Retail Sales Predictions

1. Introduction

📊 About the project

In retail, being able to predict sales accurately is essential for making smart business decisions. This project focuses on building models to forecast sales by using a dataset that includes various factors affecting sales, such as discounts, marketing spend, and seasonal trends like holidays.

This project will not only predict sales but also evaluate the performance of different models such as linear regression, decision trees, and gradient boosting methods, among others. The comparison will help identify the most effective model for improving sales forecasting accuracy in real-world retail settings.

🔑 Key Objectives:

- Build and compare different models to predict retail sales.
- Analyze how features like marketing, discounts, and holidays affect sales.
- Find the most effective model for forecasting future sales.

2. Importing Necessary Libraries

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

from scipy import stats

from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.model_selection import RandomizedSearchCV

from sklearn.linear_model import LinearRegression, SGDRegressor, ElasticNet
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor
from xgboost import XGBRegressor

import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import MeanSquaredError

# Suppress warnings
import warnings
warnings.filterwarnings('ignore')

custom_palette = ['#1560B0', '#C33B2F', '#EAD59F', '#6AA8CD', '#D69642', '#D6E5EE', '#695D5E']
sns.set_theme(context='notebook', palette=custom_palette, style='whitegrid')
```

3. Loading and Understanding Data

In [ ]:
```
file_path = 'Retail_sales.csv'
df = pd.read_csv(file_path)
```

In [3]:
```
df.head()
```

Out[3]:

	Store ID	Product ID	Date	Units Sold	Sales Revenue (USD)	Discount Percentage	Marketing Spend (USD)	Store Location	Product Category	Day of the Week	Holiday Effect
0	Spearsland	52372247	2022-01-01	9	2741.69	20	81	Tanzania	Furniture	Saturday	False
1	Spearsland	52372247	2022-01-02	7	2665.53	0	0	Mauritania	Furniture	Sunday	False
2	Spearsland	52372247	2022-01-03	1	380.79	0	0	Saint Pierre and Miquelon	Furniture	Monday	False
3	Spearsland	52372247	2022-01-04	4	1523.16	0	0	Australia	Furniture	Tuesday	False
4	Spearsland	52372247	2022-01-05	2	761.58	0	0	Swaziland	Furniture	Wednesday	False

In [4]:
```
df.shape
```

Out[4]:
```
(30000, 11)
```

In [5]:
```
df.describe()
```

Out[5]:

	Product ID	Units Sold	Sales Revenue (USD)	Discount Percentage	Marketing Spend (USD)
count	3.000000e+04	30000.000000	30000.000000	30000.000000	30000.000000
mean	4.461294e+07	6.161967	2749.509593	2.973833	49.944033
std	2.779759e+07	3.323929	2568.639288	5.974530	64.401655
min	3.636541e+06	0.000000	0.000000	0.000000	0.000000
25%	2.228600e+07	4.000000	882.592500	0.000000	0.000000
50%	4.002449e+07	6.000000	1902.420000	0.000000	1.000000
75%	6.559352e+07	8.000000	3863.920000	0.000000	100.000000
max	9.628253e+07	56.000000	27165.880000	20.000000	199.000000

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 11 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Store ID              30000 non-null  object 
 1   Product ID            30000 non-null  int64  
 2   Date                  30000 non-null  object 
 3   Units Sold            30000 non-null  int64  
 4   Sales Revenue (USD)   30000 non-null  float64
 5   Discount Percentage   30000 non-null  int64  
 6   Marketing Spend (USD) 30000 non-null  int64  
 7   Store Location        30000 non-null  object 
 8   Product Category      30000 non-null  object 
 9   Day of the Week       30000 non-null  object 
 10  Holiday Effect        30000 non-null  bool   
dtypes: bool(1), float64(1), int64(4), object(5)
memory usage: 2.3+ MB
```

In [7]:
```
# Check for missing values
df.isnull().sum()
```

Out[7]:
```
Store ID                0
Product ID              0
Date                    0
Units Sold              0
Sales Revenue (USD)     0
Discount Percentage     0
Marketing Spend (USD)   0
Store Location          0
Product Category        0
Day of the Week         0
Holiday Effect          0
dtype: int64
```

In [8]:
```
df['Store ID'].nunique()
```

Out[8]:
```
1
```

In [9]:
```
df['Store Location'].nunique()
```

Out[9]:
```
243
```

In [10]:
```
df['Product ID'].nunique()
```

Out[10]:
```
42
```

In [11]:
```
df['Product Category'].nunique()
```

Out[11]:
```
4
```

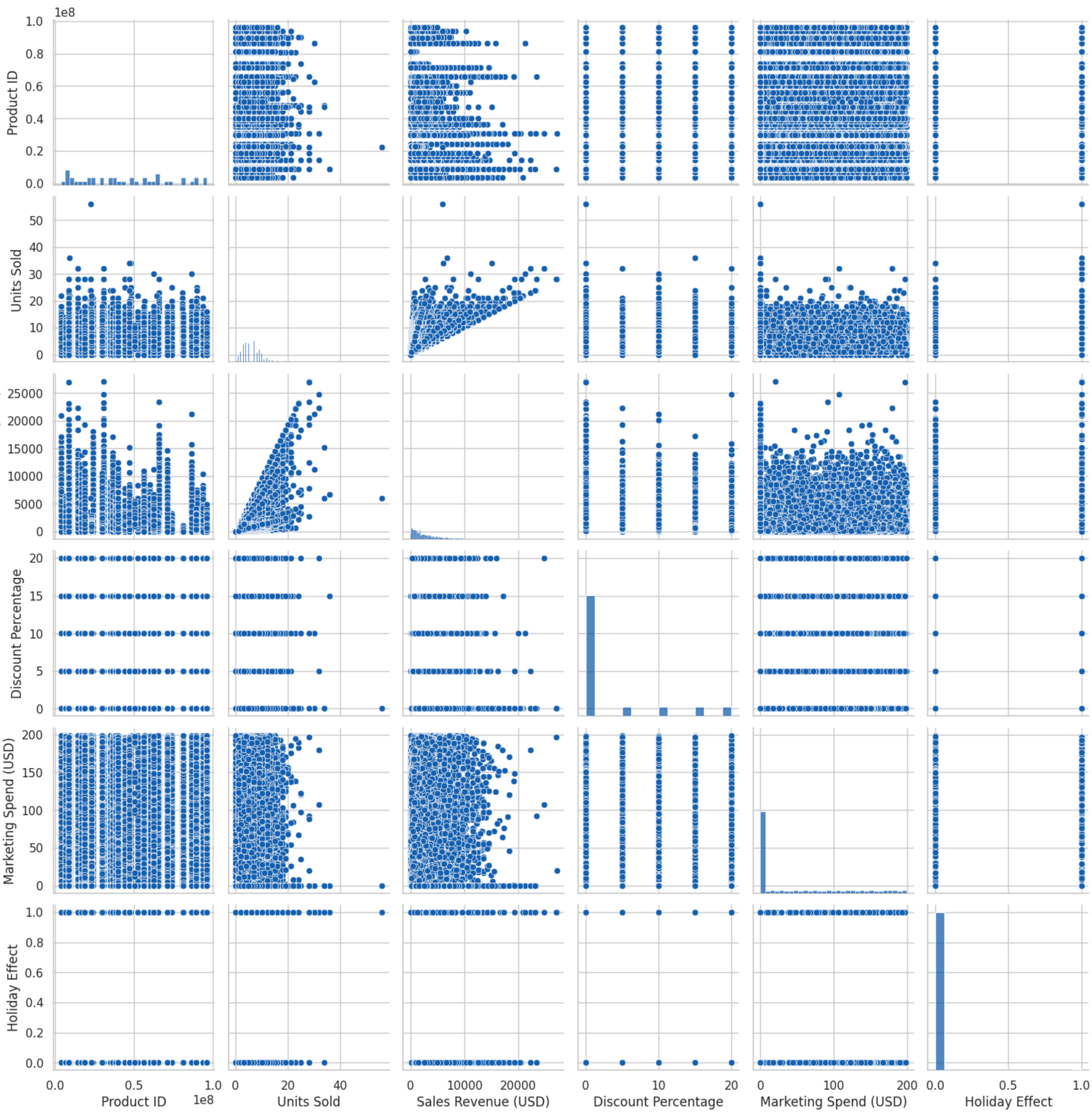
Initial data exploration reveals the following key points:

- 📊 **Observations:** The dataset consists of 30,000 rows and 11 columns.
- ⚠️ **Missing Values:** No missing values are present in the dataset.
- 📊 **Column Types:** The dataset includes a mix of integers, floats, objects, and boolean values.
- 📉 **Marketing and Discounts:** Most days had no marketing spend or discounts applied.
- 💰 **Maximum Values:** The highest marketing spend was $199, and the largest discount offered was 20%.
- 🏪 **Store ID:** There is only one unique value in 'Store ID', so this column can be dropped.
- 🔢 **Product ID:** While 'Product ID' is numerical, it contains 42 unique values.
- 🌍 **Product Location:** There are 243 unique product locations in the dataset.

