Problem Statement

Business Context

A sales forecast is a prediction of future sales revenue based on historical data, industry trends, and the status of the current sales pipeline. Businesses use the sales forecast to estimate weekly, monthly, quarterly, and annual sales totals. A company needs to make an accurate sales forecast as it adds value across an organization and helps the different verticals to chalk out their future course of action.

Forecasting helps an organization plan its sales operations by region and provides valuable insights to the supply chain team regarding the procurement of goods and materials. An accurate sales forecast process has many benefits which include improved decision-making about the future and reduction of sales pipeline and forecast risks. Moreover, it helps to reduce the time spent in planning territory coverage and establish benchmarks that can be used to assess trends in the future.

Objective

SuperKart is a retail chain operating supermarkets and food marts across various tier cities, offering a wide range of products. To optimize its inventory management and make informed decisions around regional sales strategies, SuperKart wants to accurately forecast the sales revenue of its outlets for the upcoming quarter.

To operationalize these insights at scale, the company has partnered with a data science firm—not just to build a predictive model based on historical sales data, but to develop and deploy a robust forecasting solution that can be integrated into SuperKart's decision-making systems and used across its network of stores.

Data Description

The data contains the different attributes of the various products and stores.The detailed data dictionary is given below.

- **Product_Id** - unique identifier of each product, each identifier having two letters at the beginning followed by a number.
- **Product_Weight** - weight of each product
- **Product_Sugar_Content** - sugar content of each product like low sugar, regular and no sugar
- **Product_Allocated_Area** - ratio of the allocated display area of each product to the total display area of all the products in a store
- **Product_Type** - broad category for each product like meat, snack foods, hard drinks, dairy, canned, soft drinks, health and hygiene, baking goods, bread, breakfast, frozen foods, fruits and vegetables, household, seafood, starchy foods, others
- **Product_MRP** - maximum retail price of each product
- **Store_Id** - unique identifier of each store
- **Store_Establishment_Year** - year in which the store was established
- **Store_Size** - size of the store depending on sq. feet like high, medium and low
- **Store_Location_City_Type** - type of city in which the store is located like Tier 1, Tier 2 and Tier 3. Tier 1 consists of cities where the standard of living is comparatively higher than its Tier 2 and Tier 3 counterparts.
- **Store_Type** - type of store depending on the products that are being sold there like Departmental Store, Supermarket Type 1, Supermarket Type 2 and Food Mart
- **Product_Store_Sales_Total** - total revenue generated by the sale of that particular product in that particular store

Please read the instructions carefully before starting the project.

This is a commented Python Notebook file in which all the instructions and tasks to be performed are mentioned.

- Blanks ''***are provided in the notebook that needs to be filled with an appropriate code to get the correct result. With every '*** blank, there is a comment that briefly describes what needs to be filled in the blank space.
- Identify the task to be performed correctly, and only then proceed to write the required code.
- Please run the codes in a sequential manner from the beginning to avoid any unnecessary errors.
- Add the results/observations (wherever mentioned) derived from the analysis in the presentation and submit the same. Any mathematical or computational details which are a graded part of the project can be included in the Appendix section of the presentation.

Installing and Importing the necessary libraries

```
#Installing the libraries with the specified versions
!pip install numpy==2.0.2 pandas==2.2.2 scikit-learn==1.6.1 matplotlib==3.10.0 seaborn==0.13.2 joblib==1.4.2 xgboost==2.1.4 requests==2.32.3
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 301.8/301.8 kB 20.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 481.2/481.2 kB 30.2 MB/s eta 0:00:00
```

Note:

- After running the above cell, kindly restart the notebook kernel (for Jupyter Notebook) or runtime (for Google Colab) and run all cells sequentially from the next cell.
- On executing the above line of code, you might see a warning regarding package dependencies. This error message can be ignored as the above code ensures that all necessary libraries and their dependencies are maintained to successfully execute the code in this notebook.

```
# Libraries to help with reading and manipulating data
import numpy as np
import pandas as pd

# For splitting the dataset
from sklearn.model_selection import train_test_split

# Libaries to help with data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Removes the limit for the number of displayed columns
pd.set_option("display.max_columns", None)
# Sets the limit for the number of displayed rows
pd.set_option("display.max_rows", 100)


# Libraries different ensemble classifiers
from sklearn.ensemble import (
    BaggingRegressor,
    RandomForestRegressor,
    AdaBoostRegressor,
    GradientBoostingRegressor,
)
from xgboost import XGBRegressor
from sklearn.tree import DecisionTreeRegressor

# Libraries to get different metric scores
from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    mean_squared_error,
    mean_absolute_error,
    r2_score,
    mean_absolute_percentage_error
)

# To create the pipeline
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline,Pipeline

# To tune different models and standardize
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler,OneHotEncoder

# To serialize the model
import joblib

# os related functionalities
import os

# API request
```

```
import requests

# for hugging face space authentication to upload files
from huggingface_hub import login, HfApi

import warnings
warnings.filterwarnings("ignore")
```

Loading the dataset

```
# Uncomment the below snippet of code if the drive needs to be mounted
#from google.colab import drive
#drive.mount('/content/drive')
```

```
kart = pd.read_csv("SuperKart.csv") #Complete the code to read the data
```

```
# copying data to another variable to avoid any changes to original data
data = kart.copy()
```

Data Overview

View the first and last 5 rows of the dataset

```
data.head() #Complete the code to display the first 5 rows of the dataset
```

	Product_Id	Product_Weight	Product_Sugar_Content	Product_Allocated_Area	Product_Type	Product_MRP	Store_Id	Store_Establishment_Y
0	FD6114	12.66	Low Sugar	0.027	Frozen Foods	117.08	OUT004	2
1	FD7839	16.54	Low Sugar	0.144	Dairy	171.43	OUT003	1
2	FD5075	14.28	Regular	0.031	Canned	162.08	OUT001	1

```
data.tail() #Complete the code to display the last 5 rows of the dataset
```

	Product_Id	Product_Weight	Product_Sugar_Content	Product_Allocated_Area	Product_Type	Product_MRP	Store_Id	Store_Establishmen
8758	NC7546	14.80	No Sugar	0.016	Health and Hygiene	140.53	OUT004	
8759	NC584	14.06	No Sugar	0.142	Household	144.51	OUT004	
8760	NC2471	13.48	No Sugar	0.017	Health and Hygiene	88.58	OUT001	

Understand the shape of the dataset

```
print(f"There are {data.shape[0]} rows and {data.shape[1]} columns.")  #Complete the code to display the number of columns
```

```
There are 8763 rows and 12 columns.
```

Check the data types of the columns for the dataset

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8763 entries, 0 to 8762
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype
 ---  ------                    --------------  -----  
  0   Product_Id                 8763 non-null   object 
  1   Product_Weight             8763 non-null   float64
  2   Product_Sugar_Content      8763 non-null   object 
  3   Product_Allocated_Area     8763 non-null   float64
  4   Product_Type               8763 non-null   object 
  5   Product_MRP                8763 non-null   float64
  6   Store_Id                   8763 non-null   object 
  7   Store_Establishment_Year   8763 non-null   int64  
  8   Store_Size                 8763 non-null   object 
  9   Store_Location_City_Type   8763 non-null   object 
  10  Store_Type                 8763 non-null   object 
  11  Product_Store_Sales_Total  8763 non-null   float64
 dtypes: float64(4), int64(1), object(7)
 memory usage: 821.7+ KB
```

Statistical summary of the data

Let's check the statistical summary of the data.

```
data.describe(include="all").T
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Product_Id	8763	8763	FD306	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Weight	8763.0	NaN	NaN	NaN	12.653792	2.21732	4.0	11.15	12.66	14.18	22.0
Product_Sugar_Content	8763	4	Low Sugar	4885	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Allocated_Area	8763.0	NaN	NaN	NaN	0.068786	0.048204	0.004	0.031	0.056	0.096	0.298
Product_Type	8763	16	Fruits and Vegetables	1249	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_MRP	8763.0	NaN	NaN	NaN	147.032539	30.69411	31.0	126.16	146.74	167.585	266.0
Store_Id	8763	4	OUT004	4676	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Establishment_Year	8763.0	NaN	NaN	NaN	2002.032751	8.388381	1987.0	1998.0	2009.0	2009.0	2009.0
Store_Size	8763	3	Medium	6025	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Location_City_Type	8763	3	Tier 2	6262	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Type	8763	4	Supermarket Type2	4676	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Store_Sales_Total	8763.0	NaN	NaN	NaN	3464.00364	1065.630494	33.0	2761.715	3452.34	4145.165	8000.0

Checking for duplicate values

```
# checking for duplicate values
data.duplicated().sum()
```

```
np.int64(0)
```

Checking for missing values

```
# checking for missing values in the data
data.isnull().sum() #Complete the code to compute the number of missing values.
```

	0
Product_Id	0
Product_Weight	0
Product_Sugar_Content	0
Product_Allocated_Area	0
Product_Type	0
Product_MRP	0
Store_Id	0
Store_Establishment_Year	0
Store_Size	0
Store_Location_City_Type	0
Store_Type	0
Product_Store_Sales_Total	0

dtype: int64

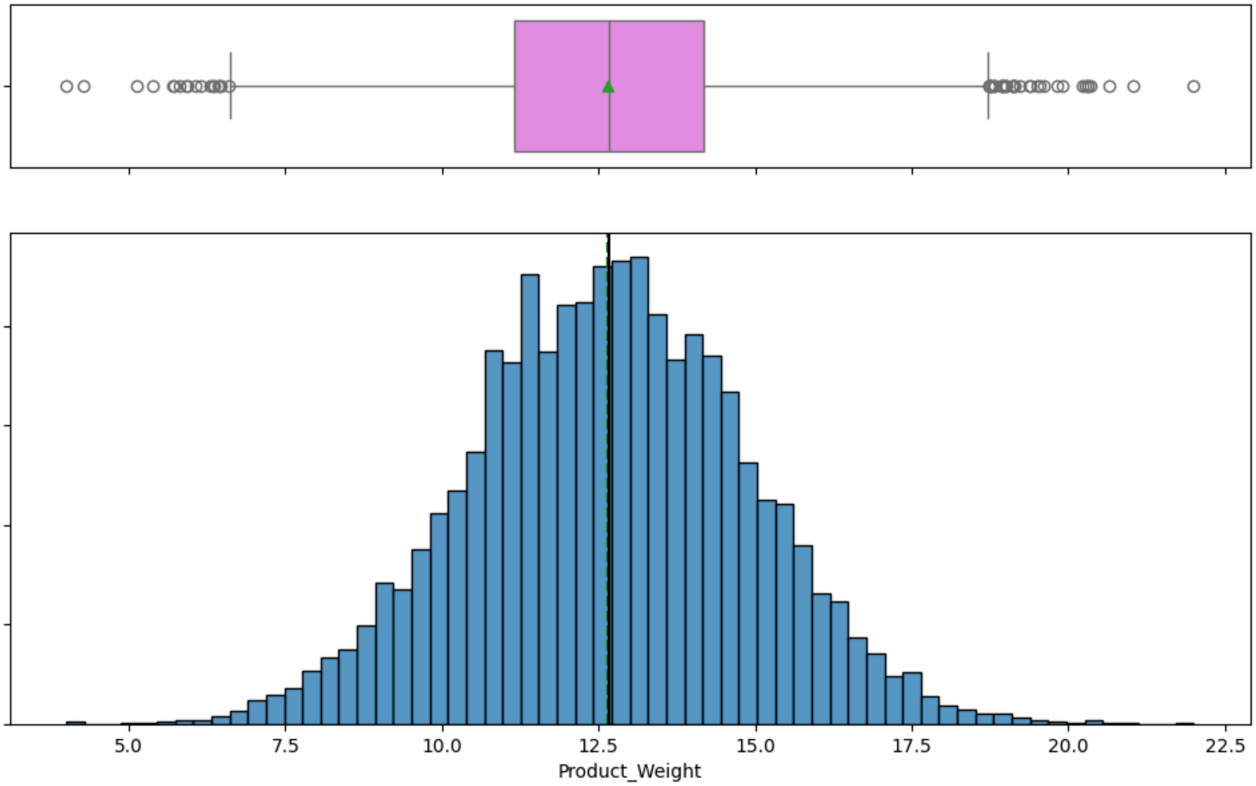
Exploratory Data Analysis (EDA)

Univariate Analysis

```
# function to plot a boxplot and a histogram along the same scale.

def histogram_boxplot(data, feature, figsize=(12, 7), kde=False, bins=None):
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to the show density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,  # Number of rows of the subplot grid= 2
        sharex=True,  # x-axis will be shared among all subplots
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )  # creating the 2 subplots
    sns.boxplot(
        data=data, x=feature, ax=ax_box2, showmeans=True, color="violet"
    )  # boxplot will be created and a star will indicate the mean value of the column
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins, palette="winter"
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2
    )  # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color="green", linestyle="--"
    )  # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram
```

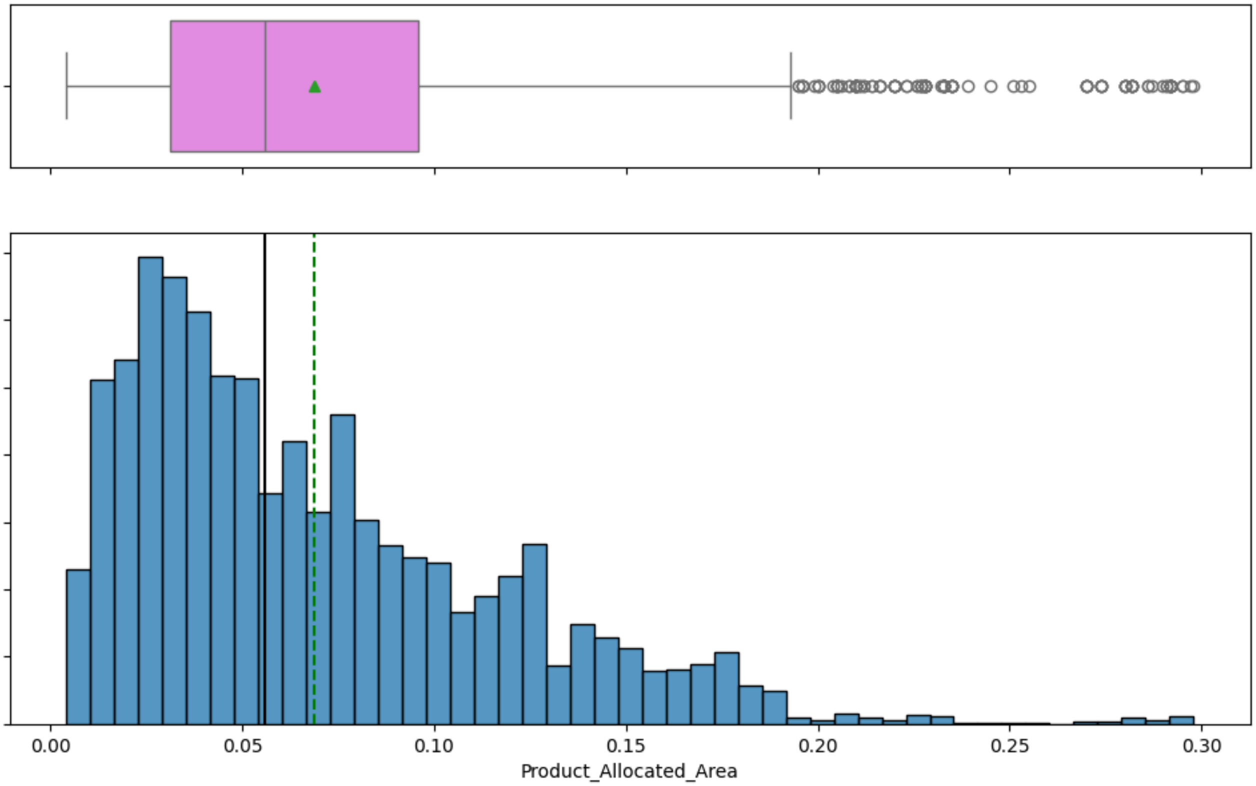
Product_Weight

```
histogram_boxplot(data, "Product_Weight")
```



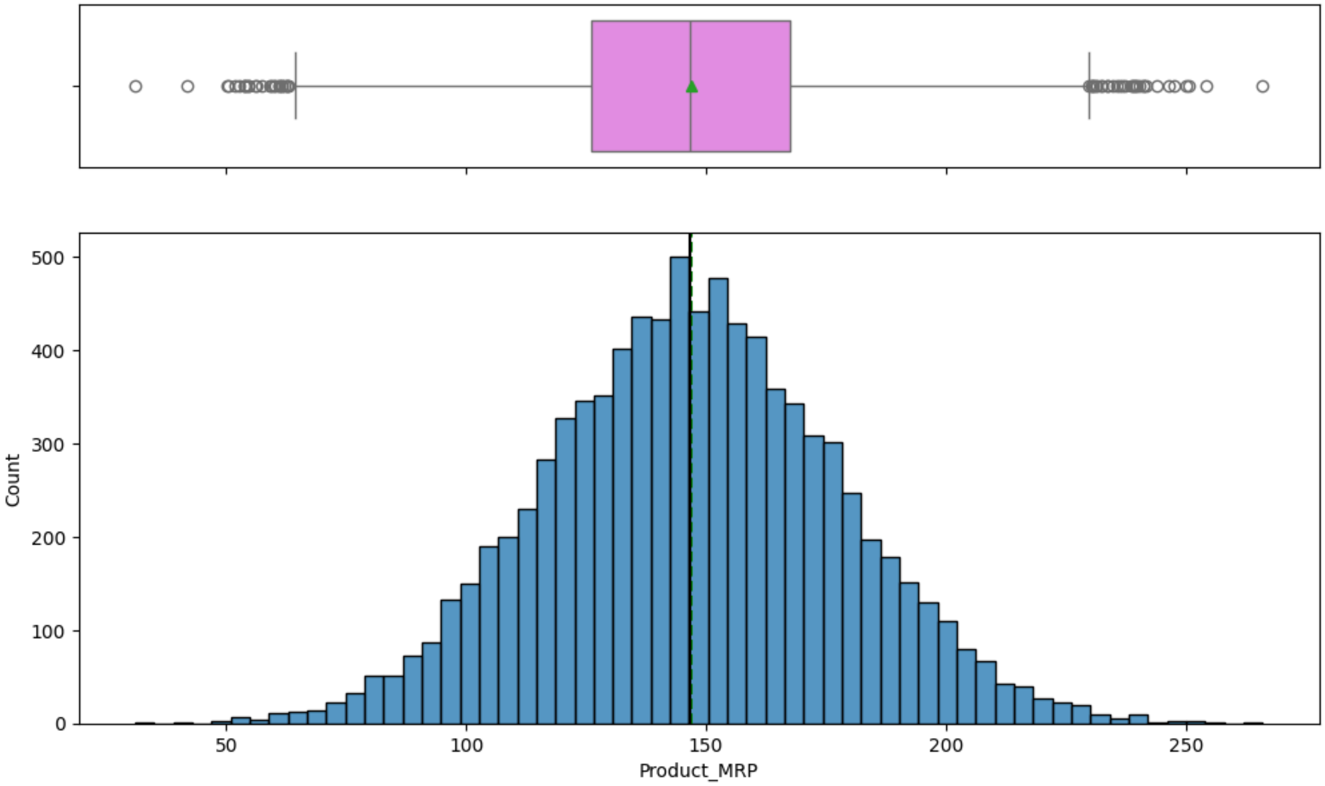
Product_Allocated_Area

```
histogram_boxplot(data, "Product_Allocated_Area") #Complete the code to plot the boxplot and histogram of Product_Allocated_Area
```



