FRANKFURT UNIVERSITY OF APPLIED SCIENCES

COMPUTER SCIENCE

Project

Author:
Vo Cong Minh - 10421040
Vo Pham Khang Huy - 10421082

Faculty 2. Computer Science and Engineering.

March 5, 2025

Declaration of Authorship

Vo Cong Minh - 10421040
Vo Pham Khang Huy - 10421082

We hereby certify that the project report I am submitting is entirely our own original work except where otherwise indicated. We did not submit this work anywhere else before. We are aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Contents

Chapter 1

Introduction

1.1 Background

The surge of international supermarkets and the growth of online shopping have intensified competition among retail outlets. To attract more customers, many malls and marts offer personalized and short-term promotions, such as discounts, which are typically managed through various strategies, including asset management, logistics, and transportation services. Advanced machine learning algorithms now provide sophisticated methods for predicting long-term sales demand, aiding in budget management and program development.

This report focuses on predicting sales for large marts by analyzing historical data from various supermarkets and products. Machine learning algorithms, such as linear regression and random forests, are employed to forecast sales volumes. Effective marketing is crucial for any organization, and accurate sales forecasting plays a pivotal role in retail operations. It assists in developing business strategies, understanding market dynamics, and enhancing market knowledge. Regular sales forecasting enables in-depth analysis of current conditions, facilitating informed decisions regarding customer acquisition, resource allocation, and strategic planning for upcoming periods.

Sales forecasts are based on historical data, requiring a comprehensive understanding of past trends to identify and capitalize on market opportunities, regardless of external circumstances. Ongoing research in the retail sector aims to predict long-term sales demand. Traditional mathematical methods have been used for sales prediction; however, they are time-consuming and often struggle with complex data. To address these limitations, machine learning techniques are utilized, as they can efficiently handle large and intricate datasets.

1.2 Problem Statement

With escalating competition, many malls and large retail chains strive to maintain a competitive edge. To identify the various factors influencing sales and to develop strategies for increased profitability, a reliable predictive model is essential. Such a model can provide valuable insights and contribute to profit maximization.

1.3 Objectives

The objectives of this project are:

1. To forecast future sales using a given dataset.

2. To identify key features that significantly impact the sales of specific products.

3. To determine the most effective algorithm for accurate sales prediction.

1.4 Methodology

Figure 1.1 illustrates the steps involved in constructing the predictive model. The methodology comprises the following stages:

FIGURE 1.1: Process of building a predictive model

1. **Data Collection:** The initial step involves gathering data. For this project, data was sourced from Kaggle, accessible at: `https://www.kaggle.com/brijbhushannanda1979/bigmart-sales-data/code`
2. **Data Preprocessing:** This phase focuses on cleaning the dataset, such as identifying and handling missing values. In this dataset, attributes like `Item Weight` and `Outlet Size` contained missing entries.
3. **Exploratory Data Analysis (EDA):** EDA is crucial for uncovering significant insights from the data. In this project, libraries such as `klib` and `dtale` were utilized for analysis.
4. **Algorithm Testing:** Various algorithms, including simple linear regression and XGBoost, were applied to determine the most effective method for sales prediction.
5. **Model Building:** After completing the previous steps, the dataset was ready for model construction. The developed model is now prepared to forecast Big Mart sales.
6. **Web Deployment:** To enhance user accessibility, the predictive model was deployed as a web application.

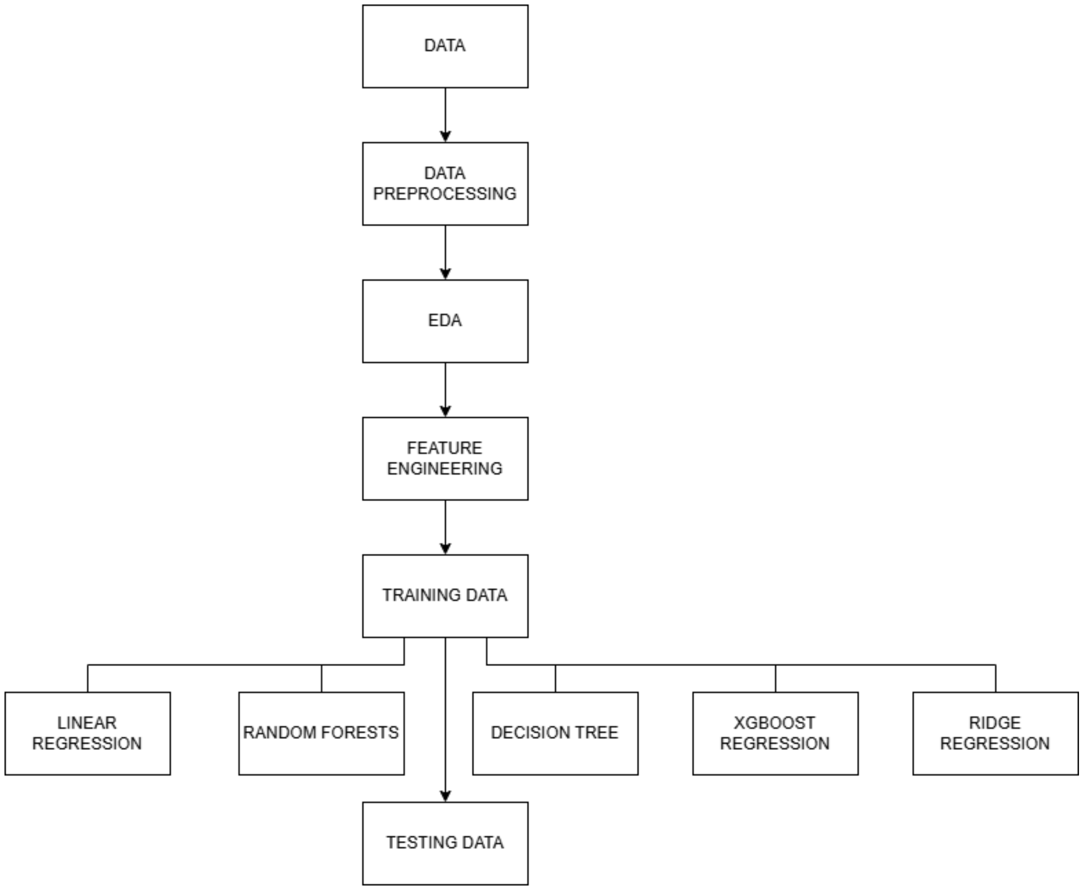


FIGURE 1.2: Working procedure of proposed model

Chapter 2

Models Development

2.1 Phases of Model Development

2.1.1 Data Collection

For this study, the 2013 Big Mart sales dataset was used. It consists of 12 features, including:

- Item attributes: Fat content, type, MRP, weight, visibility.
- Outlet characteristics: Type, size, establishment year, location.
- Target variable: Sales volume.

The dataset comprises 8,523 product records spanning multiple regions and cities. It incorporates store-level factors such as population density, capacity, and location, alongside product-level factors like advertising and demand trends. The dataset was divided into training (80%) and testing (20%) subsets.