4. Exploratory Data Analysis

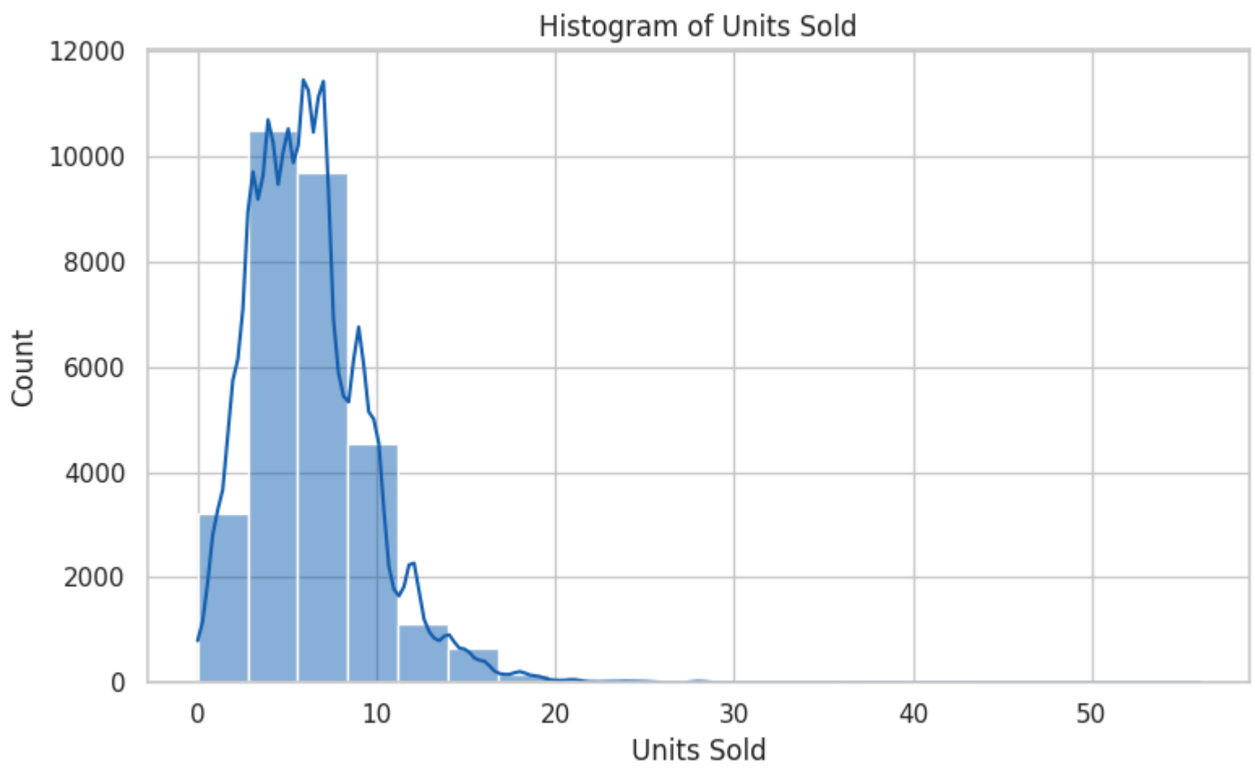
✨ Pairplot

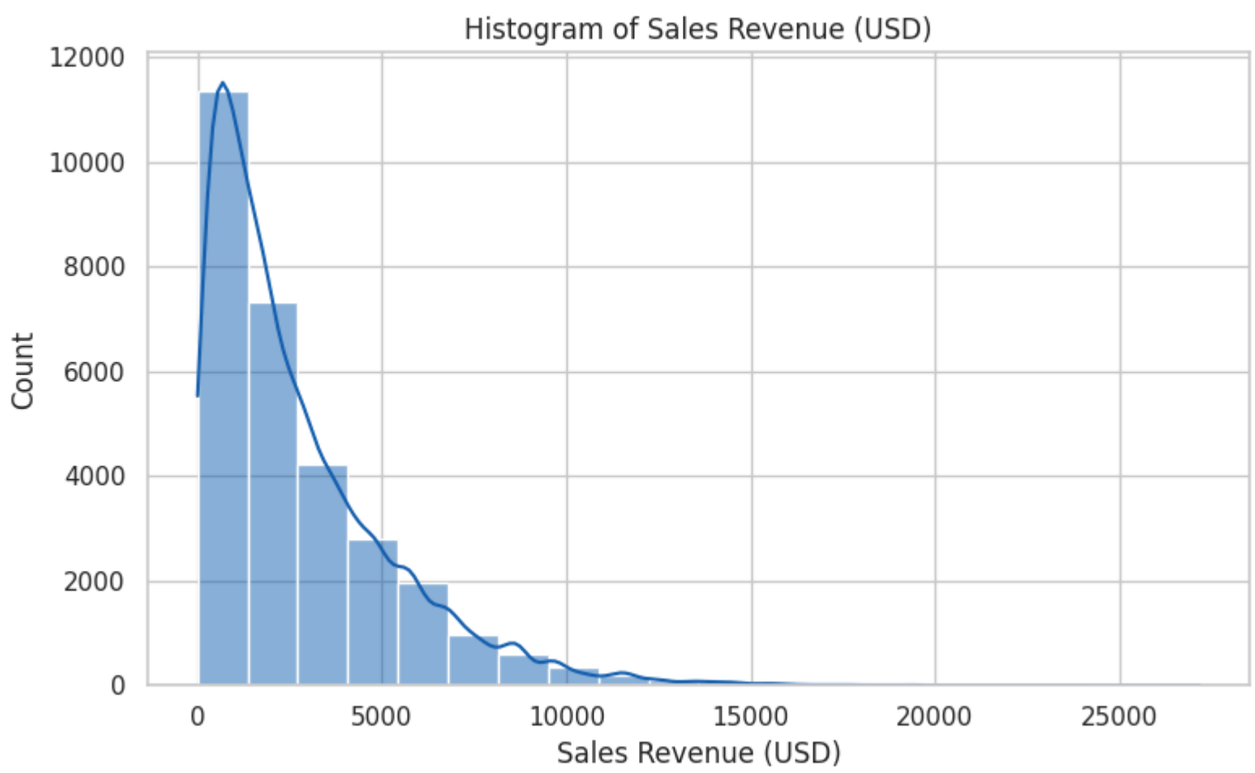
In [12]:
```
sns.pairplot(df)
plt.show()
```