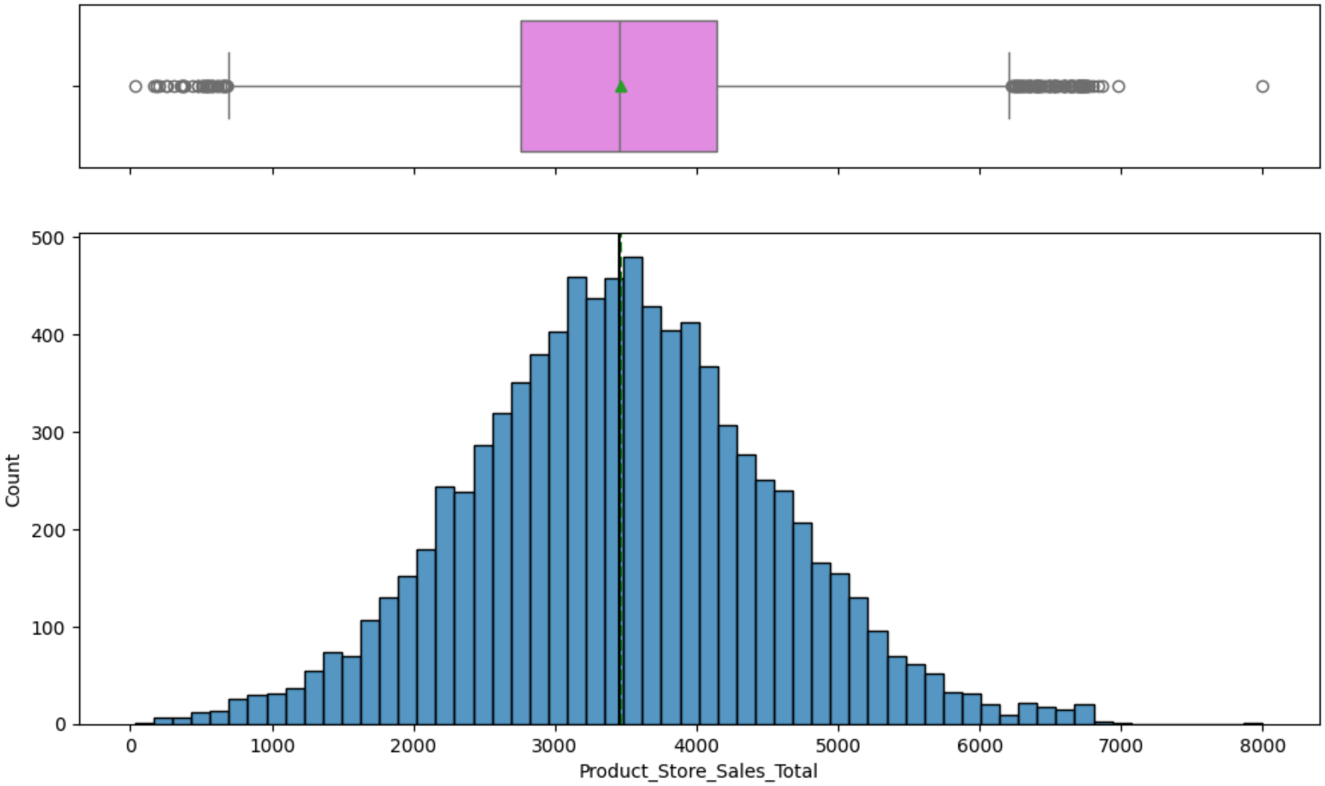
Product_MRP

```
histogram_boxplot(data, "Product_MRP") #Complete the code to plot the boxplot and histogram of Product_MRP
```



Product_Store_Sales_Total

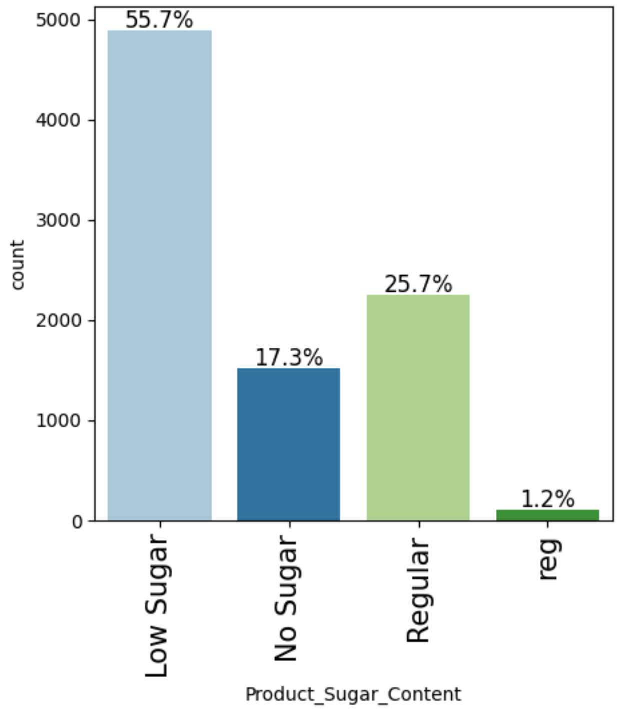
```
histogram_boxplot(data, "Product_Store_Sales_Total") #Complete the code to plot the boxplot and histogram of Product_Store_Sales_Total
```



```
# function to create labeled barplots


def labeled_barplot(data, feature, perc=False, n=None):
    """
    Barplot with percentage at the top

    data: dataframe
    feature: dataframe column
    perc: whether to display percentages instead of count (default is False)
    n: displays the top n category levels (default is None, i.e., display all levels)
    """

    total = len(data[feature])  # length of the column
    count = data[feature].nunique()
    if n is None:
        plt.figure(figsize=(count + 1, 5))
    else:
        plt.figure(figsize=(n + 1, 5))

    plt.xticks(rotation=90, fontsize=15)
    ax = sns.countplot(
        data=data,
        x=feature,
        palette="Paired",
        order=data[feature].value_counts().index[:n].sort_values(),
    )

    for p in ax.patches:
        if perc == True:
            label = "{:.1f}%".format(
                100 * p.get_height() / total
            )  # percentage of each class of the category
        else:
            label = p.get_height()  # count of each level of the category

        x = p.get_x() + p.get_width() / 2  # width of the plot
        y = p.get_height()  # height of the plot

        ax.annotate(
            label,
            (x, y),
            ha="center",
            va="center",
            size=12,
            xytext=(0, 5),
            textcoords="offset points",
        )  # annotate the percentage

    plt.show()  # show the plot
```

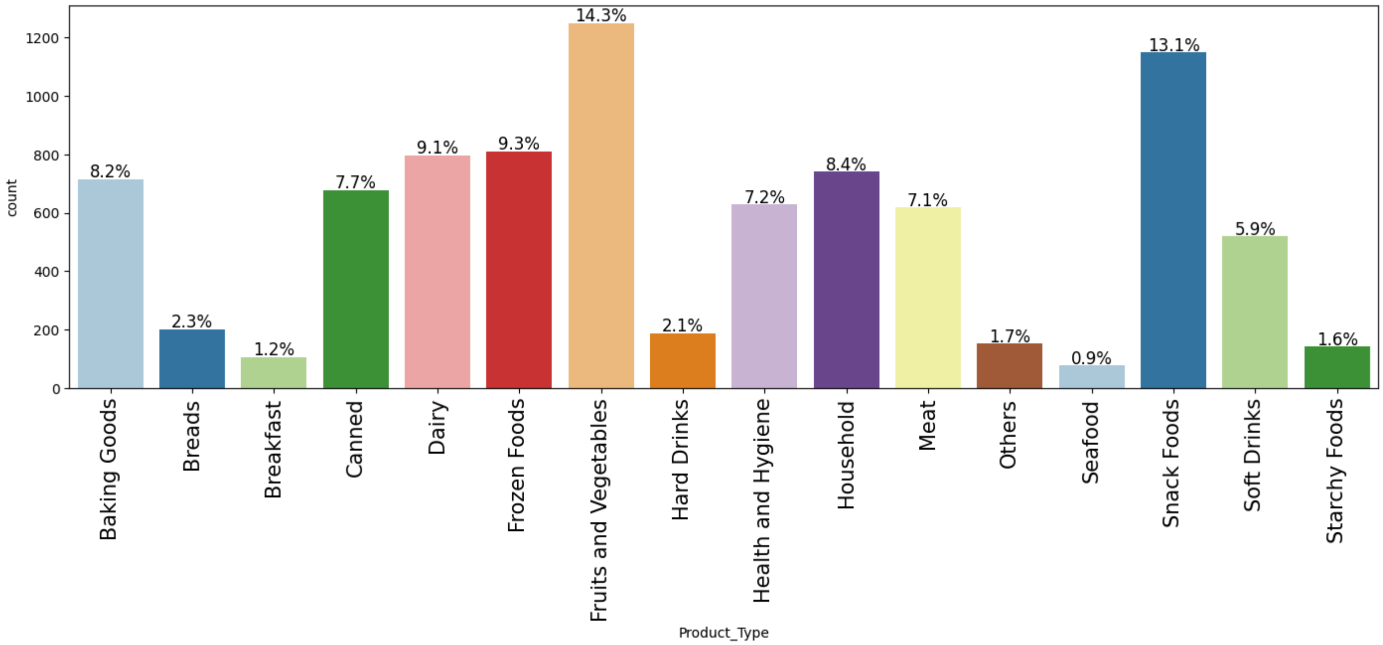
Product_Sugar_Content

```
labeled_barplot(data, "Product_Sugar_Content", perc=True)
```



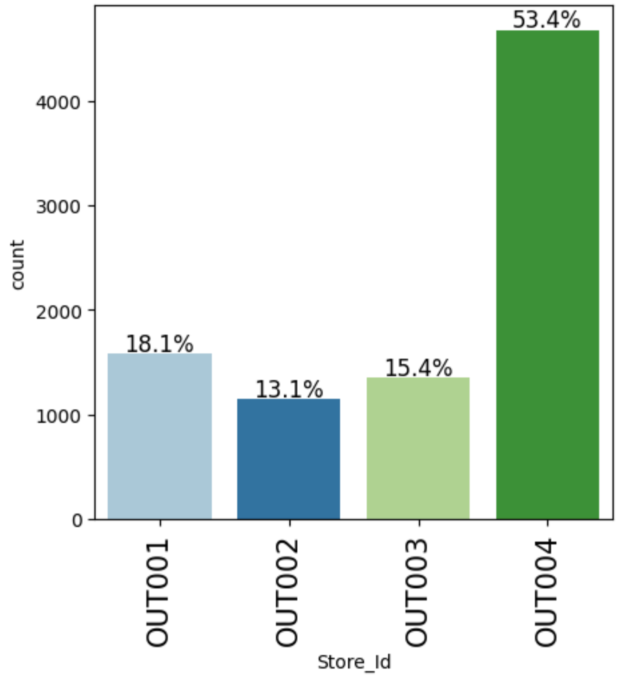
Product_Type

```
labeled_barplot(data, "Product_Type", perc=True) #Complete the code to plot the labelled barplot of Product_Type with the percentages being
```



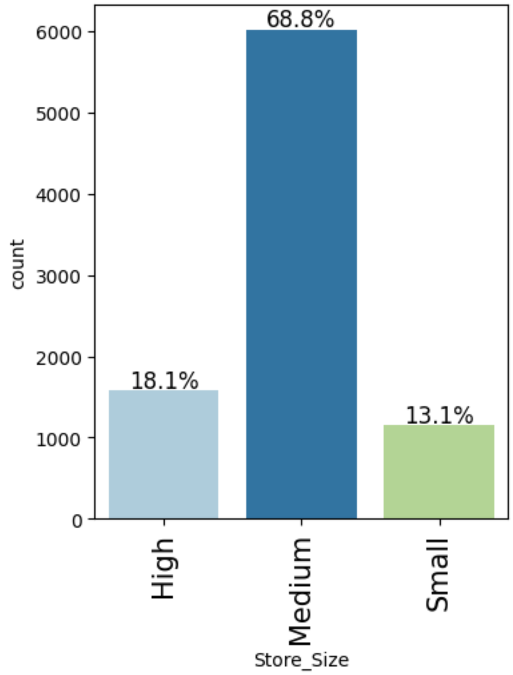
Store_Id

```
labeled_barplot(data, "Store_Id", perc=True) #Complete the code to plot the labelled barplot of Store_Id with the percentages being displaye
```



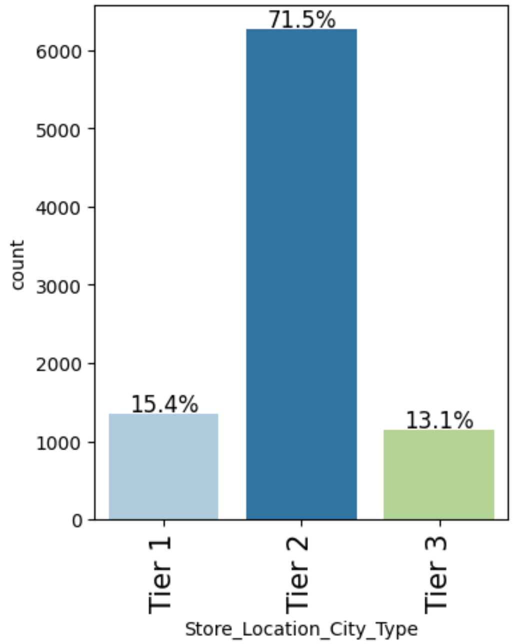
Store_Size

```
labeled_barplot(data, "Store_Size", perc=True) #Complete the code to plot the labelled barplot of Store_Size with the percentages being disp
```



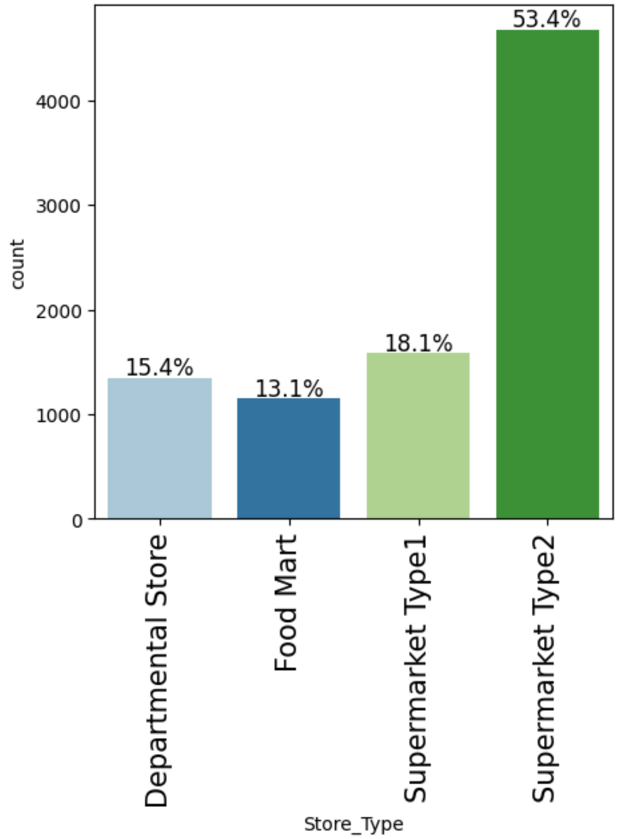
Store_Location_City_Type

```
labeled_barplot(data, "Store_Location_City_Type", perc=True) #Complete the code to plot the labelled barplot of Store_Location_City_Type wit
```



Store_Type

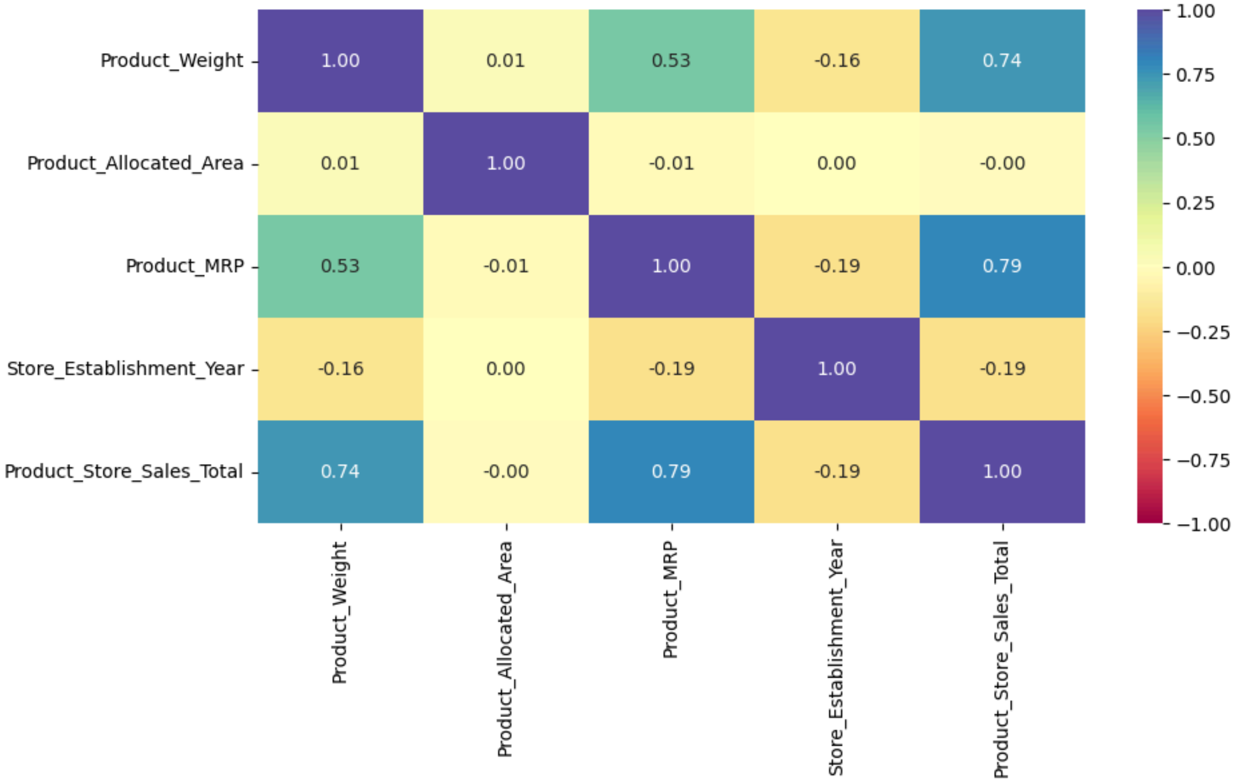
```
labeled_barplot(data, "Store_Type", perc=True) #Complete the code to plot the labelled barplot of Store_Type with the percentages being disp
```



