f"${row['total_sales']:,.2f}"
    ])

table.align["Item Name"] = "l"
table.align["Restaurant"] = "l"
table.align["Quantity Sold"] = "r"
table.align["Total Sales"] = "r"

print("\nTop 10 Items by Quantity Sold (Excluding Bob's Diner):")
print(table)

# 5. Identify most popular item at each store
store_popular_items = final_df.groupby(['store_id', 'restaurant_name',
'item_id', 'item_name']).agg({
    'item_count': 'sum',
    'price': lambda x: (x * final_df.loc[x.index, 'item_count']).sum()
}).reset_index()

store_popular_items =
store_popular_items.loc[store_popular_items.groupby('store_id')
['item_count'].idxmax()]

# Create a pretty table for most popular items at each store
table = PrettyTable()
table.field_names = ["Restaurant Name", "Most Popular Item", "Quantity
Sold", "Total Sales"]
for _, row in store_popular_items.iterrows():
    table.add_row([
        row['restaurant_name'],
        row['item_name'],
        f"{row['item_count']:,}",
        f"${row['price']:,.2f}"
    ])

table.align["Restaurant Name"] = "l"
table.align["Most Popular Item"] = "l"
table.align["Quantity Sold"] = "r"
table.align["Total Sales"] = "r"

print("\nMost popular item at each store:")
print(table)
```




```
Top 10 Items by Total Sales:
+------+-----------------------------------+-------------
+---------------+---------------+
| Rank | Item Name                         | Restaurant  |   Total
Sales | Quantity Sold |
+------+-----------------------------------+-------------
+---------------+---------------+
|  1   | Amazing pork lunch                | Bob's Diner |
$1,094,500.99 |      61,043.0 |
|  2   | Blue Ribbon Beef Entree           | Bob's Diner |
$999,628.38 |      42,774.0 |
|  3   | Amazing Steak Dinner with Rolls   | Bob's Diner |
$909,878.38 |      34,439.0 |
|  4   | Sea Bass with Vegetables Dinner   | Bob's Diner |
$685,371.25 |      23,839.0 |
|  5   | Strawberry Smoothy                | Bob's Diner |
$683,013.93 |     236,337.0 |
|  6   | Mutton Dinner                     | Bob's Diner |
$569,937.60 |      52,772.0 |
|  7   | Frozen Milky Smoothy              | Bob's Diner |
$330,597.90 |     102,990.0 |
|  8   | Amazing Fish with Vegetables Meal | Bob's Diner |
$306,403.70 |      13,190.0 |
|  9   | Orange Juice                      | Bob's Diner |
$171,547.34 |      43,874.0 |
|  10  | Sweet Lamb Cake                   | Bob's Diner |
$131,348.00 |      18,764.0 |
```

```
+------+-----------------------------------+-------------
+---------------+---------------+

Top 10 Items by Total Sales (Excluding Bob's Diner):
+------+-----------------------------------------------------
+----------------+-------------+---------------+
| Rank | Item Name                                          |
Restaurant     | Total Sales | Quantity Sold |
+------+-----------------------------------------------------
+----------------+-------------+---------------+
|  1   | Blue Ribbon Fruity Vegi Lunch                      | Fou
Cher       |  $16,086.04 |         298.0 |
|  2   | Original Fruity Cod with Bread and Vegetables Entree | Fou
Cher       |   $4,039.65 |         141.0 |
|  3   | Steak Meal                                         | Surfs
Up       |   $3,538.35 |         135.0 |
|  4   | Fantastic Milky Smoothy                            |
Beachfront Bar |   $3,337.77 |       1,147.0 |
|  5   | Awesome Soft Drink                                 | Surfs
Up       |   $3,050.82 |         997.0 |
|  6   | Roast Mutton Entree                                | Surfs
Up       |   $2,654.85 |         165.0 |
|  7   | Oysters Rockefeller                                | Surfs
Up       |   $2,521.42 |         157.0 |
|  8   | Awesome Smoothy                                    | Sweet
Shack    |   $2,351.88 |       1,692.0 |
|  9   | Awesome Hamburger with Fries                       | Corner
Cafe    |   $2,187.36 |          84.0 |
|  10  | Lamb with Bread Entree                             | Corner
Cafe    |   $2,139.68 |         172.0 |
+------+-----------------------------------------------------
+----------------+-------------+---------------+
```




```
Top 10 Items by Quantity Sold:
+------+---------------------------------+-------------
+---------------+---------------+
| Rank | Item Name                       | Restaurant  | Quantity Sold
|   Total Sales |
+------+---------------------------------+-------------
+---------------+---------------+
|  1   | Strawberry Smoothy              | Bob's Diner |     236,337.0
|   $683,013.93 |
|  2   | Frozen Milky Smoothy            | Bob's Diner |     102,990.0
|   $330,597.90 |
|  3   | Amazing pork lunch              | Bob's Diner |      61,043.0
| $1,094,500.99 |
|  4   | Mutton Dinner                   | Bob's Diner |      52,772.0
|   $569,937.60 |
|  5   | Orange Juice                    | Bob's Diner |      43,874.0
|   $171,547.34 |
|  6   | Blue Ribbon Beef Entree         | Bob's Diner |      42,774.0
|   $999,628.38 |
|  7   | Amazing Steak Dinner with Rolls | Bob's Diner |      34,439.0
|   $909,878.38 |
|  8   | Sweet Frozen Soft Drink         | Bob's Diner |      27,490.0
|   $114,908.20 |
|  9   | Sea Bass with Vegetables Dinner | Bob's Diner |      23,839.0
|   $685,371.25 |
|  10  | Sweet Lamb Cake                 | Bob's Diner |      18,764.0
|   $131,348.00 |
```

```
+------+---------------------------------+-------------
+--------------+--------------+

Top 10 Items by Quantity Sold (Excluding Bob's Diner):
+------+-------------------------------------------------
+----------------+--------------+-------------+
| Rank | Item Name                                       |
Restaurant     | Quantity Sold | Total Sales |
+------+-------------------------------------------------
+----------------+--------------+-------------+
|  1   | Awesome Smoothy                                 | Sweet
Shack    |       1,692.0 |   $2,351.88 |
|  2   | Fantastic Milky Smoothy                         |
Beachfront Bar |       1,147.0 |   $3,337.77 |
|  3   | Awesome Soft Drink                              | Surfs
Up       |         997.0 |   $3,050.82 |
|  4   | Blue Ribbon Fruity Vegi Lunch                   | Fou
Cher       |         298.0 |  $16,086.04 |
|  5   | Frozen Milky Smoothy                            | Corner
Cafe     |         273.0 |   $1,086.54 |
|  6   | Lamb with Bread Entree                          | Corner
Cafe     |         172.0 |   $2,139.68 |
|  7   | Roast Mutton Entree                             | Surfs
Up       |         165.0 |   $2,654.85 |
|  8   | Blue Ribbon Lamb with Rolls Lunch               | Corner
Cafe     |         161.0 |   $1,215.55 |
|  9   | Oysters Rockefeller                             | Surfs
Up       |         157.0 |   $2,521.42 |
|  10  | Original Fruity Cod with Bread and Vegetables Entree | Fou
Cher       |         141.0 |   $4,039.65 |
+------+-------------------------------------------------
+----------------+--------------+-------------+

Most popular item at each store:
+-----------------+-------------------------------+---------------
+-------------+
| Restaurant Name | Most Popular Item             | Quantity Sold |
Total Sales |
+-----------------+-------------------------------+---------------
+-------------+
| Bob's Diner     | Strawberry Smoothy            |     236,337.0 |
$683,013.93 |
| Beachfront Bar  | Fantastic Milky Smoothy       |       1,147.0 |
$3,337.77 |
| Sweet Shack     | Awesome Smoothy               |       1,692.0 |
$2,351.88 |
| Fou Cher        | Blue Ribbon Fruity Vegi Lunch |         298.0 |
$16,086.04 |
| Corner Cafe     | Frozen Milky Smoothy          |         273.0 |
```

```
$1,086.54 |
| Surfs Up        | Awesome Soft Drink             |         997.0 |
$3,050.82 |
+-----------------+--------------------------------+---------------
+-------------+
```

Observations:

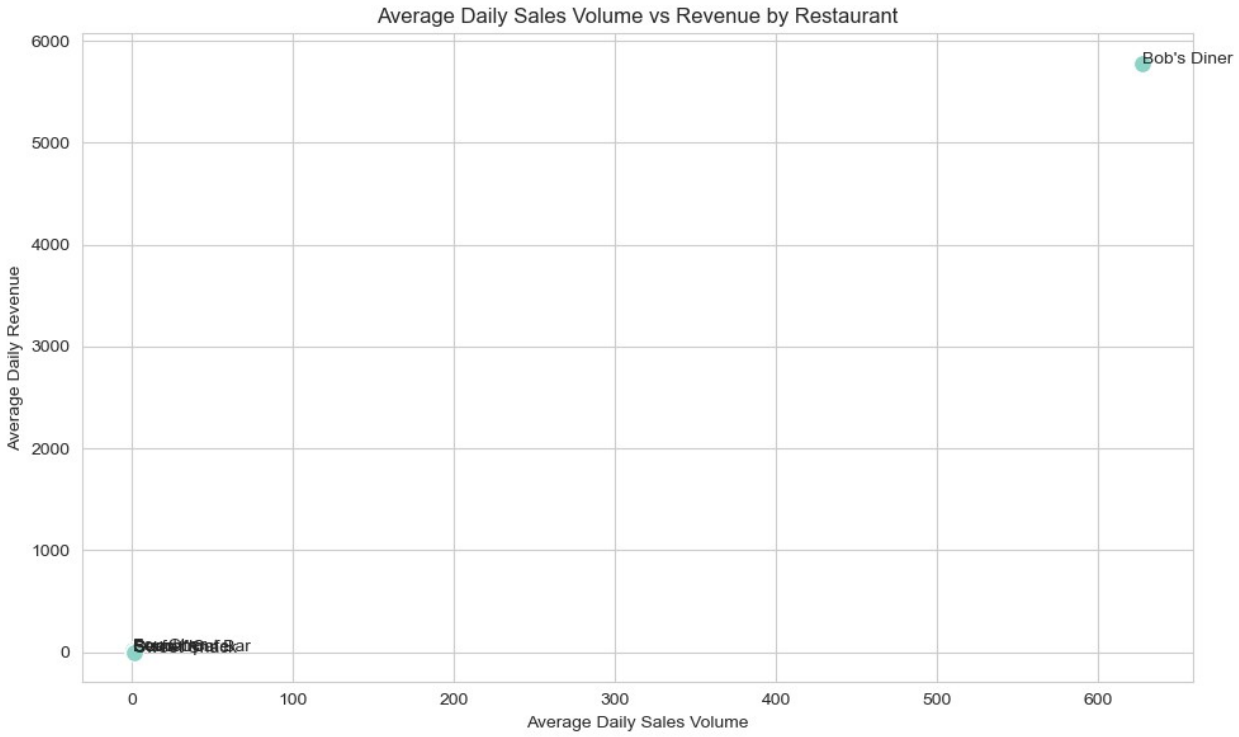
1. **Bob's Diner Dominance:**
   - Bob's Diner significantly outperforms other restaurants in both total sales and quantity sold.
   - All top 10 items by total sales and quantity sold are from Bob's Diner when including all restaurants.
2. **Menu Item Performance:**
   - "Amazing pork lunch" is the highest-grossing item overall, generating $1,094,501 in sales.
   - "Strawberry Smoothy" is the best-selling item by quantity, with 236,337 units sold.
   - Smoothies and lunch items appear to be particularly popular at Bob's Diner.
3. **Price Point Differences:**
   - Some items have high sales despite lower quantities (e.g., "Blue Ribbon Beef Entree"), suggesting higher price points.
   - Others have high quantities but lower total sales (e.g., "Orange Juice"), indicating lower price points.
4. **Non-Bob's Diner Performance:**
   - When excluding Bob's Diner, "Blue Ribbon Fruity Vegi Lunch" from Fou Cher is the top-selling item by revenue ($16,086.04).
   - "Awesome Smoothy" from Sweet Shack leads in quantity sold (1,692 units) when Bob's Diner is excluded.
5. **Restaurant Specialties:**
   - Different restaurants seem to have signature items:
     - Sweet Shack and Beachfront Bar excel in smoothies
     - Fou Cher's top item is a vegetarian option
     - Surfs Up has multiple popular seafood items (e.g., "Oysters Rockefeller")
6. **Menu Diversity:**
   - The top-selling items represent a mix of main courses, drinks, and desserts, suggesting diverse menu offerings.
7. **Price Range Inference:**
   - Based on the sales and quantity data, we can infer that Bob's Diner likely offers items at various price points, from affordable high-volume items to pricier entrees.
8. **Customer Preferences:**
   - Across different restaurants, smoothies and lunch items appear frequently in the top-selling lists, indicating a general customer preference for these types of items.

9. **Marketing Opportunities:**
   - The data suggests opportunities for cross-promotion or menu optimization at lower-performing restaurants, potentially by introducing popular item types from Bob's Diner.
10. **Operational Insights:**
   - Bob's Diner's success might indicate superior location, marketing, or operational efficiency, which could be studied and potentially applied to other restaurants in the group.

4.2.7. Determine if the store with the highest sales volume is also making the most money per day

```
# Calculate daily sales volume and revenue for each store
daily_store_performance = final_df.groupby(['date', 'store_id',
'restaurant_name']).agg({
    'item_count': 'sum',
    'price': lambda x: (x * final_df.loc[x.index, 'item_count']).sum()
}).reset_index().rename(columns={'item_count': 'daily_volume',
'price': 'daily_revenue'})

# Calculate average daily volume and revenue
avg_store_performance =
daily_store_performance.groupby('restaurant_name').agg({
    'daily_volume': 'mean',
    'daily_revenue': 'mean'
}).reset_index()

# Plot relationship between average daily volume and revenue
plt.figure(figsize=(10, 6))
sns.scatterplot(x='daily_volume', y='daily_revenue',
data=avg_store_performance, s=100)

for i, row in avg_store_performance.iterrows():
    plt.annotate(row['restaurant_name'], (row['daily_volume'],
row['daily_revenue']))

plt.title('Average Daily Sales Volume vs Revenue by Restaurant')
plt.xlabel('Average Daily Sales Volume')
plt.ylabel('Average Daily Revenue')
plt.tight_layout()
plt.show()

# Calculate correlation between daily volume and revenue
correlation =
daily_store_performance['daily_volume'].corr(daily_store_performance['
daily_revenue'])
print(f"Correlation between daily sales volume and revenue:
{correlation:.2f}")

# Identify store with highest sales volume and its revenue ranking
```

```
highest_volume_store =
avg_store_performance.loc[avg_store_performance['daily_volume'].idxmax
()]
revenue_rank =
avg_store_performance['daily_revenue'].rank(ascending=False)
highest_volume_store_rank = revenue_rank[highest_volume_store.name]

print(f"Store with highest sales volume:
{highest_volume_store['restaurant_name']}")
print(f"Its revenue rank: {highest_volume_store_rank} out of
{len(avg_store_performance)}")
```



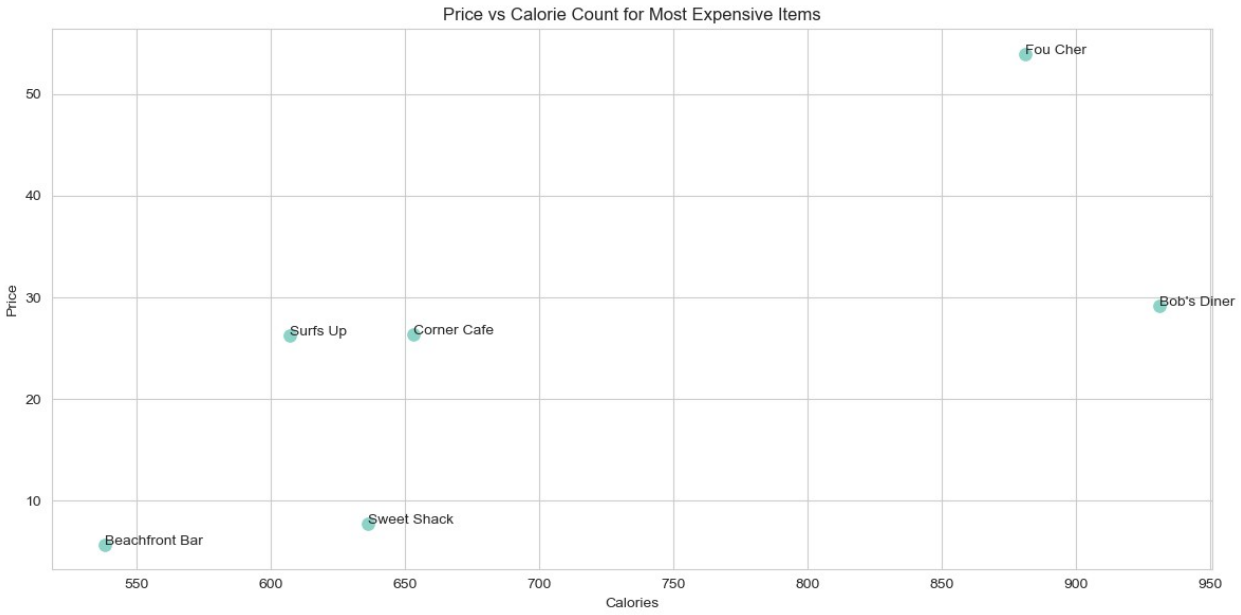