A	B	C	D	E	F	G	H	I	J	K	L
Item_Identifier	Item_Weight	Item_Fat_	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location	Outlet_Type	Item_Outlet_Sales
FDA15	9.3	Low Fat	0.016047301	Dairy	249.8092	OUT049	1999	Medium	Tier 1	Supermarket Ty	3735.138
DRC01	5.92	Regular	0.019278216	Soft Drinks	48.2692	OUT018	2009	Medium	Tier 3	Supermarket Ty	443.4228
FDN15	17.5	Low Fat	0.016760075	Meat	141.618	OUT049	1999	Medium	Tier 1	Supermarket Ty	2097.27
FDX07	19.2	Regular	0	Fruits and Veget	182.095	OUT010	1998		Tier 3	Grocery Store	732.38
NCD19	8.93	Low Fat	0	Household	53.8614	OUT013	1987	High	Tier 3	Supermarket Ty	994.7052
FDP36	10.395	Regular	0	Baking Goods	51.4008	OUT018	2009	Medium	Tier 3	Supermarket Ty	556.6088
FDO10	13.65	Regular	0.012741089	Snack Foods	57.6588	OUT013	1987	High	Tier 3	Supermarket Ty	343.5528
FDP10		Low Fat	0.127469857	Snack Foods	107.7622	OUT027	1985	Medium	Tier 3	Supermarket Ty	4022.7636
FDH17	16.2	Regular	0.016687114	Frozen Foods	96.9726	OUT045	2002		Tier 2	Supermarket Ty	1076.5986
FDU28	19.2	Regular	0.09444959	Frozen Foods	187.8214	OUT017	2007		Tier 2	Supermarket Ty	4710.535
FDY07	11.8	Low Fat	0	Fruits and Veget	45.5402	OUT049	1999	Medium	Tier 1	Supermarket Ty	1516.0266
FDA03	18.5	Regular	0.045463773	Dairy	144.1102	OUT046	1997	Small	Tier 1	Supermarket Ty	2187.153
FDX32	15.1	Regular	0.1000135	Fruits and Veget	145.4786	OUT049	1999	Medium	Tier 1	Supermarket Ty	1589.2646
FDS46	17.6	Regular	0.047257328	Snack Foods	119.6782	OUT046	1997	Small	Tier 1	Supermarket Ty	2145.2076
FDF32	16.35	Low Fat	0.0680243	Fruits and Veget	196.4426	OUT013	1987	High	Tier 3	Supermarket Ty	1977.426
FDP49	9	Regular	0.069088961	Breakfast	56.3614	OUT046	1997	Small	Tier 1	Supermarket Ty	1547.3192
NCB42	11.8	Low Fat	0.008596051	Health and Hygi	115.3492	OUT018	2009	Medium	Tier 3	Supermarket Ty	1621.8888
FDP49	9	Regular	0.069196376	Breakfast	54.3614	OUT049	1999	Medium	Tier 1	Supermarket Ty	718.3982

FIGURE 2.1: Dataset overview

A. Product Features

- `Item_Identifier` - Unique ID for each product
- `Item_Weight` - Weight of the product
- `Item_Fat_Content` - Whether the product is low fat or regular
- `Item_Visibility` - Percentage of total display area allocated to this product in the store
- `Item_Type` - Category of the product (e.g., Dairy, Snacks, Fruits, etc.)
- `Item_MRP` - Maximum Retail Price (MRP) of the product

B. Store Features

- `Outlet_Identifier` - Unique ID for each store
- `Outlet_Establishment_Year` - Year the store was opened
- `Outlet_Size` - Size of the store (Small, Medium, Large)
- `Outlet_Location_Type` - Type of city (Tier 1, Tier 2, Tier 3)
- `Outlet_Type` - Whether the store is a supermarket, grocery store, etc.

C. Target Variable

- `Item_Outlet_Sales` - Sales of a particular product in a particular store (This is the value we need to predict)

FIGURE 2.2: Features of the dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
```

FIGURE 2.3: How libraries are imported

```
train.head()
```
✓ 0.0s Python

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	1999	Medium	Tier 1	Supermarket Type1	3735.1380
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009	Medium	Tier 3	Supermarket Type2	443.4228
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	1999	Medium	Tier 1	Supermarket Type1	2097.2700
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	OUT010	1998	NaN	Tier 3	Grocery Store	732.3800
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	OUT013	1987	High	Tier 3	Supermarket Type1	994.7052

FIGURE 2.4: First five dataset

Outlet_Size includes some missing data, Item_Identifier is a character string with some special code used by the bigmart, and Item_Visibility has some values of 0 that have no meaning.

```
train.info()
✓ 0.0s
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8522 entries, 0 to 8521
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Item_Identifier            8522 non-null   object
 1   Item_Weight                7059 non-null   float64
 2   Item_Fat_Content           8522 non-null   object
 3   Item_Visibility            8522 non-null   float64
 4   Item_Type                  8522 non-null   object
 5   Item_MRP                   8522 non-null   float64
 6   Outlet_Identifier          8522 non-null   object
 7   Outlet_Establishment_Year  8522 non-null   int64
 8   Outlet_Size                6112 non-null   object
 9   Outlet_Location_Type       8522 non-null   object
 10  Outlet_Type                8522 non-null   object
 11  Item_Outlet_Sales          8522 non-null   float64
dtypes: float64(4), int64(1), object(7)
memory usage: 799.1+ KB
```

FIGURE 2.5: Description of dataset using info() method

In Figure 2.5 we can clearly see that there are totally 12 features out of which Numeric data count is 5 and Categorical data count is 7.

```
train.describe()
✓ 0.0s
```

	Item_Weight	Item_Visibility	Item_MRP	Outlet_Establishment_Year	Item_Outlet_Sales
count	7059.000000	8522.000000	8522.000000	8522.000000	8522.000000
mean	12.857370	0.066135	141.000471	1997.831964	2181.455027
std	4.643728	0.051600	62.274675	8.372247	1706.530835
min	4.555000	0.000000	31.290000	1985.000000	33.290000
25%	8.772500	0.026988	93.844250	1987.000000	834.913200
50%	12.600000	0.053935	143.014100	1999.000000	1794.331000
75%	16.850000	0.094594	185.652250	2004.000000	3101.296400
max	21.350000	0.328391	266.888400	2009.000000	13086.964800

FIGURE 2.6: Description of dataset

In Figure 2.6 Item_Visibility feature has a minimum value of 0.00 and Item_weight has count of 7059.

2.2 Exploratory Data Analysis (EDA)

EDA was performed using multiple libraries:

- **D-Tale**: A Flask and React-based tool for analyzing Pandas data structures.