Bivariate Analysis

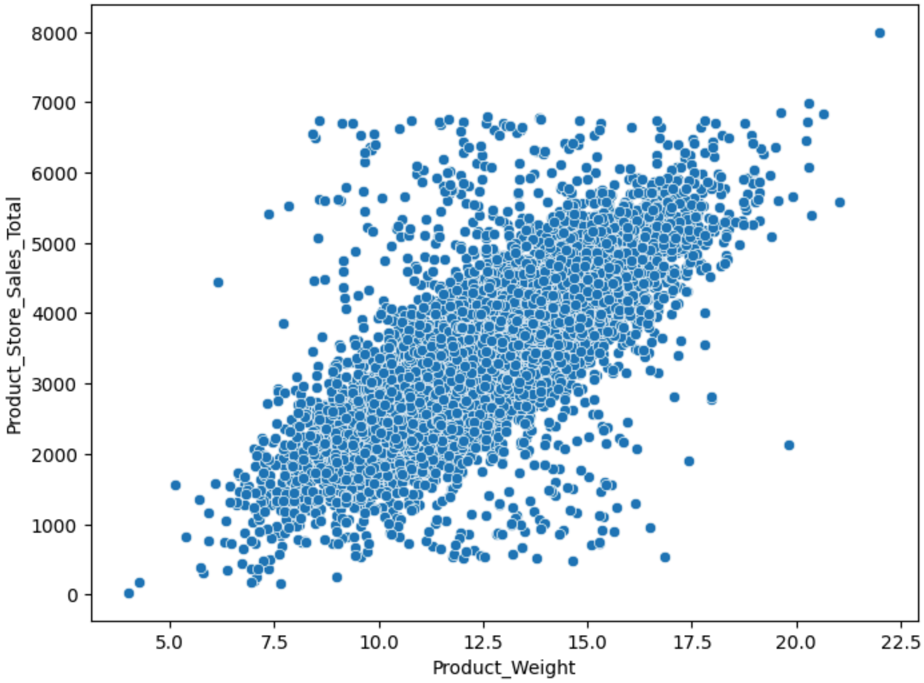
Correlation matrix

```
cols_list = data.select_dtypes(include=np.number).columns.tolist()

plt.figure(figsize=(10, 5))
sns.heatmap(
    data[cols_list].corr(), annot=True, vmin=-1, vmax=1, fmt=".2f", cmap="Spectral"
)
plt.show()
```

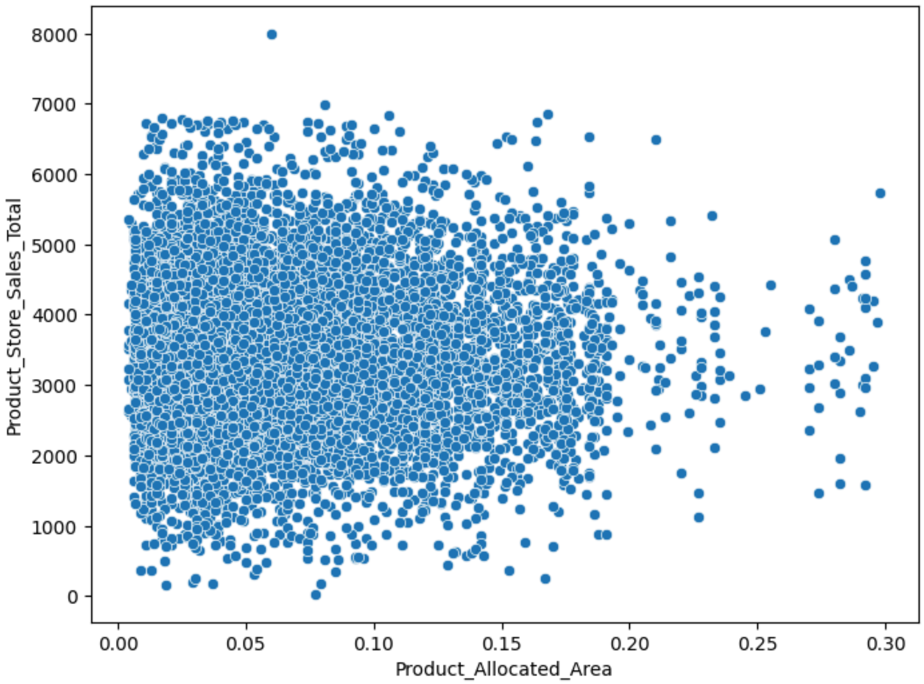


Let's check the distribution of our target variable i.e Product_Store_Sales_Total with the numeric columns

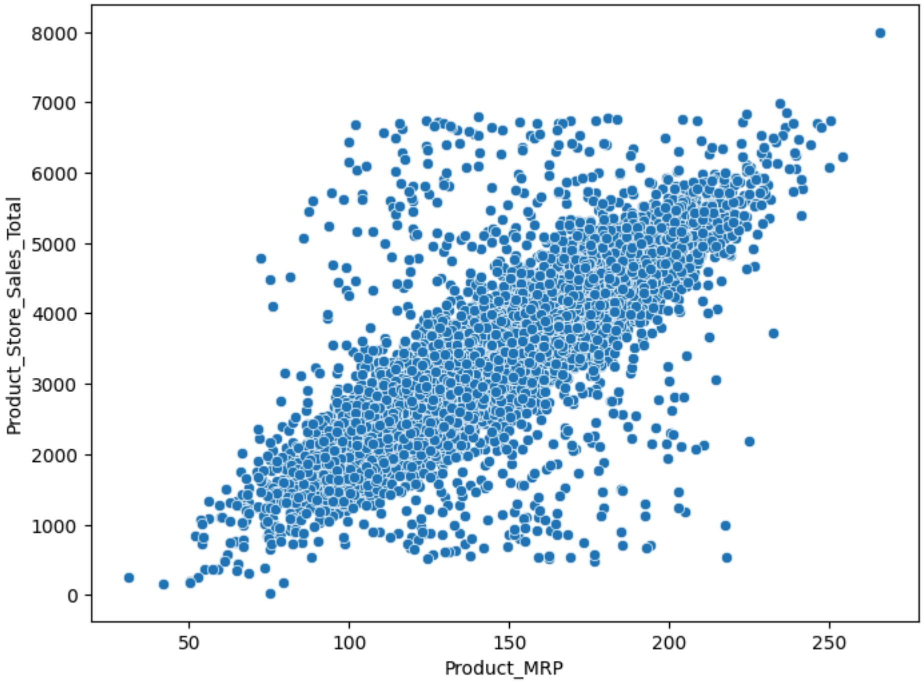
```
plt.figure(figsize=[8, 6])
sns.scatterplot(x=data.Product_Weight, y=data.Product_Store_Sales_Total)
plt.show()
```



```
plt.figure(figsize=[8, 6])
sns.scatterplot(x=data.Product_Allocated_Area, y=data.Product_Store_Sales_Total) #Complete the code to plot a scatterplot of Product_Allocat
plt.show()
```



```
plt.figure(figsize=[8, 6])
sns.scatterplot(x=data.Product_MRP, y=data.Product_Store_Sales_Total) #Complete the code to plot a scatterplot of Product_MRP and Product_St
plt.show()
```

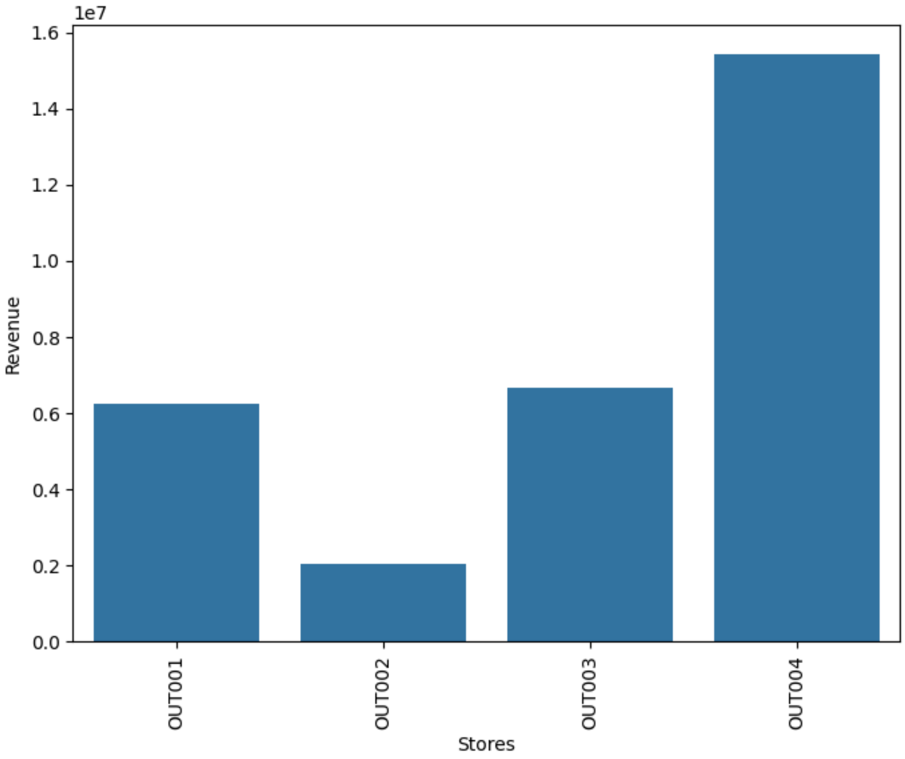


Let us see from which product type the company is generating most of the revenue

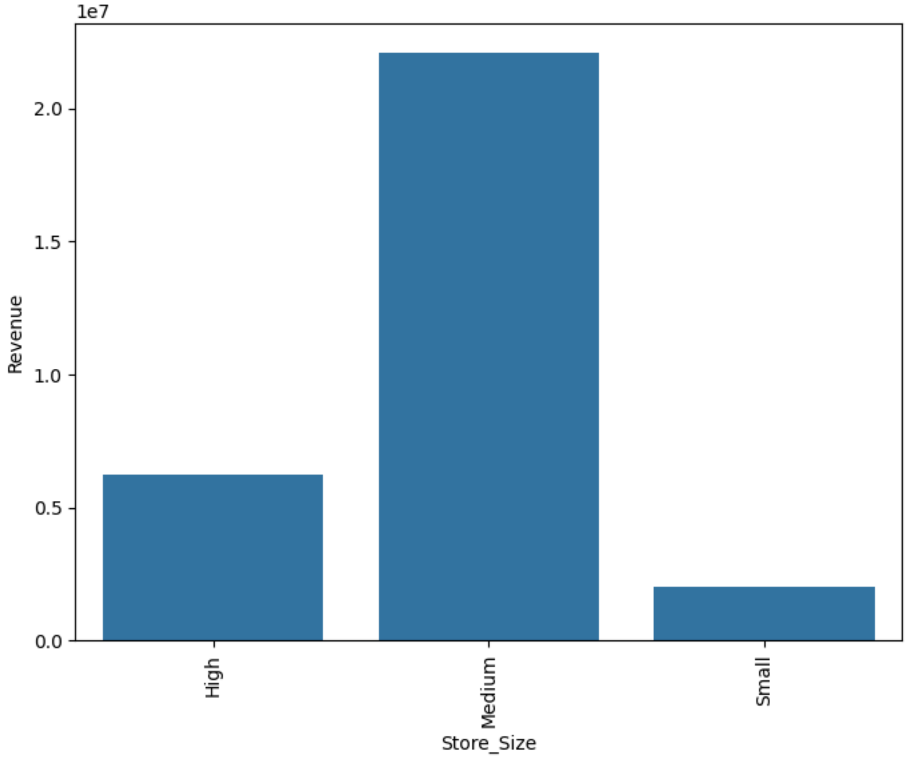
[ ] ↳ 2 cells hidden

Let us see from which type of stores and locations the revenue generation is more.

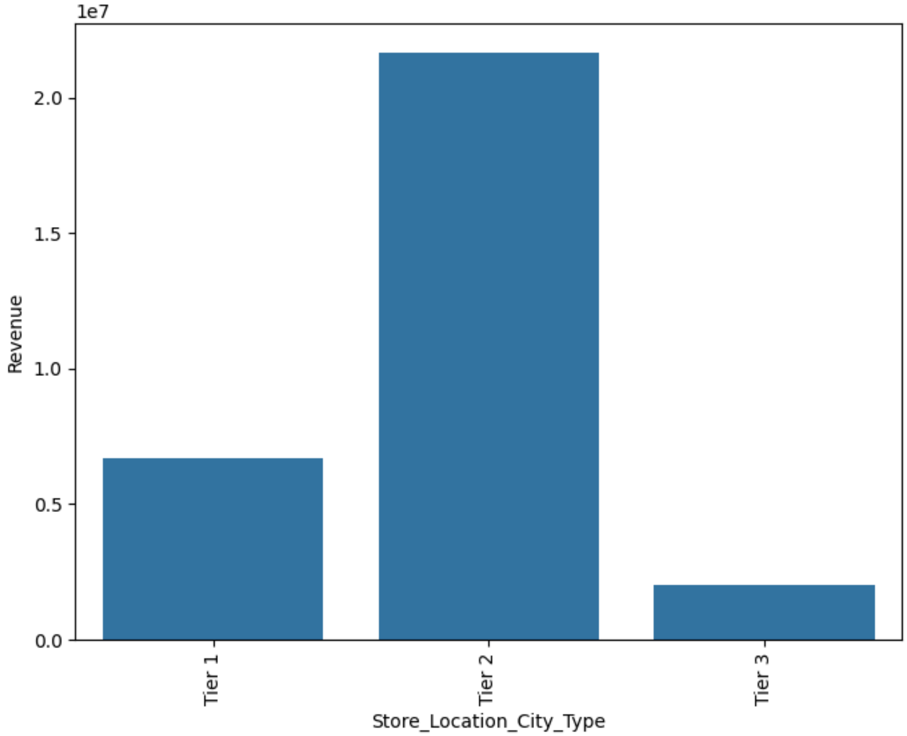
```
df_store_revenue = data.groupby(["Store_Id"], as_index=False)[
    "Product_Store_Sales_Total"
].sum() #Complete the code to perform a groupby on Store_Id and select Product_Store_Sales_Total
plt.figure(figsize=[8, 6])
plt.xticks(rotation=90)
r = sns.barplot(
    x=df_store_revenue.Store_Id, y=df_store_revenue.Product_Store_Sales_Total
)
r.set_xlabel("Stores")
r.set_ylabel("Revenue")
plt.show()
```



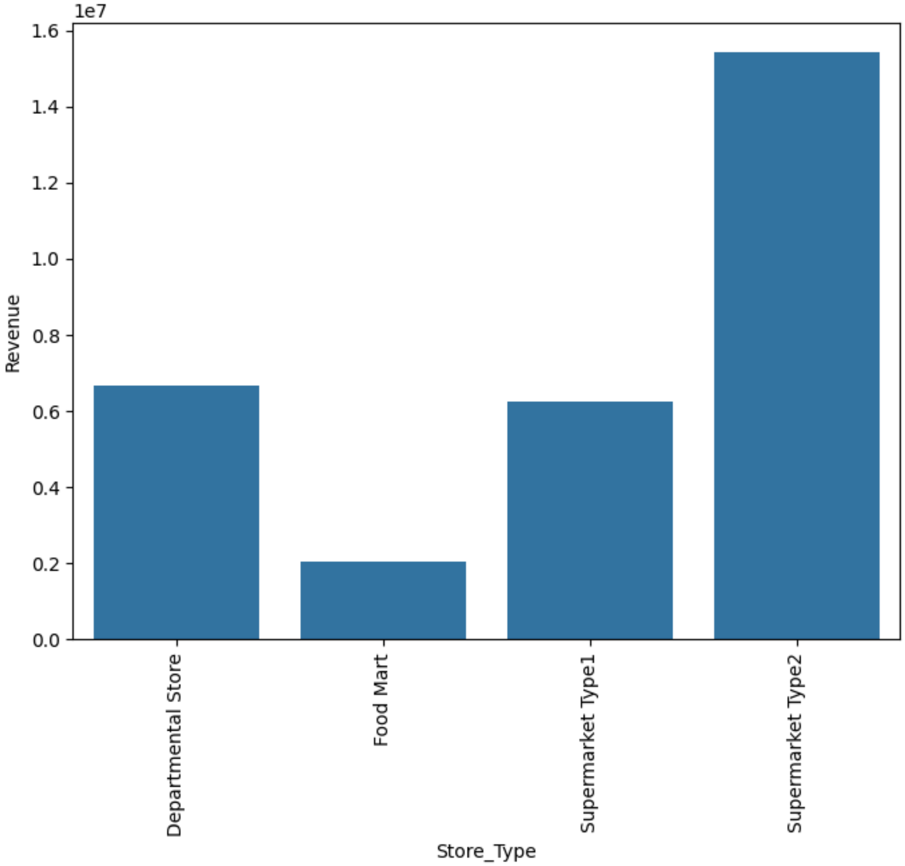
```
df_revenue3 = data.groupby(["Store_Size"], as_index=False)[
    "Product_Store_Sales_Total"
].sum() #Complete the code to perform a groupby on Store_Size and select Product_Store_Sales_Total
plt.figure(figsize=[8, 6])
plt.xticks(rotation=90)
c = sns.barplot(x=df_revenue3.Store_Size, y=df_revenue3.Product_Store_Sales_Total)
c.set_xlabel("Store_Size")
c.set_ylabel("Revenue")
plt.show()
```



```
df_revenue4 = data.groupby(["Store_Location_City_Type"], as_index=False)[
    "Product_Store_Sales_Total"
].sum() #Complete the code to perform a groupby on Store_Location_City_Type and select Product_Store_Sales_Total
plt.figure(figsize=[8, 6])
plt.xticks(rotation=90)
d = sns.barplot(
    x=df_revenue4.Store_Location_City_Type, y=df_revenue4.Product_Store_Sales_Total
)
d.set_xlabel("Store_Location_City_Type")
d.set_ylabel("Revenue")
plt.show()
```

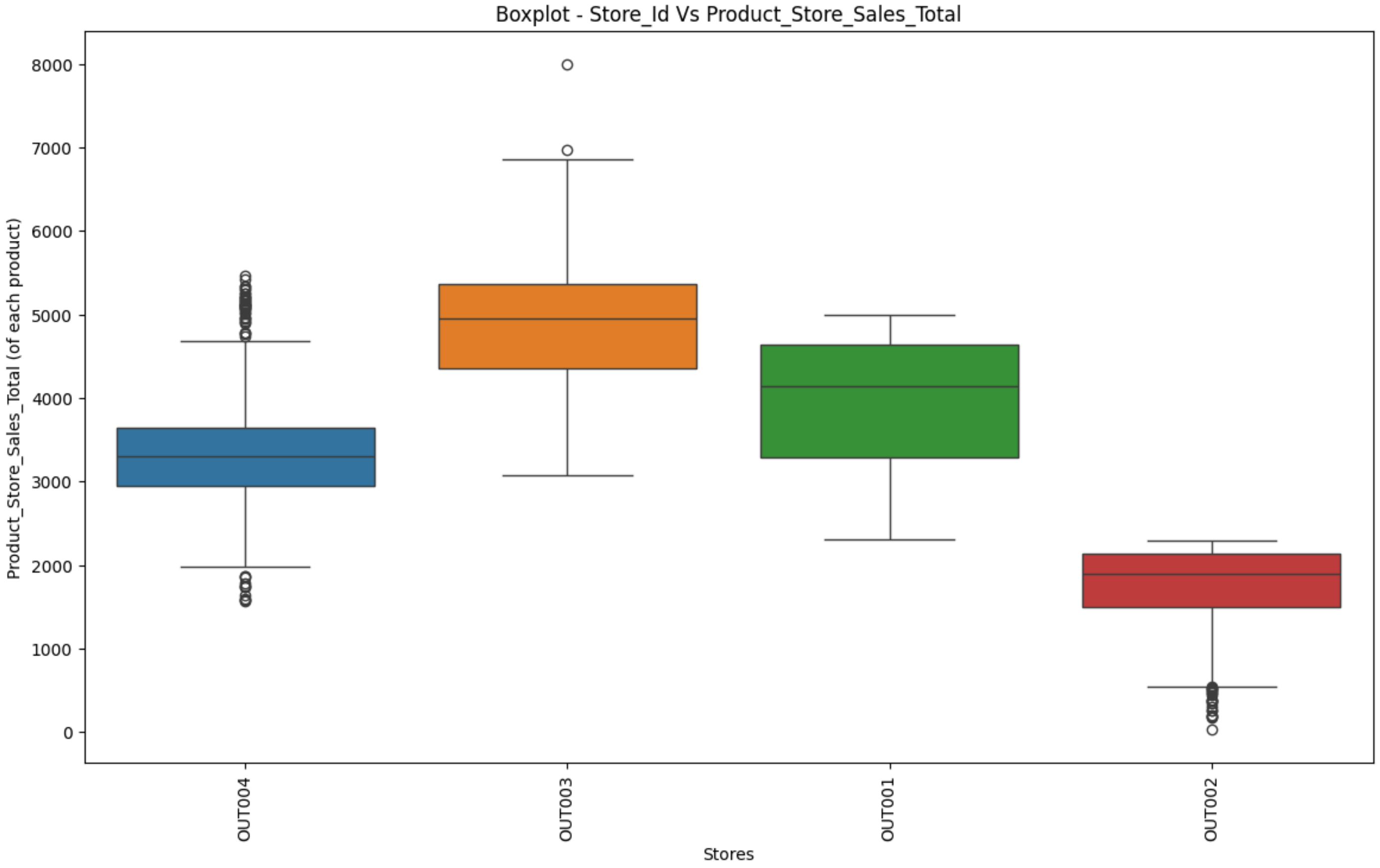


```
df_revenue5 = data.groupby(["Store_Type"], as_index=False)[
    "Product_Store_Sales_Total"
].sum() #Complete the code to perform a groupby on Store_Type and select Product_Store_Sales_Total
plt.figure(figsize=[8, 6])
plt.xticks(rotation=90)
e = sns.barplot(x=df_revenue5.Store_Type, y=df_revenue5.Product_Store_Sales_Total)
e.set_xlabel("Store_Type")
e.set_ylabel("Revenue")
plt.show()
```