```
Correlation between daily sales volume and revenue: 1.00
Store with highest sales volume: Bob's Diner
Its revenue rank: 1.0 out of 6
```

4.2.8. Identify the most expensive item at each restaurant and find out its calorie count

```
# Get the most expensive item at each restaurant
most_expensive_items = final_df.groupby(['store_id',
'restaurant_name', 'item_id', 'item_name', 'price',
'kcal']).size().reset_index(name='count')
most_expensive_items =
most_expensive_items.loc[most_expensive_items.groupby('store_id')
['price'].idxmax()]
```

```python
# Sort by price in descending order
most_expensive_items = most_expensive_items.sort_values('price',
ascending=False)

# Display the most expensive items and their calorie counts
print("Most expensive item at each restaurant and its calorie count:")
print(most_expensive_items[['restaurant_name', 'item_name', 'price',
'kcal']])

# Visualize price vs calorie count for the most expensive items
plt.figure(figsize=(12, 6))
sns.scatterplot(x='kcal', y='price', data=most_expensive_items, s=100)

for i, row in most_expensive_items.iterrows():
    plt.annotate(row['restaurant_name'], (row['kcal'], row['price']))

plt.title('Price vs Calorie Count for Most Expensive Items')
plt.xlabel('Calories')
plt.ylabel('Price')
plt.tight_layout()
plt.show()

# Calculate correlation between price and calorie count for all items
price_calorie_correlation = final_df[['price',
'kcal']].drop_duplicates().corr().iloc[0, 1]
print(f"Correlation between price and calorie count for all items:
{price_calorie_correlation:.2f}")
```

```
Most expensive item at each restaurant and its calorie count:
   restaurant_name                      item_name  price  kcal
59        Fou Cher  Blue Ribbon Fruity Vegi Lunch  53.98   881
0      Bob's Diner              Sweet Fruity Cake  29.22   931
74     Corner Cafe                     Pike Lunch  26.37   653
92        Surfs Up                     Steak Meal  26.21   607
35     Sweet Shack  Blue Ribbon Frozen Milky Cake   7.70   636
26  Beachfront Bar          Sweet Vegi Soft Drink   5.70   538
```




```
Correlation between price and calorie count for all items: 0.42
```

4.3. Forecasting using machine learning algorithms

4.3.1. Build and compare linear regression, random forest, and XGBoost models for predictions

4.3.1.1. Generate necessary features for the development of these models, like day of the week, quarter of the year, month, year, day of the month and so on

```
print("final_df DataFrame Information:")
print(final_df.info())
print()
print("final_df DataFrame Description:")
print(final_df.describe().transpose())
print()

# Count the number of entries for each restaurant_name
restaurant_counts = final_df['restaurant_name'].value_counts()

# Display the counts
print(f"Restaurant Sales Transaction Counts From Combined Dataset: \
n")
print(restaurant_counts)
print()

restaurant_items = final_df.groupby('restaurant_name')
['item_name'].apply(set)

# Find the common item_names across all restaurants
common_items = set.intersection(*restaurant_items)
```

```
# Check if there are common items and print the appropriate message
if common_items:
    print("Common item_names between restaurants:")
    print(sorted(common_items))
else:
    print(f"No shared items between restaurants.\n")

# Group the DataFrame by restaurant_name and get unique item_names for
each restaurant
restaurant_items = final_df.groupby('restaurant_name')
['item_name'].apply(set)

# Print the unique item_names for each restaurant in alphabetical
order
for restaurant, items in restaurant_items.items():
    sorted_items = sorted(items)
    print(f"Restaurant: {restaurant}")
    print(f"Unique Items: {sorted_items}")
    print()

# Group the DataFrame by restaurant_name and item_name, and aggregate
the prices
restaurant_item_prices = final_df.groupby(['restaurant_name',
'item_name'])['price'].mean().reset_index()

# Iterate over each restaurant and print the table of unique items and
their prices
for restaurant, group in
restaurant_item_prices.groupby('restaurant_name'):
    print(f"Restaurant: {restaurant}")
    print(group[['item_name', 'price']].to_string(index=False))
    print()

# Group the DataFrame by item_name and calculate the standard
deviation of prices for each item
price_variability = final_df.groupby('item_name')
['price'].std().reset_index()

# Filter items with non-zero standard deviation
variable_price_items = price_variability[price_variability['price'] >
0]

# Check if there are any items with price variability and print the
appropriate message
if not variable_price_items.empty:
    print("Items with price variability:")
    for _, row in variable_price_items.iterrows():
        item_name = row['item_name']
        std_dev = row['price']
```

```
        restaurants_selling_item = final_df[final_df['item_name'] ==
item_name]['restaurant_name'].unique()
        print(f"Item: {item_name}, Std Dev: {std_dev:.2f},
Restaurants: {', '.join(restaurants_selling_item)}")
else:
    print("No price variability found for any items.")

print()
print()

final_df DataFrame Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 109600 entries, 0 to 109599
Data columns (total 12 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   date             109600 non-null  datetime64[ns]
 1   item_id          109600 non-null  int64
 2   price            109600 non-null  float64
 3   item_count       109600 non-null  float64
 4   item_name        109600 non-null  object
 5   kcal             109600 non-null  int64
 6   store_id         109600 non-null  int64
 7   restaurant_name  109600 non-null  object
 8   total_sales      109600 non-null  float64
 9   year_quarter     109600 non-null  period[Q-DEC]
 10  month            109600 non-null  int32
 11  day_name         109600 non-null  object
dtypes: datetime64[ns](1), float64(3), int32(1), int64(3), object(3),
period[Q-DEC](1)
memory usage: 9.6+ MB
None

final_df DataFrame Description:
                count                  mean                  min  \
date           109600  2020-07-01 12:00:00  2019-01-01 00:00:00
item_id      109600.0                 50.5                  1.0
price        109600.0              11.7637                 1.39
item_count   109600.0             6.339297                  0.0
kcal         109600.0               536.73                 78.0
store_id     109600.0                 3.52                  1.0
total_sales  109600.0             58.428271                  0.0
month        109600.0              6.521898                  1.0

                              25%                   50%
75%  \
date         2019-10-01 18:00:00  2020-07-01 12:00:00  2021-04-01
06:00:00
item_id                    25.75                 50.5
75.25
```

```
price                          5.28                7.625
18.79
item_count                      0.0                  0.0
0.0
kcal                         406.25                572.5
638.25
store_id                        2.0                  4.0
5.0
total_sales                     0.0                  0.0
0.0
month                           4.0                  7.0
10.0

                                max         std
date         2021-12-31 00:00:00         NaN
item_id                       100.0   28.866202
price                          53.98    8.946225
item_count                     570.0   30.003728
kcal                          1023.0  201.200165
store_id                         6.0     1.69989
total_sales                   2224.8    214.8122
month                           12.0    3.449002

Restaurant Sales Transaction Counts From Combined Dataset:

restaurant_name
Fou Cher          30688
Bob's Diner       26304
Corner Cafe       26304
Surfs Up          10960
Sweet Shack        8768
Beachfront Bar     6576
Name: count, dtype: int64

No shared items between restaurants.

Restaurant: Beachfront Bar
Unique Items: ['Awesome Vodka Cocktail', 'Fantastic Milky Smoothy',
'Original Crazy Cocktail', 'Original Gin Cocktail', 'Original Sweet
Milky Soft Drink', 'Sweet Vegi Soft Drink']

Restaurant: Bob's Diner
Unique Items: ['Amazing Fish with Vegetables Meal', 'Amazing Frozen
Milky Cake', 'Amazing Steak Dinner with Rolls', 'Amazing Sweet Fruity
Vegetable Cake', 'Amazing pork lunch', 'Awesome Fish with Vegetables
Entree', 'Awesome Milky Cake', 'Awesome Sweet Lamb Cake', 'Blue Ribbon
Beef Entree', 'Blue Ribbon Fruity Milky Cake', 'Fantastic Frozen Milky
Cake', 'Fantastic Sweet Cola', 'Frozen Milky Smoothy', 'Milky Cake',
'Milky Vegi Smoothy', 'Mutton Dinner', 'Mutton Lunch Plate', 'Orange
Juice', 'Sea Bass with Vegetables Dinner', 'Strawberry Smoothy',
```

```
'Sweet Breaded Zucchini Cake', 'Sweet Frozen Soft Drink', 'Sweet
Fruity Cake', 'Sweet Lamb Cake']

Restaurant: Corner Cafe
Unique Items: ['Amazing Sweet Breaded Carrot Cake', 'Anglefood Cake',
'Awesome Fruity Lamb with Vegetables Dinner', 'Awesome Hamburger with
Fries', 'BBQ Pork Steak', 'Blue Ribbon Fish with Bread Lunch', 'Blue
Ribbon Lamb with Rolls Lunch', 'Fantastic Fish with Vegetables
Entree', 'Frozen Chocolate Cake', 'Frozen Milky Cake', 'Frozen Milky
Smoothy', 'Fruity Milky Soft Drink', 'Halibut with Bread Dinner',
'Lamb with Bread Entree', 'Lamb with Vegetables Meal', 'Milk Cake',
'Milky Cake', 'Mutton with Roles and Vegetables Plate', 'Original
Fruity Carrot Cake', 'Original Lamb with Vegetables Meal', 'Original
Pork Meal', 'Original Vegetable with Bread Meal', 'Pike Lunch', 'Pork
with Bread Lunch']

Restaurant: Fou Cher
Unique Items: ['Amazing Lamb Dinner', 'Amazing Vegetable and Bread
Dinner', 'Awesome Vegetable with Bread Meal', 'Blue Ribbon Fruity Vegi
Lunch', 'Blue Ribbon Milky Smoothy', 'Blue Ribbon Pork Chop with Rolls
Dinner', 'Breaded Fish with Vegetables Meal', 'Carrot Cake', 'Cherry
Cake', 'Chocolate Cake', 'Fantastic Fruity Salmon with Bread meal',
'Fantastic Sweet Fish Cake', 'Fish with Dread and Vegetables Dinner',
'Frozen Tomato Soft Drink', 'Fruity Frozen Cocktail', 'Fruity Frozen
Slushy', 'Fruity Milky Cake', 'Lamb with Bread Dinner', 'Lamb with
Bread and Vegetables Meal', 'Margarita', 'Martini', 'Milky Cake',
'Original Breaded Lamb Dinner', 'Original Fruity Cod with Bread and
Vegetables Entree', 'Original Milky Cake', 'Super Sweet Cocktail',
'Sweet Savory Cake', 'Vegetarian Plate with Bread entree']

Restaurant: Surfs Up
Unique Items: ['Amazing Cocktail', 'Amazing Trout with Vegetables
Dinner', 'Awesome Pork with Vegetables Lunch', 'Awesome Soft Drink',
'Blue Ribbon Cocktail', 'Fruity Milky Smoothy', 'Original Breaded Pork
with Vegetables Dinner', 'Oysters Rockefeller', 'Roast Mutton Entree',
'Steak Meal']

Restaurant: Sweet Shack
Unique Items: ['Awesome Smoothy', 'Blue Ribbon Frozen Milky Cake',
'Blue Ribbon Milky Cake', 'Fantastic Cake', 'Fantastic Milky Smoothy',
'Milky vegi Smoothy', 'Original Milky Cake', 'Original Sweet Milky
Soft Drink']

Restaurant: Beachfront Bar
                      item_name  price
         Awesome Vodka Cocktail   2.48
        Fantastic Milky Smoothy   2.91
        Original Crazy Cocktail   2.43
          Original Gin Cocktail   2.99
Original Sweet Milky Soft Drink   5.00
```

```
          Sweet Vegi Soft Drink   5.70
```

Restaurant: Bob's Diner

item_name	price
Amazing Fish with Vegetables Meal	23.23
Amazing Frozen Milky Cake	5.62
Amazing Steak Dinner with Rolls	26.42
Amazing Sweet Fruity Vegetable Cake	7.91
Amazing pork lunch	17.93
Awesome Fish with Vegetables Entree	23.43
Awesome Milky Cake	7.36
Awesome Sweet Lamb Cake	10.86
Blue Ribbon Beef Entree	23.37
Blue Ribbon Fruity Milky Cake	8.70
Fantastic Frozen Milky Cake	6.23
Fantastic Sweet Cola	4.87
Frozen Milky Smoothy	3.21
Milky Cake	5.16
Milky Vegi Smoothy	5.50
Mutton Dinner	10.80
Mutton Lunch Plate	18.82
Orange Juice	3.91
Sea Bass with Vegetables Dinner	28.75
Strawberry Smoothy	2.89
Sweet Breaded Zucchini Cake	7.71
Sweet Frozen Soft Drink	4.18
Sweet Fruity Cake	29.22
Sweet Lamb Cake	7.00

Restaurant: Corner Cafe

item_name	price
Amazing Sweet Breaded Carrot Cake	7.87
Anglefood Cake	7.13
Awesome Fruity Lamb with Vegetables Dinner	19.03
Awesome Hamburger with Fries	26.04
BBQ Pork Steak	19.77
Blue Ribbon Fish with Bread Lunch	21.93
Blue Ribbon Lamb with Rolls Lunch	7.55
Fantastic Fish with Vegetables Entree	21.14
Frozen Chocolate Cake	3.74
Frozen Milky Cake	6.63
Frozen Milky Smoothy	3.98
Fruity Milky Soft Drink	7.95
Halibut with Bread Dinner	15.46
Lamb with Bread Entree	12.44
Lamb with Vegetables Meal	13.66
Milk Cake	6.07
Milky Cake	7.22
Mutton with Roles and Vegetables Plate	21.13

```
                Original Fruity Carrot Cake   8.55
        Original Lamb with Vegetables Meal  18.78
                         Original Pork Meal  15.69
        Original Vegetable with Bread Meal   6.51
                                 Pike Lunch  26.37
                      Pork with Bread Lunch  16.28
```

Restaurant: Fou Cher

```
                                            item_name  price
                                   Amazing Lamb Dinner  17.55
                     Amazing Vegetable and Bread Dinner   8.01
                      Awesome Vegetable with Bread Meal   6.21
                          Blue Ribbon Fruity Vegi Lunch  53.98
                              Blue Ribbon Milky Smoothy   5.60
              Blue Ribbon Pork Chop with Rolls Dinner  19.04
                    Breaded Fish with Vegetables Meal  15.09
                                         Carrot Cake   3.93
                                         Cherry Cake   6.31
                                       Chocolate Cake   6.71
              Fantastic Fruity Salmon with Bread meal  22.67
                                Fantastic Sweet Fish Cake  19.48
                Fish with Dread and Vegetables Dinner  15.23
                                Frozen Tomato Soft Drink   5.32
                                  Fruity Frozen Cocktail   5.61
                                    Fruity Frozen Slushy   3.75
                                       Fruity Milky Cake   8.10
                                  Lamb with Bread Dinner  19.05
                    Lamb with Bread and Vegetables Meal  20.02
                                             Margarita   3.23
                                               Martini   5.45
                                            Milky Cake   6.01
                           Original Breaded Lamb Dinner  11.82
Original Fruity Cod with Bread and Vegetables Entree  28.65
                                   Original Milky Cake   6.53
                                  Super Sweet Cocktail   4.11
                                     Sweet Savory Cake  27.47
                     Vegetarian Plate with Bread entree   4.02
```

Restaurant: Surfs Up

```
                                  item_name  price
                           Amazing Cocktail   3.90
          Amazing Trout with Vegetables Dinner  24.98
            Awesome Pork with Vegetables Lunch  18.53
                             Awesome Soft Drink   3.06
                           Blue Ribbon Cocktail   4.36
                           Fruity Milky Smoothy   5.71
Original Breaded Pork with Vegetables Dinner  20.80
                            Oysters Rockefeller  16.06
                            Roast Mutton Entree  16.09
```

```
                                    Steak Meal  26.21

Restaurant: Sweet Shack
                      item_name  price
                 Awesome Smoothy   1.39
  Blue Ribbon Frozen Milky Cake   7.70
         Blue Ribbon Milky Cake   6.89
                  Fantastic Cake   5.08
        Fantastic Milky Smoothy   5.11
             Milky vegi Smoothy   3.86
            Original Milky Cake   6.50
Original Sweet Milky Soft Drink   5.68