```python
import dtale
```

✓ 0.0s

```python
import dtale.app as dtale_app
dtale_app.USE_COLAB = True
dtale.show(train)
```

✓ 0.2s

FIGURE 2.7: D-Tale library

12 / 8522	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales
0	FDA15	9.30	Low Fat	0.02	Dairy	249.81	OUT049	1999	Medium	Tier 1	Supermarket Type1	3735.14
1	DRC01	5.92	Regular	0.02	Soft Drinks	48.27	OUT018	2009	Medium	Tier 3	Supermarket Type2	443.42
2	FDN15	17.50	Low Fat	0.02	Meat	141.62	OUT049	1999	Medium	Tier 1	Supermarket Type1	2097.27
3	FDX07	19.20	Regular	0.00	Fruits and Vegetables	182.10	OUT010	1998	nan	Tier 3	Grocery Store	732.38
4	NCD19	8.93	Low Fat	0.00	Household	53.86	OUT013	1987	High	Tier 3	Supermarket Type1	994.71
5	FDP36	10.40	Regular	0.00	Baking Goods	51.40	OUT018	2009	Medium	Tier 3	Supermarket Type2	556.61
6	FDO10	13.65	Regular	0.01	Snack Foods	57.66	OUT013	1987	High	Tier 3	Supermarket Type1	343.55
7	FDP10	12.86	Low Fat	0.13	Snack Foods	107.76	OUT027	1985	Medium	Tier 3	Supermarket Type3	4022.76
8	FDH17	16.20	Regular	0.02	Frozen Foods	96.97	OUT045	2002	nan	Tier 2	Supermarket Type1	1076.60
9	FDU28	19.20	Regular	0.09	Frozen Foods	187.82	OUT017	2007	nan	Tier 2	Supermarket Type1	4710.54
10	FDY07	11.80	Low Fat	0.00	Fruits and Vegetables	45.54	OUT049	1999	Medium	Tier 1	Supermarket Type1	1516.03
11	FDA03	18.50	Regular	0.05	Dairy	144.11	OUT046	1997	Small	Tier 1	Supermarket Type1	2187.15
12	FDX32	15.10	Regular	0.10	Fruits and Vegetables	145.48	OUT049	1999	Medium	Tier 1	Supermarket Type1	1589.26
13	FDS46	17.60	Regular	0.05	Snack Foods	119.68	OUT046	1997	Small	Tier 1	Supermarket Type1	2145.21
14	FDF32	16.35	Low Fat	0.07	Fruits and Vegetables	196.44	OUT013	1987	High	Tier 3	Supermarket Type1	1977.43
15	FDP49	9.00	Regular	0.07	Breakfast	56.36	OUT046	1997	Small	Tier 1	Supermarket Type1	1547.32
16	NCB42	11.80	Low Fat	0.01	Health and Hygiene	115.35	OUT018	2009	Medium	Tier 3	Supermarket Type2	1621.89
17	FDP49	9.00	Regular	0.07	Breakfast	54.36	OUT049	1999	Medium	Tier 1	Supermarket Type1	718.40
18	DRI11	12.86	Low Fat	0.03	Hard Drinks	113.28	OUT027	1985	Medium	Tier 3	Supermarket Type3	2303.67
19	FDU02	13.35	Low Fat	0.10	Dairy	230.54	OUT035	2004	Small	Tier 2	Supermarket Type1	2748.42
20	FDN22	18.85	Regular	0.14	Snack Foods	250.87	OUT013	1987	High	Tier 3	Supermarket Type1	3775.09
21	FDW12	12.86	Regular	0.04	Baking Goods	144.54	OUT027	1985	Medium	Tier 3	Supermarket Type3	4064.04
22	NCB30	14.60	Low Fat	0.03	Household	196.51	OUT035	2004	Small	Tier 2	Supermarket Type1	1587.27
23	FDC37	12.86	Low Fat	0.06	Baking Goods	107.69	OUT019	1985	Small	Tier 1	Grocery Store	214.39
24	FDR28	13.85	Regular	0.03	Frozen Foods	165.02	OUT046	1997	Small	Tier 1	Supermarket Type1	4078.03
25	NCD06	13.00	Low Fat	0.10	Household	45.91	OUT017	2007	nan	Tier 2	Supermarket Type1	838.91
26	FDV10	7.65	Regular	0.07	Snack Foods	42.31	OUT035	2004	Small	Tier 2	Supermarket Type1	1065.28
27	DRJ59	11.65	low fat	0.02	Hard Drinks	39.12	OUT013	1987	High	Tier 3	Supermarket Type1	308.93
28	FDE51	5.93	Regular	0.16	Dairy	45.51	OUT010	1998	nan	Tier 3	Grocery Store	178.43
29	FDC14	12.86	Regular	0.07	Canned	43.65	OUT019	1985	Small	Tier 1	Grocery Store	125.84
30	FDV38	19.25	Low Fat	0.17	Dairy	55.80	OUT010	1998	nan	Tier 3	Grocery Store	163.79
31	NCS17	18.60	Low Fat	0.08	Health and Hygiene	96.44	OUT018	2009	Medium	Tier 3	Supermarket Type2	2741.76
32	FDP33	18.70	Low Fat	0.00	Snack Foods	256.67	OUT018	2009	Medium	Tier 3	Supermarket Type2	3068.01
33	FDO23	17.85	Low Fat	0.00	Breads	93.14	OUT045	2002	nan	Tier 2	Supermarket Type1	2174.50

FIGURE 2.8: D-Tale window

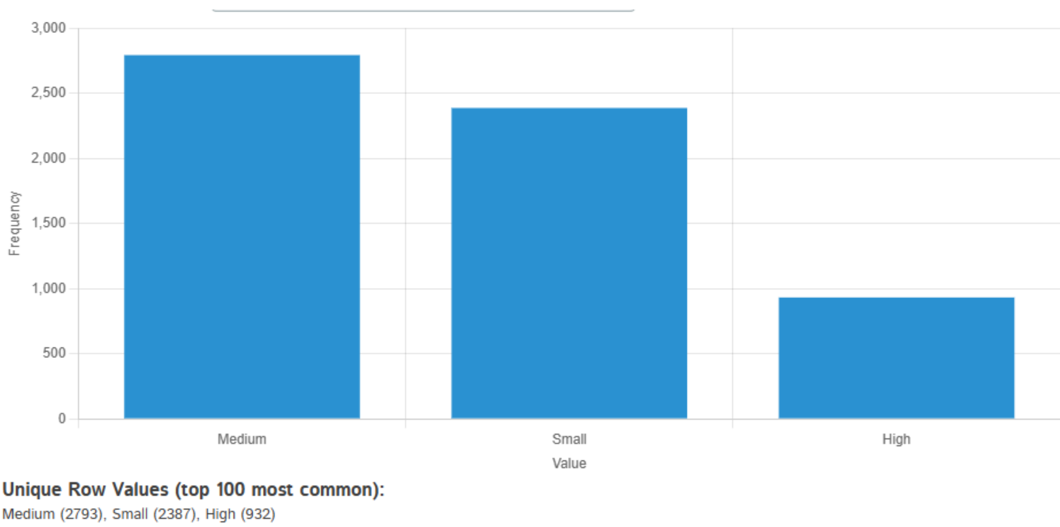


FIGURE 2.9: Frequency of values in Outlet_Size

- **Klib**: A Python library for data cleaning, visualization, and feature correlation.