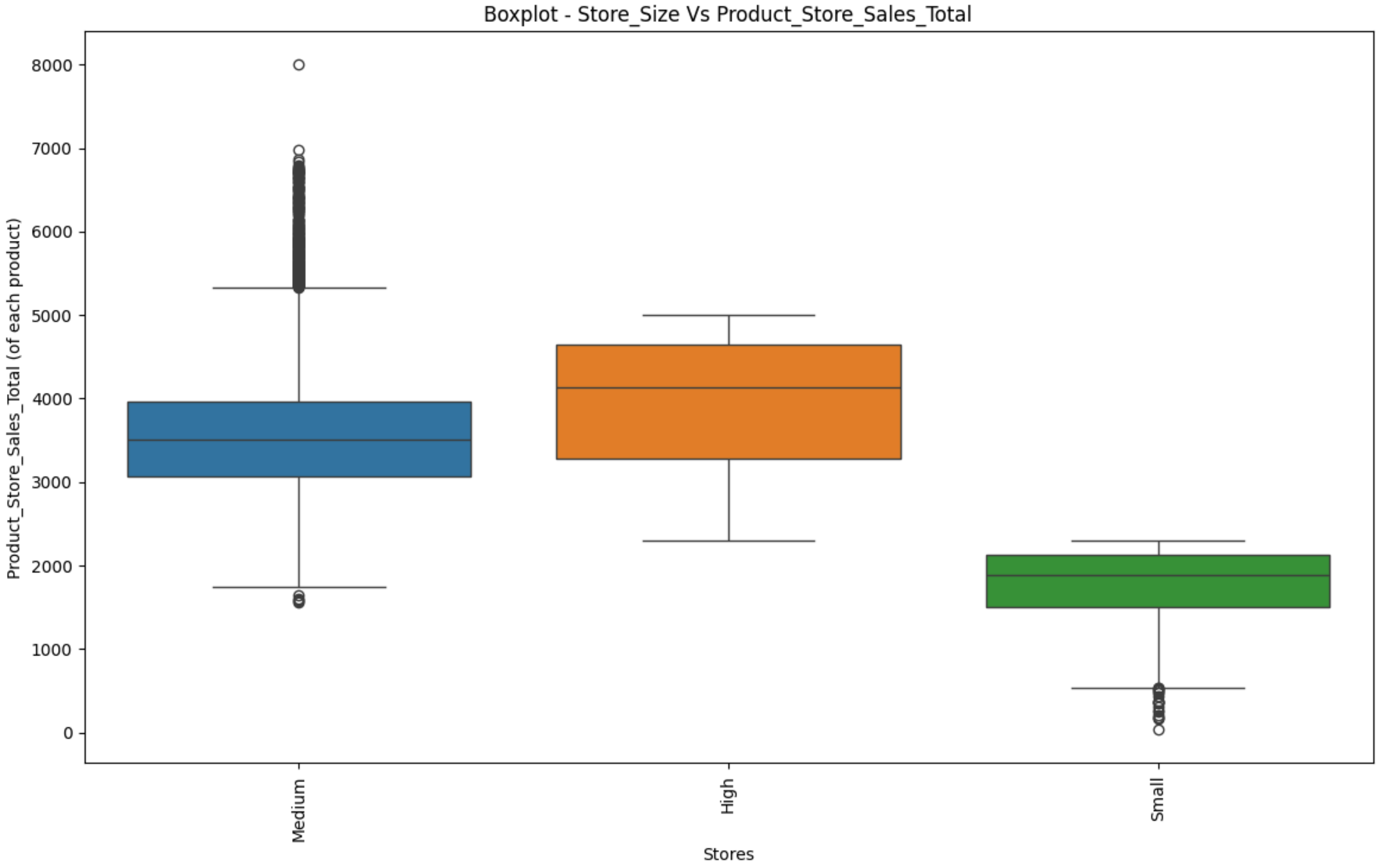


Let's check the distribution of our target variable i.e Product_Store_Sales_Total with the other categorical columns

```
plt.figure(figsize=[14, 8])
sns.boxplot(data=data, x="Store_Id", y="Product_Store_Sales_Total", hue = "Store_Id")
plt.xticks(rotation=90)
plt.title("Boxplot - Store_Id Vs Product_Store_Sales_Total")
plt.xlabel("Stores")
plt.ylabel("Product_Store_Sales_Total (of each product)")
plt.show()
```

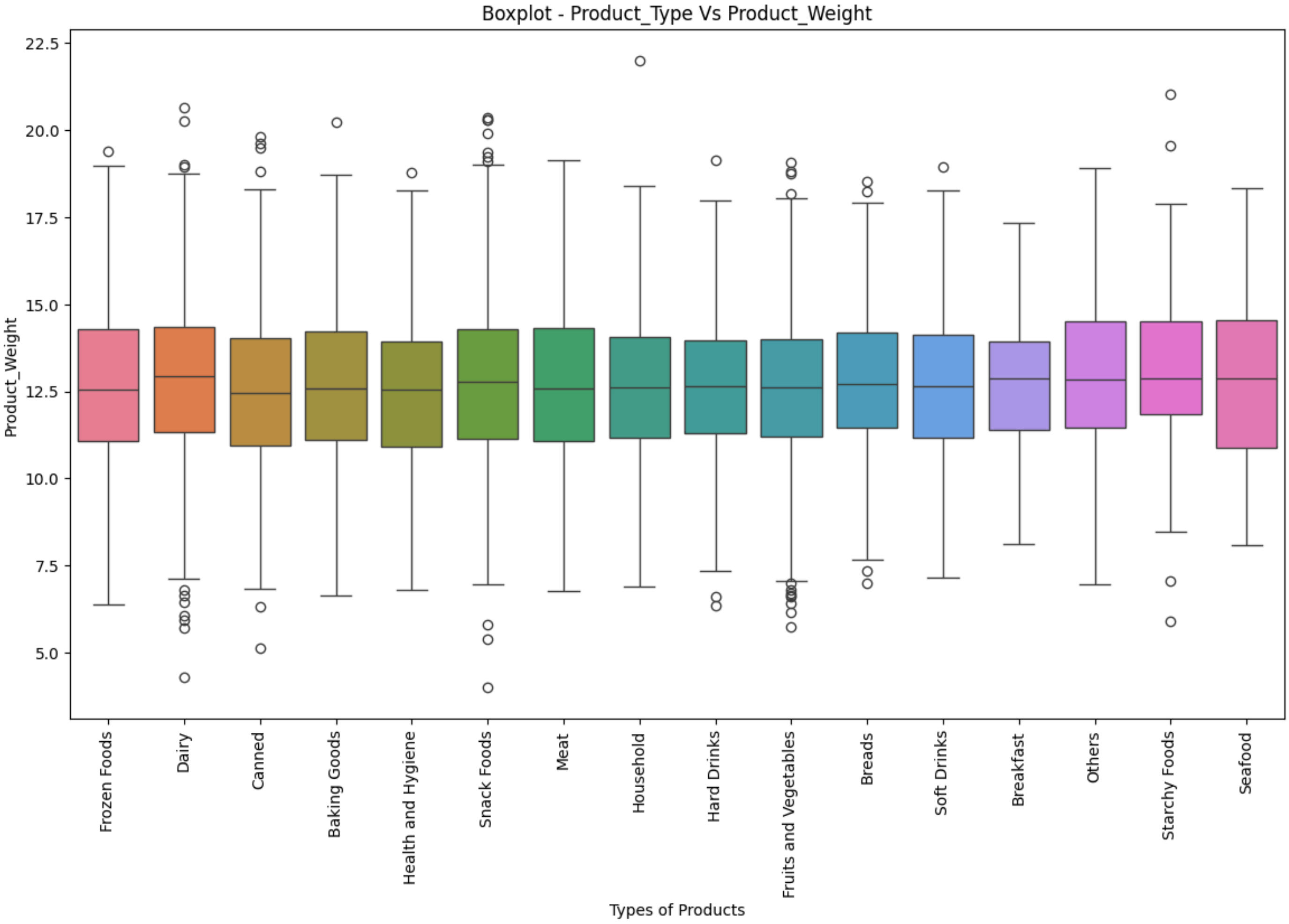


```
plt.figure(figsize=[14, 8])
sns.boxplot(data = data, x = "Store_Size", y = "Product_Store_Sales_Total", hue = "Store_Size") #Complet the code to plot the boxplot with x
plt.xticks(rotation=90)
plt.title("Boxplot - Store_Size Vs Product_Store_Sales_Total")
plt.xlabel("Stores")
plt.ylabel("Product_Store_Sales_Total (of each product)")
plt.show()
```



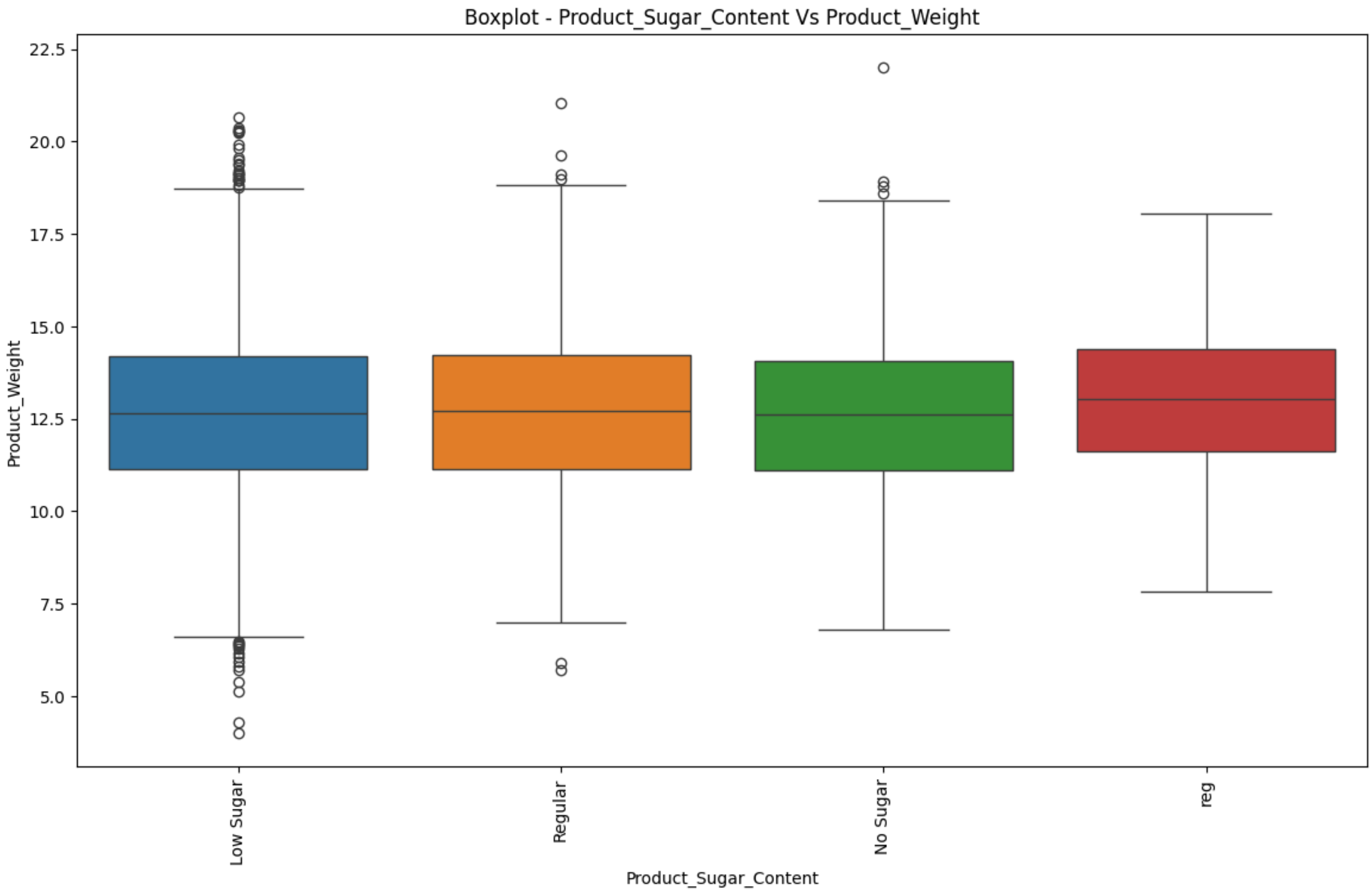
Let's now try to find out some relationship between the other columns

```
plt.figure(figsize=[14, 8])
sns.boxplot(data = data, x = "Product_Type", y = "Product_Weight", hue = "Product_Type") #Complete the code to plot the boxplot with x as Pr
plt.xticks(rotation=90)
plt.title("Boxplot - Product_Type Vs Product_Weight")
plt.xlabel("Types of Products")
plt.ylabel("Product_Weight")
plt.show()
```



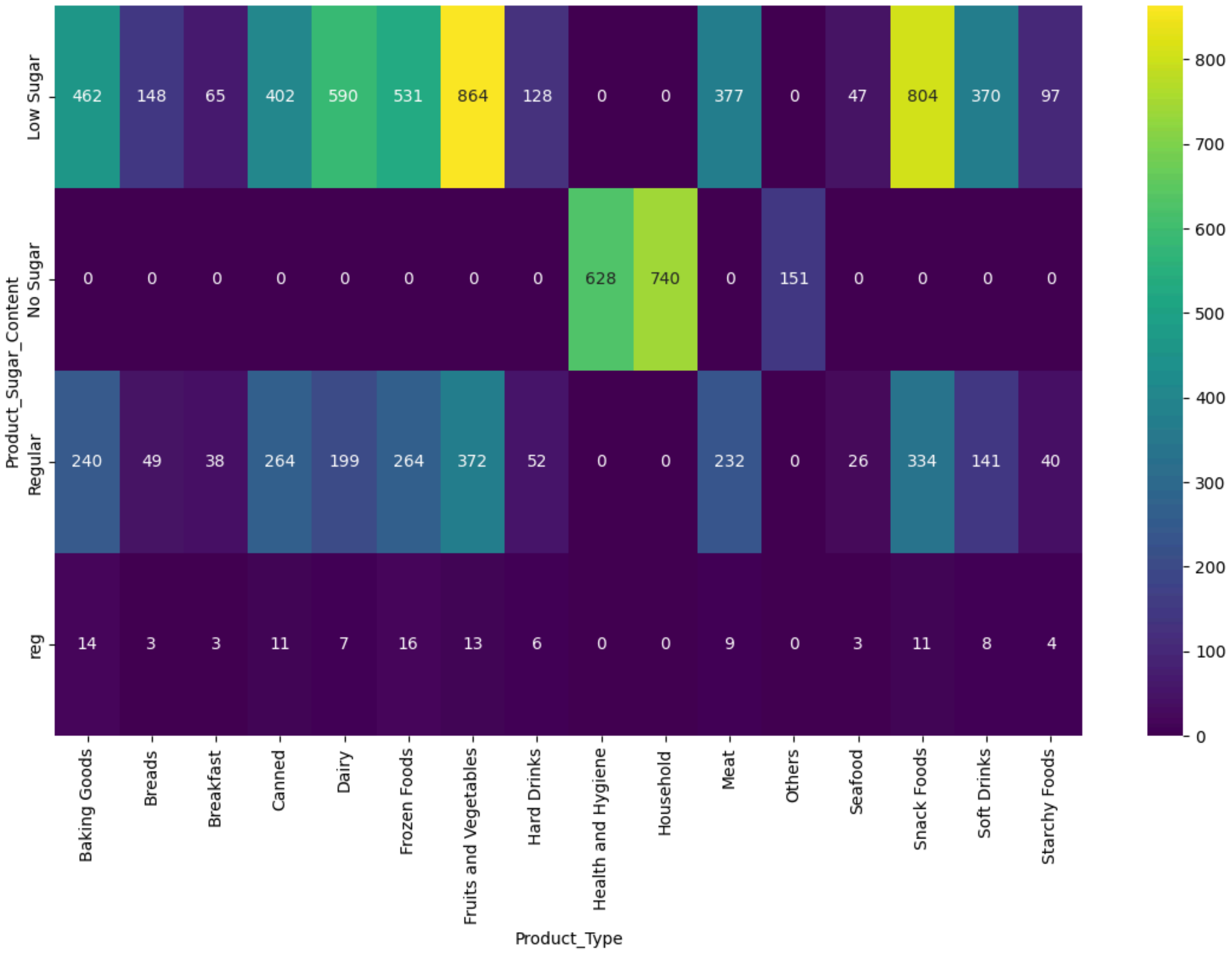
Let's find out whether there is some relationship between the weight of the product and its sugar content

```
plt.figure(figsize=[14, 8])
sns.boxplot(data = data, x = "Product_Sugar_Content", y = "Product_Weight", hue = "Product_Sugar_Content") #Complete the code to plot the bc
plt.xticks(rotation=90)
plt.title("Boxplot - Product_Sugar_Content Vs Product_Weight")
plt.xlabel("Product_Sugar_Content")
plt.ylabel("Product_Weight")
plt.show()
```