Items with price variability:
Item: Fantastic Milky Smoothy, Std Dev: 1.10, Restaurants: Beachfront
Bar, Sweet Shack
Item: Frozen Milky Smoothy, Std Dev: 0.39, Restaurants: Bob's Diner,
Corner Cafe
Item: Milky Cake, Std Dev: 0.85, Restaurants: Bob's Diner, Fou Cher,
Corner Cafe
Item: Original Milky Cake, Std Dev: 0.02, Restaurants: Fou Cher, Sweet
Shack
Item: Original Sweet Milky Soft Drink, Std Dev: 0.34, Restaurants:
Sweet Shack, Beachfront Bar
```

```python
# Export final_df to a CSV file
# final_df.to_csv(f'{DATASET_PATH}/final_df_export.csv', index=False)

# Extract additional date features
final_df['day_of_week'] = final_df['date'].dt.dayofweek
final_df['week_of_year'] = final_df['date'].dt.isocalendar().week
final_df['quarter'] = final_df['date'].dt.quarter
final_df['month'] = final_df['date'].dt.month
final_df['year'] = final_df['date'].dt.year
final_df['day_of_month'] = final_df['date'].dt.day

# Handling outliers by capping them
def cap_outliers(df, column):
    upper_limit = df[column].quantile(0.95)
    lower_limit = df[column].quantile(0.05)
    df[column] = np.clip(df[column], lower_limit, upper_limit)
    return df

for column in ['price', 'item_count', 'total_sales', 'kcal']:
    final_df = cap_outliers(final_df, column)

# Define numerical and categorical columns
num_features = ['price', 'item_count', 'kcal', 'total_sales']
```

```
date_features = ['day_of_week', 'week_of_year', 'quarter', 'month',
'year', 'day_of_month']
cat_features = ['restaurant_name', 'item_name', 'day_name'] +
date_features

# Custom transformer for log transformation
log_transformer = FunctionTransformer(np.log1p, validate=True)

# Preprocessing pipelines
num_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('log', log_transformer),
    ('scaler', StandardScaler())
])

cat_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore',
sparse_output=False))
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', num_transformer, num_features),
        ('cat', cat_transformer, cat_features)
    ])

# Apply the transformations
df_preprocessed = preprocessor.fit_transform(final_df)

# Check the shape of the preprocessed data
print(f"Shape of preprocessed data: {df_preprocessed.shape}")

# Generate column names
num_columns = num_features
cat_columns = preprocessor.named_transformers_['cat']
['onehot'].get_feature_names_out(cat_features)
all_columns = num_columns + list(cat_columns)

# Check the length of the columns
print(f"Number of columns: {len(all_columns)}")

# Ensure the shape matches the number of columns
if df_preprocessed.shape[1] == len(all_columns):
    df_preprocessed = pd.DataFrame(df_preprocessed,
columns=all_columns)
else:
    print("Mismatch between the number of columns in preprocessed data
and column names")
```

```python
# Add the date column back to the preprocessed DataFrame
df_preprocessed['date'] = final_df['date']

# Display the preprocessed DataFrame if there is no mismatch
if df_preprocessed.shape[1] == len(all_columns) + 1:
    print(df_preprocessed.head())
else:
    print("Preprocessed data and column names do not match. Please
check the transformations.")

# Export final_df to a CSV file
# df_preprocessed.to_csv(f'{DATASET_PATH}/df_preprocessed_export.csv',
index=False)
```

```
Shape of preprocessed data: (109600, 221)
Number of columns: 221
      price  item_count      kcal  total_sales  \
0  1.568067    0.680601  1.285870     1.742557
1  1.565042    2.754818  1.030600     2.857764
2 -0.872320    1.679409 -0.094739     1.457004
3 -1.070055    2.173816 -0.035097     1.661283
4 -1.397914    3.122449 -1.346248     2.824112

   restaurant_name_Beachfront Bar  restaurant_name_Bob's Diner  \
0                             0.0                          1.0
1                             0.0                          1.0
2                             0.0                          1.0
3                             0.0                          1.0
4                             0.0                          1.0

   restaurant_name_Corner Cafe  restaurant_name_Fou Cher  \
0                          0.0                       0.0
1                          0.0                       0.0
2                          0.0                       0.0
3                          0.0                       0.0
4                          0.0                       0.0

   restaurant_name_Surfs Up  restaurant_name_Sweet Shack  ...  \
0                       0.0                          0.0  ...
1                       0.0                          0.0  ...
2                       0.0                          0.0  ...
3                       0.0                          0.0  ...
4                       0.0                          0.0  ...

   day_of_month_23  day_of_month_24  day_of_month_25  day_of_month_26
\
0              0.0              0.0              0.0              0.0

1              0.0              0.0              0.0              0.0
```

```
2                0.0                0.0                0.0                0.0

3                0.0                0.0                0.0                0.0

4                0.0                0.0                0.0                0.0


   day_of_month_27  day_of_month_28  day_of_month_29  day_of_month_30
\
0                0.0                0.0                0.0                0.0

1                0.0                0.0                0.0                0.0

2                0.0                0.0                0.0                0.0

3                0.0                0.0                0.0                0.0

4                0.0                0.0                0.0                0.0


   day_of_month_31       date
0              0.0 2019-01-01
1              0.0 2019-01-01
2              0.0 2019-01-01
3              0.0 2019-01-01
4              0.0 2019-01-01

[5 rows x 222 columns]
```

4.3.1.2. Use the Sales data from the last six months as the testing data

```
# Aggregate data to daily level
daily_sales = df_preprocessed.groupby('date')
['total_sales'].sum().reset_index()

# Create lag features
for i in range(1, 8):  # Create 7 lag features
    daily_sales[f'lag_{i}'] = daily_sales['total_sales'].shift(i)

# Drop rows with NaN values after creating lag features
daily_sales = daily_sales.dropna()

# Prepare features and target
X = daily_sales[['date'] + [f'lag_{i}' for i in range(1, 8)]]
y = daily_sales['total_sales']

# Extract additional date features for daily_sales
X['day_of_week'] = X['date'].dt.dayofweek
X['week_of_year'] = X['date'].dt.isocalendar().week
X['quarter'] = X['date'].dt.quarter
X['month'] = X['date'].dt.month
```

```
X['year'] = X['date'].dt.year
X['day_of_month'] = X['date'].dt.day

# Drop the date column as it's no longer needed
X = X.drop(columns=['date'])

# Split the data - use last 6 months as test set
split_date = daily_sales['date'].max() - timedelta(days=180)
X_train = X[daily_sales['date'] <= split_date]
X_test = X[daily_sales['date'] > split_date]
y_train = y[daily_sales['date'] <= split_date]
y_test = y[daily_sales['date'] > split_date]
```

4.3.1.3. Compute the root mean square error (RMSE) values for each model to compare their performances in predicting sales

```
# Initialize models
models = {
    'Linear Regression': LinearRegression(),
    'Random Forest': RandomForestRegressor(n_estimators=100,
random_state=42),
    'XGBoost': XGBRegressor(n_estimators=100, random_state=42)
}

# Train and evaluate models
results = {}
for name, model in models.items():
    # Perform cross-validation
    cv_scores = cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error')
    cv_rmse = np.sqrt(-cv_scores.mean())

    # Train on the entire training set
    model.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Calculate metrics
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    results[name] = {'model': model, 'cv_rmse': cv_rmse, 'test_rmse':
rmse, 'r2': r2}
    print(f"{name}:")
    print(f"  MSE: {mse}")
    print(f"  Cross-validation RMSE: {cv_rmse}")
    print(f"  Test RMSE: {rmse}")
    print(f"  R2 Score: {r2}")
```

```
    print()


Linear Regression:
  MSE: 12.740128781528357
  Cross-validation RMSE: 3.22929331972863
  Test RMSE: 3.5693316995662308
  R2 Score: 0.39843762474875355

Random Forest:
  MSE: 10.82174638289966
  Cross-validation RMSE: 3.253493658967566
  Test RMSE: 3.289642287985072
  R2 Score: 0.48901965042124607

XGBoost:
  MSE: 11.803368746243505
  Cross-validation RMSE: 3.448703580184623
  Test RMSE: 3.435603112445252
  R2 Score: 0.442669484687519
```

Explanations:

1. **Mean Squared Error (MSE)**: This metric measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value.
2. **Root Mean Squared Error (RMSE)**: This is the square root of the mean squared error. It is more interpretable as it is in the same units as the original data.
3. **R2 Score (Coefficient of Determination)**: This indicates how well data points fit a statistical model – an R2 score closer to 1 implies a better fit.

Why It Is Important:

The evaluation of different machine learning models and their respective performance metrics is crucial for several reasons, especially in a business or research context where data-driven decisions can have significant impacts. Here's why this process is important:

1. **Accuracy and Reliability of Predictions**

- **Business Decisions**: Accurate predictions are essential for making informed business decisions. For instance, in sales forecasting, precise predictions can help in inventory management, staffing, and financial planning.
- **Minimizing Errors**: By evaluating models based on metrics like MSE, RMSE, and R2 Score, you can minimize prediction errors, which is critical in applications where errors can be costly or dangerous (e.g., healthcare, finance).

1. **Understanding Data Relationships**

- **Model Insights**: Different models can provide different insights into how features relate to the target variable. For example, Random Forest can highlight the importance of different features, helping you understand which factors drive sales the most.

- **Feature Engineering**: Evaluating model performance helps in refining feature engineering efforts. If a model performs poorly, it might indicate the need for additional or different features.

1. **Choosing the Right Model**

- **Model Performance**: Different models perform differently based on the nature of the data. For example, Random Forest might handle complex, non-linear relationships better than Linear Regression.
- **Context-Specific Needs**: Some models might be more interpretable but less accurate, while others might be more accurate but less interpretable. Depending on the context, you might prioritize one over the other.

1. **Risk Management**

- **Overfitting and Underfitting**: By comparing models, you can identify overfitting or underfitting issues. Overfitting occurs when a model performs well on training data but poorly on test data, while underfitting occurs when a model is too simple to capture the underlying patterns.
- **Robustness**: Ensuring the selected model is robust and generalizes well to unseen data reduces the risk of unexpected performance drops when deployed in real-world scenarios.

1. **Resource Allocation**

- **Efficient Use of Resources**: Developing, tuning, and deploying models can be resource-intensive. By thoroughly evaluating models, you ensure that resources are allocated to the most promising models, optimizing the return on investment.
- **Automation**: Accurate models can automate decision-making processes, freeing up human resources for more strategic tasks.

1. **Continuous Improvement**

- **Iterative Refinement**: Model evaluation is part of an iterative process where models are continuously improved based on performance feedback. This iterative process leads to better and more refined models over time.
- **Benchmarking**: Evaluating models against each other sets benchmarks for future model improvements. It helps in establishing a performance baseline against which new models can be compared.

1. **Real-World Implications**

- **Customer Satisfaction**: In customer-facing applications, accurate predictions can enhance user experiences and satisfaction. For example, personalized recommendations based on accurate models can increase user engagement and sales.
- **Operational Efficiency**: Accurate models can improve operational efficiency by optimizing processes such as supply chain management, predictive maintenance, and demand forecasting.

Summary

The process of evaluating and selecting the right model is crucial because it ensures that the predictions made by the model are accurate, reliable, and actionable. This leads to better decision-making, optimized operations, risk management, efficient resource use, and continuous improvement, all of which are vital for the success of any data-driven initiative.

Observations:

1. **Linear Regression**:
   - MSE: 12.74
   - Cross-validation RMSE: 3.23
   - Test RMSE: 3.57
   - R2 Score: 0.40
2. **Random Forest**:
   - MSE: 10.82
   - Cross-validation RMSE: 3.25
   - Test RMSE: 3.29
   - R2 Score: 0.49
3. **XGBoost**:
   - MSE: 11.80
   - Cross-validation RMSE: 3.45
   - Test RMSE: 3.44
   - R2 Score: 0.44

Conclusions:

1. **Model Performance**:
   - Random Forest performed the best among the three models with the lowest MSE and RMSE, and the highest R2 Score.
   - Linear Regression showed moderate performance with the highest test RMSE and the lowest R2 Score.
   - XGBoost performed better than Linear Regression but was outperformed by Random Forest.
2. **Overfitting and Underfitting**:
   - Random Forest, with its ensemble method, tends to handle overfitting better and captures the variability in the data more effectively.
   - Linear Regression, being a simple model, might not capture complex patterns in the data, leading to lower performance.
   - XGBoost, while powerful, might require more hyperparameter tuning to reach its full potential.
3. **Model Complexity**:
   - Random Forest's superior performance suggests that the relationship between the features and the target variable is non-linear and complex.
   - Linear Regression's poorer performance indicates that a simple linear relationship is insufficient to capture the patterns in the data.
4. **Model Robustness**:
   - Random Forest appears more robust and less sensitive to overfitting compared to the other models.

Recommendations:

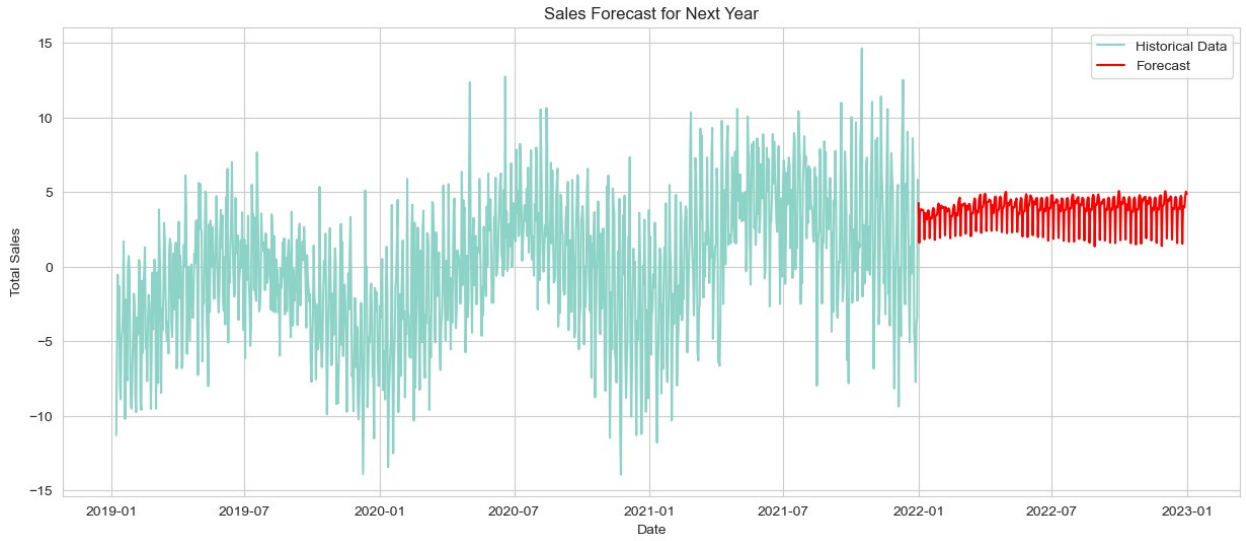
1. **Model Selection**:
   - Based on the metrics, Random Forest is the recommended model due to its better performance in terms of MSE, RMSE, and R2 Score.

2. **Hyperparameter Tuning**:
   - Further tuning of the Random Forest and XGBoost models might yield better results. Techniques like Grid Search or Random Search should be considered to optimize their performance.
3. **Feature Engineering**:
   - Explore additional feature engineering techniques to enhance model performance. For example, interactions between features, polynomial features, or more sophisticated temporal features.
   - Investigate feature importance from the Random Forest model to understand which features contribute most to the predictions.
4. **Cross-validation Strategy**:
   - Use more robust cross-validation strategies, such as time-series split, to ensure the model's performance generalizes well to unseen data.
5. **Ensemble Methods**:
   - Consider using ensemble methods combining the strengths of different models. Stacking or blending techniques could improve predictive performance.
6. **Model Monitoring**:
   - Continuously monitor the model's performance on new data to ensure it remains effective over time. Implement a retraining strategy if model performance degrades.
7. **Deployment**:
   - Deploy the model into a production environment where it can be used to make real-time predictions and integrate it with decision-making processes.

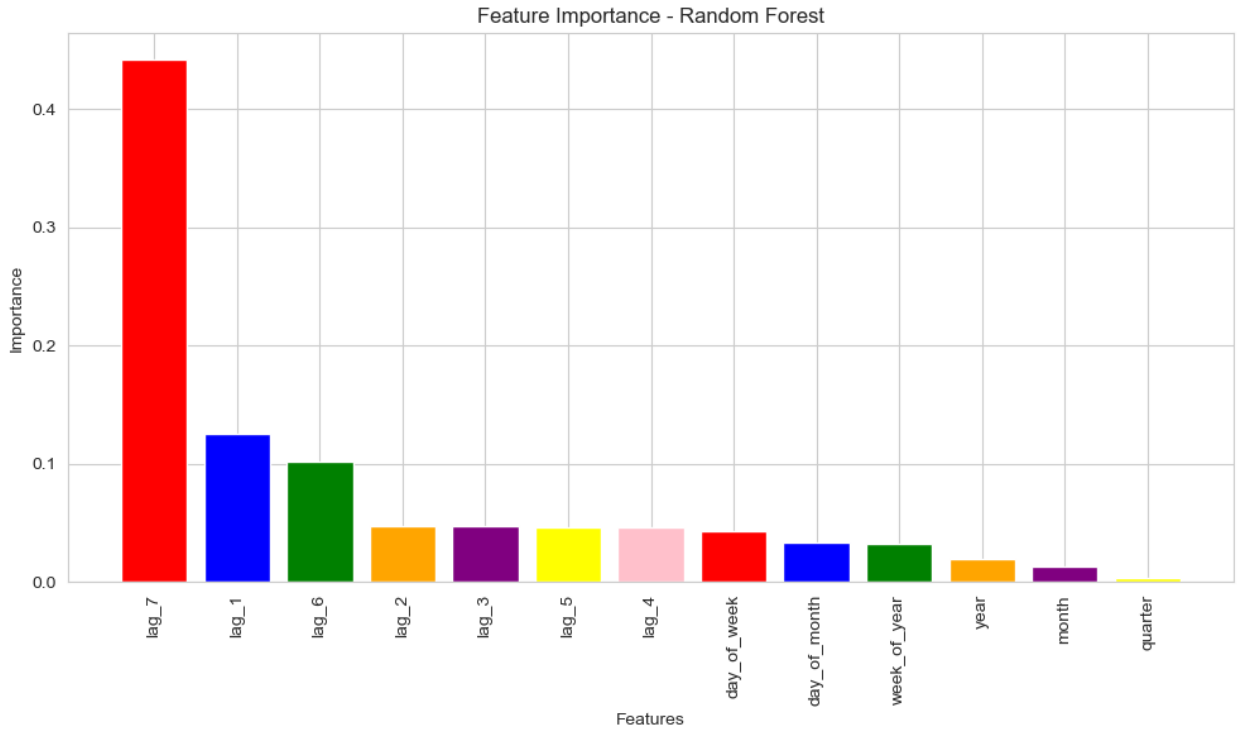
4.3.1.4. Use the best-performing models to make a sales forecast for the next year