```
import klib
klib.corr_plot(train)  # Show correlations
```

✓ 0.2s

FIGURE 2.10: Klib library

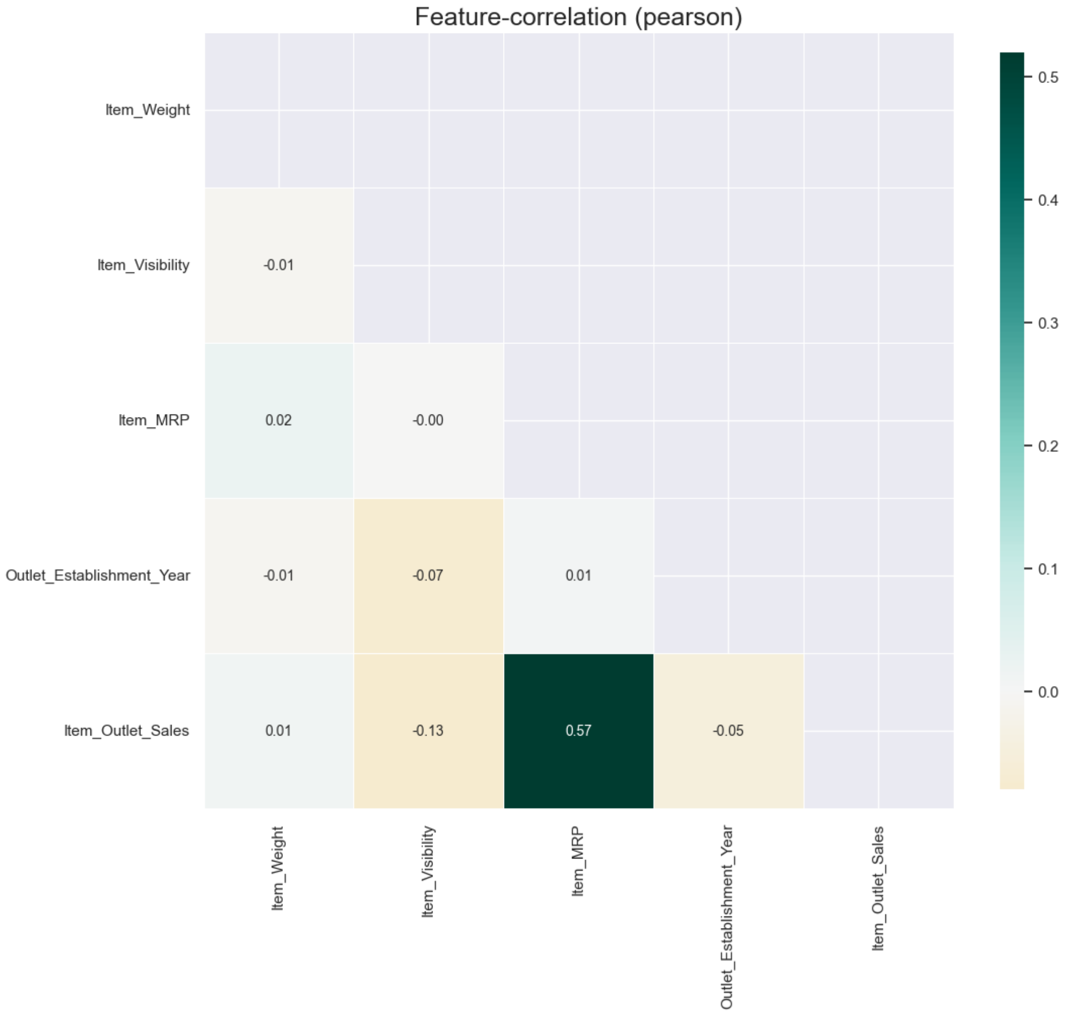


FIGURE 2.11: Feature- correlation using klib Library