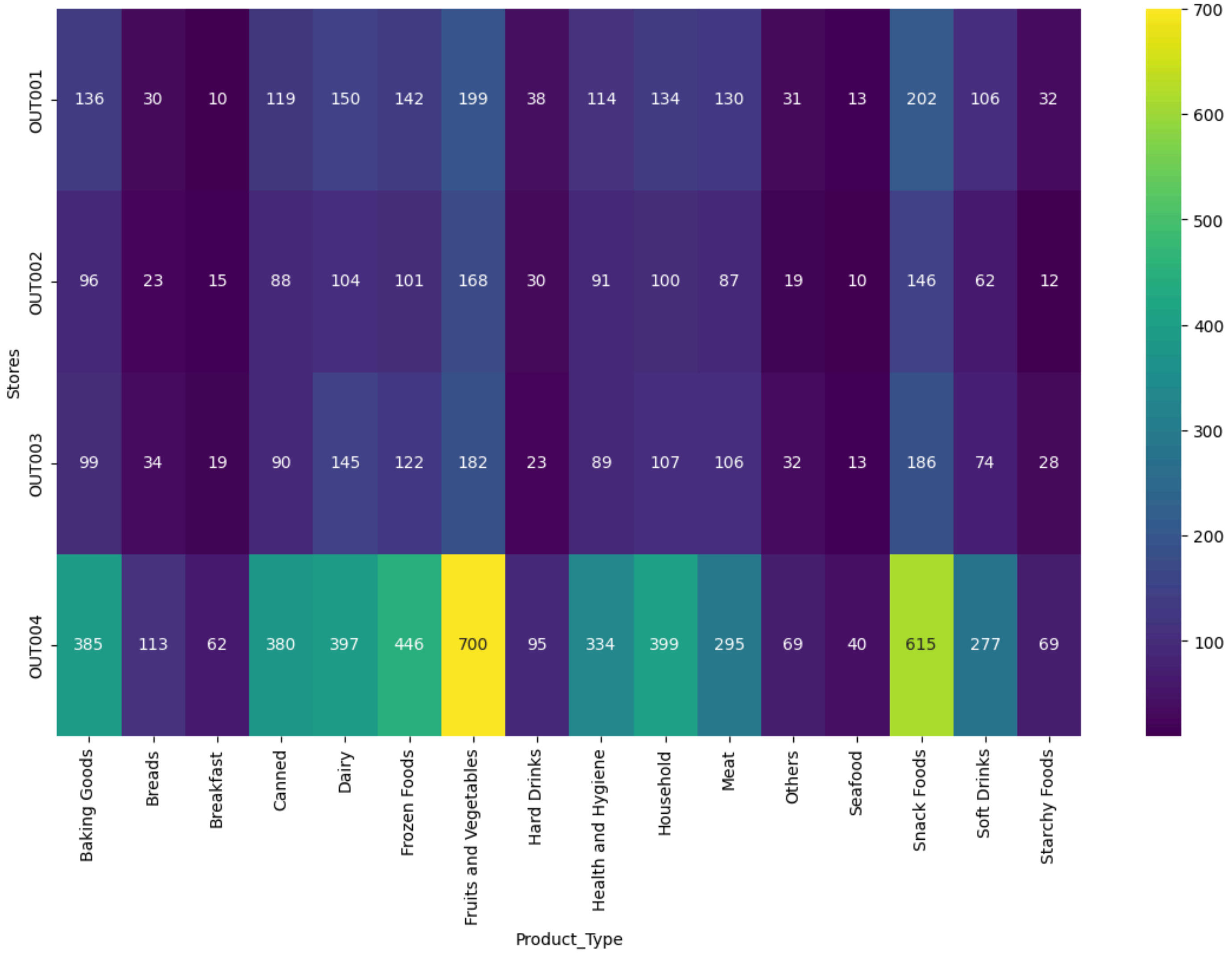
Let's analyze the sugar content of different product types

```
plt.figure(figsize=(14, 8))
sns.heatmap(
    pd.crosstab(data["Product_Sugar_Content"], data["Product_Type"]),
    annot=True,
    fmt="g",
    cmap="viridis",
)
plt.ylabel("Product_Sugar_Content")
plt.xlabel("Product_Type")
plt.show()
```



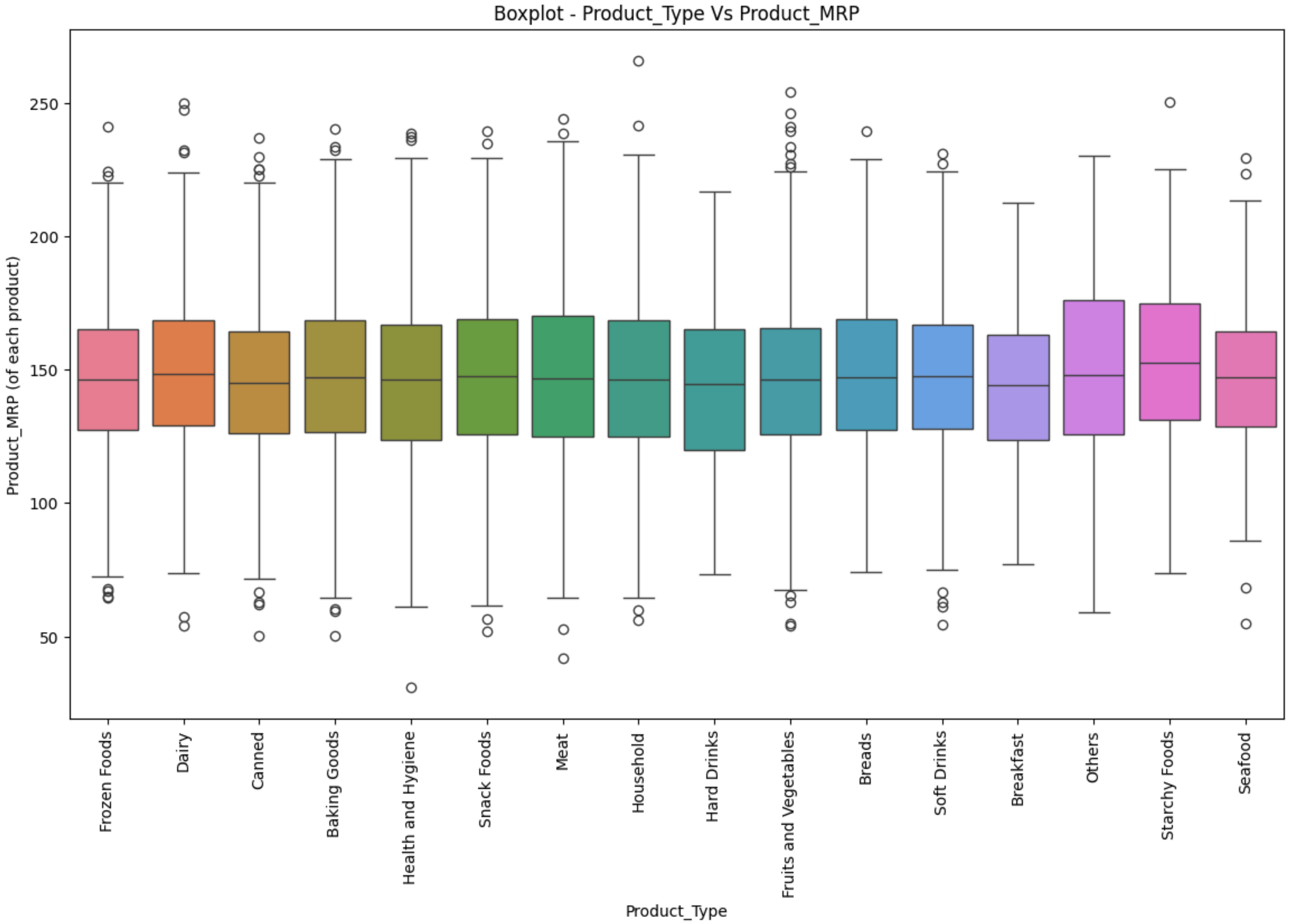
Let's find out how many items of each product type has been sold in each of the stores

```
plt.figure(figsize=(14, 8))
sns.heatmap(
    pd.crosstab(data["Store_Id"], data["Product_Type"]), #Complete the code to perform a crosstab operation between Store_Id and Product_Typ
    annot=True,
    fmt="g",
    cmap="viridis",
)
plt.ylabel("Stores")
plt.xlabel("Product_Type")
plt.show()
```



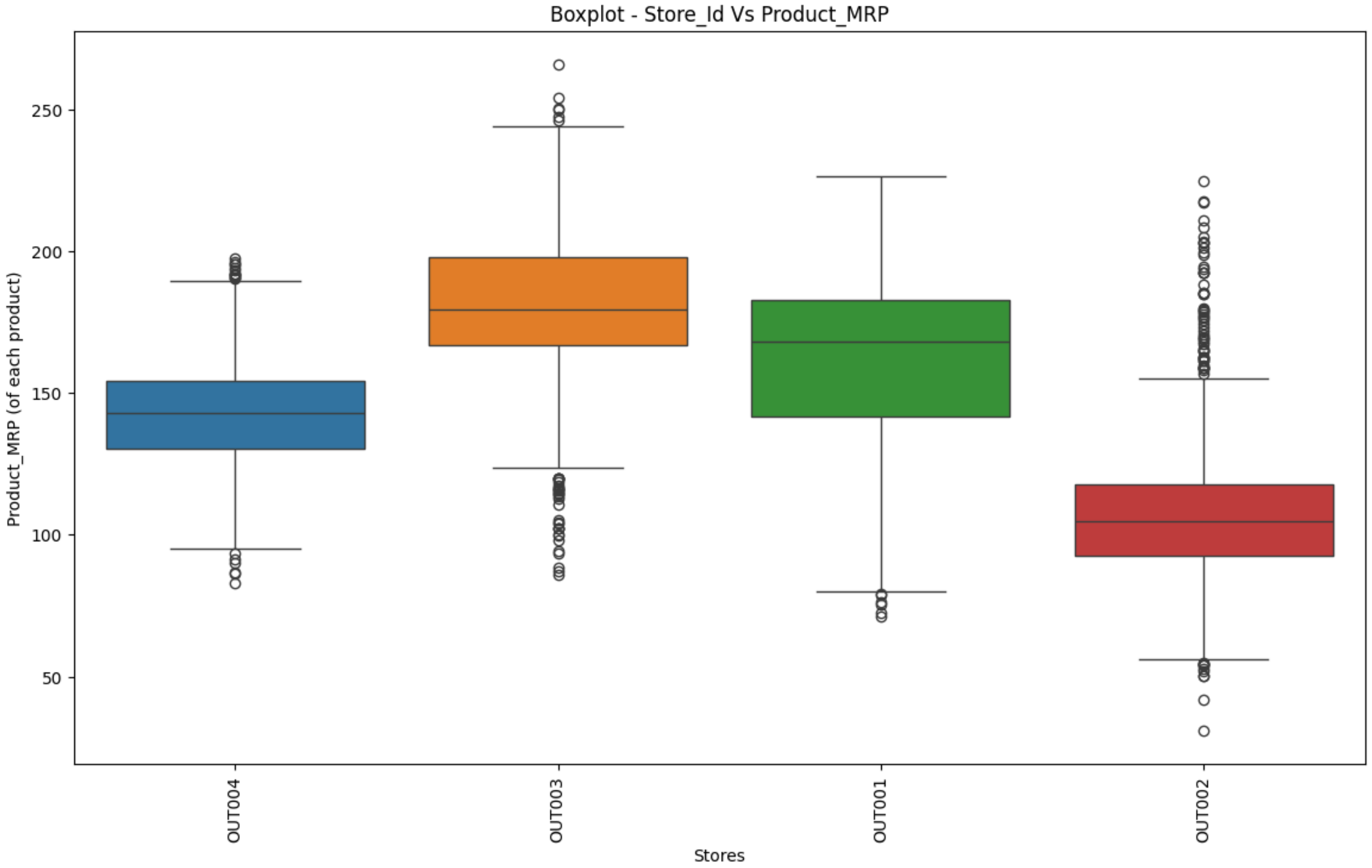
Different product types have different prices. Let's analyze the trend.

```
plt.figure(figsize=[14, 8])
sns.boxplot(data = data, x = "Product_Type", y = "Product_MRP", hue = "Product_Type") #Complete the code to plot a boxplot with x as Product
plt.xticks(rotation=90)
plt.title("Boxplot - Product_Type Vs Product_MRP")
plt.xlabel("Product_Type")
plt.ylabel("Product_MRP (of each product)")
plt.show()
```



Let's find out how the Product_MRP varies with the different stores

```
plt.figure(figsize=[14, 8])
sns.boxplot(data = data, x = "Store_Id", y = "Product_MRP", hue = "Store_Id") #Complete the code to plot the boxplot with x as Store_Id , y
plt.xticks(rotation=90)
plt.title("Boxplot - Store_Id Vs Product_MRP")
plt.xlabel("Stores")
plt.ylabel("Product_MRP (of each product)")
plt.show()
```



Let's delve deeper and do a detailed analysis of each of the stores.

OUT001

```
data.loc[data["Store_Id"] == "OUT001"].describe(include="all").T
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Product_Id	1586	1586	NC7187	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Weight	1586.0	NaN	NaN	NaN	13.458865	2.064975	6.16	12.0525	13.96	14.95	17.97
Product_Sugar_Content	1586	4	Low Sugar	845	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Allocated_Area	1586.0	NaN	NaN	NaN	0.068768	0.047131	0.004	0.033	0.0565	0.094	0.295
Product_Type	1586	16	Snack Foods	202	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_MRP	1586.0	NaN	NaN	NaN	160.514054	30.359059	71.35	141.72	168.32	182.9375	226.59
Store_Id	1586	1	OUT001	1586	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Establishment_Year	1586.0	NaN	NaN	NaN	1987.0	0.0	1987.0	1987.0	1987.0	1987.0	1987.0
Store_Size	1586	1	High	1586	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Location_City_Type	1586	1	Tier 2	1586	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Type	1586	1	Supermarket Type1	1586	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Store_Sales_Total	1586.0	NaN	NaN	NaN	3923.778802	904.62901	2300.56	3285.51	4139.645	4639.4	4997.63

Observations

- OUT001 is a store of Supermarket Type 1 which is located in a Tier 2 city and has store size as high. It was established in 1987.
- OUT001 has sold products whose MRP range from 71 to 227.

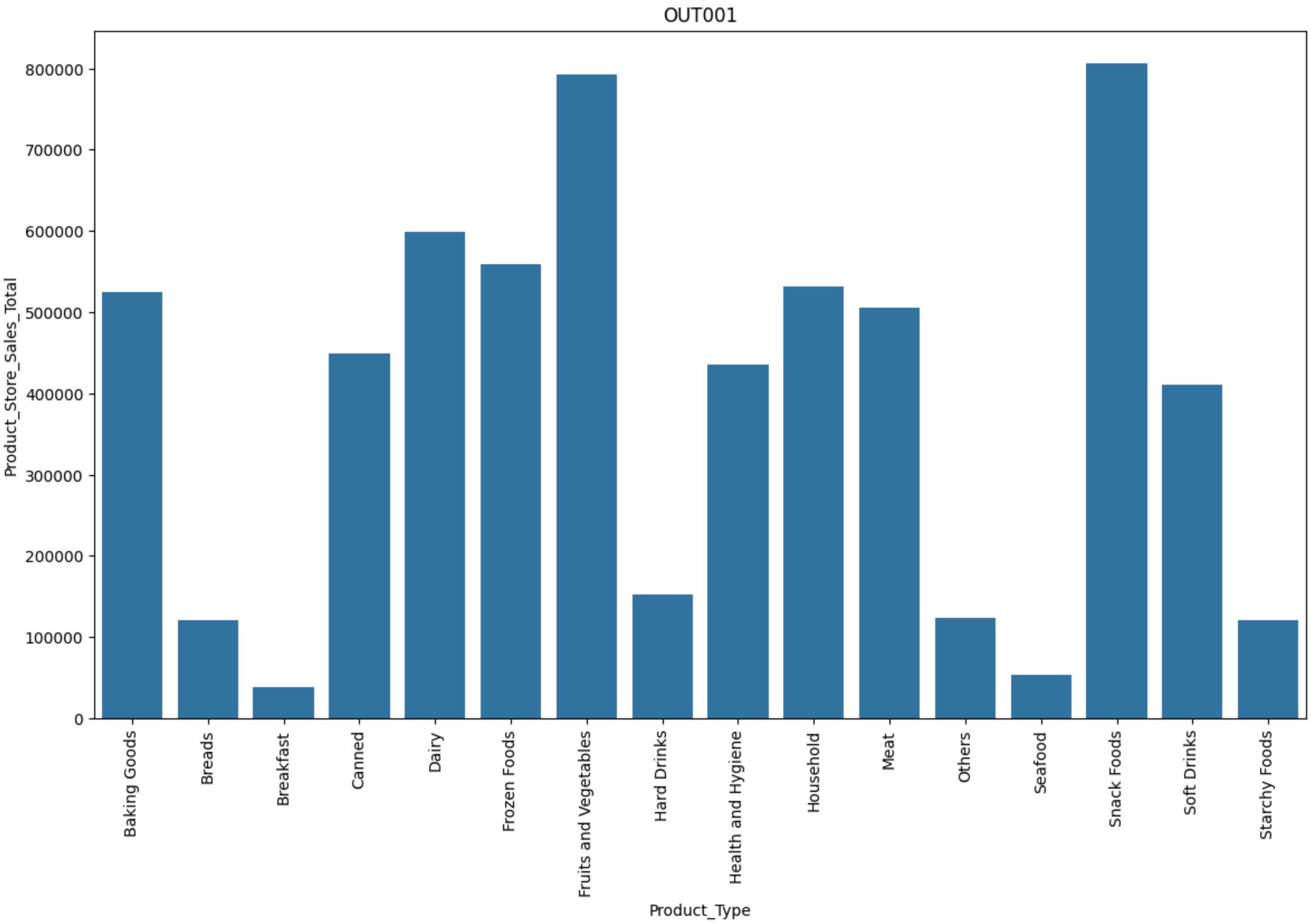
- Snack Foods have been sold the highest number of times in OUT001.
- The revenue generated from each product at OUT001 ranges from 2300 to 5000.

```
data.loc[data["Store_Id"] == "OUT001", "Product_Store_Sales_Total"].sum()
```

```
np.float64(6223113.18)
```

OUT001 has generated total revenue of 6223113 from the sales of goods.