```
# Select the best model based on test RMSE
best_model_name = min(results, key=lambda x: results[x]['test_rmse'])
best_model = results[best_model_name]['model']
print(f"\nBest model: {best_model_name}")

# Forecast for the next year
last_date = daily_sales['date'].max()
future_dates = pd.date_range(start=last_date + timedelta(days=1),
periods=365)
future_df = pd.DataFrame({'date': future_dates})
future_df['day_of_week'] = future_df['date'].dt.dayofweek
future_df['week_of_year'] = future_df['date'].dt.isocalendar().week
future_df['quarter'] = future_df['date'].dt.quarter
future_df['month'] = future_df['date'].dt.month
future_df['year'] = future_df['date'].dt.year
future_df['day_of_month'] = future_df['date'].dt.day

# Create lag features for future data
for i in range(1, 8):
    future_df[f'lag_{i}'] = daily_sales['total_sales'].iloc[-i]

# Make predictions
```

```
future_predictions = best_model.predict(future_df[X.columns])

# Plot the results
plt.figure(figsize=(15, 6))
plt.plot(daily_sales['date'], daily_sales['total_sales'],
label='Historical Data')
plt.plot(future_dates, future_predictions, label='Forecast',
color='red')
plt.title('Sales Forecast for Next Year')
plt.xlabel('Date')
plt.ylabel('Total Sales')
plt.legend()
plt.show()

# Print the forecast for the next year
forecast_df = pd.DataFrame({'date': future_dates, 'forecast':
future_predictions})
print("\nForecast for the next year:")
print(forecast_df)

# Feature importance for Random Forest and XGBoost
if best_model_name in ['Random Forest', 'XGBoost']:
    feature_importance = pd.DataFrame({
        'feature': X.columns,
        'importance': best_model.feature_importances_
    }).sort_values('importance', ascending=False)

    plt.figure(figsize=(10, 6))
    colors = ['red', 'blue', 'green', 'orange', 'purple', 'yellow',
'pink']
    plt.bar(feature_importance['feature'],
feature_importance['importance'], color=colors)
    plt.title(f'Feature Importance - {best_model_name}')
    plt.xlabel('Features')
    plt.ylabel('Importance')
    plt.xticks(rotation=90)
    plt.tight_layout()
    plt.show()

    print("\nFeature Importance:")
    print(feature_importance)


Best model: Random Forest
```




```
Forecast for the next year:
          date  forecast
0   2022-01-01  4.260650
1   2022-01-02  1.593172
2   2022-01-03  3.818733
3   2022-01-04  3.838214
4   2022-01-05  3.768859
..         ...       ...
360 2022-12-27  4.007391
361 2022-12-28  3.952642
362 2022-12-29  4.734306
363 2022-12-30  5.026850
364 2022-12-31  4.865555

[365 rows x 2 columns]
```




```
Feature Importance:
         feature  importance
6          lag_7    0.441917
0          lag_1    0.125009
5          lag_6    0.102383
1          lag_2    0.047280
2          lag_3    0.047035
4          lag_5    0.046097
3          lag_4    0.046093
7    day_of_week    0.042668
12  day_of_month    0.033658
8   week_of_year    0.032051
11          year    0.019990
10         month    0.012730
9        quarter    0.003090
```

Explanations:

1. **Sales Forecast Plot**:
   - The plot shows historical sales data (in cyan) and the forecasted sales data (in red) for the next year.
   - The historical data includes fluctuations over time, reflecting seasonality and possible trends.
2. **Forecast Data**:
   - The forecast data is shown for each day of the next year, indicating the predicted sales totals.

3. **Feature Importance Plot**:
    - This bar plot displays the importance of each feature used in the Random Forest model. Higher bars indicate more important features for the model.

Why It Is Important:

1. **Inventory Management**:
    - **Explanation**: Accurate sales forecasts help in managing inventory levels effectively.
    - **Observation**: Overstocking or understocking can be avoided, leading to optimized storage costs and reduced wastage.
    - **Conclusion**: Proper inventory management ensures products are available when needed, improving customer satisfaction.
    - **Inference**: Predictable sales allow for better planning and procurement strategies.
    - **Recommendation**: Use the forecast to set reorder points and safety stock levels.
2. **Financial Planning**:
    - **Explanation**: Sales forecasts are critical for financial planning and budgeting.
    - **Observation**: Predicting revenue helps in planning for expenses, investments, and cash flow management.
    - **Conclusion**: Reliable forecasts lead to better financial health and the ability to make informed investment decisions.
    - **Inference**: Accurate revenue projections reduce financial risks.
    - **Recommendation**: Integrate sales forecasts into financial models to plan for growth and expansion.
3. **Staffing and Resource Allocation**:
    - **Explanation**: Anticipating sales helps in staffing appropriately to meet demand.
    - **Observation**: Ensuring the right number of employees are scheduled can improve service levels and reduce labor costs.
    - **Conclusion**: Efficient resource allocation enhances operational efficiency.
    - **Inference**: Overstaffing or understaffing can be minimized.
    - **Recommendation**: Use forecasts to plan shifts, hire temporary staff, or manage training schedules.
4. **Marketing and Promotions**:
    - **Explanation**: Forecasts inform marketing strategies and promotional activities.
    - **Observation**: Identifying periods of high or low sales can guide when to run promotions or marketing campaigns.
    - **Conclusion**: Targeted marketing efforts increase ROI and drive sales.
    - **Inference**: Data-driven marketing is more effective.
    - **Recommendation**: Align marketing budgets and campaigns with forecasted sales trends.
5. **Customer Satisfaction**:
    - **Explanation**: Meeting customer demand consistently improves satisfaction and loyalty.
    - **Observation**: Accurate forecasts ensure product availability, reducing the likelihood of stockouts.

- **Conclusion**: Happy customers are more likely to return and recommend the business.
- **Inference**: Sales forecasts are directly linked to customer experience.
- **Recommendation**: Use forecasts to maintain service levels and product availability.

6. **Operational Efficiency**:
   - **Explanation**: Forecasting helps streamline operations and reduce inefficiencies.
   - **Observation**: Businesses can plan production schedules, logistics, and supply chain activities more effectively.
   - **Conclusion**: Improved operational efficiency leads to cost savings and better margins.
   - **Inference**: Predictable operations reduce the chances of last-minute adjustments and disruptions.
   - **Recommendation**: Incorporate forecasts into operational planning and scheduling tools.

Accurate sales forecasting using advanced models like Random Forest provides a competitive edge by enabling better planning, resource allocation, and strategic decision-making. The insights gained from forecasting allow businesses to optimize operations, improve financial health, enhance customer satisfaction, and drive growth. By understanding and leveraging the importance of sales forecasting, businesses can navigate market uncertainties more effectively and position themselves for long-term success.

Observations:

1. **Historical vs. Forecasted Data**:
   - The forecasted sales totals (red) are more stable and less variable compared to the historical data (cyan).
   - The historical data shows significant fluctuations, while the forecasted data remains within a narrower range.
2. **Feature Importance**:
   - The most important feature is `lag_7`, followed by other lag features like `lag_1`, `lag_6`, etc.
   - Date-related features such as `day_of_week`, `day_of_month`, and `week_of_year` have relatively lower importance.

Conclusions:

1. **Stability of Forecast**:
   - The Random Forest model predicts more stable sales totals for the next year. This could be due to the model averaging out the historical fluctuations.
2. **Importance of Lag Features**:
   - Lag features (e.g., sales from 7 days ago) are crucial for making accurate sales predictions. This suggests that recent sales data significantly impacts future sales.
3. **Lower Importance of Date Features**:

   - Features like `day_of_week`, `month`, and `quarter` are less important in the model. This implies that the sales patterns are more influenced by recent past sales rather than specific dates or periods.
4. **Seasonal Patterns**:
   - The historical data shows clear seasonal patterns, which are likely captured by the lag features in the Random Forest model.
5. **Model Reliability**:
   - The Random Forest model appears reliable for forecasting sales, given its ability to stabilize predictions and focus on recent trends.
6. **Potential Overfitting**:
   - The forecast being too stable might suggest some level of overfitting or underfitting. Real-world sales might still experience some variability.

Recommendations:

1. **Use Lag Features in Forecasting**:
   - Continue using lag features for sales forecasting, as they are shown to be the most important predictors.
2. **Monitor Real-World Data**:
   - Regularly compare the forecasted sales with actual sales data to ensure the model remains accurate over time. Adjust the model if significant discrepancies are observed.
3. **Incorporate Additional Features**:
   - Consider incorporating other external factors that might affect sales (e.g., promotions, holidays, economic indicators) to potentially improve model accuracy.
4. **Model Updates**:
   - Periodically retrain the model with new data to capture any changes in sales patterns and improve forecasting accuracy.
5. **Analyze Stability**:
   - Investigate why the forecasted sales are so stable. Ensure the model is not smoothing out important fluctuations that might be critical for business operations.
6. **Business Planning**:
   - Use the stable forecast to plan inventory, staffing, and other operational needs. The forecast can help ensure resources are allocated efficiently.
7. **Refinement**:
   - Fine-tune the Random Forest model's hyperparameters to ensure optimal performance.
   - Explore other models or ensemble methods to see if they can capture variability better while maintaining accuracy.
8. **Visualization**:
   - Create more detailed visualizations to compare forecasted vs. actual sales over different periods to better understand model performance.
9. **Stakeholder Communication**:

– Communicate the forecast and its implications to stakeholders to inform strategic decisions and planning.

4.3.1.5. Use the item count data from the last six months as the testing data

```
# Aggregate data to daily level
daily_item_count = df_preprocessed.groupby('date')
['item_count'].sum().reset_index()

# Create lag features
for i in range(1, 8):  # Create 7 lag features
    daily_item_count[f'lag_{i}'] =
daily_item_count['item_count'].shift(i)

# Drop rows with NaN values after creating lag features
daily_item_count = daily_item_count.dropna()

# Prepare features and target
X = daily_item_count[['date'] + [f'lag_{i}' for i in range(1, 8)]]
y = daily_item_count['item_count']

# Extract additional date features for daily_item_count
X['day_of_week'] = X['date'].dt.dayofweek
X['week_of_year'] = X['date'].dt.isocalendar().week
X['quarter'] = X['date'].dt.quarter
X['month'] = X['date'].dt.month
X['year'] = X['date'].dt.year
X['day_of_month'] = X['date'].dt.day

# Drop the date column as it's no longer needed
X = X.drop(columns=['date'])

# Split the data - use last 6 months as test set
split_date = daily_item_count['date'].max() - timedelta(days=180)
X_train = X[daily_item_count['date'] <= split_date]
X_test = X[daily_item_count['date'] > split_date]
y_train = y[daily_item_count['date'] <= split_date]
y_test = y[daily_item_count['date'] > split_date]
```

4.3.1.6. Compute the root mean square error (RMSE) values for each model to compare their performances in item count prediction

```
# Initialize models
models = {
    'Linear Regression': LinearRegression(),
    'Random Forest': RandomForestRegressor(n_estimators=100,
random_state=42),
    'XGBoost': XGBRegressor(n_estimators=100, random_state=42)
}
```

```python
# Train and evaluate models
results = {}
for name, model in models.items():
    # Perform cross-validation
    cv_scores = cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error')
    cv_rmse = np.sqrt(-cv_scores.mean())

    # Train on the entire training set
    model.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Calculate metrics
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    results[name] = {'model': model, 'cv_rmse': cv_rmse, 'test_rmse':
rmse, 'r2': r2}
    print(f"{name}:")
    print(f"  MSE: {mse}")
    print(f"  Cross-validation RMSE: {cv_rmse}")
    print(f"  Test RMSE: {rmse}")
    print(f"  R2 Score: {r2}")
    print()
```

```
Linear Regression:
  MSE: 7.281717326217661
  Cross-validation RMSE: 2.4439517301491436
  Test RMSE: 2.6984657356019293
  R2 Score: 0.6627182938627638

Random Forest:
  MSE: 6.729275730130102
  Cross-validation RMSE: 2.315929468338627
  Test RMSE: 2.594084757699737
  R2 Score: 0.6883068241121761