```
klib.corr_mat(train)  # Show correlations
```

✓ 0.0s

	Item_Weight	Item_Visibility	Item_MRP	Outlet_Establishment_Year	Item_Outlet_Sales
Item_Weight	1.00	-0.01	0.02	-0.01	0.01
Item_Visibility	-0.01	1.00	-0.00	-0.07	-0.13
Item_MRP	0.02	-0.00	1.00	0.01	0.57
Outlet_Establishment_Year	-0.01	-0.07	0.01	1.00	-0.05
Item_Outlet_Sales	0.01	-0.13	0.57	-0.05	1.00

FIGURE 2.12: Color- encoded correlation matrix.

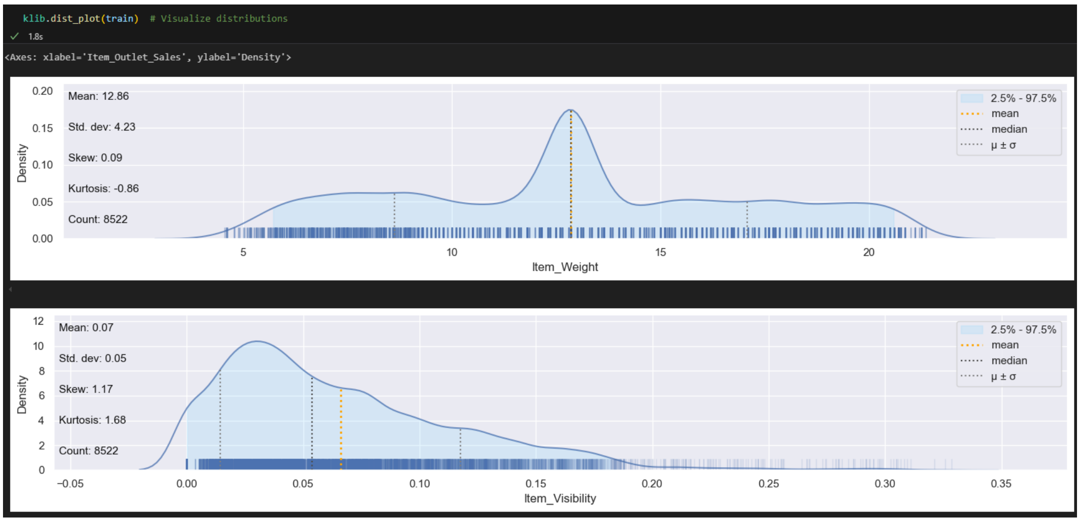


FIGURE 2.13: Distribution plot for Item_Weight and Item_Visibility

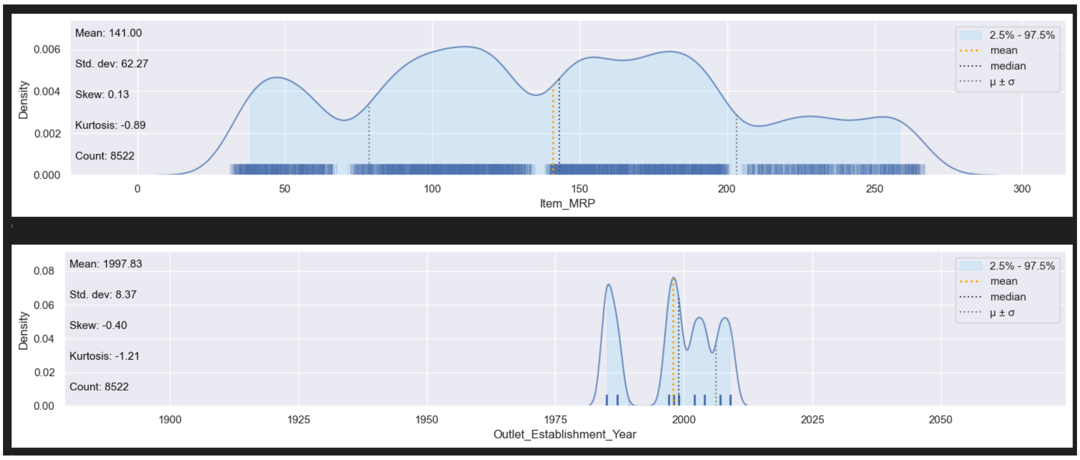


FIGURE 2.14: Distribution plot for Item_MRP and Outlet_Establishment_Year

- **Seaborn**: Used for visualizing feature relationships and correlations.

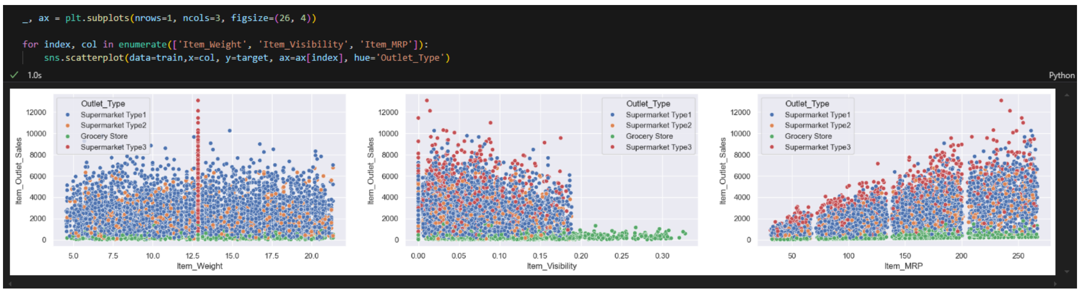


FIGURE 2.15: Relationship between different features

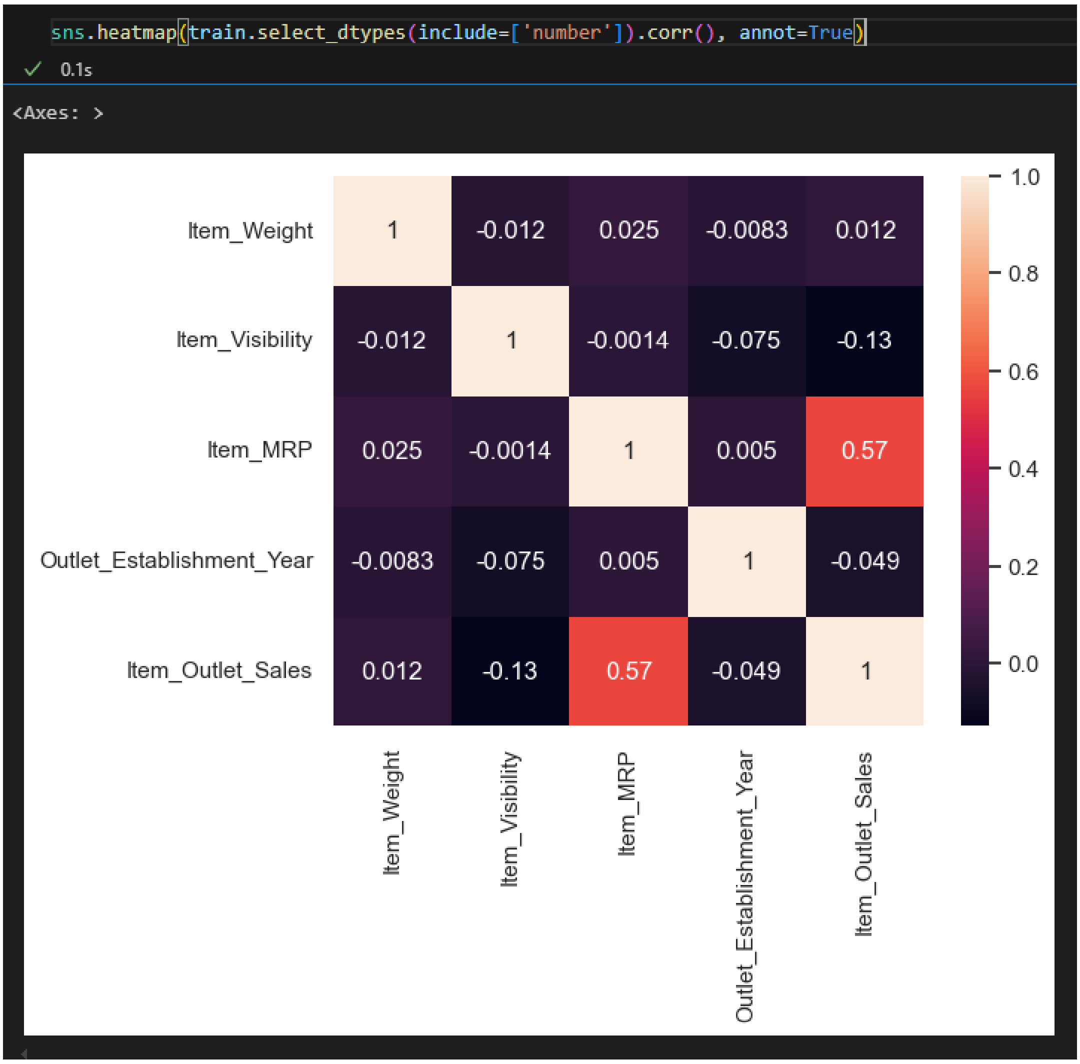


FIGURE 2.16: Correlation between different features

Findings included:

- `Item Visibility` had the lowest correlation with the target variable.
- `Item MRP` exhibited a strong positive correlation (0.57) with sales.

2.3 Data Preprocessing

2.3.1 Handling Missing Values

Analysis revealed missing values in `Item Weight` and `Outlet Size` columns. Instead of dropping records, which could reduce predictive accuracy, imputation techniques were applied:

- Numerical features (e.g., `Item Weight`) were filled using mean imputation.
- Categorical features (e.g., `Outlet Size`) were imputed using mode imputation.