```
df_OUT001 = (
    data.loc[data["Store_Id"] == "OUT001"]
    .groupby(["Product_Type"], as_index=False)["Product_Store_Sales_Total"]
    .sum()
)
plt.figure(figsize=[14, 8])
plt.xticks(rotation=90)
plt.xlabel("Product_Type")
plt.ylabel("Product_Store_Sales_Total")
plt.title("OUT001")
sns.barplot(x=df_OUT001.Product_Type, y=df_OUT001.Product_Store_Sales_Total)
plt.show()
```



- OUT001 has generated the highest revenue from the sale of fruits and vegetables and snack foods. Both the categories have contributed around 800000 each.

OUT002

```
data.loc[data["Store_Id"] == "OUT002"].describe(include="all").T
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Product_Id	1152	1152	NC2769	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Weight	1152.0	NaN	NaN	NaN	9.911241	1.799846	4.0	8.7675	9.795	10.89	19.82
Product_Sugar_Content	1152	4	Low Sugar	658	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Allocated_Area	1152.0	NaN	NaN	NaN	0.067747	0.047567	0.006	0.031	0.0545	0.09525	0.292
Product_Type	1152	16	Fruits and Vegetables	168	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_MRP	1152.0	NaN	NaN	NaN	107.080634	24.912333	31.0	92.8275	104.675	117.8175	224.93
Store_Id	1152	1	OUT002	1152	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Establishment_Year	1152.0	NaN	NaN	NaN	1998.0	0.0	1998.0	1998.0	1998.0	1998.0	1998.0
Store_Size	1152	1	Small	1152	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Location_City_Type	1152	1	Tier 3	1152	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Type	1152	1	Food Mart	1152	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Store_Sales_Total	1152.0	NaN	NaN	NaN	1762.942465	462.862431	33.0	1495.4725	1889.495	2133.6225	2299.63

Observations

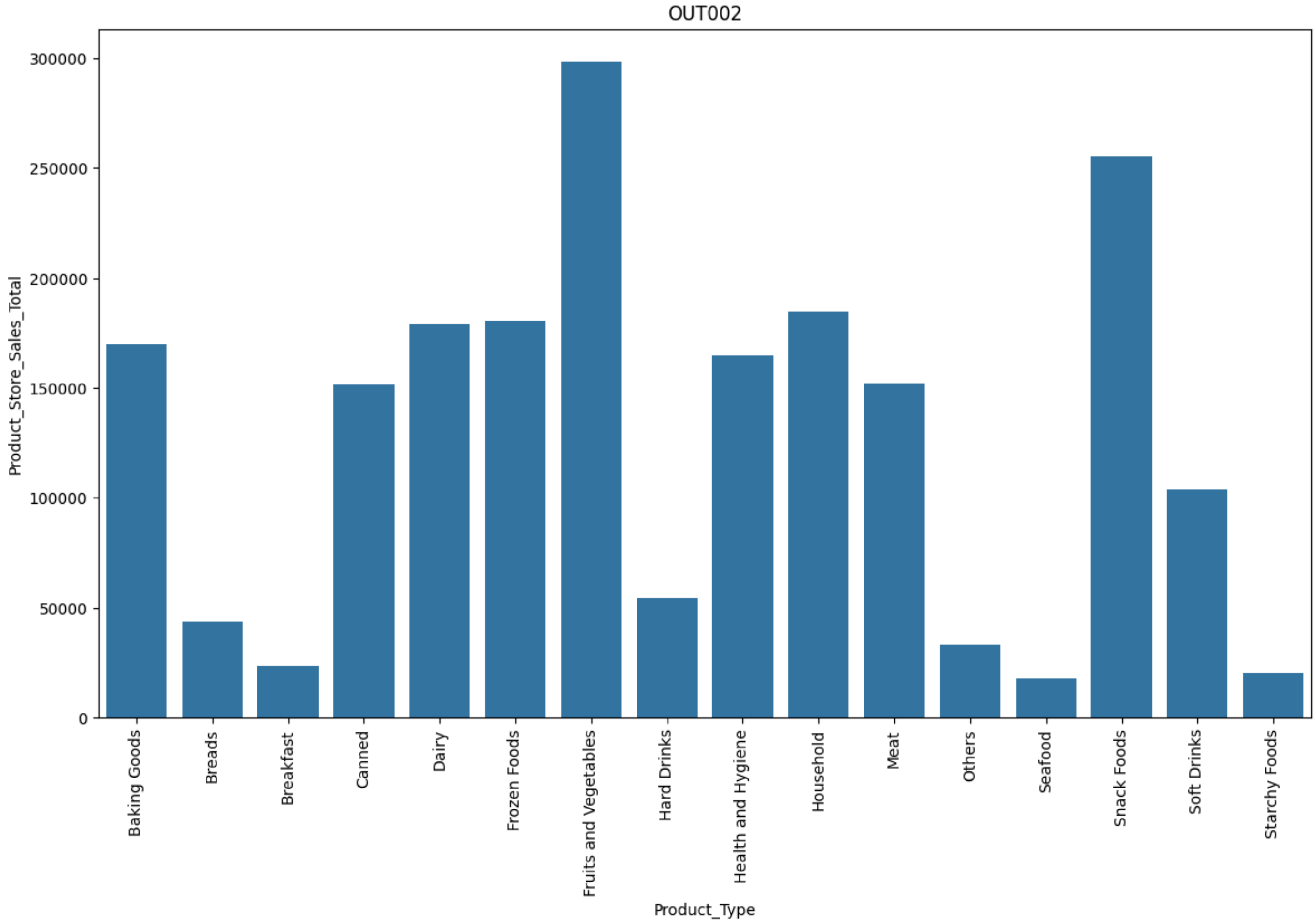
- OUT002 is a food mart which is located in a Tier 3 city and has store size as small. It was established in 1998.
- OUT002 has sold products whose MRP range from 31 to 225.
- Fruits and vegetables have been sold the highest number of times in OUT002.
- The revenue generated from each product at OUT002 ranges from 33 to 2300.

```
data.loc[data["Store_Id"] == "OUT002", "Product_Store_Sales_Total"].sum()
```

```
np.float64(2030909.72)
```

OUT002 has generated total revenue of 2030910 from the sales of goods.

```
df_OUT002 = (
    data.loc[data["Store_Id"] == "OUT002"]
    .groupby(["Product_Type"], as_index=False)["Product_Store_Sales_Total"]
    .sum()
)
plt.figure(figsize=[14, 8])
plt.xticks(rotation=90)
plt.xlabel("Product_Type")
plt.ylabel("Product_Store_Sales_Total")
plt.title("OUT002")
sns.barplot(x=df_OUT002.Product_Type, y=df_OUT002.Product_Store_Sales_Total)
plt.show()
```



- OUT002 has generated the highest revenue from the sale of fruits and vegetables (~ 300000) followed by snack foods (~ 250000).

OUT003

```
data.loc[data["Store_Id"] == "OUT003"].describe(include="all").T
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Product_Id	1349	1349	NC522	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Weight	1349.0	NaN	NaN	NaN	15.103692	1.893531	7.35	14.02	15.18	16.35	22.0
Product_Sugar_Content	1349	4	Low Sugar	750	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Allocated_Area	1349.0	NaN	NaN	NaN	0.068637	0.048708	0.004	0.031	0.057	0.094	0.298
Product_Type	1349	16	Snack Foods	186	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_MRP	1349.0	NaN	NaN	NaN	181.358725	24.796429	85.88	166.92	179.67	198.07	266.0
Store_Id	1349	1	OUT003	1349	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Establishment_Year	1349.0	NaN	NaN	NaN	1999.0	0.0	1999.0	1999.0	1999.0	1999.0	1999.0
Store_Size	1349	1	Medium	1349	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Location_City_Type	1349	1	Tier 1	1349	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Type	1349	1	Departmental Store	1349	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Store_Sales_Total	1349.0	NaN	NaN	NaN	4946.966323	677.539953	3069.24	4355.39	4958.29	5366.59	8000.0

Observations

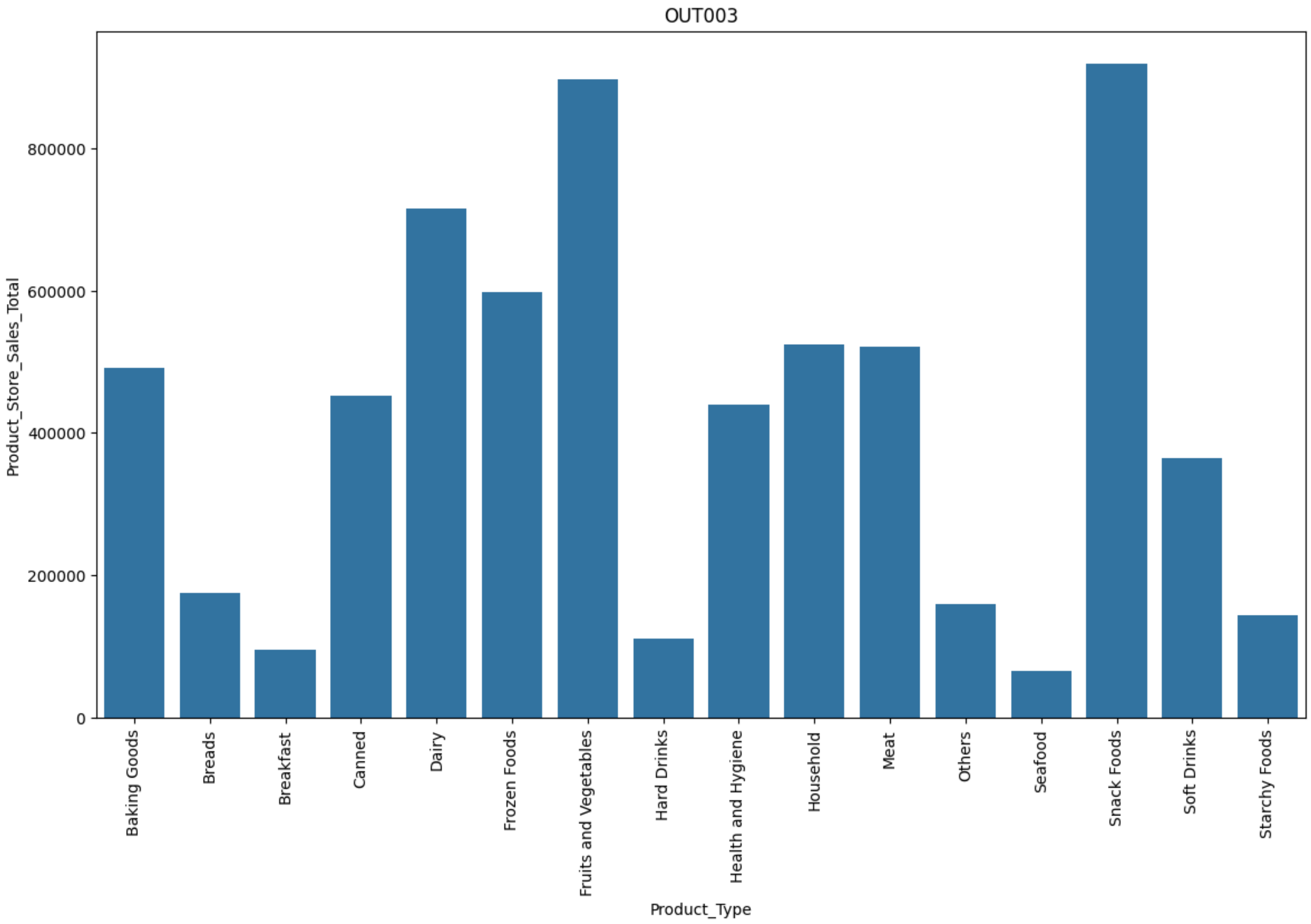
- OUT003 is a Departmental store which is located in a Tier 1 city and has store size as medium. It was established in 1999.
- OUT003 has sold products whose MRP range from 86 to 266.
- Snack Foods have been sold the highest number of times in OUT003.
- The revenue generated from each product at OUT003 ranges from 3070 to 8000.

```
data.loc[data["Store_Id"] == "OUT003", "Product_Store_Sales_Total"].sum()
```

```
np.float64(6673457.57)
```

OUT003 has generated total revenue of 6673458 from the sales of goods.

```
df_OUT003 = (
    data.loc[data["Store_Id"] == "OUT003"]
    .groupby(["Product_Type"], as_index=False)["Product_Store_Sales_Total"]
    .sum()
)
plt.figure(figsize=[14, 8])
plt.xticks(rotation=90)
plt.xlabel("Product_Type")
plt.ylabel("Product_Store_Sales_Total")
plt.title("OUT003")
sns.barplot(x=df_OUT003.Product_Type, y=df_OUT003.Product_Store_Sales_Total)
plt.show()
```



- OUT003 has generated the highest revenue from the sale of snack foods followed by fruits and vegetables, both the categories contributing around 800000 each.

OUT004

```
data.loc[data["Store_Id"] == "OUT004"].describe(include="all").T
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Product_Id	4676	4676	NC584	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Weight	4676.0	NaN	NaN	NaN	12.349613	1.428199	7.34	11.37	12.37	13.3025	17.79
Product_Sugar_Content	4676	4	Low Sugar	2632	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Allocated_Area	4676.0	NaN	NaN	NaN	0.069092	0.048584	0.004	0.031	0.056	0.097	0.297
Product_Type	4676	16	Fruits and Vegetables	700	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_MRP	4676.0	NaN	NaN	NaN	142.399709	17.513973	83.04	130.54	142.82	154.1925	197.66
Store_Id	4676	1	OUT004	4676	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Establishment_Year	4676.0	NaN	NaN	NaN	2009.0	0.0	2009.0	2009.0	2009.0	2009.0	2009.0
Store_Size	4676	1	Medium	4676	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Location_City_Type	4676	1	Tier 2	4676	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Store_Type	4676	1	Supermarket Type2	4676	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Product_Store_Sales_Total	4676.0	NaN	NaN	NaN	3299.312111	468.271692	1561.06	3042.225	3304.18	3549.3075	5462.86

Observations

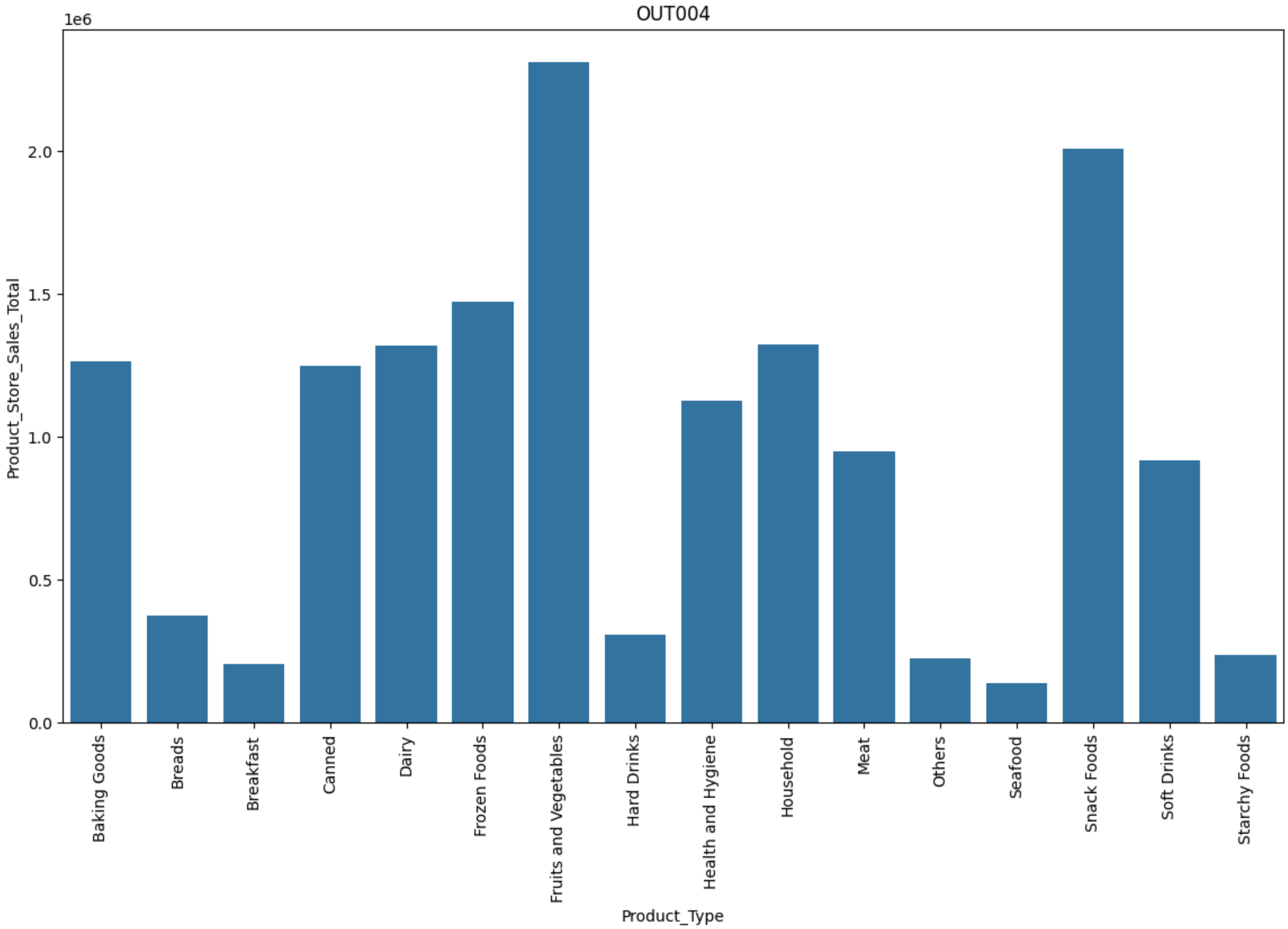
- OUT004 is a store of Supermarket Type2 which is located in a Tier 2 city and has store size as medium. It was established in 2009.
- OUT004 has sold products whose MRP range from 83 to 198.
- Fruits and vegetables have been sold the highest number of times in OUT004.
- The revenue generated from each product at OUT004 ranges from 1561 to 5463.

```
data.loc[data["Store_Id"] == "OUT004", "Product_Store_Sales_Total"].sum()
```

```
np.float64(15427583.43)
```

OUT004 has generated total revenue of 15427583 from the sales of goods which is highest among all the 4 stores.

```
df_OUT004 = (
    data.loc[data["Store_Id"] == "OUT004"]
    .groupby(["Product_Type"], as_index=False)["Product_Store_Sales_Total"]
    .sum()
)
plt.figure(figsize=[14, 8])
plt.xticks(rotation=90)
plt.xlabel("Product_Type")
plt.ylabel("Product_Store_Sales_Total")
plt.title("OUT004")
sns.barplot(x=df_OUT004.Product_Type, y=df_OUT004.Product_Store_Sales_Total)
plt.show()
```



- OUT004 has generated the highest revenue from the sale of fruits and vegetables (~ 2500000) followed by snack foods (~ 2000000).

Let's find out the revenue generated by the stores from each of the product types.

```
df1 = data.groupby(["Product_Type", "Store_Id"], as_index=False)[
    "Product_Store_Sales_Total"
].sum()
df1
```

	Product_Type	Store_Id	Product_Store_Sales_Total
0	Baking Goods	OUT001	525131.04
1	Baking Goods	OUT002	169860.50
2	Baking Goods	OUT003	491908.20
3	Baking Goods	OUT004	1266086.26
4	Breads	OUT001	121274.09
5	Breads	OUT002	43419.47
6	Breads	OUT003	175391.93
7	Breads	OUT004	374856.75
8	Breakfast	OUT001	38161.10
9	Breakfast	OUT002	23396.10
10	Breakfast	OUT003	95634.08
11	Breakfast	OUT004	204939.13
12	Canned	OUT001	449016.38
13	Canned	OUT002	151467.66
14	Canned	OUT003	452445.17
15	Canned	OUT004	1247153.50
16	Dairy	OUT001	598767.62
17	Dairy	OUT002	178888.18
18	Dairy	OUT003	715814.94
19	Dairy	OUT004	1318447.30
20	Frozen Foods	OUT001	558556.81
21	Frozen Foods	OUT002	180295.95
22	Frozen Foods	OUT003	597608.42
23	Frozen Foods	OUT004	1473519.65
24	Fruits and Vegetables	OUT001	792992.59
25	Fruits and Vegetables	OUT002	298503.56
26	Fruits and Vegetables	OUT003	897437.46
27	Fruits and Vegetables	OUT004	2311899.66
28	Hard Drinks	OUT001	152920.74
29	Hard Drinks	OUT002	54281.85
30	Hard Drinks	OUT003	110760.30
31	Hard Drinks	OUT004	307851.73
32	Health and Hygiene	OUT001	435005.31
33	Health and Hygiene	OUT002	164660.81
34	Health and Hygiene	OUT003	439139.18
35	Health and Hygiene	OUT004	1124901.91
36	Household	OUT001	531371.38
37	Household	OUT002	184665.65
38	Household	OUT003	523981.64
39	Household	OUT004	1324721.50
40	Meat	OUT001	505867.28
41	Meat	OUT002	151800.01
42	Meat	OUT003	520939.68
43	Meat	OUT004	950604.97
44	Others	OUT001	123977.09
45	Others	OUT002	32835.73

46	Others	OUT003	159963.75
47	Others	OUT004	224719.73
48	Seafood	OUT001	52936.84
49	Seafood	OUT002	17663.35
50	Seafood	OUT003	65337.48
51	Seafood	OUT004	136466.37
52	Snack Foods	OUT001	806142.24
53	Snack Foods	OUT002	255317.57
54	Snack Foods	OUT003	918510.44
55	Snack Foods	OUT004	2009026.70
56	Soft Drinks	OUT001	410548.69
57	Soft Drinks	OUT002	103808.35
58	Soft Drinks	OUT003	365046.30
59	Soft Drinks	OUT004	917641.38
60	Starchy Foods	OUT001	120443.98
61	Starchy Foods	OUT002	20044.98
62	Starchy Foods	OUT003	143538.60
63	Starchy Foods	OUT004	234746.89

- In all the product types, the revenue generated by OUT004 has been the highest which seems quite logical since around 53% of the total products were brought from this store.
- In all the product categories, the revenue generated by OUT002 has been the lowest which seems quite obvious since it is small store in a Tier 3 city.