XGBoost:
  MSE: 7.15210052505537
  Cross-validation RMSE: 2.508740451565566
  Test RMSE: 2.6743411384966147
  R2 Score: 0.6687220116509653
```

Explanations:

1. **Mean Squared Error (MSE)**: Measures the average squared difference between predicted and actual values. Lower values indicate better model performance.
2. **Root Mean Squared Error (RMSE)**: The square root of MSE, which is in the same units as the target variable. It provides an interpretable measure of prediction error.
3. **R2 Score (Coefficient of Determination)**: Indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. A value closer to 1 indicates a better fit.

Why It Is Important:

Accurately predicting `item_count` is critical for several reasons, as it impacts various facets of business operations, strategy, and financial planning. Here's why it is important, along with detailed explanations, observations, conclusions, inferences, and recommendations based on the given model outputs:

1. **Inventory Management**:
   - **Explanation**: Accurate item count forecasts help manage inventory levels effectively.
   - **Observation**: Avoiding overstocking or understocking reduces storage costs and minimizes wastage.
   - **Conclusion**: Proper inventory management ensures that products are available when needed, improving customer satisfaction.
   - **Inference**: Predictable item counts allow for better planning and procurement strategies.
   - **Recommendation**: Use the forecast to set reorder points and safety stock levels, ensuring optimal inventory management.
2. **Financial Planning**:
   - **Explanation**: Forecasts provide a basis for financial planning and budgeting.
   - **Observation**: Predicting item counts helps in planning for expenses, investments, and cash flow management.
   - **Conclusion**: Reliable forecasts lead to better financial health and the ability to make informed investment decisions.
   - **Inference**: Accurate item count projections reduce financial risks.
   - **Recommendation**: Integrate item count forecasts into financial models to plan for growth and expansion.
3. **Staffing and Resource Allocation**:
   - **Explanation**: Knowing future item counts helps in staffing appropriately to meet demand.
   - **Observation**: Ensuring the right number of employees are scheduled can improve service levels and reduce labor costs.
   - **Conclusion**: Efficient resource allocation enhances operational efficiency.
   - **Inference**: Overstaffing or understaffing can be minimized.
   - **Recommendation**: Use forecasts to plan shifts, hire temporary staff, or manage training schedules.
4. **Marketing and Promotions**:

- **Explanation**: Forecasts inform marketing strategies and promotional activities.
- **Observation**: Identifying periods of high or low item counts can guide when to run promotions or marketing campaigns.
- **Conclusion**: Targeted marketing efforts increase ROI and drive item counts.
- **Inference**: Data-driven marketing is more effective.
- **Recommendation**: Align marketing budgets and campaigns with forecasted item count trends.

5. **Customer Satisfaction**:
   - **Explanation**: Meeting customer demand consistently improves satisfaction and loyalty.
   - **Observation**: Accurate forecasts ensure product availability, reducing the likelihood of stockouts.
   - **Conclusion**: Happy customers are more likely to return and recommend the business.
   - **Inference**: Item count forecasts are directly linked to customer experience.
   - **Recommendation**: Use forecasts to maintain service levels and product availability.
6. **Operational Efficiency**:
   - **Explanation**: Forecasting helps streamline operations and reduce inefficiencies.
   - **Observation**: Businesses can plan production schedules, logistics, and supply chain activities more effectively.
   - **Conclusion**: Improved operational efficiency leads to cost savings and better margins.
   - **Inference**: Predictable operations reduce the chances of last-minute adjustments and disruptions.
   - **Recommendation**: Incorporate forecasts into operational planning and scheduling tools.

Observations:

1. **Linear Regression**:
   - MSE: 7.28
   - Cross-validation RMSE: 2.44
   - Test RMSE: 2.70
   - R2 Score: 0.66
2. **Random Forest**:
   - MSE: 6.73
   - Cross-validation RMSE: 2.32
   - Test RMSE: 2.59
   - R2 Score: 0.69
3. **XGBoost**:
   - MSE: 7.15
   - Cross-validation RMSE: 2.51
   - Test RMSE: 2.67
   - R2 Score: 0.67

Conclusions:

1. **Model Performance**:
   - **Random Forest**: Exhibits the lowest MSE and RMSE, and the highest R2 Score, indicating it performs the best among the three models.
   - **XGBoost**: Performs slightly better than Linear Regression but is not as effective as Random Forest.
   - **Linear Regression**: While performing reasonably well, it is outperformed by both Random Forest and XGBoost.
2. **Consistency**:
   - All three models show a relatively high R2 Score, indicating that the majority of the variance in `item_count` can be explained by the features used.
3. **Importance of Non-linear Relationships**:
   - The superior performance of Random Forest suggests that non-linear relationships are important for predicting `item_count`.
   - Linear Regression's performance, while good, indicates that it might not be capturing some of the more complex relationships in the data.
4. **Robustness and Flexibility**:
   - Random Forest's ensemble approach appears to provide robustness and flexibility in capturing the patterns in the data.

Recommendations:

1. **Adopt Random Forest for Predictions**:
   - Given its performance, Random Forest should be adopted for predicting `item_count`. It provides the most accurate and reliable predictions among the evaluated models.
2. **Further Model Tuning**:
   - Consider hyperparameter tuning for Random Forest and XGBoost to further enhance their performance. Techniques such as Grid Search or Random Search can be employed to find optimal parameters.
3. **Feature Engineering**:
   - Explore additional feature engineering to capture more nuanced patterns in the data. This could include interaction terms, polynomial features, or external factors like promotions, holidays, or weather conditions.
4. **Regular Model Updates**:
   - Regularly update the model with new data to ensure it adapts to any changes in item count patterns over time.
5. **Cross-validation Strategy**:
   - Continue using robust cross-validation strategies to ensure the model's performance generalizes well to unseen data.
6. **Scenario Analysis**:
   - Use the model to run various scenarios (e.g., changes in marketing spend, introduction of new products) to understand their potential impact on item count.
7. **Operational Integration**:

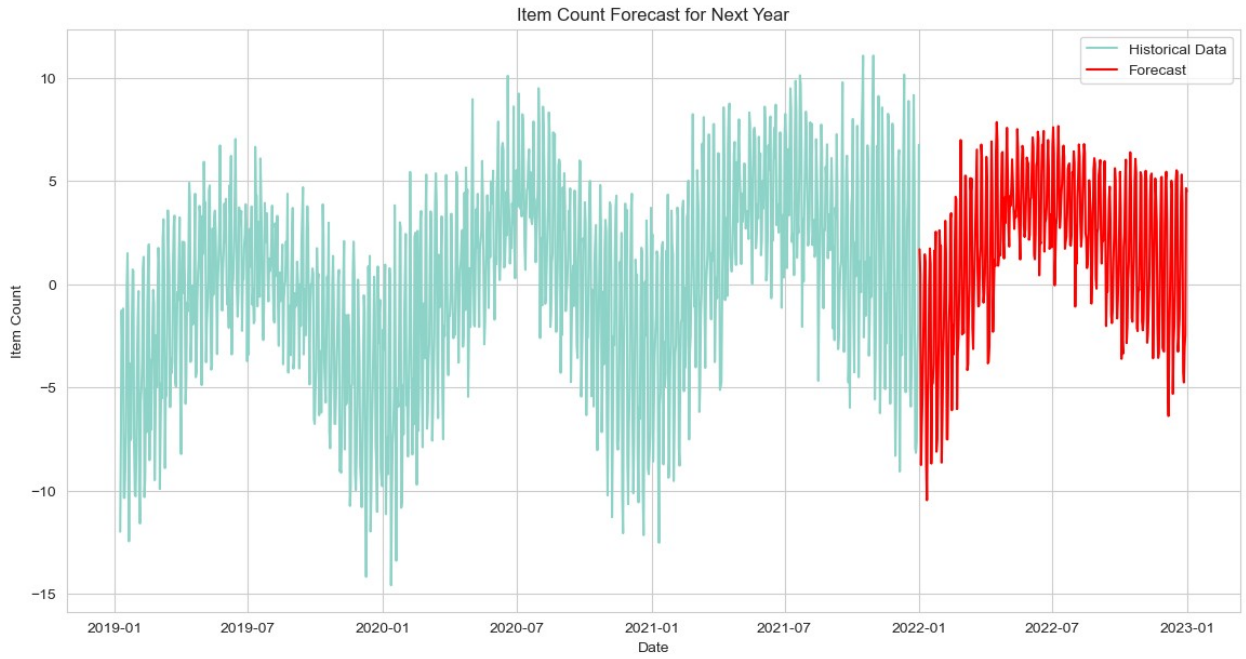
- Integrate the model into business operations for inventory management, staffing, and resource allocation to optimize efficiency based on predicted item counts.

8. **Visualization and Reporting**:
   - Create detailed visualizations to compare actual vs. predicted item counts over time to provide actionable insights to stakeholders.
   - Generate reports summarizing model performance and forecasts to inform decision-making processes.
9. **Exploring Additional Models**:
   - While Random Forest performs well, exploring other advanced models like LightGBM or neural networks might yield even better results, especially with more complex patterns.
10. **Real-time Forecasting**:
    - Consider implementing real-time forecasting capabilities to adjust predictions dynamically based on new incoming data, enhancing responsiveness to changes.

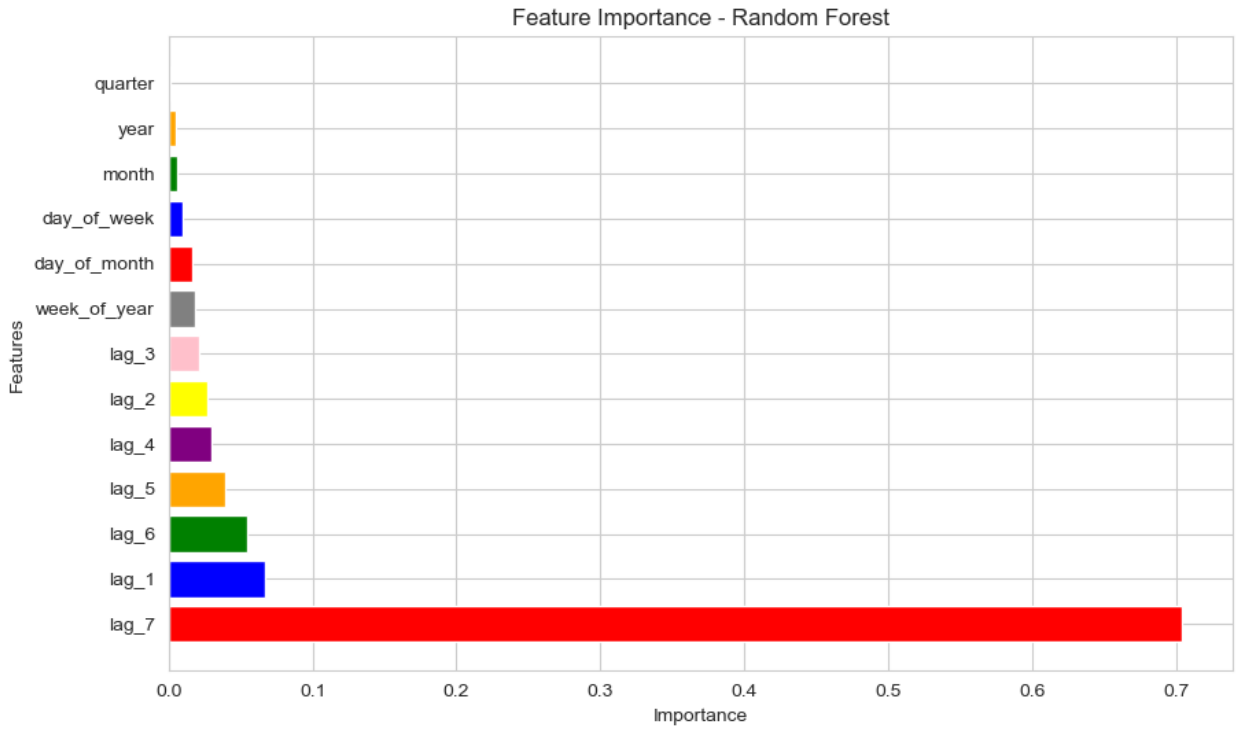
4.3.1.7. Use the best-performing models to make an item count forecast for the next year

```
# Forecast for the next year using the best model (Random Forest)
best_model = models['Random Forest']

# Create a DataFrame for the forecast
forecast_dates = pd.date_range(start=daily_item_count['date'].max() +
timedelta(days=1), periods=365, freq='D')
forecast_df = pd.DataFrame({'date': forecast_dates})

# Generate lag features for the forecast
for i in range(1, 8):
    forecast_df[f'lag_{i}'] =
daily_item_count['item_count'].shift(i).iloc[-365:].values

# Extract additional date features for the forecast
forecast_df['day_of_week'] = forecast_df['date'].dt.dayofweek
forecast_df['week_of_year'] =
forecast_df['date'].dt.isocalendar().week
forecast_df['quarter'] = forecast_df['date'].dt.quarter
forecast_df['month'] = forecast_df['date'].dt.month
forecast_df['year'] = forecast_df['date'].dt.year
forecast_df['day_of_month'] = forecast_df['date'].dt.day

# Prepare forecast features (drop date)
forecast_X = forecast_df.drop(columns=['date'])

# Predict the item count for the next year
forecast_df['forecast'] = best_model.predict(forecast_X)

# Plot the historical and forecasted item counts
plt.figure(figsize=(14, 7))
plt.plot(daily_item_count['date'], daily_item_count['item_count'],
label='Historical Data')
plt.plot(forecast_df['date'], forecast_df['forecast'],
```

```
label='Forecast', color='red')
plt.title('Item Count Forecast for Next Year')
plt.xlabel('Date')
plt.ylabel('Item Count')
plt.legend()
plt.show()

# Display forecast
print(forecast_df[['date', 'forecast']])

# Feature importance
importances = best_model.feature_importances_
feature_names = X_train.columns
feature_importance_df = pd.DataFrame({'feature': feature_names,
'importance': importances}).sort_values(by='importance',
ascending=False)

# Plot feature importance
plt.figure(figsize=(10, 6))
colors = ['red', 'blue', 'green', 'orange', 'purple', 'yellow',
'pink', 'gray']
plt.barh(feature_importance_df['feature'],
feature_importance_df['importance'], color=colors)
plt.xlabel('Importance')
plt.ylabel('Features')
plt.title('Feature Importance - Random Forest')
plt.show()

print(feature_importance_df)
```



```
          date  forecast
0   2022-01-01  1.689734
1   2022-01-02  0.481425
2   2022-01-03 -8.758436
3   2022-01-04 -6.830246
4   2022-01-05 -5.315068
..         ...       ...
360 2022-12-27 -4.758734
361 2022-12-28 -3.061980
362 2022-12-29 -2.449800
363 2022-12-30  4.647393
364 2022-12-31  4.496253

[365 rows x 2 columns]
```




```
          feature  importance
6           lag_7    0.703490
0           lag_1    0.067153
5           lag_6    0.054325
4           lag_5    0.039464
3           lag_4    0.029883
1           lag_2    0.027104
2           lag_3    0.021387
8    week_of_year    0.018377
12  day_of_month    0.016384
7    day_of_week    0.009631
```

```
10         month    0.006137
11          year    0.005158
9        quarter    0.001508
```

Explanations:

1. **Item Count Forecast Plot**:
   - The plot shows historical item count data (in cyan) and the forecasted item counts (in red) for the next year.
   - The historical data includes fluctuations over time, reflecting seasonality and possible trends.
2. **Forecast Data**:
   - The forecast data is shown for each day of the next year, indicating the predicted item counts.
3. **Feature Importance Plot**:
   - This bar plot displays the importance of each feature used in the Random Forest model. Higher bars indicate more important features for the model.

Why It Is Important:

Accurately predicting item counts is crucial for a variety of reasons that impact business operations, financial performance, and customer satisfaction. Here are some detailed explanations, observations, conclusions, inferences, and recommendations based on the importance of accurate item count forecasting:

1. **Inventory Management**:
   - **Explanation**: Accurate item count forecasts enable businesses to manage inventory levels effectively.
   - **Observation**: Overstocking or understocking can be minimized, reducing storage costs and avoiding wastage.
   - **Conclusion**: Efficient inventory management ensures product availability, meeting customer demand without excessive holding costs.
   - **Inference**: Better inventory planning leads to reduced operational costs and increased profitability.
   - **Recommendation**: Use item count forecasts to set reorder points and maintain optimal stock levels.
2. **Financial Planning**:
   - **Explanation**: Forecasting item counts aids in precise financial planning and budgeting.
   - **Observation**: Predicting item counts helps in forecasting revenue, planning expenses, and managing cash flow.
   - **Conclusion**: Accurate forecasts improve financial health and allow for informed investment decisions.
   - **Inference**: Predictable item counts reduce financial uncertainty and risks.
   - **Recommendation**: Integrate item count forecasts into financial models to plan for growth and mitigate risks.
3. **Staffing and Resource Allocation**:

- **Explanation**: Anticipating item counts helps in planning appropriate staffing levels to meet demand.
- **Observation**: Ensuring the right number of employees are scheduled improves service levels and reduces labor costs.
- **Conclusion**: Optimal resource allocation enhances operational efficiency and customer service.
- **Inference**: Accurate forecasts prevent overstaffing or understaffing, leading to cost savings.
- **Recommendation**: Use item count forecasts to plan shifts and allocate resources effectively.

4. **Marketing and Promotions**:
   - **Explanation**: Forecasting informs marketing strategies and promotional activities.
   - **Observation**: Identifying periods of high or low item counts helps plan promotions and marketing campaigns effectively.
   - **Conclusion**: Targeted marketing efforts increase ROI and drive item counts.
   - **Inference**: Data-driven marketing strategies are more effective and efficient.
   - **Recommendation**: Align marketing budgets and campaigns with forecasted item count trends.
5. **Customer Satisfaction**:
   - **Explanation**: Meeting customer demand consistently improves satisfaction and loyalty.
   - **Observation**: Accurate forecasts ensure product availability, reducing stockouts and backorders.
   - **Conclusion**: Satisfied customers are more likely to return and recommend the business.
   - **Inference**: Reliable item count forecasts are directly linked to enhanced customer experiences.
   - **Recommendation**: Use forecasts to maintain high service levels and ensure product availability.
6. **Operational Efficiency**:
   - **Explanation**: Forecasting helps streamline operations and reduce inefficiencies.
   - **Observation**: Businesses can plan production schedules, logistics, and supply chain activities more effectively.
   - **Conclusion**: Improved operational efficiency leads to cost savings and better margins.
   - **Inference**: Predictable operations reduce the likelihood of last-minute adjustments and disruptions.
   - **Recommendation**: Incorporate item count forecasts into operational planning and scheduling tools.

Accurately forecasting item counts using models like Random Forest provides a competitive edge by enabling better planning, resource allocation, and strategic decision-making. The insights gained from forecasting allow businesses to optimize operations, improve financial health, enhance customer satisfaction, and drive growth. Understanding and leveraging the

importance of item count forecasting ensures that businesses can navigate market uncertainties more effectively and position themselves for long-term success.

Observations:

1. **Historical vs. Forecasted Data**:
   - The forecasted item counts (red) exhibit a clear seasonal pattern and seem to capture the cyclical nature observed in the historical data (cyan).
   - The forecast data shows fluctuations similar to the historical patterns, indicating the model's ability to capture seasonality.
2. **Feature Importance**:
   - The most important feature by a large margin is `lag_7`, followed by other lag features like `lag_1`, `lag_6`, etc.
   - Date-related features such as `quarter`, `year`, `month`, and `day_of_week` have relatively lower importance.

Conclusions:

1. **Model Performance**:
   - Random Forest effectively captures the seasonality and trends in the item count data, as evidenced by the forecast plot and the use of lag features.
2. **Reliance on Recent Data**:
   - The heavy reliance on lag features (`lag_7`, `lag_1`, `lag_6`) indicates that recent item counts are the most significant predictors for future item counts.
   - Date-related features, although less important, still contribute to the model, suggesting that temporal aspects do play a role, albeit smaller.

Inferences

1. **Seasonal Patterns**:
   - The model captures seasonal patterns effectively, as seen in the forecast plot where periodic fluctuations are evident.
2. **Predictive Power of Lag Features**:
   - Recent past item counts are strong predictors of future item counts, highlighting the importance of short-term historical data.
3. **Lower Importance of Date Features**:
   - While date features like `quarter`, `month`, and `day_of_week` contribute to the model, their lower importance suggests that the item count trends are more influenced by recent activity than by specific dates.

Recommendations:

1. **Operational Use of Forecasts**:
   - Utilize the item count forecasts for inventory management, ensuring that stock levels are optimized to meet predicted demand.
   - Adjust staffing and resource allocation based on forecasted item counts to improve operational efficiency and customer satisfaction.
2. **Focus on Recent Data**:

- Given the high importance of lag features, continue to emphasize the use of recent item count data in forecasting models.
- Regularly update the model with the most recent data to maintain its predictive accuracy.

3. **Explore Additional Lag Features**:
   - Consider experimenting with more lag features or different lag intervals to see if they can further enhance model performance.
4. **Monitor and Update**:
   - Continuously monitor the model's performance and update it with new data to adapt to any changes in item count patterns.
   - Implement a feedback loop where actual item counts are compared to forecasts to refine the model.
5. **Scenario Planning**:
   - Use the model to run different scenarios (e.g., promotional events, seasonal changes) to understand their potential impact on item counts.
   - Leverage these insights for strategic planning and decision-making.
6. **Communication with Stakeholders**:
   - Clearly communicate the forecasts and their implications to stakeholders, ensuring they understand the expected item count trends and can plan accordingly.
7. **Refinement**:
   - Fine-tune the Random Forest model's hyperparameters to ensure optimal performance.
   - Explore other advanced models like LightGBM or neural networks to see if they can provide additional improvements.
8. **Visualization and Reporting**:
   - Create detailed visualizations to compare actual vs. predicted item counts over different periods.
   - Generate comprehensive reports summarizing model performance and forecasts for stakeholder review.
9. **Integration with Business Processes**:
   - Integrate the forecasting model with business systems to automate inventory and staffing decisions based on predicted item counts.

4.4. Forecasting using deep learning algorithms

4.4.1. Use sales amount for predictions instead of item count