```
train.isnull().sum()
✓ 0.0s

Item_Identifier                 0
Item_Weight                  1463
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type            0
Outlet_Type                     0
Item_Outlet_Sales               0
dtype: int64
```

FIGURE 2.17: Description of train dataset using isNull() method

```
test.isnull().sum()
✓ 0.0s

Item_Identifier                 0
Item_Weight                   976
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  1606
Outlet_Location_Type            0
Outlet_Type                     0
dtype: int64
```

FIGURE 2.18: Description of test dataset using isNull() method

From Figure 2.18 above that the column names Item_Weight and Outlet_Size, respectively, contain 976 and 1606 missing values.

In order to deal with these missing values, such as removing the rows that include missing values or utilizing various techniques to fill in the missing value with appropriate values. Given the size of our dataset 8522 rows dropping would not be the best option because it would result in a lower prediction accuracy.

```
train.info()
✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8522 entries, 0 to 8521
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Item_Identifier            8522 non-null   object
 1   Item_Weight                7059 non-null   float64
 2   Item_Fat_Content           8522 non-null   object
 3   Item_Visibility            8522 non-null   float64
 4   Item_Type                  8522 non-null   object
 5   Item_MRP                   8522 non-null   float64
 6   Outlet_Identifier          8522 non-null   object
 7   Outlet_Establishment_Year  8522 non-null   int64
 8   Outlet_Size                6112 non-null   object
 9   Outlet_Location_Type       8522 non-null   object
 10  Outlet_Type                8522 non-null   object
 11  Item_Outlet_Sales          8522 non-null   float64
dtypes: float64(4), int64(1), object(7)
memory usage: 799.1+ KB
```

FIGURE 2.19: Datatype of various features of dataset

Since Item_Weight is a numerical feature, filling its missing value using the average imputation method.

```
train['Item_Weight'].fillna(train['Item_Weight'].mean(), inplace=True)
test['Item_Weight'].fillna(test['Item_Weight'].mean(), inplace=True)
✓ 0.0s

train.isnull().sum()
✓ 0.0s

Item_Identifier                 0
Item_Weight                     0
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type            0
Outlet_Type                     0
Item_Outlet_Sales               0
dtype: int64
```

FIGURE 2.20: Missing value in Outlet_Size column = 2410

Finally, we have this

```
train['Outlet_Size'].fillna(train['Outlet_Size'].mode(), inplace=True)
test['Outlet_Size'].fillna(test['Outlet_Size'].mode(), inplace=True)
```
✓ 0.0s

```
train.isnull().sum()
```
✓ 0.0s

```
Item_Identifier              0
Item_Weight                  0
Item_Fat_Content             0
Item_Visibility              0
Item_Type                    0
Item_MRP                     0
Outlet_Identifier            0
Outlet_Establishment_Year    0
Outlet_Size                  0
Outlet_Location_Type         0
Outlet_Type                  0
Item_Outlet_Sales            0
dtype: int64
```

FIGURE 2.21: Filling Values in Outlet_Size

2.3.2 Handling Categorical Variables

```
train.replace({'Item_Fat_Content': {'low fat': 'Low Fat', 'LF': 'Low Fat', 'reg':'Regular'}}, inplace=True)
```
✓ 0.0s

FIGURE 2.22: Handling Categorical Variables

2.3.3 Data Cleaning and Feature Engineering

Data cleaning was conducted using the Klib library, ensuring that records were free from inconsistencies and anomalies.

```
klib.data_cleaning(train)
```
✓ 0.0s

```
Shape of cleaned data: (8192, 12) - Remaining NAs: 0


Dropped rows: 0
     of which 0 duplicates. (Rows (first 150 shown): [])

Dropped columns: 0
     of which 0 single valued.     Columns: []
Dropped missing values: 0
Reduced memory by at least: 0.86 MB (-81.13%)
```

	item_identifier	item_weight	item_fat_content	item_visibility	item_type	item_mrp	outlet_identifier	outlet_establishment_year	outlet_size	outlet_location_type	outlet_type	item_outlet_sales
0	156	9.300000	0	0.016047	4	249.809204	9	1999	1	0	1	3735.137939
1	8	5.920000	1	0.019278	14	48.269199	3	2009	1	2	2	443.422791
2	662	17.500000	0	0.016760	10	141.617996	9	1999	1	0	1	2097.270020
3	1121	19.200001	1	0.000000	6	182.095001	0	1998	3	2	0	732.380005
4	1297	8.930000	0	0.000000	9	53.861401	1	1987	0	2	1	994.705200
...	...	...	...	...	...	...	...	...	...	...	...	...
8187	389	20.750000	1	0.083607	5	178.831802	8	1997	2	0	1	3608.635986
8188	370	6.865000	0	0.056783	13	214.521805	1	1987	0	2	1	2778.383301
8189	897	8.380000	1	0.046982	0	108.156998	7	2002	3	1	1	549.284973
8190	1357	10.600000	0	0.035186	8	85.122398	6	2004	2	1	1	1193.113647
8191	681	7.210000	1	0.145221	13	103.133202	3	2009	1	2	2	1845.597656

8192 rows × 12 columns

FIGURE 2.23: Data Cleaning using Klib

Feature engineering included:

- Encoding categorical variables using label encoding.
- Replacing incorrect or illogical values (e.g., setting `Item Visibility` to its mean where it was zero).
- Standardizing numerical features to enhance model performance.

Label Encoder

```
encoder = LabelEncoder()
```
✓ 0.0s

```
train['Item_Identifier']  = encoder.fit_transform(train['Item_Identifier'])
train['Item_Fat_Content']  = encoder.fit_transform(train['Item_Fat_Content'])
train['Item_Type']  = encoder.fit_transform(train['Item_Type'])
train['Outlet_Identifier']  = encoder.fit_transform(train['Outlet_Identifier'])
train['Outlet_Size']  = encoder.fit_transform(train['Outlet_Size'])
train['Outlet_Location_Type']  = encoder.fit_transform(train['Outlet_Location_Type'])
train['Outlet_Type']  = encoder.fit_transform(train['Outlet_Type'])
```
✓ 0.0s

```
train.head()
```
✓ 0.0s

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales
0	156	9.30	0	0.016047	4	249.8092	9	1999	1	0	1	3735.1380
1	8	5.92	1	0.019278	14	48.2692	3	2009	1	2	2	443.4228
2	662	17.50	0	0.016760	10	141.6180	9	1999	1	0	1	2097.2700
3	1121	19.20	1	0.000000	6	182.0950	0	1998	3	2	0	732.3800
4	1297	8.93	0	0.000000	9	53.8614	1	1987	0	2	1	994.7052

FIGURE 2.24: Label Encoding Code

Model Development

The dataset was split into training and testing subsets:

- `X_train` and `X_test` for input features.
- `Y_train` and `Y_test` for target labels.