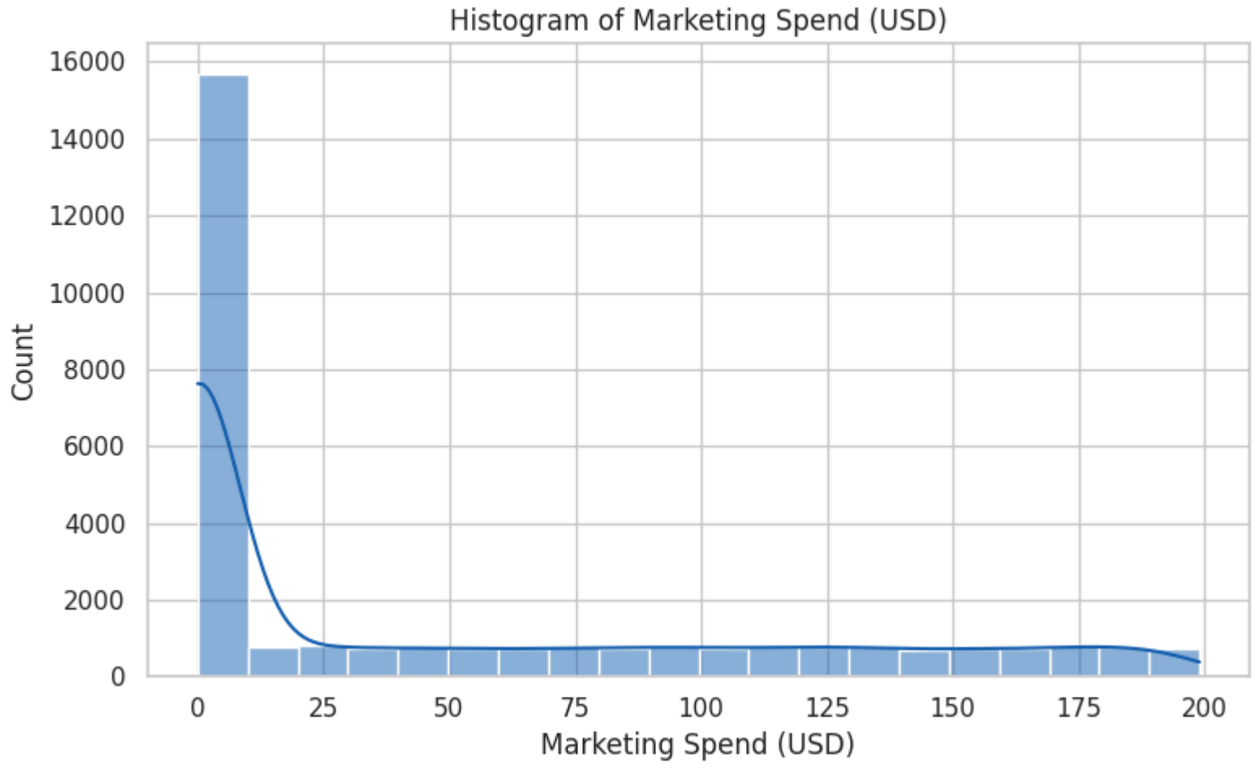


Histograms

In [13]:
```
numerical_cols_to_plot = ['Units Sold', 'Sales Revenue (USD)','Marketing Spend (USD)']

# Plot histograms for each numerical column
for column in numerical_cols_to_plot:

    plt.figure(figsize=(8, 5))
    sns.histplot(data=df, x=column, kde=True, bins=20)
    plt.title(f'Histogram of {column}')

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

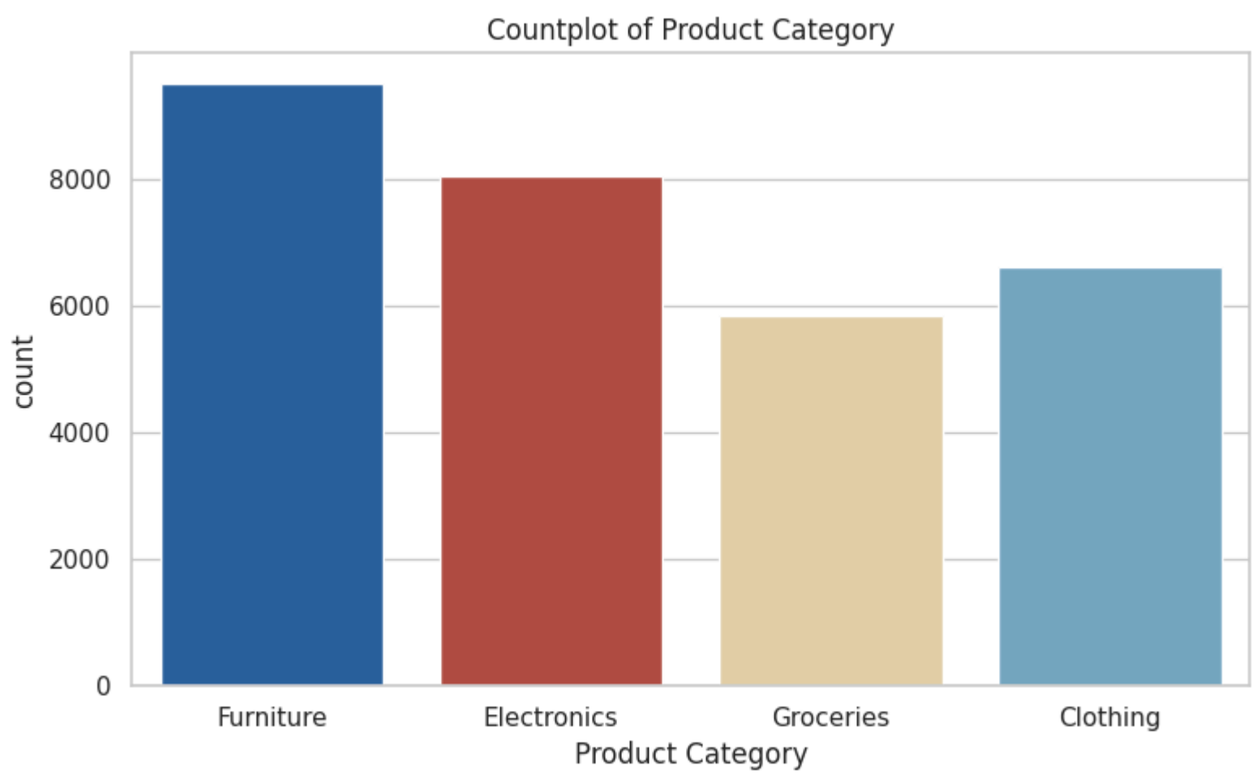


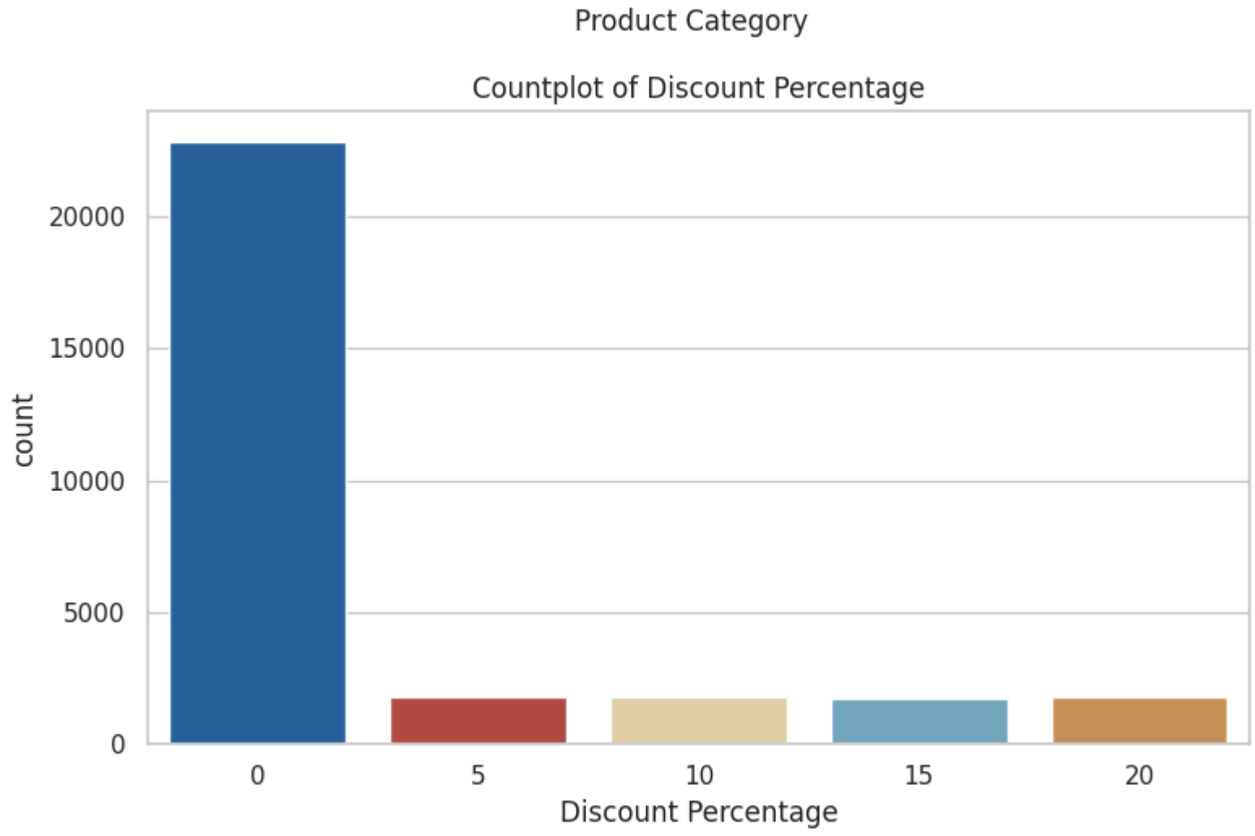


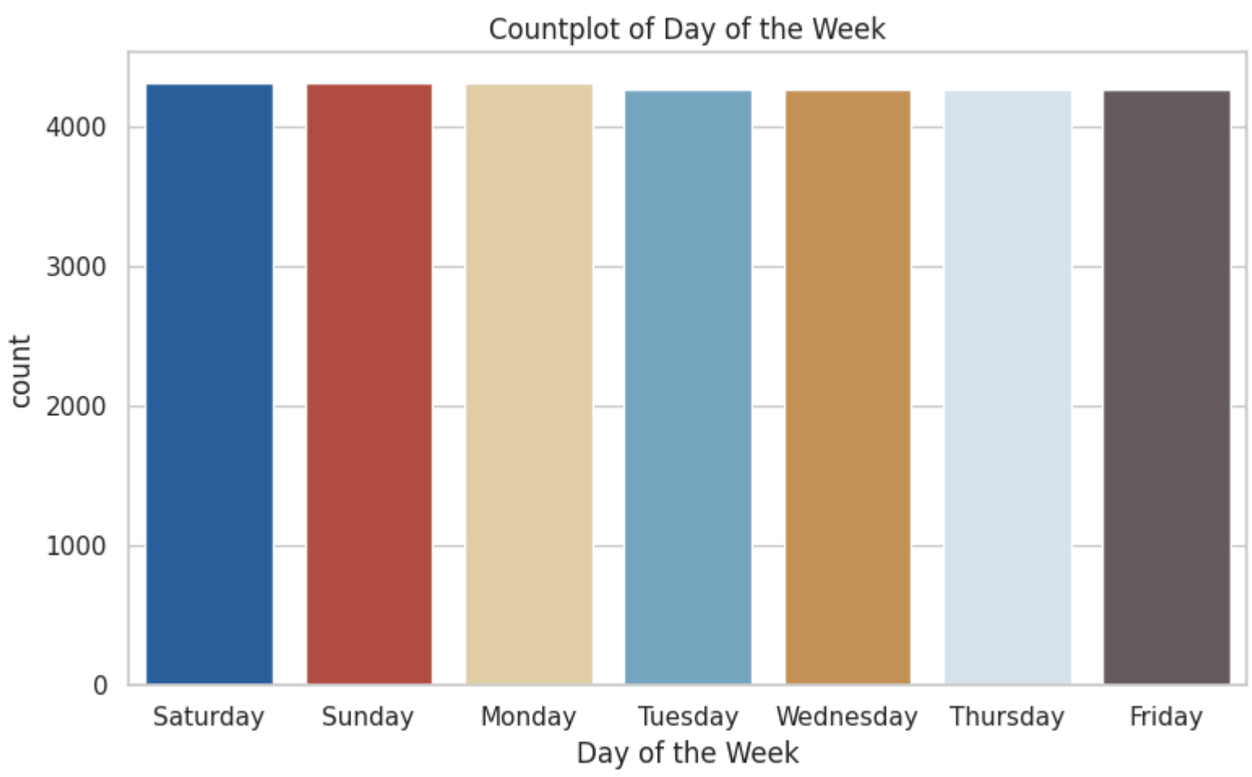


Countplots

In [14]:
```python
categorical_columns_to_plot = ['Product Category', 'Discount Percentage', 'Day of the Week']

# Plot countplots for each categorical column
for column in categorical_columns_to_plot:
    plt.figure(figsize=(8, 5))
    sns.countplot(data=df, x=column)
    plt.title(f'Countplot of {column}')