Let's find out the revenue generated by the stores from products having different levels of sugar content.

```
df2 = data.groupby(["Product_Sugar_Content", "Store_Id"], as_index=False)[
    "Product_Store_Sales_Total"
].sum()
df2
```

	Product_Sugar_Content	Store_Id	Product_Store_Sales_Total
0	Low Sugar	OUT001	3300834.93
1	Low Sugar	OUT002	1156758.85
2	Low Sugar	OUT003	3706903.24
3	Low Sugar	OUT004	8658908.78
4	No Sugar	OUT001	1090353.78
5	No Sugar	OUT002	382162.19
6	No Sugar	OUT003	1123084.57
7	No Sugar	OUT004	2674343.14
8	Regular	OUT001	1749444.51
9	Regular	OUT002	472112.50
10	Regular	OUT003	1743566.35
11	Regular	OUT004	3902547.93
12	reg	OUT001	82479.96
13	reg	OUT002	19876.18
14	reg	OUT003	99903.41
15	reg	OUT004	191783.58

- The trend is the same as that which was present in the revenue analysis of stores for product types.

Data Preprocessing

Replacing the values in the Product_Sugar_Content column

We can observe that in the Product_Sugar_Content column, there are 3 types - Low Sugar, Regular and reg.

It seems quite obvious that Regular and reg are referring to the same category. So let's replace reg with Regular.

```
# Replacing reg with Regular
data.Product_Sugar_Content.replace(to_replace=["reg"], value=["Regular"], inplace=True)
```

```
data.Product_Sugar_Content.value_counts()
```

Product_Sugar_Content	count
Low Sugar	4885
Regular	2359
No Sugar	1519

dtype: int64

Exploring Patterns in Product_IDs

We can see that the Product_Id column has two characters followed by a number.

Let's delve deeper and see whether they are having any relationship with the other columns or not

```
## extracting the first two characters from the Product_Id column and storing it in another column
data["Product_Id_char"] = data["Product_Id"].str[:2]
data.head()
```

	Product_Id	Product_Weight	Product_Sugar_Content	Product_Allocated_Area	Product_Type	Product_MRP	Store_Id	Store_Establishment_Y
0	FD6114	12.66	Low Sugar	0.027	Frozen Foods	117.08	OUT004	2
1	FD7839	16.54	Low Sugar	0.144	Dairy	171.43	OUT003	1
2	FD5075	14.28	Regular	0.031	Canned	162.08	OUT001	1
3	FD8233	12.10	Low Sugar	0.112	Baking Goods	186.31	OUT001	1
4	NC1180	9.57	No Sugar	0.010	Health and Hygiene	123.67	OUT002	1

```
data["Product_Id_char"].unique()
```

```
array(['FD', 'NC', 'DR'], dtype=object)
```

```
data.loc[data.Product_Id_char == "FD", "Product_Type"].unique()
```

```
array(['Frozen Foods', 'Dairy', 'Canned', 'Baking Goods', 'Snack Foods',
       'Meat', 'Fruits and Vegetables', 'Breads', 'Breakfast',
       'Starchy Foods', 'Seafood'], dtype=object)
```

```
data.loc[data.Product_Id_char == "Product_Id_char+", "Product_Type"].unique() #Complete the code to select the rows where Product_Id_char is
```

```
array([], dtype=object)
```

```
data.loc[data.Product_Id_char == "Product_Id_char", "Product_Type"].unique() #Complete the code to select the rows where Product_Id_char is
```

```
array([], dtype=object)
```

Store's Age

A store which has been in the business for a long duration is more trustworthy than the newly established ones.

On the other hand, older stores may sometimes lack infrastructure if proper attention is not given. So let us calculate the current age of the store and incorporate that in our model.

```
# Outlet Age
data["Store_Age_Years"] = 2025 - data.Store_Establishment_Year
```

Grouping Product Types into Perishables and Non-Perishables.

We have 16 different product types in our dataset.

So let us make two broad categories, perishables and non perishables, in order to reduce the number of product types

```
perishables = [
    "Dairy",
    "Meat",
    "Fruits and Vegetables",
    "Breakfast",
    "Breads",
    "Seafood",
]
```

```
def change(x):
    if x in perishables:
        return "Perishables"
    else:
        return "Non Perishables"
```

```
data['Product_Type_Category'] = data['Product_Type'].apply(change)
```

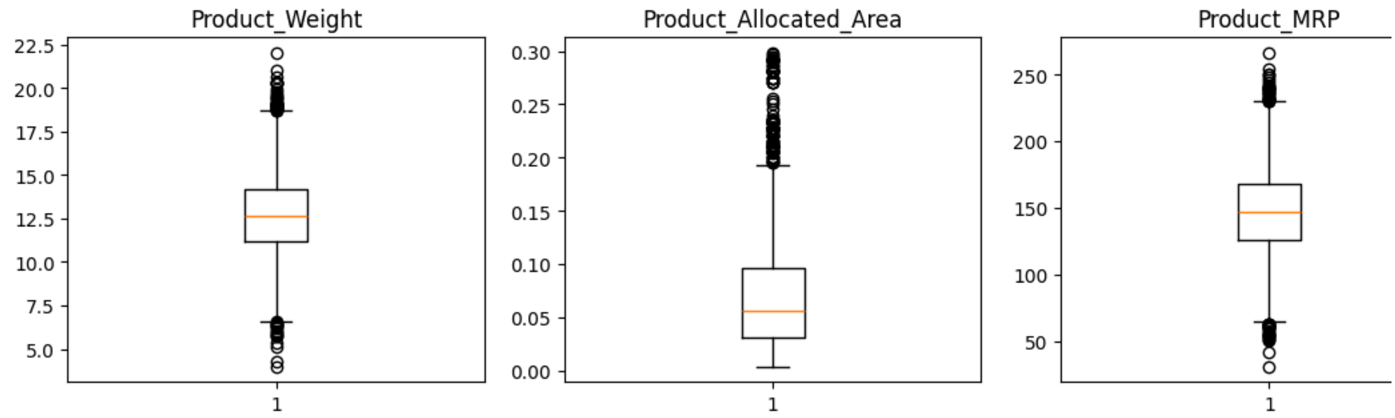
```
data.head()
```

	Product_Id	Product_Weight	Product_Sugar_Content	Product_Allocated_Area	Product_Type	Product_MRP	Store_Id	Store_Establishment_Y
0	FD6114	12.66	Low Sugar	0.027	Frozen Foods	117.08	OUT004	2
1	FD7839	16.54	Low Sugar	0.144	Dairy	171.43	OUT003	1
2	FD5075	14.28	Regular	0.031	Canned	162.08	OUT001	1
3	FD8233	12.10	Low Sugar	0.112	Baking Goods	186.31	OUT001	1
4	NC1180	9.57	No Sugar	0.010	Health and Hygiene	123.67	OUT002	1

Outlier Check

```
# outlier detection using boxplot
numeric_columns = data.select_dtypes(include=np.number).columns.tolist()
numeric_columns.remove("Store_Establishment_Year")
numeric_columns.remove("Store_Age_Years")
```

```
plt.figure(figsize=(15, 12))

for i, variable in enumerate(numeric_columns):
    plt.subplot(4, 4, i + 1)
    plt.boxplot(data[variable], whis=1.5)
    plt.tight_layout()
    plt.title(variable)

plt.show()
```



Data Preparation for Modeling

- We aim to forecast the Product_Store_Sales_Total.
- Before building the model, we'll drop unnecessary columns and encode the categorical features.
- We'll then split the data into training and testing sets to evaluate the model's performance on unseen data.

```
data.head()
```

	Product_Id	Product_Weight	Product_Sugar_Content	Product_Allocated_Area	Product_Type	Product_MRP	Store_Id	Store_Establishment_Y
0	FD6114	12.66	Low Sugar	0.027	Frozen Foods	117.08	OUT004	2
1	FD7839	16.54	Low Sugar	0.144	Dairy	171.43	OUT003	1
2	FD5075	14.28	Regular	0.031	Canned	162.08	OUT001	1
3	FD8233	12.10	Low Sugar	0.112	Baking Goods	186.31	OUT001	1
4	NC1180	9.57	No Sugar	0.010	Health and Hygiene	123.67	OUT002	1

Let's remove the columns that are not required.

```
data = data.drop(["Product_Id","Product_Type","Store_Id","Store_Establishment_Year"], axis=1) #Complete the code to drop the columns "Produc
```

```
data.shape
```

```
(8763, 11)
```

```
data.head()
```

	Product_Weight	Product_Sugar_Content	Product_Allocated_Area	Product_MRP	Store_Size	Store_Location_City_Type	Store_Type	Produc
0	12.66	Low Sugar	0.027	117.08	Medium	Tier 2	Supermarket Type2	
1	16.54	Low Sugar	0.144	171.43	Medium	Tier 1	Departmental Store	
2	14.28	Regular	0.031	162.08	High	Tier 2	Supermarket Type1	

```
# Separating features and the target column
X = data.drop("Product_Store_Sales_Total", axis=1) #Complete the code to drop the target variable
y = data["Product_Store_Sales_Total"] #Complete the code to select the target variable
```

```
# Splitting the data into train and test sets in 70:30 ratio
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=1, shuffle=True #Complete the code to define the test_size
)
```

```
X_train.shape, X_test.shape
```

```
((6134, 10), (2629, 10))
```

Data Pre-processing Pipeline

```
categorical_features = data.select_dtypes(include=['object', 'category']).columns.tolist()
categorical_features
```

```
['Product_Sugar_Content',
 'Store_Size',
 'Store_Location_City_Type',
 'Store_Type',
 'Product_Id_char',
 'Product_Type_Category']
```

```
# Create a preprocessing pipeline for the categorical features

preprocessor = make_column_transformer(
    (Pipeline([('encoder', OneHotEncoder(handle_unknown='ignore'))]), categorical_features)
)
```

Model Building

Note: As per the rubric, you are required to build two ML models for this project. We have provided the code required for model building for six different ML models, each under a separate markdown section. You may choose any two models from the ones provided below, uncomment the corresponding code for the model, and then run the code to build the model, and check its performance.

Define functions for Model Evaluation

- We'll fit different models on the train data and observe their performance.
- We'll try to improve that performance by tuning some hyperparameters available for that algorithm.
- We'll use GridSearchCv for hyperparameter tuning and `r_2 score` to optimize the model.
- R-square - `Coefficient of determination` is used to evaluate the performance of a regression model. It is the amount of the variation in the output dependent attribute which is predictable from the input independent variables.
- Let's start by creating a function to get model scores, so that we don't have to use the same codes repeatedly.

```
# function to compute adjusted R-squared
def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))


# function to compute different metrics to check performance of a regression model
def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performance

    model: regressor
    predictors: independent variables
    target: dependent variable
    """

    # predicting using the independent variables
    pred = model.predict(predictors)

    r2 = r2_score(target, pred)  # to compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)  # to compute adjusted R-squared
    rmse = np.sqrt(mean_squared_error(target, pred))  # to compute RMSE
    mae = mean_absolute_error(target, pred)  # to compute MAE
    mape = mean_absolute_percentage_error(target, pred)  # to compute MAPE

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )

    return df_perf
```

Decision Tree Model

```
# Uncomment the below snippet of code if decision tree regressor is to be used

# dtree = DecisionTreeRegressor(random_state=1)
# dtree = make_pipeline(preprocessor,dtree)
# dtree.fit(X_train, y_train)
```

```
# Libraries to help with data visualization
import matplotlib.pyplot as plt
import seaborn as sns

numerical_features = ['Product_Weight', 'Product_Allocated_Area', 'Product_MRP']

# Check for non-numeric values in numerical columns of X_train
for col in numerical_features:
    # Attempt to convert to numeric, coercing errors to NaN
    # Then check if there are any NaN values (which indicate non-numeric original values)
    non_numeric_count = pd.to_numeric(X_train[col], errors='coerce').isnull().sum()
    if non_numeric_count > 0:
        print(f"Column '{col}' in X_train contains {non_numeric_count} non-numeric values.")
        # Optionally, print the non-numeric values themselves
        # print(X_train[pd.to_numeric(X_train[col], errors='coerce').isnull()][col])
    else:
        print(f"Column '{col}' in X_train contains no non-numeric values.")

# Now, re-run the preprocessor and model training after this check.
# This part of the code is repeated from the previous step where the error occurred
# to ensure the full pipeline is executed.

# Re-define helper functions for model evaluation (if not already defined in this session)
# It's safer to include them for robustness if the environment resets
def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))

def model_performance_regression(model, predictors, target):
    pred = model.predict(predictors)
    r2 = r2_score(target, pred)
    adjr2 = adj_r2_score(predictors, target, pred)
    rmse = np.sqrt(mean_squared_error(target, pred))
    mae = mean_absolute_error(target, pred)
    mape = mean_absolute_percentage_error(target, pred)
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse, "MAE": mae, "R-squared": r2,
            "Adj. R-squared": adjr2, "MAPE": mape,
        },
        index=[0],
    )
    return df_perf

# Libraries for preprocessing and pipeline
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error, mean_absolute_percentage_error
import numpy as np
import pandas as pd # Import pandas for DataFrame creation

# Define numerical and categorical features (re-confirming for clarity)
numerical_features = ['Product_Weight', 'Product_Allocated_Area', 'Product_MRP']
categorical_features = ['Product_Sugar_Content', 'Store_Size', 'Store_Location_City_Type', 'Store_Type']

# Create the preprocessor
preprocessor = make_column_transformer(
    (StandardScaler(), numerical_features),
    (OneHotEncoder(handle_unknown='ignore'), categorical_features)
)

# Train the Decision Tree Regressor pipeline
dtree = DecisionTreeRegressor(random_state=1)
dtree_pipeline = make_pipeline(preprocessor, dtree)
dtree_pipeline.fit(X_train, y_train)

# Evaluate the Decision Tree model
print("\nDecision Tree Regressor - Model Performance on Training Set:")
print(model_performance_regression(dtree_pipeline, X_train, y_train))

print("\nDecision Tree Regressor - Model Performance on Test Set:")
print(model_performance_regression(dtree_pipeline, X_test, y_test))
```

```
Column 'Product_Weight' in X_train contains no non-numeric values.
Column 'Product_Allocated_Area' in X_train contains no non-numeric values.
Column 'Product_MRP' in X_train contains no non-numeric values.

Decision Tree Regressor - Model Performance on Training Set:
   RMSE  MAE  R-squared  Adj. R-squared  MAPE
0   0.0  0.0        1.0             1.0   0.0

Decision Tree Regressor - Model Performance on Test Set:
        RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  359.44104  123.382385     0.8871        0.886668  0.052663
```

Checking model performance on training set

```
# Uncomment the below snippet of code if decision tree regressor is to be used

#dtree_model_train_perf = model_performance_regression(dtree, X_train, y_train)
#dtree_model_train_perf


# Re-define helper functions for model evaluation (if not already defined in this session)
# It's safer to include them for robustness if the environment resets
import numpy as np
import pandas as pd
```

```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error, mean_absolute_percentage_error

def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))

def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performance

    model: regressor
    predictors: independent variables
    target: dependent variable
    """
    # predicting using the independent variables
    pred = model.predict(predictors)

    r2 = r2_score(target, pred)  # to compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)  # to compute adjusted R-squared
    rmse = np.sqrt(mean_squared_error(target, pred))  # to compute RMSE
    mae = mean_absolute_error(target, pred)  # to compute MAE
    mape = mean_absolute_percentage_error(target, pred)  # to compute MAPE

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )
    return df_perf

# Uncomment the below snippet of code if decision tree regressor is to be used

dtree_model_train_perf = model_performance_regression(dtree_pipeline, X_train, y_train) # Use dtree_pipeline as it's the fitted model
dtree_model_train_perf
```

```
   RMSE  MAE  R-squared  Adj. R-squared  MAPE
0   0.0  0.0        1.0             1.0   0.0
```

Checking model performance on test set

```
# Uncomment the below snippet of code if decision tree regressor is to be used

#dtree_model_test_perf = model_performance_regression(dtree, X_test, y_test)
#dtree_model_test_perf
```

```
# Re-define helper functions for model evaluation (if not already defined in this session)
# It's safer to include them for robustness if the environment resets
import numpy as np
import pandas as pd
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error, mean_absolute_percentage_error

def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))

def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performance

    model: regressor
    predictors: independent variables
    target: dependent variable
    """
    # predicting using the independent variables
    pred = model.predict(predictors)

    r2 = r2_score(target, pred)  # to compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)  # to compute adjusted R-squared
    rmse = np.sqrt(mean_squared_error(target, pred))  # to compute RMSE
    mae = mean_absolute_error(target, pred)  # to compute MAE
    mape = mean_absolute_percentage_error(target, pred)  # to compute MAPE

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )
    return df_perf

# Uncomment the below snippet of code if decision tree regressor is to be used

dtree_model_test_perf = model_performance_regression(dtree_pipeline, X_test, y_test) # Use dtree_pipeline as it's the fitted model
dtree_model_test_perf
```

```
        RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  359.44104  123.382385     0.8871        0.886668  0.052663
```

Bagging Regressor

[ ] ↳ 8 cells hidden

Random Forest Model

[ ] ↳ 8 cells hidden

AdaBoost Regressor

[ ] ↳ 8 cells hidden

Gradient Boosting Regressor

[ ] ↳ 8 cells hidden

XGBoost Regressor

```
# Uncomment the below snippet of code if xgboost regressor is to be used

#xgb_estimator = XGBRegressor(random_state=1)
#xgb_estimator = make_pipeline(preprocessor,xgb_estimator)
#xgb_estimator.fit(X_train, y_train)


from xgboost import XGBRegressor

# Define the model
xgb_estimator = XGBRegressor(random_state=1, verbosity=0)

# Create pipeline with preprocessing
xgb_pipeline = make_pipeline(preprocessor, xgb_estimator)

# Fit the model
xgb_pipeline.fit(X_train, y_train)
```

```
# Evaluate the model
print("\nXGBoost Regressor - Model Performance on Training Set:")
print(model_performance_regression(xgb_pipeline, X_train, y_train))

print("\nXGBoost Regressor - Model Performance on Test Set:")
print(model_performance_regression(xgb_pipeline, X_test, y_test))
```

```

XGBoost Regressor - Model Performance on Training Set:
         RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  127.766962  58.911644   0.985573        0.985549  0.019671

XGBoost Regressor - Model Performance on Test Set:
         RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  309.775996  134.783796   0.916144        0.915823  0.056372
```

Checking model performance on training set

```
# Uncomment the below snippet of code if xgboost regressor is to be used

#xgb_estimator_model_train_perf = model_performance_regression(xgb_estimator, X_train, y_train)
#xgb_estimator_model_train_perf
```

```
# Check performance on training set
xgb_estimator_model_train_perf = model_performance_regression(xgb_pipeline, X_train, y_train)
xgb_estimator_model_train_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	127.766962	58.911644	0.985573	0.985549	0.019671

Checking model performance on test set

```
# Uncomment the below snippet of code if xgboost regressor is to be used

#xgb_estimator_model_test_perf = model_performance_regression(xgb_estimator, X_test, y_test)
#xgb_estimator_model_test_perf
```

```
# Evaluate XGBoost Regressor on the test set
xgb_estimator_model_test_perf = model_performance_regression(xgb_pipeline, X_test, y_test)
xgb_estimator_model_test_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	309.775996	134.783796	0.916144	0.915823	0.056372

Model Performance Improvement - Hyperparameter Tuning

Note:

1. As per the rubric, you are required to tune two ML models for this project. We have provided the code required for model building for six different ML models, each under a separate markdown section. You may choose the two models you built previously, uncomment the corresponding code for the model, and then run the code to tune the model, and check its performance.