```
X = train.drop('Item_Outlet_Sales', axis=1)
y = train['Item_Outlet_Sales']
```
✓ 0.0s

```
X.head()
```
✓ 0.0s

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type
0	156	9.30	0	0.016047	4	249.8092	9	1999	1	0	1
1	8	5.92	1	0.019278	14	48.2692	3	2009	1	2	2
2	662	17.50	0	0.016760	10	141.6180	9	1999	1	0	1
3	1121	19.20	1	0.000000	6	182.0950	0	1998	2	2	0
4	1297	8.93	0	0.000000	9	53.8614	1	1987	0	2	1

FIGURE 2.25: X and y

```
y.head()
```
✓ 0.0s

```
0    3735.1380
1     443.4228
2    2097.2700
3     732.3800
4     994.7052
Name: Item_Outlet_Sales, dtype: float64
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```
✓ 0.0s

FIGURE 2.26: Splitting of data into train and test data set

Standardization

Standardization techniques were applied to normalize input features.

```
X.describe()
```

✓ 0.0s

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type
count	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000	8192.000000
mean	780.187744	12.863652	0.352051	0.063224	7.229980	139.111799	4.762817	1998.089844	1.449463	1.106567	1.201416
std	450.102233	4.267190	0.477639	0.046287	4.205454	61.536177	2.844914	8.287443	0.686572	0.808782	0.770255
min	0.000000	4.555000	0.000000	0.000000	0.000000	31.290000	0.000000	1985.000000	0.000000	0.000000	0.000000
25%	394.750000	9.300000	0.000000	0.026817	4.000000	93.027800	2.000000	1987.000000	1.000000	0.000000	1.000000
50%	783.500000	12.857370	0.000000	0.052735	6.000000	141.148300	5.000000	1999.000000	2.000000	1.000000	1.000000
75%	1168.000000	16.100000	1.000000	0.092790	10.000000	183.663400	7.000000	2004.000000	2.000000	2.000000	1.000000
max	1558.000000	21.350000	1.000000	0.195721	15.000000	266.888400	9.000000	2009.000000	2.000000	2.000000	3.000000

FIGURE 2.27: Standardization of dataset

Scaling method

```
X_train_std = scaler.fit_transform(X_train)
```

✓ 0.0s

```
X_train_std
```

✓ 0.0s

```
array([[-0.2530511 ,  1.52242505, -0.73933292, ...,  1.27199331,
        -0.13788025, -0.2631656 ],
       [-1.28238654, -0.4632596 , -0.73933292, ..., -1.74551803,
         1.10152889, -0.2631656 ],
       [-1.5069284 ,  0.28944523, -0.73933292, ..., -0.73968092,
        -1.37728938, -0.2631656 ],
       ...,
       [ 0.60287578, -0.36814401, -0.73933292, ...,  0.2661562 ,
        -0.13788025, -0.2631656 ],
       [-0.4531379 ,  1.26408642, -0.73933292, ..., -1.74551803,
         1.10152889, -0.2631656 ],
       [ 1.19424345, -0.90830661, -0.73933292, ..., -0.73968092,
         1.10152889,  1.02861281]])
```

FIGURE 2.28: X_train_std array

```
X_test_std = scaler.fit_transform(X_test)
✓ 0.0s

X_test_std
✓ 0.0s

array([[ 6.85898527e-01, -1.43127826e+00,  1.37315818e+00, ...,
        -7.28957605e-01,  1.11781520e+00,  1.07098502e+00],
       [ 1.28896426e+00,  2.16061017e-03, -7.28248221e-01, ...,
        -7.28957605e-01,  1.11781520e+00,  2.39677387e+00],
       [ 1.59936573e+00,  1.39498587e+00, -7.28248221e-01, ...,
         2.85271185e-01, -1.07669156e-01, -2.54803837e-01],
       ...,
       [ 2.51336457e-01, -1.78339916e+00,  1.37315818e+00, ...,
         1.29949997e+00, -1.07669156e-01, -2.54803837e-01],
       [-6.90953745e-01,  2.21249532e-01, -7.28248221e-01, ...,
         2.85271185e-01, -1.33315351e+00, -2.54803837e-01],
       [ 1.23131827e+00, -9.87350262e-01, -7.28248221e-01, ...,
        -7.28957605e-01,  1.11781520e+00,  1.07098502e+00]])
```

FIGURE 2.29: X_test_std array

```
y_train
✓ 0.0s

5412    3705.8428
1682    5536.7928
6138    1842.9344
7287     217.7166
5664    5086.7120
           ...
5928     980.0576
8328    5518.1504
1417    2768.3964
1613    4288.4178
5158    1518.0240
Name: Item_Outlet_Sales, Length: 6553, dtype: float64
```

```
y_test
✓ 0.0s

7743    1411.4960
400      717.7324
1801    2360.9268
4708     455.4072
4576    1955.4546
           ...
8165    4327.7000
3984     591.8962
6005    1693.7952
5801    3179.8608
526      283.6308
Name: Item_Outlet_Sales, Length: 1639, dtype: float64
```

FIGURE 2.30: y_train and y_test array

In Figure 2.28, Figure 2.29 and Figure 2.30 we just split the train and test data into X_train_std , Y_train, X_test_std and Y_test.

Chapter 3

Model Development and Evaluation

3.1 Model Selection

After performing data preprocessing and feature transformation, the dataset is ready for model fitting. The training dataset is provided to the algorithm for learning, and the testing dataset is used to evaluate predictive performance. The models implemented in this study include:

- Linear Regression (LR)
- Decision Tree Regressor (DTR)
- Random Forest Regressor (RFR)
- XGBoost Regressor (XGB)
- Gradient Boosting Regressor (GBR)

3.1.1 Linear Regression (LR)

Linear Regression establishes a relationship between the dependent variable (Y) and independent variables (X) through the equation:

$$Y = \beta_0 + \beta_1 X + \epsilon \tag{3.1}$$

where:

- Y is the predicted sales value.
- X represents features such as Item MRP, Item Visibility, Outlet Size, etc.
- β_0 is the intercept.
- β_1 is the coefficient (slope) representing the impact of X on Y.
- ϵ is the residual error.

3.1.2 Decision Tree Regressor (DTR)

Decision Tree Regression is a non-parametric model that splits the data into hierarchical structures based on feature conditions, making it useful for capturing non-linear relationships. The model recursively partitions the dataset into homogenous groups by minimizing variance.

The prediction for a given input is computed as:

$$\hat{Y} = \frac{1}{N}\sum_{i=1}^{N} Y_i \tag{3.2}$$

where:

- $\hat{Y}$ is the predicted sales value.
- N is the number of samples in a given leaf node.
- Y_i represents the actual sales values in that node.

The tree splits at each node based on minimizing the Mean Squared Error (MSE):

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y})^2 \tag{3.3}$$

3.1.3 Random Forest Regressor (RFR)

Random Forest Regression is an ensemble method that builds multiple decision trees and aggregates their outputs to improve prediction accuracy and reduce overfitting. Each tree is trained on a random subset of data and features (Bootstrap Aggregation).

The final prediction is computed as the average prediction from all decision trees:

$$\hat{Y} = \frac{1}{T}\sum_{t=1}^{T} f_t(X) \tag{3.4}$$

where:

- T is the number of trees in the forest.
- $f_t(X)$ is the prediction from the t-th tree.
- $\hat{Y}$ is the final aggregated prediction.