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

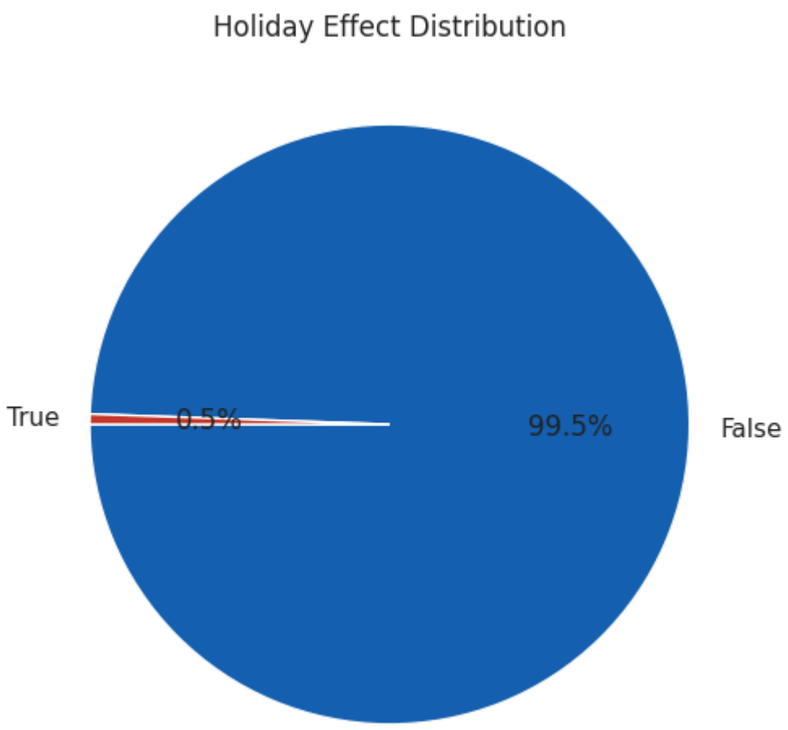






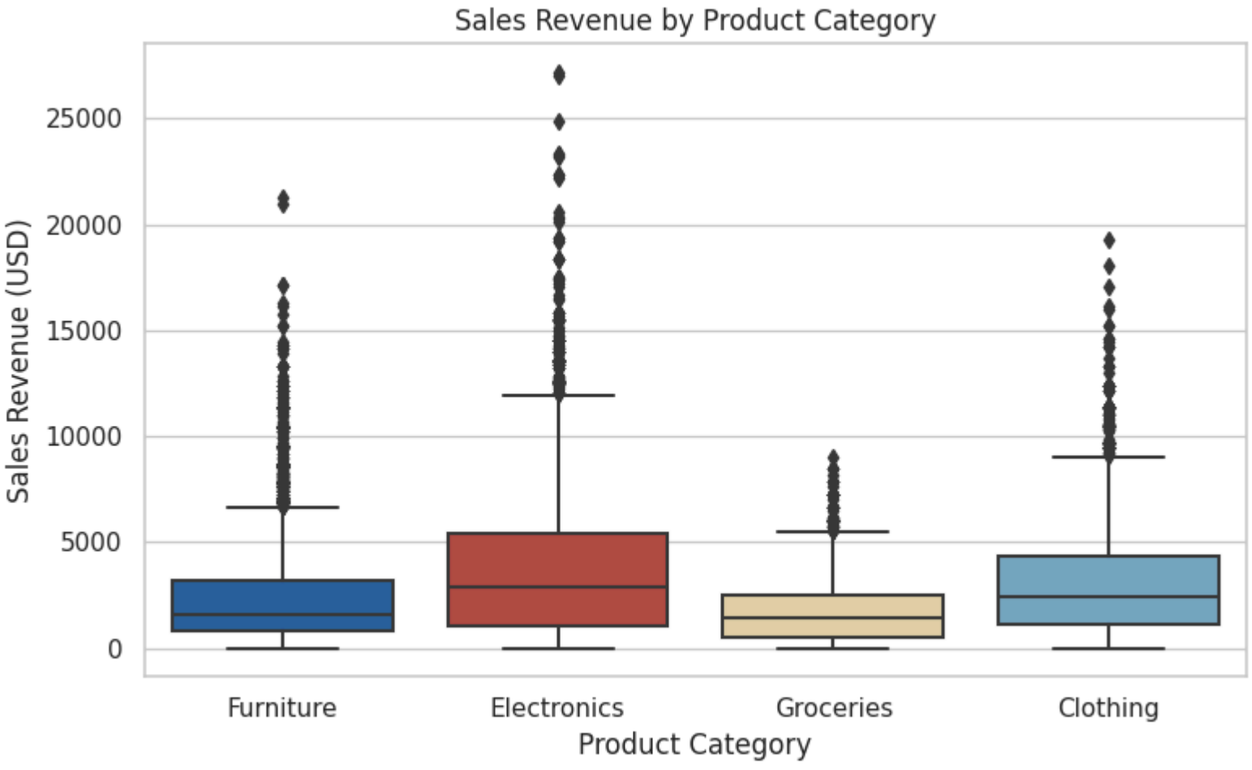
🍘 Pie Chart

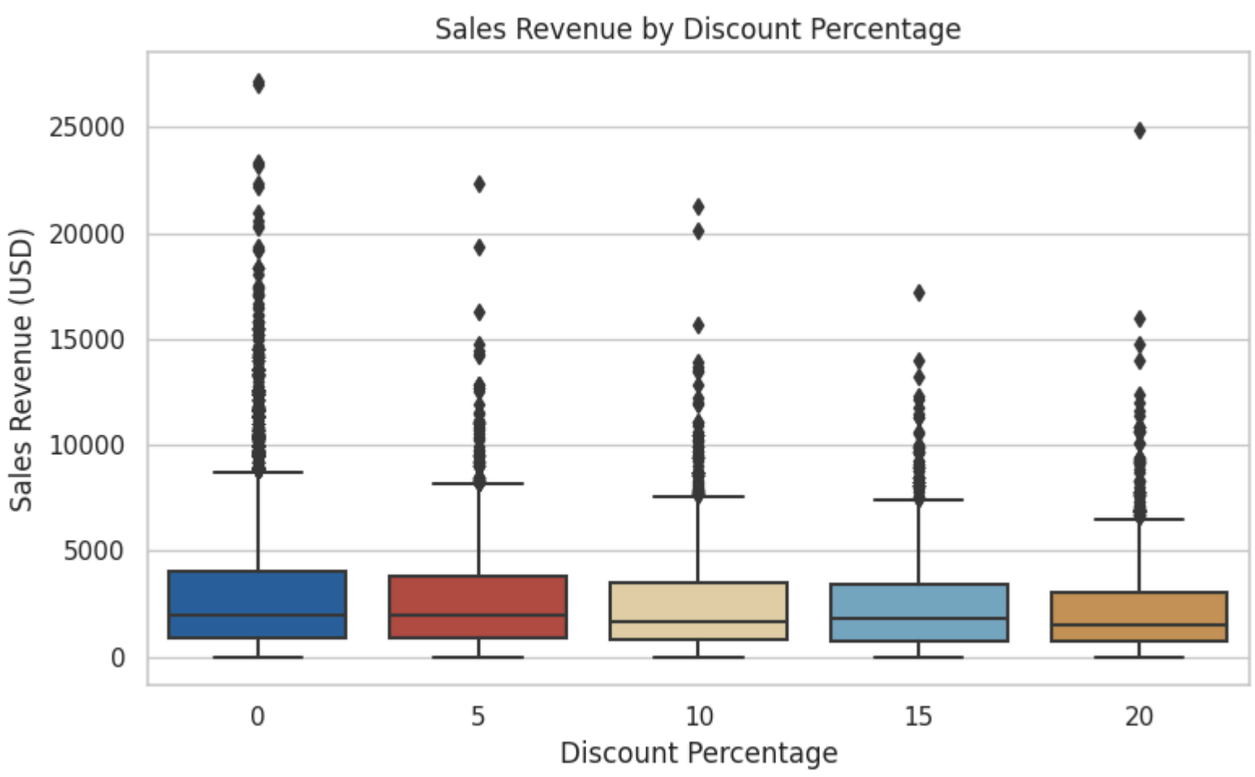
In [15]:
```
# Define the Holiday Effect categories and count occurences
categories = [False, True]
counts = df['Holiday Effect'].value_counts().tolist()

# Plot the pie chart with the counts of each Holiday Effect category
plt.figure(figsize=(6, 6))
plt.pie(counts, labels=categories, autopct='%1.1f%%', startangle=180, colors=custom_palette)
plt.title('Holiday Effect Distribution')
plt.show()
```