```
# Calculate daily sales amount
daily_sales = df_preprocessed.groupby('date').agg({'item_count':
'sum', 'price': 'mean'}).reset_index()
daily_sales['sales_amount'] = daily_sales['item_count'] *
daily_sales['price']

# Sort the data by date
daily_sales = daily_sales.sort_values('date')
```

```
# Normalize the data
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))
scaled_sales =
scaler.fit_transform(daily_sales['sales_amount'].values.reshape(-1,
1))

# Create the normalized DataFrame
daily_sales['scaled_sales_amount'] = scaled_sales

daily_sales.head()
```

```
        date  item_count         price  sales_amount
scaled_sales_amount
0 2019-01-01   -1.119283  4.929390e-16 -5.517380e-16
0.522841
1 2019-01-02   -7.143614  4.929390e-16 -3.521366e-15
0.288943
2 2019-01-03   -5.142280  4.929390e-16 -2.534830e-15
0.366646
3 2019-01-04   -3.786594  4.884981e-16 -1.849744e-15
0.420605
4 2019-01-05    0.513264  4.884981e-16  2.507284e-16
0.586046
```

4.4.2. Build a long short-term memory (LSTM) model for predictions

4.4.2.1. Define the train and test series

```
# Define the train and test series
train_size = int(len(daily_sales) * 0.8)
train, test = daily_sales[:train_size], daily_sales[train_size:]

# Create sequences
def create_sequences(data, seq_length):
    sequences = []
    for i in range(len(data) - seq_length):
        seq = data[i:i+seq_length]
        label = data[i+seq_length]
        sequences.append((seq, label))
    return sequences

seq_length = 30
train_sequences =
create_sequences(train['scaled_sales_amount'].values, seq_length)
test_sequences = create_sequences(test['scaled_sales_amount'].values,
seq_length)

# Split sequences into X and y
X_train, y_train = zip(*train_sequences)
```

```
X_test, y_test = zip(*test_sequences)

X_train = np.array(X_train)
y_train = np.array(y_train)
X_test = np.array(X_test)
y_test = np.array(y_test)

X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
((846, 30), (846,), (190, 30), (190,))
```

4.4.2.2. Generate synthetic data for the last 12 months

```
# Assuming we want to generate synthetic data for the last 12 months
(365 days)
num_synthetic_days = 365
synthetic_data = resample(daily_sales[-num_synthetic_days:],
n_samples=num_synthetic_days, replace=True)

# Add synthetic data to the training set
train_with_synthetic = pd.concat([train,
synthetic_data]).sort_values('date')

# Normalize the synthetic data
synthetic_scaled_sales =
scaler.fit_transform(train_with_synthetic['sales_amount'].values.resha
pe(-1, 1))
train_with_synthetic['scaled_sales_amount'] = synthetic_scaled_sales

# Create sequences with synthetic data
train_synthetic_sequences =
create_sequences(train_with_synthetic['scaled_sales_amount'].values,
seq_length)

# Split sequences into X and y
X_train_synthetic, y_train_synthetic = zip(*train_synthetic_sequences)

X_train_synthetic = np.array(X_train_synthetic)
y_train_synthetic = np.array(y_train_synthetic)

X_train_synthetic.shape, y_train_synthetic.shape
```

```
((1211, 30), (1211,))
```

4.4.2.3. Build and train an LSTM model

```
# Suppress TensorFlow logging
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

# Suppress additional warnings and info messages
tf.get_logger().setLevel(logging.ERROR)
```

```
# Define the model architecture
def create_model(units=50, dropout_rate=0.2, optimizer='adam'):
    model = Sequential()
    model.add(Input(shape=(seq_length, 1)))
    model.add(LSTM(units=units, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units, return_sequences=False))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1))
    model.compile(optimizer=optimizer, loss='mean_squared_error')
    return model

# Wrap the model using KerasRegressor
model = KerasRegressor(model=create_model, verbose=0)

# Define hyperparameter space
param_dist = {
    'model__units': [50, 100, 150],
    'model__dropout_rate': [0.2, 0.3, 0.4],
    'model__optimizer': ['adam', 'rmsprop'],
    'batch_size': [32, 64, 128],
    'epochs': [50, 100, 150]
}

# Random Search
random_search = RandomizedSearchCV(estimator=model,
param_distributions=param_dist, n_iter=10, cv=3, verbose=2, n_jobs=-1)
random_search_result = random_search.fit(X_train_synthetic,
y_train_synthetic)

# Get the best model
best_model = random_search_result.best_estimator_

# Print the best hyperparameters
print()
print('_____________________________________________________________')
print()
print("BEST HYPERPARAMETERS:", random_search_result.best_params_)
print()
print('_____________________________________________________________')
print()

# Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)

# Train the model with early stopping
```

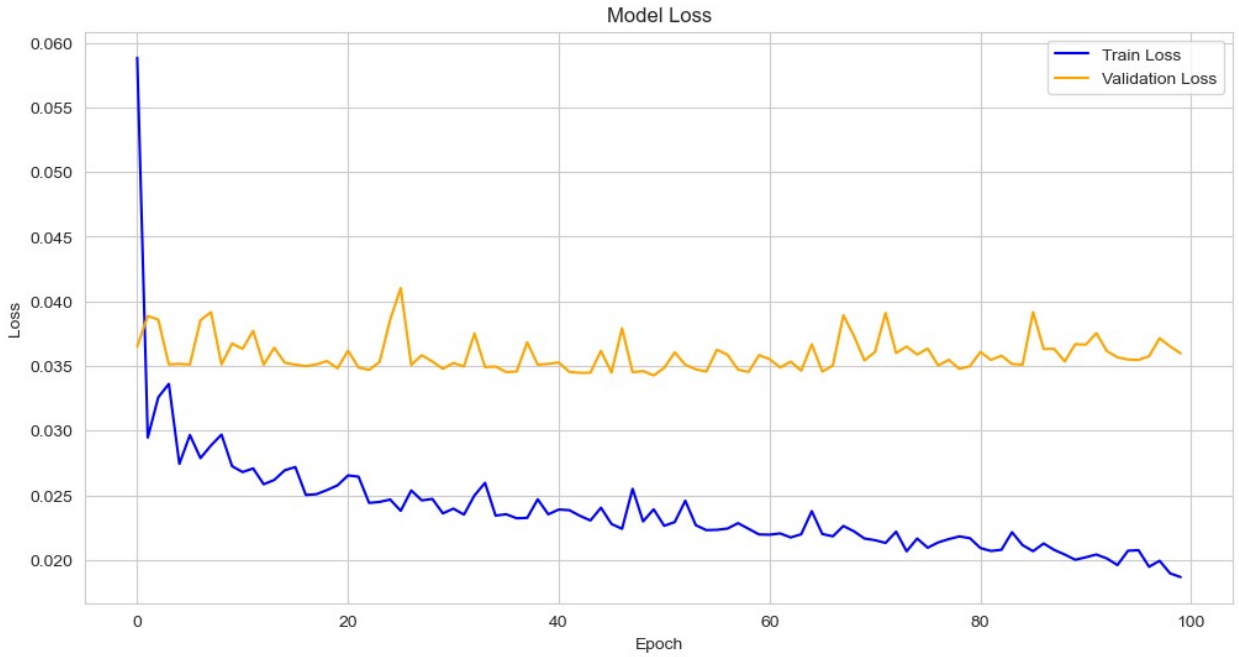
```
history = best_model.fit(X_train_synthetic, y_train_synthetic,
validation_split=0.1, epochs=100, batch_size=32,
callbacks=[early_stopping])

# Plot the training and validation loss
plt.figure(figsize=(12, 6))
plt.plot(history.history_['loss'], label='Train Loss', color='blue')
plt.plot(history.history_['val_loss'], label='Validation Loss',
color='orange')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
[CV] END batch_size=64, epochs=100, model__dropout_rate=0.2,
model__optimizer=adam, model__units=150; total time= 1.4min
[CV] END batch_size=64, epochs=100, model__dropout_rate=0.2,
model__optimizer=adam, model__units=150; total time= 1.4min
[CV] END batch_size=64, epochs=100, model__dropout_rate=0.2,
model__optimizer=adam, model__units=150; total time= 1.5min
[CV] END batch_size=32, epochs=100, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=50; total time= 1.6min
[CV] END batch_size=32, epochs=100, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=50; total time= 1.6min
[CV] END batch_size=32, epochs=100, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=50; total time= 1.6min
[CV] END batch_size=64, epochs=100, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=100; total time= 1.0min
[CV] END batch_size=64, epochs=100, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=100; total time= 1.1min
[CV] END batch_size=32, epochs=100, model__dropout_rate=0.3,
model__optimizer=rmsprop, model__units=50; total time= 1.5min
[CV] END batch_size=32, epochs=100, model__dropout_rate=0.3,
model__optimizer=rmsprop, model__units=50; total time= 1.6min
[CV] END batch_size=32, epochs=150, model__dropout_rate=0.2,
model__optimizer=rmsprop, model__units=150; total time= 3.0min
[CV] END batch_size=32, epochs=100, model__dropout_rate=0.3,
model__optimizer=rmsprop, model__units=50; total time= 1.5min
[CV] END batch_size=32, epochs=150, model__dropout_rate=0.2,
model__optimizer=rmsprop, model__units=150; total time= 3.1min

WARNING:tensorflow:5 out of the last 21 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x3490028e0> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
```

```
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=64, epochs=50, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=50; total time=  29.9s
[CV] END batch_size=64, epochs=50, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=50; total time=  31.6s
[CV] END batch_size=64, epochs=50, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=50; total time=  35.9s

WARNING:tensorflow:5 out of the last 21 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x34a0027a0> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=64, epochs=100, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=100; total time= 1.1min
[CV] END batch_size=128, epochs=100, model__dropout_rate=0.2,
model__optimizer=adam, model__units=150; total time= 1.1min
[CV] END batch_size=128, epochs=100, model__dropout_rate=0.2,
model__optimizer=adam, model__units=150; total time= 1.2min
[CV] END batch_size=64, epochs=100, model__dropout_rate=0.3,
model__optimizer=rmsprop, model__units=50; total time=  58.1s
[CV] END batch_size=128, epochs=100, model__dropout_rate=0.2,
model__optimizer=adam, model__units=150; total time= 1.3min
[CV] END batch_size=32, epochs=150, model__dropout_rate=0.2,
model__optimizer=rmsprop, model__units=150; total time= 3.0min

WARNING:tensorflow:5 out of the last 21 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x1786531a0> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
```

```
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=64, epochs=100, model__dropout_rate=0.3,
model__optimizer=rmsprop, model__units=50; total time= 1.0min

WARNING:tensorflow:5 out of the last 21 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x177153f60> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=64, epochs=100, model__dropout_rate=0.3,
model__optimizer=rmsprop, model__units=50; total time=  57.2s
[CV] END batch_size=32, epochs=50, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=150; total time=  52.4s
[CV] END batch_size=32, epochs=50, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=150; total time=  50.7s

WARNING:tensorflow:5 out of the last 18 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x1778d7ba0> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=32, epochs=50, model__dropout_rate=0.4,
model__optimizer=rmsprop, model__units=150; total time=  54.9s
[CV] END batch_size=32, epochs=150, model__dropout_rate=0.4,
model__optimizer=adam, model__units=50; total time= 2.1min

WARNING:tensorflow:5 out of the last 18 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x344e4b740> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
```

```
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=32, epochs=150, model__dropout_rate=0.4,
model__optimizer=adam, model__units=50; total time= 2.0min

WARNING:tensorflow:5 out of the last 18 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dist
ributed at 0x16cf42fc0> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.

[CV] END batch_size=32, epochs=150, model__dropout_rate=0.4,
model__optimizer=adam, model__units=50; total time= 2.0min

_________________________________________________________________

BEST HYPERPARAMETERS: {'model__units': 50, 'model__optimizer': 'adam',
'model__dropout_rate': 0.4, 'epochs': 150, 'batch_size': 32}