2. We've provided a sample parameter grid for tuning. You may add/remove parameters or parameter values to check for during tuning as per your requirements.

Hyperparameter Tuning - Decision Tree

```
# Uncomment the below snippet of code if decision tree regressor is to be used

# # Choose the type of classifier.
# dtree_tuned = DecisionTreeRegressor(random_state=1)
# dtree_tuned = make_pipeline(preprocessor,dtree_tuned)

# # Grid of parameters to choose from
# parameters = {
#     "decisiontreeregressor__max_depth": list(np.arange(2, 6)),
#     "decisiontreeregressor__min_samples_leaf": [1, 3, 5],
#     "decisiontreeregressor__max_leaf_nodes": [2, 3, 5, 10, 15],
#     "decisiontreeregressor__min_impurity_decrease": [0.001, 0.01, 0.1],
# }

# # Run the grid search
# grid_obj = GridSearchCV(dtree_tuned, parameters, scoring=r2_score, cv=3, n_jobs =-1)
# grid_obj = grid_obj.fit(X_train, y_train)

# # Set the clf to the best combination of parameters
# dtree_tuned = grid_obj.best_estimator_

# # Fit the best algorithm to the data.
# dtree_tuned.fit(X_train, y_train)


# Choose the type of regressor
dtree_tuned = DecisionTreeRegressor(random_state=1)
dtree_tuned = make_pipeline(preprocessor, dtree_tuned)

# Grid of parameters to tune
parameters = {
    "decisiontreeregressor__max_depth": list(np.arange(2, 6)),
    "decisiontreeregressor__min_samples_leaf": [1, 3, 5],
    "decisiontreeregressor__max_leaf_nodes": [2, 3, 5, 10, 15],
    "decisiontreeregressor__min_impurity_decrease": [0.001, 0.01, 0.1],
}

# Run GridSearchCV
grid_obj = GridSearchCV(dtree_tuned, parameters, scoring='r2', cv=3, n_jobs=-1)
grid_obj = grid_obj.fit(X_train, y_train)

# Get the best estimator
dtree_tuned = grid_obj.best_estimator_

# Fit the best estimator
dtree_tuned.fit(X_train, y_train)

# Evaluate performance
print("\nTuned Decision Tree - Model Performance on Training Set:")
print(model_performance_regression(dtree_tuned, X_train, y_train))

print("\nTuned Decision Tree - Model Performance on Test Set:")
print(model_performance_regression(dtree_tuned, X_test, y_test))
```

```
Tuned Decision Tree - Model Performance on Training Set:
         RMSE         MAE  R-squared  Adj. R-squared     MAPE
0  390.807182  263.756623   0.865019        0.864799  0.09531

Tuned Decision Tree - Model Performance on Test Set:
         RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  418.906939  277.126837   0.846653        0.846067  0.113007
```

Checking model performance on training set

```
# Uncomment the below snippet of code if decision tree regressor is to be used

#dtree_tuned_model_train_perf = model_performance_regression(dtree_tuned, X_train, y_train)
#dtree_tuned_model_train_perf
```

```
# Evaluate tuned Decision Tree Regressor on the training set
dtree_tuned_model_train_perf = model_performance_regression(dtree_tuned, X_train, y_train)
dtree_tuned_model_train_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	390.807182	263.756623	0.865019	0.864799	0.09531

Checking model performance on test set

```
# Uncomment the below snippet of code if decision tree regressor is to be used

#dtree_tuned_model_test_perf = model_performance_regression(dtree_tuned, X_test, y_test)
#dtree_tuned_model_test_perf
```

```
# Evaluate tuned Decision Tree Regressor on the test set
dtree_tuned_model_test_perf = model_performance_regression(dtree_tuned, X_test, y_test)
dtree_tuned_model_test_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	418.906939	277.126837	0.846653	0.846067	0.113007

Hyperparameter Tuning - Bagging Regressor

```
#Uncomment the below snippet of code if bagging regressor is to be used

# # Choose the type of regressor.
# bagging_estimator_tuned = BaggingRegressor(random_state=1)
# bagging_estimator_tuned = make_pipeline(preprocessor,bagging_estimator_tuned)

# # Grid of parameters to choose from
# parameters = {
#     "baggingregressor__max_samples": ______, #Complete the code to define the list of values to be tuned
#     "baggingregressor__max_features": _____, #Complete the code to define the list of values to be tuned
#     "baggingregressor__n_estimators": _____, #Complete the code to define the list of values to be tuned
# }

# # Run the grid search
# grid_obj = GridSearchCV(bagging_estimator_tuned, parameters, scoring=r2_score, cv=3, n_jobs = -1)
# grid_obj = grid_obj.fit(X_train, y_train)

# # Set the clf to the best combination of parameters
# bagging_estimator_tuned = grid_obj.best_estimator_

# # Fit the best algorithm to the data.
# bagging_estimator_tuned.fit(X_train, y_train)
```

```
# Choose the type of regressor
bagging_estimator_tuned = BaggingRegressor(random_state=1)
bagging_estimator_tuned = make_pipeline(preprocessor, bagging_estimator_tuned)

# Grid of parameters to tune
parameters = {
    "baggingregressor__max_samples": [0.5, 0.7, 1.0],
    "baggingregressor__max_features": [0.5, 0.7, 1.0],
    "baggingregressor__n_estimators": [10, 50, 100],
}

# Run the grid search
grid_obj = GridSearchCV(bagging_estimator_tuned, parameters, scoring='r2', cv=3, n_jobs=-1)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the regressor to the best combination of parameters
bagging_estimator_tuned = grid_obj.best_estimator_

# Fit the best estimator to the data
bagging_estimator_tuned.fit(X_train, y_train)
```

```
# Evaluate model performance
print("\nTuned Bagging Regressor - Model Performance on Training Set:")
print(model_performance_regression(bagging_estimator_tuned, X_train, y_train))

print("\nTuned Bagging Regressor - Model Performance on Test Set:")
print(model_performance_regression(bagging_estimator_tuned, X_test, y_test))
```

```
Tuned Bagging Regressor - Model Performance on Training Set:
         RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  169.596607  64.404086    0.97458        0.974538  0.022975

Tuned Bagging Regressor - Model Performance on Test Set:
         RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  294.038734  111.894844   0.924447        0.924159  0.050528
```

Checking model performance on training set

```
#Uncomment the below snippet of code if bagging regressor is to be used
```

```
# bagging_estimator_tuned_model_train_perf = model_performance_regression(bagging_estimator_tuned, X_train, y_train)
# bagging_estimator_tuned_model_train_perf
```

```
# Evaluate tuned Bagging Regressor on the training set
bagging_estimator_tuned_model_train_perf = model_performance_regression(bagging_estimator_tuned, X_train, y_train)
bagging_estimator_tuned_model_train_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	169.596607	64.404086	0.97458	0.974538	0.022975

Checking model performance on test set

```
#Uncomment the below snippet of code if bagging regressor is to be used

# bagging_estimator_tuned_model_test_perf = model_performance_regression(bagging_estimator_tuned, X_test, y_test)
# bagging_estimator_tuned_model_test_perf
```

```
# Evaluate tuned Bagging Regressor on the test set
bagging_estimator_tuned_model_test_perf = model_performance_regression(bagging_estimator_tuned, X_test, y_test)
bagging_estimator_tuned_model_test_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	294.038734	111.894844	0.924447	0.924159	0.050528

Hyperparameter Tuning - Random Forest

```
#Uncomment the below snippet of code if random forest regressor is to be used
```

```
# # Choose the type of classifier.
# rf_tuned = RandomForestRegressor(random_state=1)
# rf_tuned = make_pipeline(preprocessor,rf_tuned)

# # Grid of parameters to choose from
# parameters = {
#     "randomforestregressor__max_depth": ______, #Complete the code to define the list of values to be tuned
#     "randomforestregressor__max_features":______, #Complete the code to define the list of values to be tuned
#     "randomforestregressor__n_estimators": ______, #Complete the code to define the list of values to be tuned
# }

# # Run the grid search
# grid_obj = GridSearchCV(rf_tuned, parameters, scoring=r2_score, cv=3, n_jobs = -1)
# grid_obj = grid_obj.fit(X_train, y_train)

# # Set the clf to the best combination of parameters
# rf_tuned = grid_obj.best_estimator_

# # Fit the best algorithm to the data.
# rf_tuned.fit(X_train, y_train)


# Choose the type of regressor
rf_tuned = RandomForestRegressor(random_state=1)
rf_tuned = make_pipeline(preprocessor, rf_tuned)

# Grid of parameters to tune
parameters = {
    "randomforestregressor__max_depth": [5, 10, 15],
    "randomforestregressor__max_features": ['auto', 'sqrt'],
    "randomforestregressor__n_estimators": [50, 100, 200],
}

# Run GridSearchCV
grid_obj = GridSearchCV(rf_tuned, parameters, scoring='r2', cv=3, n_jobs=-1)
grid_obj = grid_obj.fit(X_train, y_train)

# Get the best estimator
rf_tuned = grid_obj.best_estimator_

# Fit the best estimator
rf_tuned.fit(X_train, y_train)

# Evaluate model performance
print("\nTuned Random Forest - Model Performance on Training Set:")
print(model_performance_regression(rf_tuned, X_train, y_train))

print("\nTuned Random Forest - Model Performance on Test Set:")
print(model_performance_regression(rf_tuned, X_test, y_test))
```

```
Tuned Random Forest - Model Performance on Training Set:
         RMSE        MAE  R-squared  Adj. R-squared      MAPE
0  125.125394  61.302641   0.986163        0.986141  0.022221

Tuned Random Forest - Model Performance on Test Set:
         RMSE         MAE  R-squared  Adj. R-squared      MAPE
0  306.318185  140.711502   0.918005        0.917692  0.061945
```

Checking model performance on training set

```
#Uncomment the below snippet of code if random forest regressor is to be used

# rf_tuned_model_train_perf = model_performance_regression(rf_tuned, X_train, y_train)
# rf_tuned_model_train_perf
```

```
# Evaluate tuned Random Forest Regressor on the training set
rf_tuned_model_train_perf = model_performance_regression(rf_tuned, X_train, y_train)
rf_tuned_model_train_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	125.125394	61.302641	0.986163	0.986141	0.022221

Checking model performance on test set

```
#Uncomment the below snippet of code if random forest regressor is to be used

# rf_tuned_model_test_perf = model_performance_regression(rf_tuned, X_test, y_test)
# rf_tuned_model_test_perf
```

```
# Evaluate tuned Random Forest Regressor on the test set
rf_tuned_model_test_perf = model_performance_regression(rf_tuned, X_test, y_test)
```

```
rf_tuned_model_test_perf
```

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	306.318185	140.711502	0.918005	0.917692	0.061945

Hyperparameter Tuning - AdaBoost Regressor

[ ] ↳ 8 cells hidden

Hyperparameter Tuning - Gradient Boosting Regressor

[ ] ↳ 8 cells hidden

Hyperparameter Tuning - XGBoost Regressor

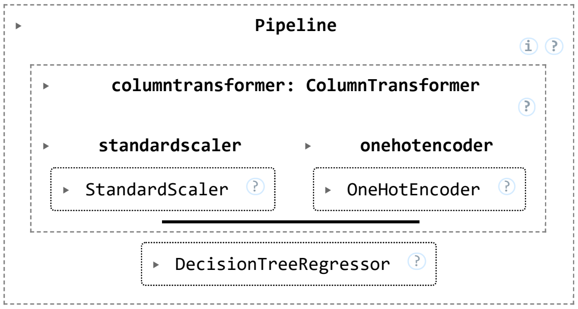
[ ] ↳ 8 cells hidden

Model Performance Comparison, Final Model Selection, and Serialization

```
 # training performance comparison

models_train_comp_df = pd.concat(
    [
        _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of the first moc
        _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of the first moc
        _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of the second mc
        _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of the second mc
    ],
    axis=1,
)

models_train_comp_df.columns = ["_____"] #Complete the code to define the names for the models

print("Training performance comparison:")
models_train_comp_df
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-152-2332552604.py in <cell line: 0>()
      3 models_train_comp_df = pd.concat(
      4     [
----> 5         _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of
the first model you have choosen . Eg, rf_model_train_perf
      6         _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of
the first model (tuned) you have choosen
      7         _____.T, #Complete the code to define the variable name of the dataframe which stores the train performance metrics of
the second model you have choosen
```

```
dtree_pipeline = make_pipeline(preprocessor, DecisionTreeRegressor(random_state=1))
dtree_pipeline.fit(X_train, y_train)
```


Pipeline

columntransformer: ColumnTransformer

standardscaler

onehotencoder

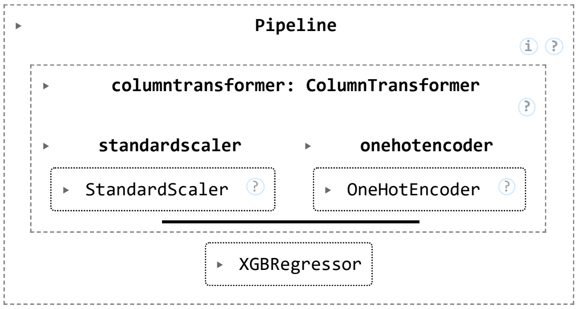
StandardScaler

OneHotEncoder

DecisionTreeRegressor

```
dtree_pipeline_model_train_perf = model_performance_regression(dtree_pipeline, X_train, y_train)
```

```
xgb_pipeline = make_pipeline(preprocessor, XGBRegressor(random_state=1))
xgb_pipeline.fit(X_train, y_train)
```



```
xgb_pipeline_model_train_perf = model_performance_regression(xgb_pipeline, X_train, y_train)
```

```
# Training performance comparison
models_train_comp_df = pd.concat(
    [
        dtree_pipeline_model_train_perf.T,
        dtree_tuned_model_train_perf.T,
        xgb_pipeline_model_train_perf.T,
        xgb_tuned_model_train_perf.T,
    ],
    axis=1,
)

models_train_comp_df.columns = [
    "Decision Tree (Base)",
    "Decision Tree (Tuned)",
    "XGBoost (Base)",
    "XGBoost (Tuned)"
]

print("Training performance comparison:")
models_train_comp_df
```

```
Training performance comparison:
```

	Decision Tree (Base)	Decision Tree (Tuned)	XGBoost (Base)	XGBoost (Tuned)
RMSE	0.0	390.807182	127.766962	137.507614
MAE	0.0	263.756623	58.911644	65.290831
R-squared	1.0	0.865019	0.985573	0.983289
Adj. R-squared	1.0	0.864799	0.985549	0.983262
MAPE	0.0	0.095310	0.019671	0.021683

```
 # training performance comparison

models_train_comp_df = pd.concat(
    [
        _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of the first mode
        _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of the first mode
        _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of the second moc
        _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of the second moc
    ],
    axis=1,
)

models_train_comp_df.columns = ["_____"] #Complete the code to define the names for the models

print("Training performance comparison:")
models_train_comp_df
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-158-882114727.py in <cell line: 0>()
      3 models_train_comp_df = pd.concat(
      4     [
----> 5         _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of
the first model you have choosen . Eg, rf_model_test_perf
      6         _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of
the first model (tuned) you have choosen
      7         _____.T, #Complete the code to define the variable name of the dataframe which stores the test performance metrics of
the second model you have choosen
```

```
dtree_pipeline_model_test_perf = model_performance_regression(dtree_pipeline, X_test, y_test)
```

```
xgb_pipeline_model_test_perf = model_performance_regression(xgb_pipeline, X_test, y_test)
```

```
# Test performance comparison
models_test_comp_df = pd.concat(
    [
        dtree_pipeline_model_test_perf.T,
        dtree_tuned_model_test_perf.T,
        xgb_pipeline_model_test_perf.T,
        xgb_tuned_model_test_perf.T,
    ],
    axis=1,
)

models_test_comp_df.columns = [
    "Decision Tree (Base)",
    "Decision Tree (Tuned)",
    "XGBoost (Base)",
    "XGBoost (Tuned)"
]

print("Test performance comparison:")
models_test_comp_df
```

```
Test performance comparison:
```

	Decision Tree (Base)	Decision Tree (Tuned)	XGBoost (Base)	XGBoost (Tuned)
RMSE	359.441040	418.906939	309.775996	305.743666
MAE	123.382385	277.126837	134.783796	135.002057
R-squared	0.887100	0.846653	0.916144	0.918313
Adj. R-squared	0.886668	0.846067	0.915823	0.918001
MAPE	0.052663	0.113007	0.056372	0.056233

```
# Create a folder for storing the files needed for web app deployment
os.makedirs("backend_files", exist_ok=True)
```

```
# Define the file path to save (serialize) the trained model along with the data preprocessing steps
saved_model_path = "backend_files/xgb_tuned_model.joblib" #Complete the code to define the name of the model
```

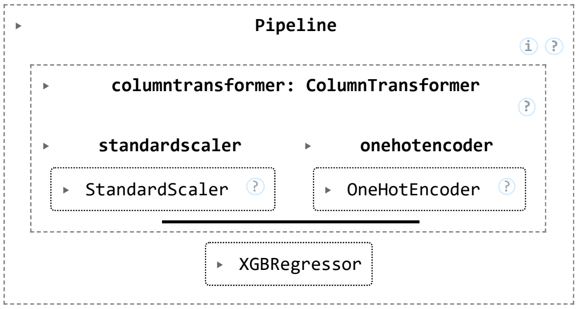
```
# Save the best trained model pipeline using joblib
joblib.dump(xgb_tuned, saved_model_path) #Complete the code to pass the variable name of the best model

print(f"Model saved successfully at {saved_model_path}")
```

```
Model saved successfully at backend_files/xgb_tuned_model.joblib
```

```
# Load the saved model pipeline from the file
saved_model = joblib.load("backend_files/xgb_tuned_model.joblib") #Complete the code to define the name of the saved model

# Confirm the model is loaded
print("Model loaded successfully.")
```

```
Model loaded successfully.
```

```
saved_model
```



Let's try making predictions on the test set using the deserialized model.

- Please ensure that the saved model is loaded before making predictions.

```
saved_model.predict(X_test) #Complete the code to pass the X_test for inference
```

```
array([3716.8042, 5929.6035, 4484.5093, ..., 3217.515 , 3116.9768,
       3532.9902], dtype=float32)
```

- As we can see, the model can be directly used for making predictions without any retraining.

Deployment - Backend

Flask Web Framework

```
%%writefile backend_files/app.py

# Import necessary libraries
import numpy as np
import joblib  # For loading the serialized model
```