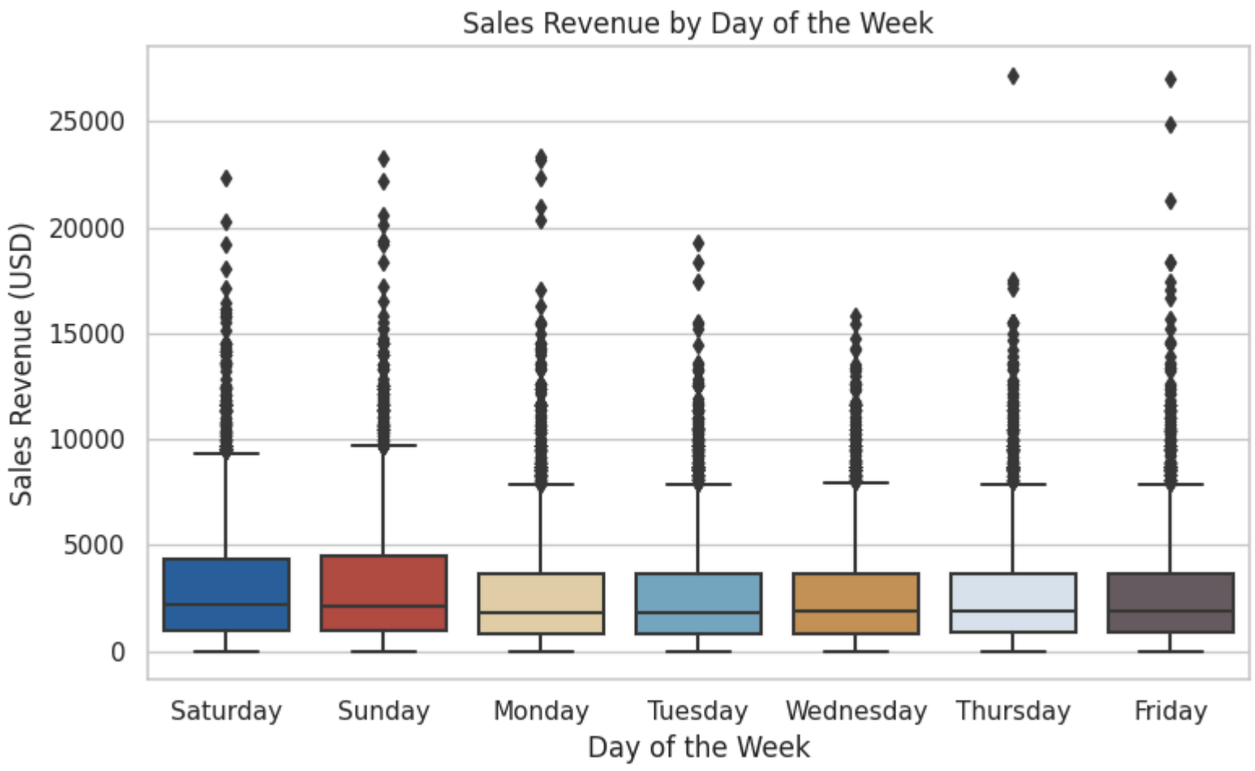


📦 Box plots

In [16]:
```
columns_to_plot = ['Product Category', 'Discount Percentage', 'Day of the Week', 'Holiday Effect']

# Plot box plots
for column in columns_to_plot:

    plt.figure(figsize=(8, 5))
    sns.boxplot(data=df, x=column, y='Sales Revenue (USD)')
    plt.title(f'Sales Revenue by {column}')

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

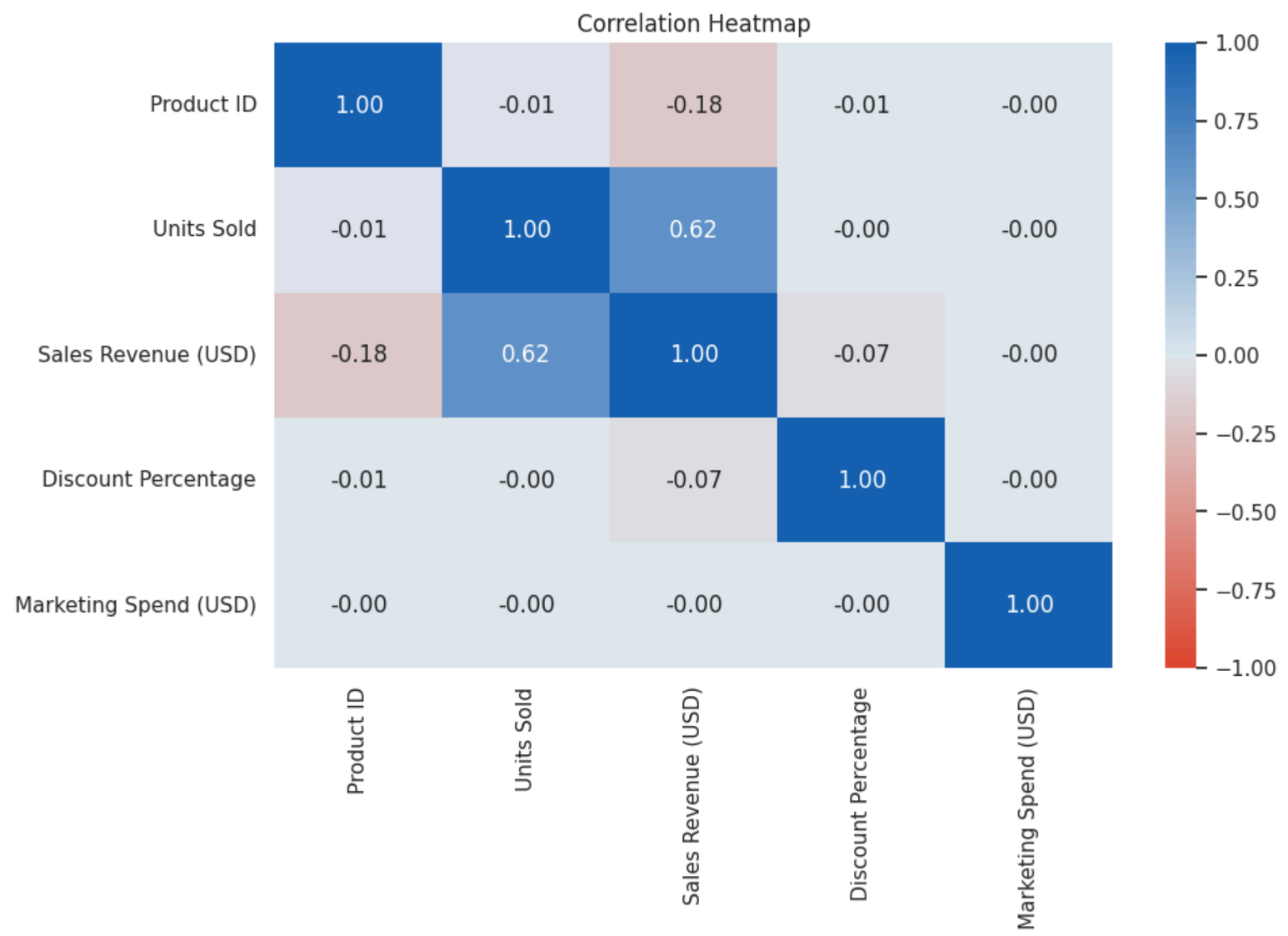






🌡️ Correlation Heatmap

In [17]:
```
# Define the custom colormap
colors = ["#DC462F", "#DCE7EE", "#1560B0"]
cmap = LinearSegmentedColormap.from_list("custom_cmap", colors)

# Create a correlation heatmap
numeric_df = df.select_dtypes(include=[np.number])
plt.figure(figsize=(10, 6))
sns.heatmap(numeric_df.corr(), annot=True, cmap=cmap, center=0, vmin=-1, fmt='.2f')
plt.title('Correlation Heatmap')
plt.show()
```



Observations from the visualization:

- 📊 **Numerical Distribution:** Numerical columns have a right-skewed distribution.
- 🗓️ **Days of the Week:** The distribution of days of the week is well balanced.
- 📉 **Discount Percentage:** The distribution of discount percentage is imbalanced, with most values being 0.
- 🛋️📦🍎👗 **Product Categories:** There are 4 product categories: Furniture, Electronics, Groceries, and Clothing.
- 💸 **Revenue by Product Categories:** Electronics and Clothing are correlated with higher revenues.
- 🎉 **Holiday Effect:** The holiday effect is mostly 'False' in 99.5% of the cases (which is expected, as holidays are not daily occurrences).
- 🎁 **Holiday Revenue:** Holidays tend to bring more revenue overall.
- 🔗 **Correlation:** The only highly correlated numerical variables are 'Units Sold' and 'Sales Revenue'.

5. Machine Learning

📏 Data Preprocessing

In [18]:
```
# Define features and target variable
X = df[['Product ID','Store Location', 'Product Category', 'Units Sold', 'Day of the Week',
        'Discount Percentage', 'Marketing Spend (USD)', 'Holiday Effect']]
y = df['Sales Revenue (USD)']
```

In [19]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [20]:
```
categorical_cols = ['Product ID', 'Store Location', 'Product Category', 'Day of the Week']
numerical_cols = ['Units Sold', 'Discount Percentage', 'Marketing Spend (USD)']
boolean_cols = ['Holiday Effect']
```

In [21]:
```
# Define your preprocessing step
preprocessor = ColumnTransformer(
    transformers=[
        ('num', MinMaxScaler(), numerical_cols),
        ('bool', 'passthrough', boolean_cols),
        ('cat', OneHotEncoder(), categorical_cols)
    ])
```

🤖 Training and Comparing 10 Different Models

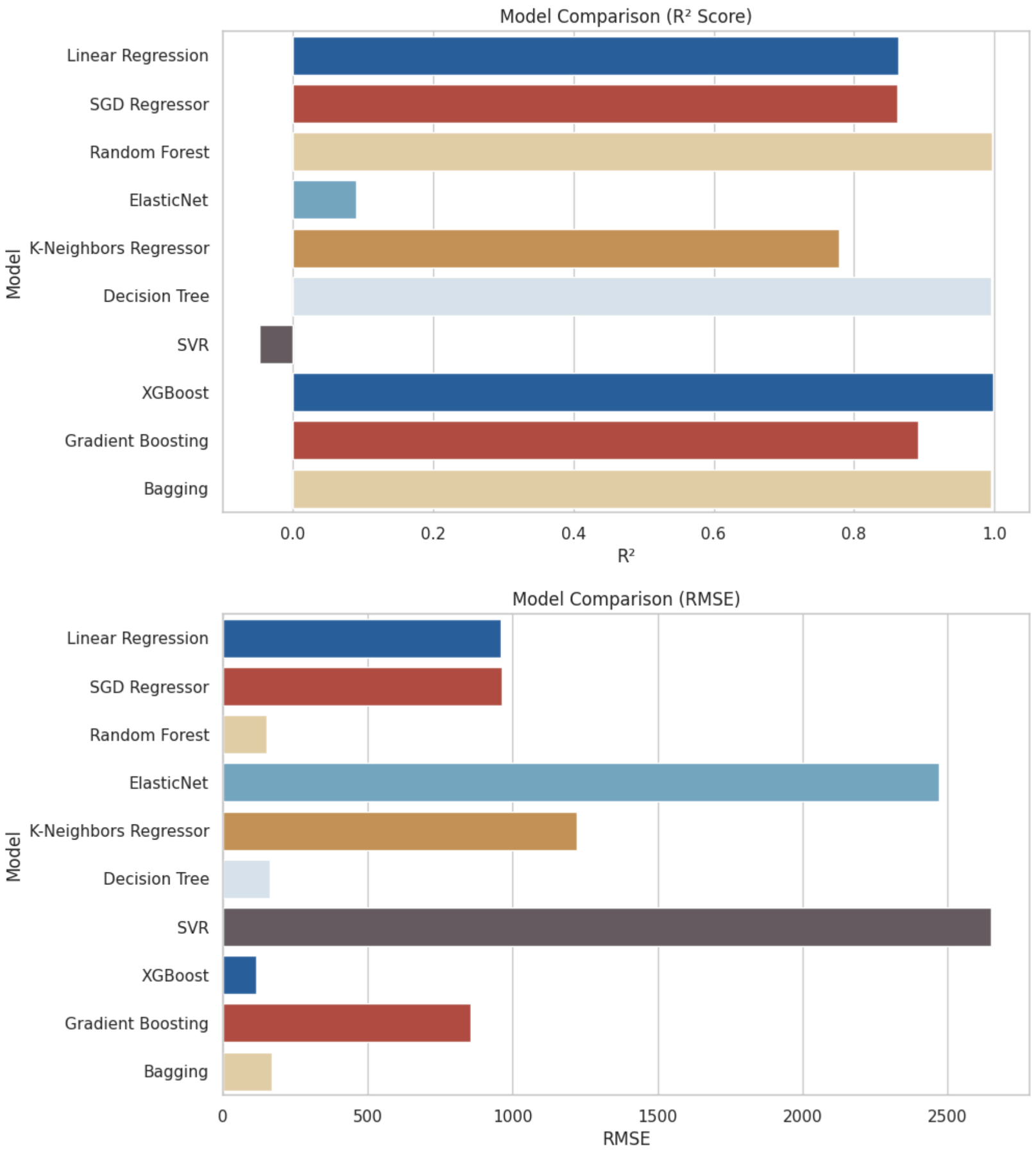
In [22]:
```
# Initialize list to collect results
results = []