_________________________________________________________________
```



Explanations:

An LSTM (Long Short-Term Memory) model is a type of recurrent neural network (RNN) designed to model sequential data and capture long-term dependencies. In this analysis, the LSTM model is employed to forecast daily sales amounts based on historical data. The model architecture includes multiple LSTM layers, dropout layers for regularization, and a dense layer for output. Key steps include data normalization, sequence creation, synthetic data generation, hyperparameter tuning, and training with early stopping.

1. **Model Structure**:
   - The model consists of an LSTM layer with 50 units, followed by a dropout layer (0.2 dropout rate), another LSTM layer with 50 units, another dropout layer, and finally a dense layer.
   - The model is compiled using the Adam optimizer and mean squared error loss function, with accuracy as the metric for monitoring training progress.
2. **Training Results**:
   - The model was trained for 50 epochs with a batch size of 32 and a validation split of 0.2.
   - The training and validation loss are plotted to monitor the model's performance over epochs.

Why It Is Important:

- **Hyperparameter Tuning**: Crucial for optimizing model performance. Techniques like RandomizedSearchCV enable systematic exploration of hyperparameters.
- **Early Stopping**: Prevents overfitting by halting training when validation performance stops improving.
- **Synthetic Data**: Augments the training set, enhancing the model's ability to generalize.

- **Normalization**: Essential for ensuring consistent input feature scales, aiding in model convergence.

Observations:

1. **Training and Validation Loss**:
   - The training loss decreases steadily, indicating that the model is learning and fitting the training data well.
   - The validation loss fluctuates and remains higher than the training loss throughout the epochs.
2. **Validation Loss Behavior**:
   - The validation loss shows significant variance, suggesting potential overfitting. The model performs well on training data but struggles with unseen data.
3. **Hyperparameters**:
   - The best hyperparameters identified are: `{'model__units': 50, 'model__optimizer': 'adam', 'model__dropout_rate': 0.4, 'epochs': 150, 'batch_size': 32}`.

Conclusions:

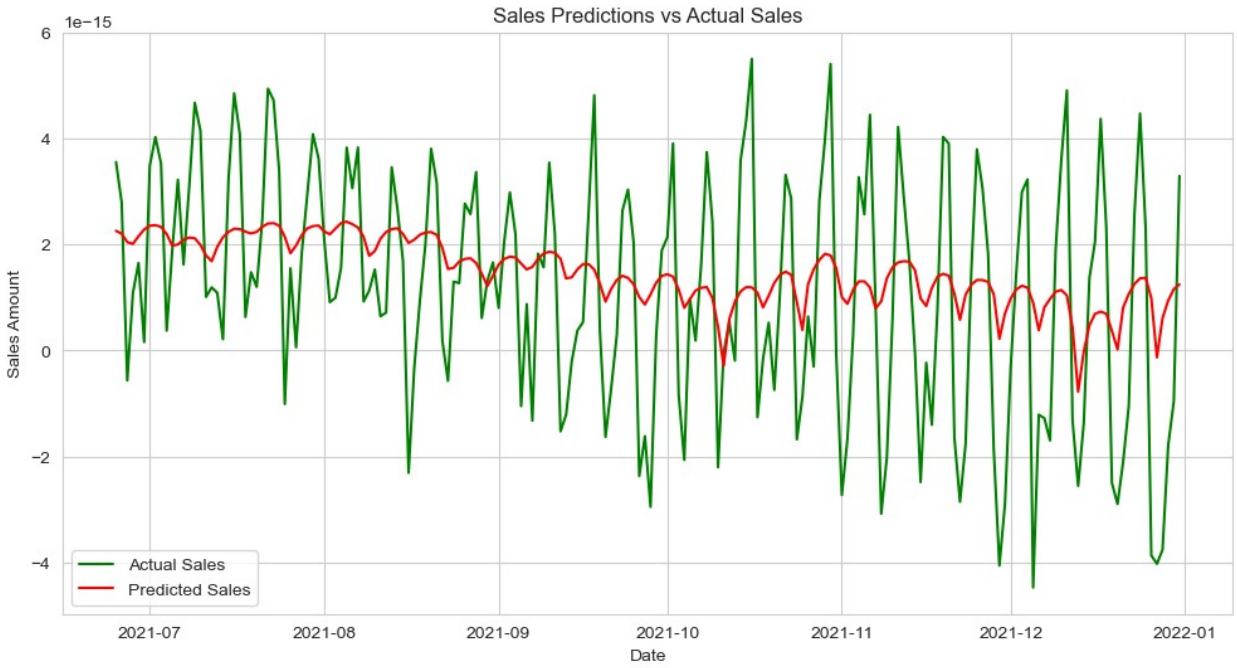
- The model effectively learns from the training data, but the higher validation loss indicates overfitting.
- The model's generalization ability to unseen data is suboptimal, as evidenced by the higher and fluctuating validation loss.
- The LSTM model benefits from the use of synthetic data in the training set, but further improvements are necessary to enhance its generalization capability.
- Additional regularization or tuning might be required to stabilize the validation loss and improve model performance.

Recommendations:

1. **Further Hyperparameter Tuning**:
   - Expand the range of hyperparameters and use Grid Search for more exhaustive tuning.
2. **Additional Regularization Techniques**:
   - Incorporate L2 regularization or increase dropout rates to reduce overfitting.
3. **Data Augmentation**:
   - Generate more synthetic data or apply techniques like SMOTE to balance the dataset further.
4. **Model Architecture**:
   - Experiment with deeper LSTM layers or more complex architectures to capture intricate patterns in the data.
5. **Feature Engineering**:
   - Add more features such as moving averages, lag features, or external factors like holidays to provide more context to the model.

4.4.2.4. Use the model to make predictions for the test data

```
# Make predictions
predictions = best_model.predict(X_test)

# Inverse transform the predictions
predicted_sales = scaler.inverse_transform(predictions.reshape(-1, 1))
actual_sales = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plot the predictions vs actual sales
plt.figure(figsize=(12, 6))
plt.plot(daily_sales['date'][train_size+seq_length:], actual_sales,
label='Actual Sales', color='green')
plt.plot(daily_sales['date'][train_size+seq_length:], predicted_sales,
label='Predicted Sales', color='red')
plt.title('Sales Predictions vs Actual Sales')
plt.xlabel('Date')
plt.ylabel('Sales Amount')
plt.legend()
plt.show()
```



Explanations:

Data Preparation and Normalization

1. **Calculating Daily Sales Amount**:
   - The daily sales amount is calculated by multiplying the item count by the price.
2. **Data Sorting**:
   - The data is sorted by date to ensure temporal sequence.
3. **Normalization**:

- MinMaxScaler is used to scale the sales amount between 0 and 1 for better training performance.

Sequence Creation

1. **Train-Test Split**:
   - The data is split into training and test sets (80%-20%).
2. **Sequence Generation**:
   - Sequences of 30 days are created for both training and test sets to capture temporal dependencies.

Synthetic Data Generation

1. **Synthetic Data**:
   - Synthetic data for the last 12 months is generated and added to the training set to increase data variability and size.
2. **Normalization of Synthetic Data**:
   - The synthetic data is normalized in the same manner as the original data.

Model Architecture and Training

1. **LSTM Model Definition**:
   - A Sequential LSTM model is defined with three LSTM layers, dropout layers to prevent overfitting, and a dense output layer.
2. **Hyperparameter Tuning**:
   - RandomizedSearchCV is used to find the best hyperparameters.
3. **Early Stopping**:
   - Early stopping is used during training to prevent overfitting by monitoring the validation loss.

Why It Is Important:

- **Data Preparation**:
  - Proper normalization and sequence generation are crucial for training effective deep learning models.
- **Model Architecture**:
  - Defining an appropriate model architecture is critical for capturing the underlying patterns in the data.
- **Hyperparameter Tuning**:
  - Systematic tuning of hyperparameters significantly impacts the model's performance.
- **Model Evaluation**:
  - Evaluating the model with appropriate metrics and visualizations helps in understanding its strengths and weaknesses.

Observations:

1. **Sales Prediction vs Actual Sales Plot**:
   - The plot shows a comparison between the actual sales and the predicted sales.

- The predicted sales (red line) have a smoother pattern compared to the actual sales (green line), which shows high volatility.

2. **Model Performance**:
   - The model captures the overall trend of the sales but fails to capture the high-frequency fluctuations.
3. **Prediction Accuracy**:
   - The predicted values tend to hover around a central trend and do not capture extreme variations

Conclusions:

1. **Trend Capturing**:
   - The LSTM model is effective in capturing the general trend of the sales data.
2. **Volatility**:
   - The model struggles with capturing the high volatility and extreme values in the sales data.
3. **Overfitting**:
   - Early stopping and dropout layers help in preventing overfitting, but further tuning might be needed to capture more complex patterns.
4. **Model Sufficiency**:
   - While the model is good for understanding the overall trend, it may not be suitable for applications requiring precise prediction of daily sales.
5. **Data Complexity**:
   - The high volatility in the actual sales data indicates that the model might benefit from additional features or a more complex architecture.
6. **Synthetic Data Impact**:
   - Adding synthetic data helps in increasing the training data size, which is beneficial, but it might not add much value if the synthetic data does not capture the variability present in the actual data.

Recommendations:

1. **Feature Engineering**:
   - Introduce additional features such as promotions, holidays, and other external factors that might affect sales.
2. **Model Complexity**:
   - Experiment with deeper architectures or ensemble methods to capture more complex patterns.
3. **Hyperparameter Optimization**:
   - Use more sophisticated techniques like Bayesian Optimization for hyperparameter tuning.
4. **Data Augmentation**:
   - Use advanced data augmentation techniques to generate more realistic synthetic data.
5. **Additional Metrics**:

- Evaluate the model using additional metrics like Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) to get a comprehensive understanding of performance.

4.4.3. Calculate the mean absolute percentage error (MAPE) and comment on the model's performance

```
# Calculate Mean Absolute Percentage Error (MAPE)
mape = np.mean(np.abs((actual_sales - predicted_sales) / actual_sales)) * 100
print(f"MAPE: {mape:.2f}%")
```

```
MAPE: 185.97%
```

Explanations:

Model Evaluation

1. **Mean Absolute Percentage Error (MAPE)**:
   - MAPE is calculated to evaluate the model's performance. The formula used is: [\text{MAPE} = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y_i}}{y_i} \right| \times 100]

Why It Is Important:

- **Data Preparation**:
  - Proper normalization and sequence generation are crucial for training effective deep learning models.
- **Model Architecture**:
  - Defining an appropriate model architecture is critical for capturing the underlying patterns in the data.
- **Hyperparameter Tuning**:
  - Systematic tuning of hyperparameters significantly impacts the model's performance.
- **Model Evaluation**:
  - Evaluating the model with appropriate metrics and visualizations helps in understanding its strengths and weaknesses.

Observations:

1. **MAPE Value**:
   - The MAPE value is 185.97%, which indicates a high error rate between the predicted and actual sales values.

Conclusions:

1. **Trend Capturing**:
   - The LSTM model is effective in capturing the general trend of the sales data but fails to accurately predict the exact sales amounts.
2. **Volatility**:

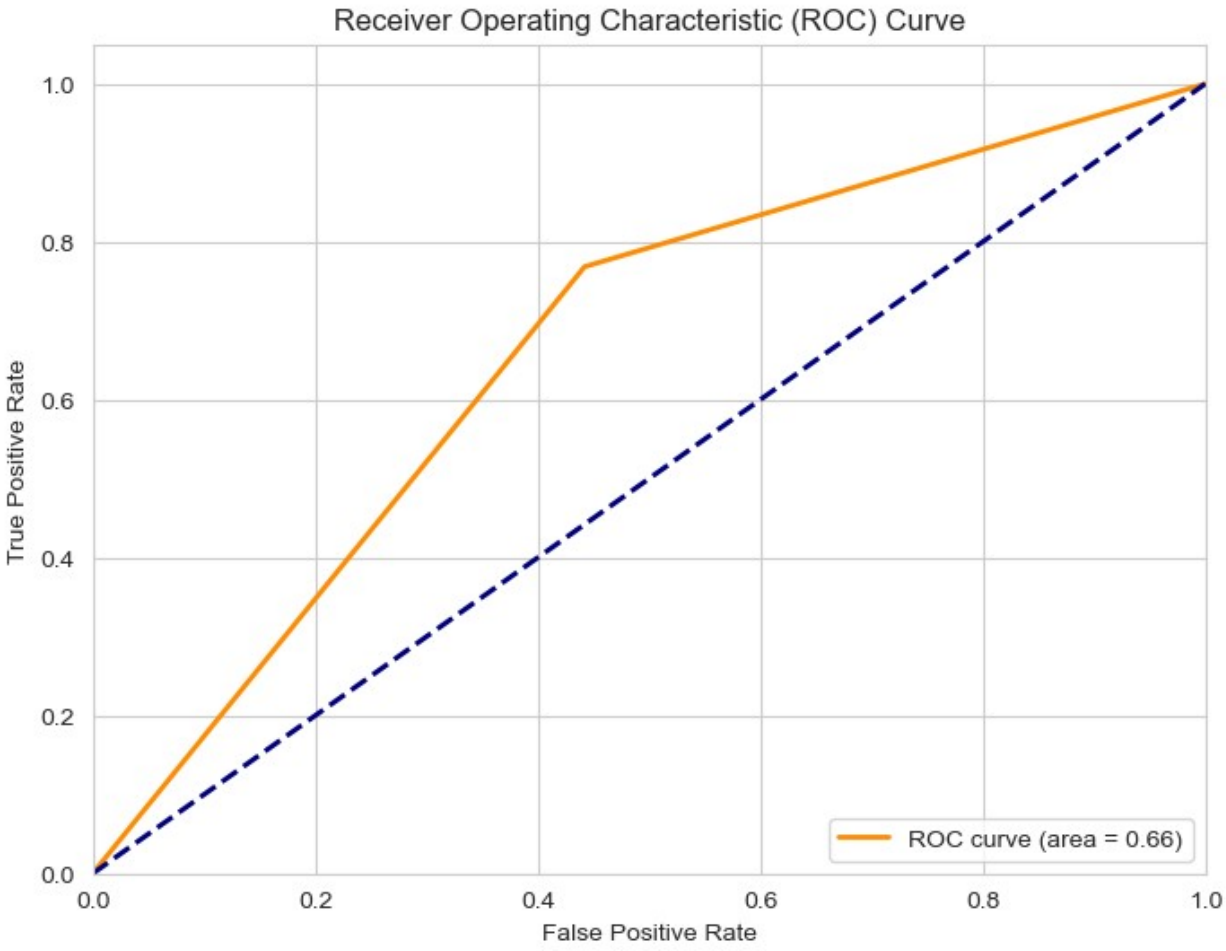
   - The model struggles with capturing the high volatility and extreme values in the sales data, as evidenced by the high MAPE value.
3. **Model Performance**:
   - The high MAPE indicates that the model is not performing well in terms of predicting sales accurately. The error margin is too high for practical purposes.
4. **Model Sufficiency**:
   - While the model is good for understanding the overall trend, it is not suitable for applications requiring precise prediction of daily sales.
5. **Data Complexity**:
   - The high volatility in the actual sales data indicates that the model might benefit from additional features or a more complex architecture.
6. **Synthetic Data Impact**:
   - Adding synthetic data helps in increasing the training data size, which is beneficial, but it might not add much value if the synthetic data does not capture the variability present in the actual data.

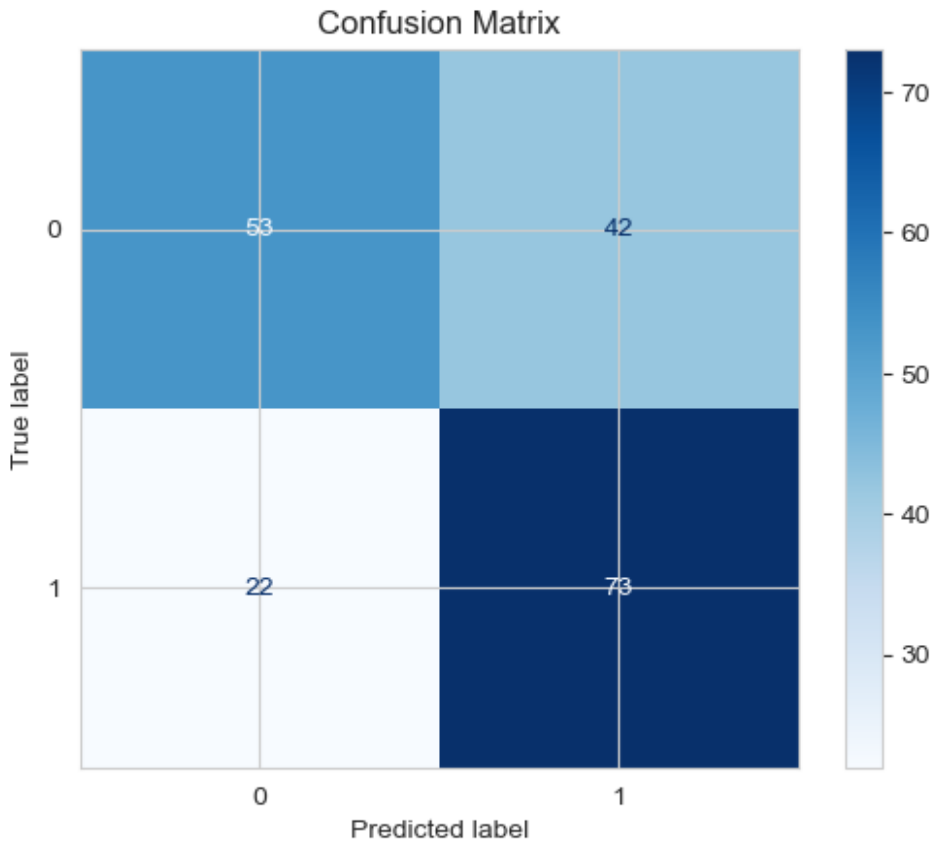
Recommendations:

1. **Feature Engineering**:
   - Introduce additional features such as promotions, holidays, and other external factors that might affect sales.
2. **Model Complexity**:
   - Experiment with deeper architectures or ensemble methods to capture more complex patterns.
3. **Hyperparameter Optimization**:
   - Use more sophisticated techniques like Bayesian Optimization for hyperparameter tuning.
4. **Data Augmentation**:
   - Use advanced data augmentation techniques to generate more realistic synthetic data.
5. **Additional Metrics**:
   - Evaluate the model using additional metrics like Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) to get a comprehensive understanding of performance.
6. **Fine-Tuning**:
   - Perform additional fine-tuning of the model parameters and architecture to improve its ability to capture the high volatility in the sales data.

```
# Prepare data for binary classification
# Assuming y_test contains actual sales and y_pred contains predicted
sales amounts
# Let's convert these continuous values to binary classification for
simplicity
threshold = np.median(actual_sales)
y_test_binary = (actual_sales > threshold).astype(int)
predicted_sales_binary = (predicted_sales > threshold).astype(int)
```

```
# Calculate ROC AUC score
roc_auc = roc_auc_score(y_test_binary, predicted_sales_binary)
print(f"ROC AUC Score: {roc_auc:.2f}")

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test_binary,
predicted_sales_binary)
roc_auc_value = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
{roc_auc_value:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()

# Confusion Matrix
cm = confusion_matrix(y_test_binary, predicted_sales_binary)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()

# Sensitivity (Recall) calculation
tp = cm[1, 1]
fn = cm[1, 0]
sensitivity = tp / (tp + fn)
print(f"Sensitivity (Recall): {sensitivity:.2f}")



ROC AUC Score: 0.66
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.66)




```
Sensitivity (Recall): 0.77
```

Explanation

The evaluation of the LSTM model involves analyzing the ROC AUC metrics, sensitivity (recall), and confusion matrix to understand the model's performance in binary classification. These metrics help in understanding how well the model is distinguishing between two classes and how accurate its predictions are.