# Function to evaluate models and store results for visualization
def evaluate_model(model, X_train, y_train, X_test, y_test, model_name):
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])

    pipeline.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred_test = pipeline.predict(X_test)

    # Calculate metrics: R² and RMSE
    test_r2 = r2_score(y_test, y_pred_test)
    test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_test))

    # Print results
    print(f"{model_name} Model:")
    print(f"Test R²: {test_r2:.4f}")
    print(f"Test RMSE: {test_rmse:.4f}\n")

    # Append results to the list for visualization
    results.append({'Model': model_name, 'R²': test_r2, 'RMSE': test_rmse})
```

In [23]:
```
# Define models to evaluate
models = {
    'Linear Regression': LinearRegression(),
    'SGD Regressor': SGDRegressor(),
    'Random Forest': RandomForestRegressor(),
    'ElasticNet': ElasticNet(),
    'K-Neighbors Regressor': KNeighborsRegressor(),
    'Decision Tree': DecisionTreeRegressor(),
    'SVR': SVR(),
    'XGBoost': XGBRegressor(),
    'Gradient Boosting': GradientBoostingRegressor(),
    'Bagging': BaggingRegressor()
}
```

In [24]:
```
# Iterate over the models to evaluate each
for model_name, model in models.items():
    evaluate_model(model, X_train, y_train, X_test, y_test, model_name)

# Convert results to a DataFrame
results_df = pd.DataFrame(results)
```

```
Linear Regression Model:
Test R²: 0.8629
Test RMSE: 959.3323

SGD Regressor Model:
Test R²: 0.8622
Test RMSE: 961.4637

Random Forest Model:
Test R²: 0.9967
Test RMSE: 149.6964

ElasticNet Model:
Test R²: 0.0894
Test RMSE: 2471.9898

K-Neighbors Regressor Model:
Test R²: 0.7782
Test RMSE: 1219.8619

Decision Tree Model:
Test R²: 0.9961
Test RMSE: 161.6051

SVR Model:
Test R²: -0.0478
Test RMSE: 2651.6621

XGBoost Model:
Test R²: 0.9981
Test RMSE: 113.9861

Gradient Boosting Model:
Test R²: 0.8913
Test RMSE: 853.9157

Bagging Model:
Test R²: 0.9958
Test RMSE: 167.4768
```

In [25]:
```
# Plot R² comparison
plt.figure(figsize=(10, 6))
sns.barplot(x='R²', y='Model', data=results_df, palette=custom_palette)
plt.title('Model Comparison (R² Score)')
plt.show()

# Plot RMSE comparison
plt.figure(figsize=(10, 6))
sns.barplot(x='RMSE', y='Model', data=results_df, palette=custom_palette)
plt.title('Model Comparison (RMSE)')
plt.show()
```



🎯 **Perfect Score, but is it a correct one?**

We achieved **outstanding results**, with an **R^2 score of 99.8%**! It's also worth noting that a simple and highly interpretable model, the **Decision Tree**, performed exceptionally well, achieving 99.6%.

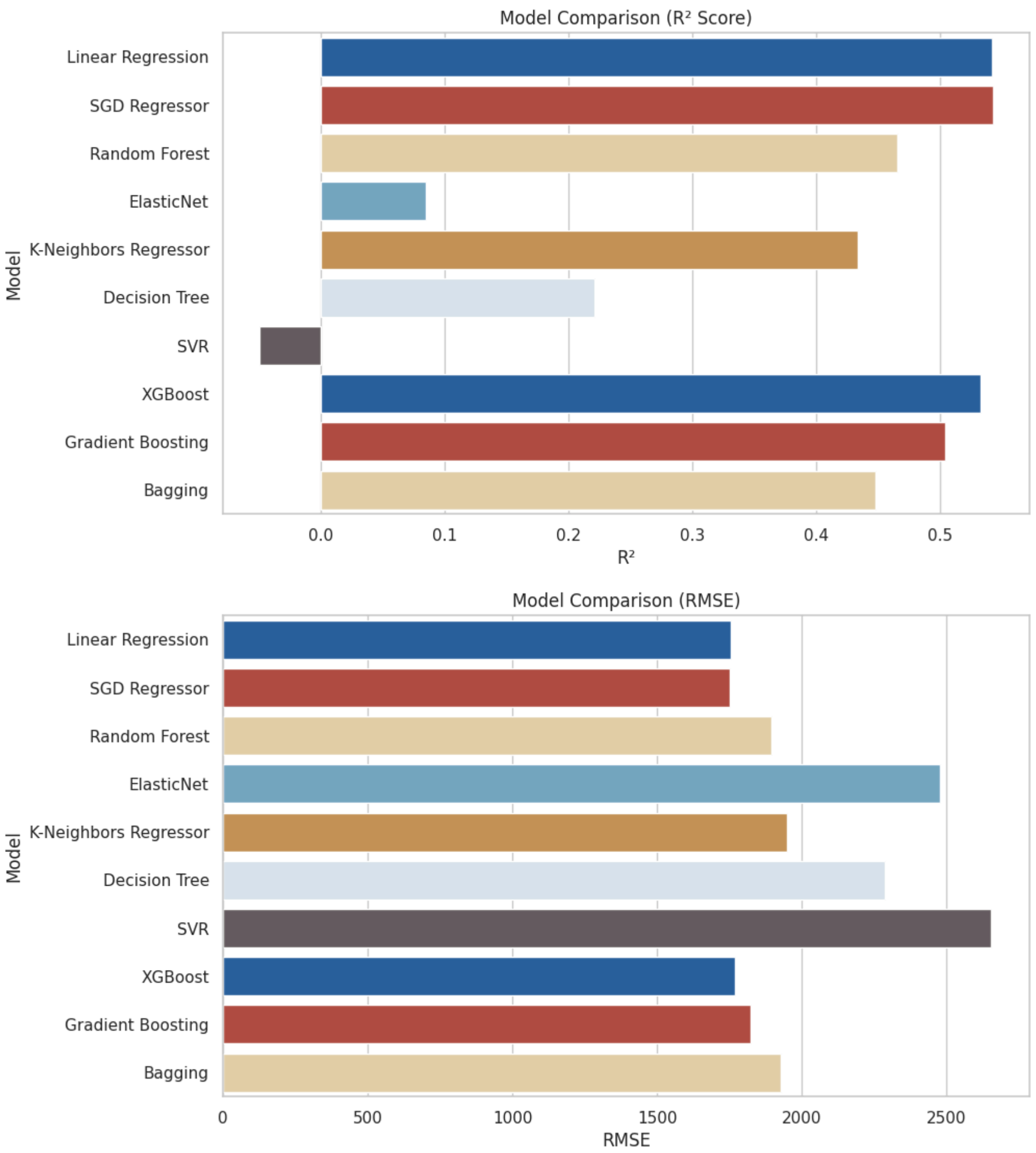
While we should be pleased with these results, there's an important issue. I didn't mention it earlier, but by including the **"Items Sold"** feature when predicting **"Sales Revenue"**, we've introduced **data leakage**. These two features are directly correlated, and in a real-world scenario, we wouldn't know the number of items sold in advance when predicting future sales. Let's remove this feature and repeat the process.

Let's Try Again

In [26]:

```
# Define correct features
X = df[['Product ID','Store Location', 'Product Category', 'Day of the Week','Discount Percentage', 'Marketing Spend (USD)', 'Holiday Effect']]

# Replace numerical columns list without Items Sold
numerical_cols = ['Discount Percentage', 'Marketing Spend (USD)']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define your preprocessing step
preprocessor = ColumnTransformer(
    transformers=[
        ('num', MinMaxScaler(), numerical_cols),
        ('bool', 'passthrough', boolean_cols),
        ('cat', OneHotEncoder(), categorical_cols)
    ])

# Initialize list to collect results
results = []

# Function to evaluate models and store results for visualization
def evaluate_model(model, X_train, y_train, X_test, y_test, model_name):
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])

    pipeline.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred_test = pipeline.predict(X_test)

    # Calculate metrics: R² and RMSE
    test_r2 = r2_score(y_test, y_pred_test)
    test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_test))

    # Print results
    print(f"{model_name} Model:")
    print(f"Test R²: {test_r2:.4f}")
    print(f"Test RMSE: {test_rmse:.4f}\n")

    # Append results to the list for visualization
    results.append({'Model': model_name, 'R²': test_r2, 'RMSE': test_rmse})

# Define models to evaluate
models = {
    'Linear Regression': LinearRegression(),
    'SGD Regressor': SGDRegressor(),
    'Random Forest': RandomForestRegressor(),
    'ElasticNet': ElasticNet(),
    'K-Neighbors Regressor': KNeighborsRegressor(),
    'Decision Tree': DecisionTreeRegressor(),
    'SVR': SVR(),
    'XGBoost': XGBRegressor(),
    'Gradient Boosting': GradientBoostingRegressor(),
    'Bagging': BaggingRegressor()
}

# Iterate over the models to evaluate each
for model_name, model in models.items():
    evaluate_model(model, X_train, y_train, X_test, y_test, model_name)

# Convert results to a DataFrame
results_df = pd.DataFrame(results)

# Plot R² comparison
plt.figure(figsize=(10, 6))
sns.barplot(x='R²', y='Model', data=results_df, palette=custom_palette)
plt.title('Model Comparison (R² Score)')
plt.show()

# Plot RMSE comparison
plt.figure(figsize=(10, 6))
sns.barplot(x='RMSE', y='Model', data=results_df, palette=custom_palette)
plt.title('Model Comparison (RMSE)')
plt.show()
```

```
Linear Regression Model:
Test R²: 0.5417
Test RMSE: 1753.7249

SGD Regressor Model:
Test R²: 0.5426
Test RMSE: 1751.9873

Random Forest Model:
Test R²: 0.4657
Test RMSE: 1893.5792

ElasticNet Model:
Test R²: 0.0846
Test RMSE: 2478.4418

K-Neighbors Regressor Model:
Test R²: 0.4335
Test RMSE: 1949.8000

Decision Tree Model:
Test R²: 0.2210
Test RMSE: 2286.3940

SVR Model:
Test R²: -0.0495
Test RMSE: 2653.7860

XGBoost Model:
Test R²: 0.5330
Test RMSE: 1770.1885

Gradient Boosting Model:
Test R²: 0.5043
Test RMSE: 1823.8799

Bagging Model:
Test R²: 0.4476
Test RMSE: 1925.3837
```



Wow! Just by removing one variable (Items Sold) the R² score dropped significantly from 99.8% to 54.3%! That's a huge difference, but now we can be confident there's no data leakage. While the results aren't as impressive, they are far more reliable.

Let's try a few more things:

- Use the Date column by extracting new features such as year, month, day, etc.
- Drop Outliers.
- Incorporate hyperparameter tuning for machine learning models to optimize performance.
- Explore Deep Learning approaches to potentially improve results further.

In [27]:
```
# remove outliers
numerical_cols = df.select_dtypes(include=['number']).columns
```

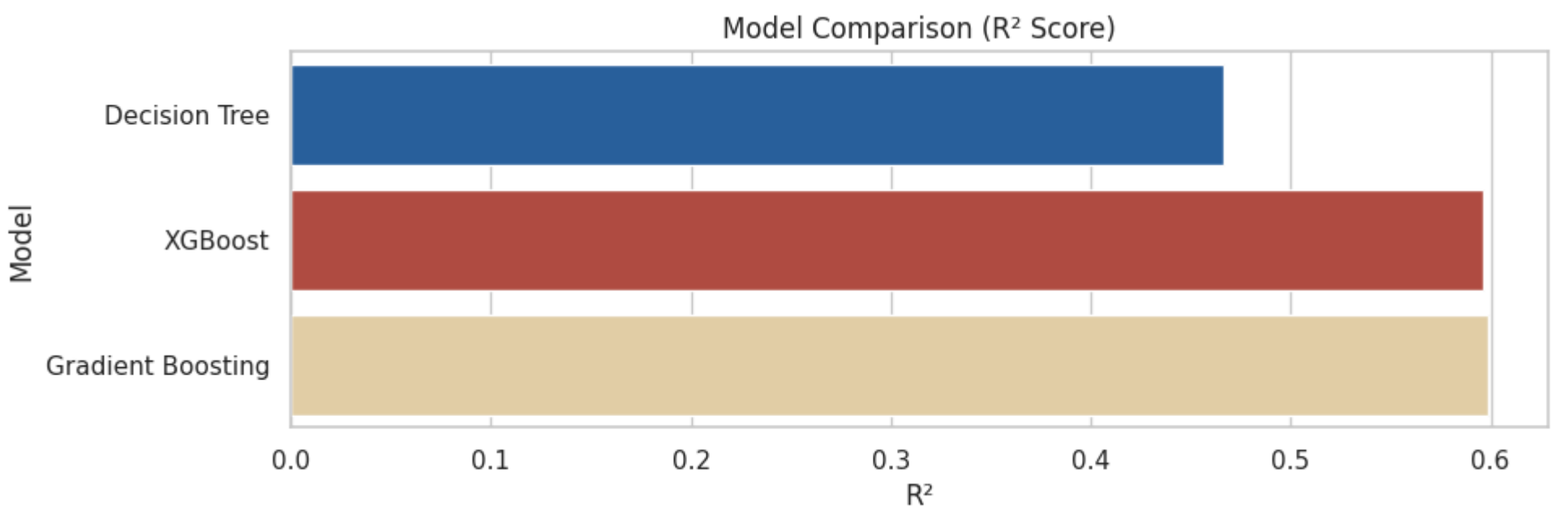
```
df = df[(np.abs(stats.zscore(df[numerical_cols])) < 3).all(axis=1)]
```

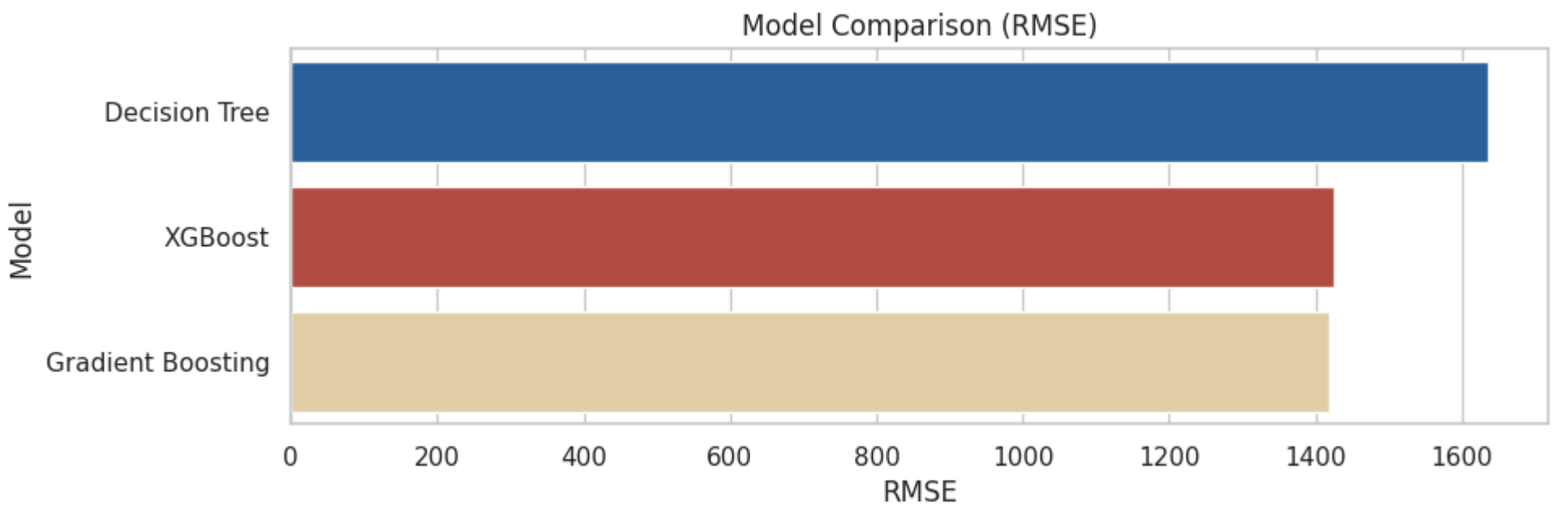
In [28]:
```
# Convert Date to DateTime Format
df['Date'] = pd.to_datetime(df['Date'])