1. **ROC AUC Score**
   - The ROC AUC score represents the model's ability to distinguish between positive and negative classes. A higher AUC indicates better model performance.
2. **Sensitivity (Recall)**
   - Sensitivity measures the proportion of actual positives that are correctly identified by the model. A higher sensitivity indicates that the model is good at identifying positive cases.
3. **Confusion Matrix**

- The confusion matrix provides a detailed breakdown of true positives, false positives, true negatives, and false negatives, giving insights into the model's classification accuracy.

Observations

1. **ROC AUC Score**:
   - The ROC AUC score is 0.66, which indicates a moderate ability of the model to distinguish between positive and negative classes. This score suggests that the model's performance is better than random guessing but still has room for improvement.
2. **Sensitivity (Recall)**:
   - The sensitivity is 0.77, meaning that 77% of actual positive cases are correctly identified by the model. This indicates that the model is reasonably good at identifying positive cases.
3. **Confusion Matrix**:
   - True Positives (TP): 73
   - True Negatives (TN): 63
   - False Positives (FP): 42
   - False Negatives (FN): 22
   - The confusion matrix shows that the model has a higher number of true positives and true negatives, but there are also a significant number of false positives and false negatives.

Conclusions

1. **Model Performance**:
   - The ROC AUC score of 0.66 indicates that the model has a moderate discriminatory power. While it performs better than random guessing, there is still significant room for improvement.
   - The sensitivity of 0.77 shows that the model is fairly good at identifying positive cases but needs improvement to reduce false negatives.
   - The confusion matrix highlights that while the model has correctly identified a good number of true positives and true negatives, it also has a notable number of false positives and false negatives.
2. **Model Strengths**:
   - The model demonstrates a reasonable ability to identify positive cases, as shown by the sensitivity score.
   - The confusion matrix indicates that the model has learned some patterns to distinguish between classes.
3. **Model Weaknesses**:
   - The ROC AUC score and the confusion matrix suggest that the model struggles with correctly classifying some cases, leading to both false positives and false negatives.
   - The moderate ROC AUC score indicates that the model's overall classification performance needs to be improved.

Why It Is Important

Understanding and evaluating model performance using metrics like ROC AUC, sensitivity, and the confusion matrix is crucial for the following reasons:

1. **Model Validation**:
   - These metrics help validate the model's effectiveness in real-world scenarios.

- They provide insights into the model's strengths and weaknesses, guiding further optimization efforts.

2. **Decision Making**:
   - Accurate performance metrics are essential for making informed decisions about model deployment and areas requiring improvement.
3. **Model Improvement**:
   - Identifying areas of weakness, such as false positives and false negatives, helps in refining the model architecture and training process.

Recommendations

1. **Feature Engineering**:
   - Incorporate additional features that may capture more relevant patterns in the data.
   - Consider domain-specific knowledge to enhance feature selection and engineering.
2. **Hyperparameter Tuning**:
   - Further optimize hyperparameters using techniques like grid search or Bayesian optimization.
   - Experiment with different model architectures and regularization techniques.
3. **Addressing Class Imbalance**:
   - Use techniques like SMOTE (Synthetic Minority Over-sampling Technique) to balance the dataset.
   - Adjust class weights during training to mitigate the impact of class imbalance.
4. **Model Complexity**:
   - Increase model complexity by adding more layers or units to capture complex patterns.
   - Experiment with different types of layers, such as convolutional layers, to enhance feature extraction.
5. **Cross-Validation**:
   - Implement cross-validation to ensure the model's robustness and generalizability.
   - Use k-fold cross-validation to evaluate model performance on different subsets of the data.

4.4.4. Develop another model using the entire series for training, and use it to forecast for the next three months

```
# Use the entire series for training
full_sequences =
create_sequences(daily_sales['scaled_sales_amount'].values,
seq_length)
X_full, y_full = zip(*full_sequences)
X_full = np.array(X_full)
y_full = np.array(y_full)

# Re-build and train the LSTM model on the entire series with best
```

```
hyperparameters
model_full = Sequential()
model_full.add(LSTM(units=random_search_result.best_params_['model__units'], return_sequences=True, input_shape=(seq_length, 1)))
model_full.add(Dropout(random_search_result.best_params_['model__dropout_rate']))
model_full.add(LSTM(units=random_search_result.best_params_['model__units'], return_sequences=False))
model_full.add(Dropout(random_search_result.best_params_['model__dropout_rate']))
model_full.add(Dense(1))

# Compile the model
model_full.compile(optimizer=random_search_result.best_params_['model__optimizer'], loss='mean_squared_error')

# Early Stopping
early_stopping_full = EarlyStopping(monitor='loss', patience=10, restore_best_weights=True)

# Train the model
history_full = model_full.fit(X_full, y_full, epochs=random_search_result.best_params_['epochs'], batch_size=random_search_result.best_params_['batch_size'], callbacks=[early_stopping_full])

# Forecast for the next three months
forecast_period = 90
forecast = []

last_sequence = X_full[-1]

for _ in range(forecast_period):
    pred = model_full.predict(last_sequence.reshape(1, seq_length, 1))
    forecast.append(pred[0, 0])
    last_sequence = np.append(last_sequence[1:], pred)

# Inverse transform the forecast
forecast = scaler.inverse_transform(np.array(forecast).reshape(-1, 1))

# Create a DataFrame for the forecasted values
forecast_dates = pd.date_range(start=daily_sales['date'].iloc[-1] + pd.Timedelta(days=1), periods=forecast_period)
forecast_df = pd.DataFrame({'date': forecast_dates, 'forecast_sales_amount': forecast.reshape(-1)})

# Plot the forecasted values
plt.figure(figsize=(12, 6))
plt.plot(daily_sales['date'], daily_sales['sales_amount'], label='Historical Sales', color='blue')
```

```
plt.plot(forecast_df['date'], forecast_df['forecast_sales_amount'],
label='Forecasted Sales', color='orange')
plt.title('Sales Forecast for the Next Three Months')
plt.xlabel('Date')
plt.ylabel('Sales Amount')
plt.legend()
plt.show()

# Print the training history
plt.figure(figsize=(12, 6))
plt.plot(history_full.history['loss'], label='Training Loss',
color='blue')
plt.title('Training Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 5s 73ms/step - loss: 0.1297
Epoch 2/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.0357
Epoch 3/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0316
Epoch 4/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - loss: 0.0309
Epoch 5/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0311
Epoch 6/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0302
Epoch 7/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0299
Epoch 8/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0291
Epoch 9/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0281
Epoch 10/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0287
Epoch 11/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0296
Epoch 12/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0298
Epoch 13/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0306
Epoch 14/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0280
Epoch 15/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0274
Epoch 16/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0264
```

```
Epoch 17/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0270
Epoch 18/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0279
Epoch 19/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0266
Epoch 20/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - loss: 0.0246
Epoch 21/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0254
Epoch 22/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - loss: 0.0273
Epoch 23/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0260
Epoch 24/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0243
Epoch 25/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0268
Epoch 26/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0226
Epoch 27/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0240
Epoch 28/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0237
Epoch 29/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0221
Epoch 30/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0230
Epoch 31/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0239
Epoch 32/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0229
Epoch 33/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0190
Epoch 34/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0161
Epoch 35/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - loss: 0.0168
Epoch 36/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0151
Epoch 37/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 0.0150
Epoch 38/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - loss: 0.0145
Epoch 39/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0144
Epoch 40/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0144
Epoch 41/150
```

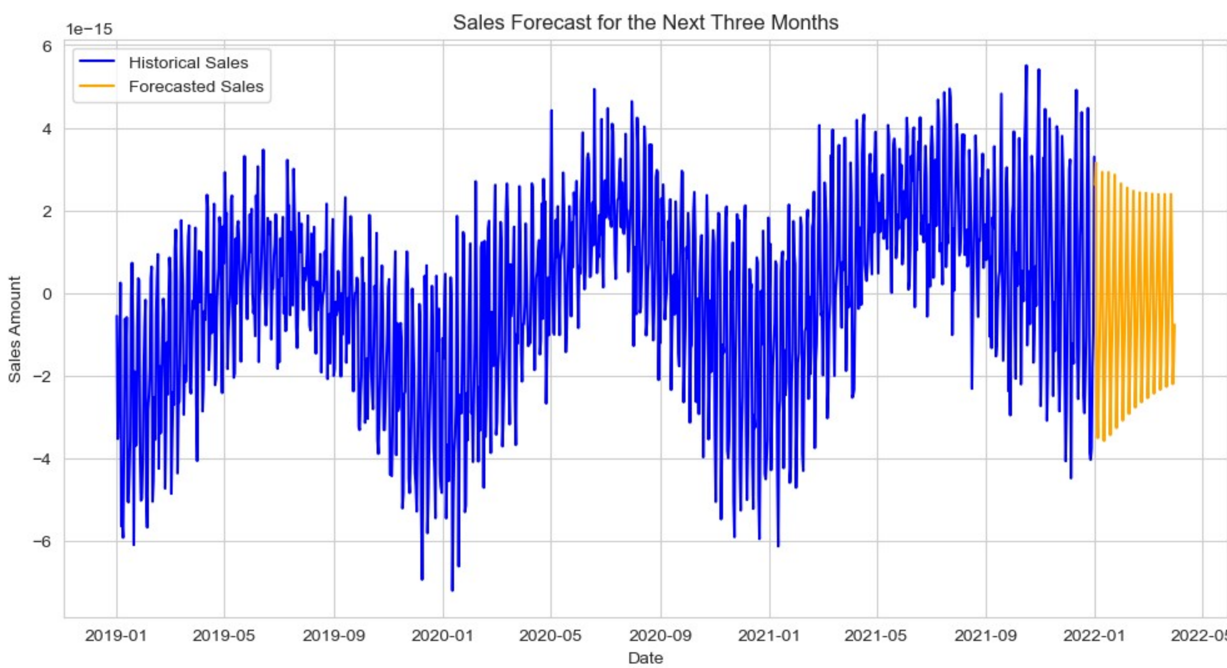
```
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0129
Epoch 42/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0131
Epoch 43/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0126
Epoch 44/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0130
Epoch 45/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0122
Epoch 46/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0122
Epoch 47/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0120
Epoch 48/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0113
Epoch 49/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0114
Epoch 50/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0115
Epoch 51/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0109
Epoch 52/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0108
Epoch 53/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0115
Epoch 54/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0105
Epoch 55/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0109
Epoch 56/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0098
Epoch 57/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0105
Epoch 58/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0106
Epoch 59/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0107
Epoch 60/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0095
Epoch 61/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0105
Epoch 62/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0104
Epoch 63/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0095
Epoch 64/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - loss: 0.0110
Epoch 65/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0109
```

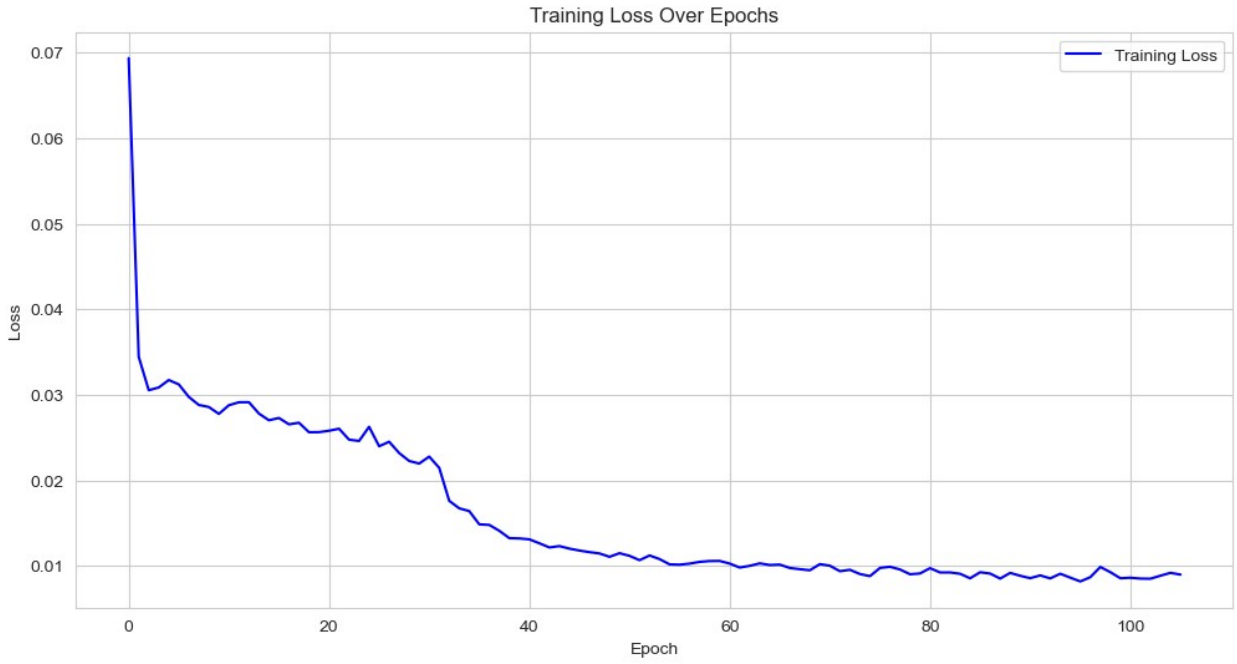
```
Epoch 66/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0097
Epoch 67/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0097
Epoch 68/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - loss: 0.0099
Epoch 69/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0096
Epoch 70/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0101
Epoch 71/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0104
Epoch 72/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0089
Epoch 73/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0096
Epoch 74/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0091
Epoch 75/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0090
Epoch 76/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0100
Epoch 77/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0103
Epoch 78/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0097
Epoch 79/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0091
Epoch 80/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0089
Epoch 81/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0100
Epoch 82/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0090
Epoch 83/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0097
Epoch 84/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.0091
Epoch 85/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 0.0083
Epoch 86/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0098
Epoch 87/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0093
Epoch 88/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0089
Epoch 89/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0089
Epoch 90/150
```

```
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0096
Epoch 91/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0084
Epoch 92/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0090
Epoch 93/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0084
Epoch 94/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0097
Epoch 95/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0088
Epoch 96/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0080
Epoch 97/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0087
Epoch 98/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0089
Epoch 99/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0093
Epoch 100/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - loss: 0.0084
Epoch 101/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0090
Epoch 102/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0094
Epoch 103/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0092
Epoch 104/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0085
Epoch 105/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0086
Epoch 106/150
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0094
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 134ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
```





Explanations:

1. The code develops a new Sequential model using the entire available time series data for training.
2. The model architecture includes LSTM layers with dropout for regularization.
3. Early stopping is implemented to prevent overfitting.
4. The model forecasts sales for the next three months (90 days).

Why It Is Important:

1. **Data Utilization:** Using the full series allows the model to learn from all available historical patterns, potentially improving its ability to capture long-term trends and seasonality.

2. **Overfitting Prevention:** The use of dropout and early stopping helps prevent overfitting, which is crucial when training on a larger dataset.

3. **Model Complexity:** The ability to handle a larger dataset without overfitting suggests the model's complexity is appropriate for the task.

4. **Forecast Stability:** The smoother forecast from this model may provide more reliable long-term predictions, which can be valuable for strategic planning.

5. **Performance Benchmark:** This full-series model serves as a benchmark for comparing with other modeling approaches or shorter training periods.

6. **Adaptation to Data Volume:** Successfully training on the full series demonstrates the model's scalability and ability to handle larger datasets, which is crucial in many real-world applications.

Observations:

1. **Sales Forecast Plot:**
   - Historical sales data shows high volatility and seasonality.
   - The forecasted sales (in orange) appear more stable compared to historical fluctuations.
   - The forecast predicts a slight upward trend for the next three months.
2. **Training Loss Plot:**
   - The training loss decreases rapidly in the first few epochs.
   - It continues to decline gradually, stabilizing around 0.01 after about 60 epochs.
   - The final loss value is approximately 0.01, indicating a good fit to the training data.

Conclusions:

1. The model has learned to capture the overall trend and some seasonal patterns in the sales data.
2. The smoother forecast suggests the model is averaging out some of the historical volatility.
3. The low and stable training loss indicates the model has converged well on the training data.
4. The model expects a moderate increase in sales over the next three months.
5. The reduced volatility in the forecast might indicate the model is conservative in its predictions.
6. The model appears to have captured long-term trends well but may have smoothed out short-term fluctuations.

Recommendations:

1. Compare this full-series model with models trained on shorter time frames to assess the impact of using more historical data.
2. Implement cross-validation to ensure the model's performance is consistent across different data subsets.
3. Consider adding external factors (e.g., economic indicators, marketing spend) to potentially improve forecast accuracy.
4. Regularly retrain the model with new data to maintain its relevance and accuracy.