# Set the Date Column as the Index
df.set_index('Date', inplace=True)

# Extract useful date features
df['Year'] = df.index.year
df['Month'] = df.index.month
df['Day'] = df.index.day
df['DayOfWeek'] = df.index.dayofweek
df['IsWeekend'] = df['DayOfWeek'].apply(lambda x: 1 if x >= 5 else 0)
```

In [29]:
```
# Define new features and target
X = df[['Product ID', 'Discount Percentage', 'Marketing Spend (USD)', 'Store Location',
       'Product Category', 'Holiday Effect', 'Year', 'Month', 'Day', 'DayOfWeek', 'IsWeekend']]
y = df['Sales Revenue (USD)']

# Split the data into training and testing sets (same as before)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Update lists of different datatypes
categorical_cols = ['Product ID', 'Store Location', 'Product Category']
numerical_cols = ['Discount Percentage', 'Marketing Spend (USD)', 'Year', 'Month', 'DayOfWeek']
boolean_cols = ['Holiday Effect', 'IsWeekend']

# Define preprocessing step (same as before)
preprocessor = ColumnTransformer(
    transformers=[
        ('num', MinMaxScaler(), numerical_cols),
        ('bool', 'passthrough', boolean_cols),
        ('cat', OneHotEncoder(), categorical_cols)
    ])

# Fit and transform the training data
X_train_processed = preprocessor.fit_transform(X_train)

# Transform the testing data
X_test_processed = preprocessor.transform(X_test)
```

In [30]:
```
# Initialize list to collect results
results = []

# Define hyperparameter grids for the selected models
param_grids = {
    'Decision Tree': {
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    },
    'XGBoost': {
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.1, 0.3],
        'max_depth': [3, 5, 7]
    },
    'Gradient Boosting': {
        'n_estimators': [100, 200],
        'learning_rate': [0.01, 0.1],
        'max_depth': [3, 5]
    }
}

# Function to evaluate models with hyperparameter tuning
def evaluate_model_with_tuning(model, param_grid, X_train, y_train, X_test, y_test, model_name):
    if param_grid:
        search = RandomizedSearchCV(model, param_grid, n_iter=10, cv=3, scoring='r2', random_state=42, n_jobs=-1)
        search.fit(X_train, y_train)
        best_model = search.best_estimator_
        print(f"Best params for {model_name}: {search.best_params_}")
    else:
        best_model = model
        best_model.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred_test = best_model.predict(X_test)

    # Calculate metrics: R² and RMSE
    test_r2 = r2_score(y_test, y_pred_test)
    test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_test))

    # Print results
    print(f"{model_name} Model:")
    print(f"Test R²: {test_r2:.4f}")
    print(f"Test RMSE: {test_rmse:.4f}\n")

    # Append results to the list for visualization
    results.append({'Model': model_name, 'R²': test_r2, 'RMSE': test_rmse})

# Define models to evaluate
models = {
    'Decision Tree': DecisionTreeRegressor(),
    'XGBoost': XGBRegressor(),
    'Gradient Boosting': GradientBoostingRegressor()
}

# Iterate over the models to evaluate each with hyperparameter tuning
for model_name, model in models.items():
    param_grid = param_grids.get(model_name, None)  # Get the param grid for the model, if available
    evaluate_model_with_tuning(model, param_grid, X_train_processed, y_train, X_test_processed, y_test, model_name)

# Convert results to a DataFrame
results_df = pd.DataFrame(results)

# Plot R² comparison
plt.figure(figsize=(10, 3))
sns.barplot(x='R²', y='Model', data=results_df, palette=custom_palette)
plt.title('Model Comparison (R² Score)')
plt.show()

# Plot RMSE comparison
plt.figure(figsize=(10, 3))
sns.barplot(x='RMSE', y='Model', data=results_df, palette=custom_palette)
plt.title('Model Comparison (RMSE)')
plt.show()
```

```
Best params for Decision Tree: {'min_samples_split': 10, 'min_samples_leaf': 4, 'max_depth': None}
Decision Tree Model:
Test R²: 0.4664
Test RMSE: 1636.7455

Best params for XGBoost: {'n_estimators': 100, 'max_depth': 7, 'learning_rate': 0.1}
XGBoost Model:
Test R²: 0.5959
Test RMSE: 1424.2376

Best params for Gradient Boosting: {'n_estimators': 200, 'max_depth': 5, 'learning_rate': 0.1}
Gradient Boosting Model:
Test R²: 0.5990
Test RMSE: 1418.7697
```





In [31]:
```
## Neural Network wth TensorFlow

# Define a Sequential model
model = Sequential([
    Dense(units=50, activation='relu'),
    Dense(units=30, activation='relu'),
    Dense(units=1, activation='relu')            # Output layer with ReLU activation (suitable for regression non-negative tasks)
])

# Compile the model with Mean Squared Error loss function and train the model
model.compile(loss = MeanSquaredError())
model.fit(X_train_processed, y_train, epochs=100)
```

```
Epoch 1/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 10874206.0000
Epoch 2/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 4743525.5000
Epoch 3/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 3506069.7500
Epoch 4/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2483249.7500
Epoch 5/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2161420.2500
Epoch 6/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2098909.0000
Epoch 7/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2078629.0000
Epoch 8/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2035885.0000
Epoch 9/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2087282.7500
Epoch 10/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2043186.8750
Epoch 11/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2054353.7500
Epoch 12/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2070369.6250
Epoch 13/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2059993.1250
Epoch 14/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2044774.2500
Epoch 15/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2037657.5000
Epoch 16/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2054639.8750
Epoch 17/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2067792.0000
Epoch 18/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2018316.2500
Epoch 19/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2058833.7500
Epoch 20/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2073933.5000
Epoch 21/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2041000.3750
Epoch 22/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2037119.8750
Epoch 23/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2092568.3750
Epoch 24/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2057042.1250
Epoch 25/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2061265.8750
Epoch 26/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2046568.5000
Epoch 27/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2043383.0000
Epoch 28/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2030527.8750
Epoch 29/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2040646.2500
Epoch 30/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2044177.5000
Epoch 31/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2059494.0000
Epoch 32/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2037781.5000
Epoch 33/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2080847.2500
Epoch 34/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2023820.7500
Epoch 35/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2026963.1250
Epoch 36/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2078676.8750
Epoch 37/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2067898.5000
Epoch 38/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2065509.3750
Epoch 39/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2091725.5000
Epoch 40/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2075420.8750
Epoch 41/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2075181.7500
Epoch 42/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2077030.5000
Epoch 43/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2041039.7500
Epoch 44/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2047110.2500
Epoch 45/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2068268.0000
Epoch 46/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2038348.2500
Epoch 47/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2014901.6250
Epoch 48/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2058075.7500
Epoch 49/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2019066.7500
Epoch 50/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2038938.6250
Epoch 51/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2062168.7500
Epoch 52/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2078326.0000
Epoch 53/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1998440.0000
Epoch 54/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2028439.0000
Epoch 55/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2053170.2500
Epoch 56/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2036982.3750
Epoch 57/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1975959.7500
Epoch 58/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1989596.1250
Epoch 59/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2053723.6250
Epoch 60/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2022052.1250
Epoch 61/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2053975.5000
Epoch 62/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2032546.3750
Epoch 63/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1996679.8750
Epoch 64/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2012127.8750
Epoch 65/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2024848.2500
Epoch 66/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2042002.0000
Epoch 67/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2058707.5000
Epoch 68/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2019930.0000
Epoch 69/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2009219.2500
```

```
Epoch 70/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2005041.8750
Epoch 71/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2030393.0000
Epoch 72/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2015399.0000
Epoch 73/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2008508.8750
Epoch 74/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2042761.0000
Epoch 75/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1992086.1250
Epoch 76/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2034266.0000
Epoch 77/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2020044.0000
Epoch 78/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1975961.7500
Epoch 79/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1951606.3750
Epoch 80/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1976508.1250
Epoch 81/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1995434.5000
Epoch 82/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1975310.7500
Epoch 83/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1969261.7500
Epoch 84/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 2023770.5000
Epoch 85/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1987033.6250
Epoch 86/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2017569.5000
Epoch 87/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2024384.1250
Epoch 88/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2029181.8750
Epoch 89/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1994954.1250
Epoch 90/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 1956240.1250
Epoch 91/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1951336.2500
Epoch 92/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1980000.6250
Epoch 93/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1964726.0000
Epoch 94/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1912321.7500
Epoch 95/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1979113.7500
Epoch 96/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1961073.2500
Epoch 97/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1992938.3750
Epoch 98/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1966620.1250
Epoch 99/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1926272.2500
Epoch 100/100
736/736 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1934807.1250
```

Out[31]: <keras.src.callbacks.history.History at 0x7ad8887381c0>

In [32]:
```
y_pred = model.predict(X_test_processed)

test_r2 = r2_score(y_test, y_pred)
test_rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f"Test R²: {test_r2:.4f}")
print(f"Test RMSE: {test_rmse:.4f}\n")
```

```
184/184 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
Test R²: 0.5758
Test RMSE: 1459.3368
```

Conclusion:

The best performing model in this analysis is the Gradient Boosting Model, with a Test R^2 of 0.5985 and a Test RMSE of 1419.7979. While these results may not seem exceptionally high, they are consistent with the findings from the exploratory data analysis (EDA), which revealed weak correlations between most features and the target variable. The exception was the number of items sold, which had a stronger correlation but was excluded from the model to avoid introducing data leakage.

Thank you for your time and support! 🎉