The algorithm reduces variance by averaging multiple tree predictions, making it more robust compared to a single decision tree.

3.1.4 XGBoost Regressor (XGBR)

Extreme Gradient Boosting (XGBoost) is an optimized version of gradient boosting that uses decision trees as weak learners and sequentially improves their predictions. It minimizes a custom loss function with regularization to prevent overfitting.

The model updates predictions iteratively using:

$$F_m(X) = F_{m-1}(X) + \gamma h_m(X) \tag{3.5}$$

where:

- $F_m(X)$ is the new prediction.
- $F_{m-1}(X)$ is the previous iteration's prediction.
- $h_m(X)$ is the new weak learner (decision tree).

- γ is the learning rate controlling contribution from $h_m(X)$.

The objective function combines loss minimization and regularization:

$$Obj = \sum_{i=1}^{N} l(Y_i, \hat{Y}i) + \lambda \sum j = 1^T ||w_j||^2 \tag{3.6}$$

where:

- $l(Y_i, \hat{Y}_i)$ is the loss function (e.g., squared error).
- $\lambda ||w_j||^2$ is the regularization term controlling model complexity.

3.1.5 Gradient Boosting Regressor (GBR)

Gradient Boosting Regression builds an additive model sequentially by optimizing residual errors. Unlike Random Forest, which trains trees independently, Gradient Boosting corrects mistakes from previous trees iteratively.

The new model is updated as:

$$F_m(X) = F_{m-1}(X) + \eta h_m(X) \tag{3.7}$$

where:

- $F_m(X)$ is the new prediction.
- $F_{m-1}(X)$ is the previous model's prediction.
- $h_m(X)$ is the weak learner (decision tree).
- η (learning rate) controls the contribution of each new tree.

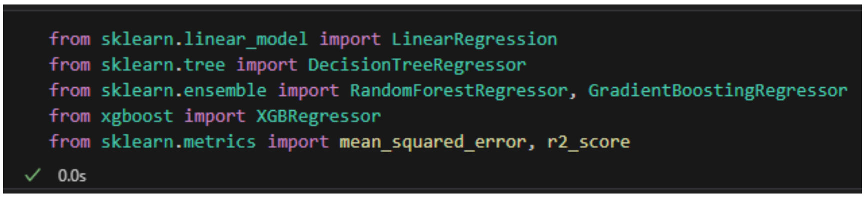
The gradient of the loss function is used to minimize errors at each stage:

$$g_m = \frac{\partial L(Y, F(X))}{\partial F(X)} \tag{3.8}$$

where:

- g_m is the gradient of the loss function.
- $L(Y, F(X))$ is the loss function (e.g., squared error).

Gradient Boosting is highly effective for structured data and outperforms traditional decision trees by focusing on difficult-to-predict examples.



```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, r2_score
```


FIGURE 3.1: Models's libraries

```python
# Initialize Models
models = {
    "Linear Regression": LinearRegression(),
    "Decision Tree": DecisionTreeRegressor(random_state=42),
    "Random Forest": RandomForestRegressor(n_estimators=100, random_state=42),
    "XGBoost": XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42),
    "Gradient Boosting": GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
}
✓ 0.0s
```

FIGURE 3.2: Initialize models

```python
# Train and Evaluate Models
results = {}

for name, model in models.items():
    model.fit(X_train, y_train)
    Y_pred = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, Y_pred))
    r2 = r2_score(y_test, Y_pred)
    results[name] = {"RMSE": rmse, "R² Score": r2}
```

FIGURE 3.3: Train and Evaluate Models

3.2 Hyperparameter Tuning and Optimization

In order to optimize parameters and enhance model performance, hyperparameter adjustment is essential. Here, we use GridSearchCV to adjust the settings for the Random Forest Regressor (RF) and Gradient Boosting Regressor (GBR). To improve predicted accuracy, the optimal parameters are chosen based on the outcomes of cross-validation.

Define Parameter Grid:

- For Random Forest (RF): n_estimators, max_depth, min_samples_split, min_samples_leaf.
- For Gradient Boosting (GBR): n_estimators, learning_rate, max_depth.

```python
# Define Parameter Grid for Random Forest
rf_param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Define Parameter Grid for Gradient Boosting Regressor
gbr_param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 10]
}
```

FIGURE 3.4: Define Parameter Grid for Random Forest and Gradient Boosting Regressor

Apply GridSearchCV:

- Evaluates different combinations of parameters using cross-validation.
- Selects the best-performing hyperparameters.

```
# Hyperparameter Tuning using GridSearchCV for Random Forest
rf = RandomForestRegressor(random_state=42)
rf_grid_search = GridSearchCV(rf, rf_param_grid, cv=5, scoring='r2', n_jobs=-1)
rf_grid_search.fit(X_train, y_train)

# Hyperparameter Tuning using GridSearchCV for Gradient Boosting
gbr = GradientBoostingRegressor(random_state=42)
gbr_grid_search = GridSearchCV(gbr, gbr_param_grid, cv=5, scoring='r2', n_jobs=-1)
gbr_grid_search.fit(X_train, y_train)

# Best Parameters and Performance
print("Best Parameters for Random Forest:", rf_grid_search.best_params_)
print("Best R² Score (Training) for RF:", rf_grid_search.best_score_)
print("Best Parameters for Gradient Boosting:", gbr_grid_search.best_params_)
print("Best R² Score (Training) for GBR:", gbr_grid_search.best_score_)
```

FIGURE 3.5: Hyperparameter Tuning using GridSearchCV

Train Models with Optimized Parameters:

- Fit the models using the best-found parameters.

```
# Train Models with Best Parameters
best_rf = RandomForestRegressor(**rf_grid_search.best_params_, random_state=42)
best_rf.fit(X_train, y_train)
Y_pred_rf = best_rf.predict(X_test)

best_gbr = GradientBoostingRegressor(**gbr_grid_search.best_params_, random_state=42)
best_gbr.fit(X_train, y_train)
Y_pred_gbr = best_gbr.predict(X_test)

# Evaluate Optimized Models
rmse_rf = np.sqrt(mean_squared_error(y_test, Y_pred_rf))
r2_rf = r2_score(y_test, Y_pred_rf)

rmse_gbr = np.sqrt(mean_squared_error(y_test, Y_pred_gbr))
r2_gbr = r2_score(y_test, Y_pred_gbr)
```

FIGURE 3.6: Train and Evaluate Models with Best Parameters

3.3 Model Evaluation

To determine the effectiveness of each model, performance metrics such as Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) are used:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.9}$$

$$RMSE = \sqrt{MSE} \tag{3.10}$$

where:

- Y_i is the actual sales value.
- $\hat{Y}_i$ is the predicted sales value.

Model	RMSE	R^2 Score
Linear Regression	1109	0.47
Decision Tree	1419	0.13
Random Forest	1051	0.52
XGBoost	1119	0.46
Gradient Boosting	996	0.57

TABLE 3.1: Model Performance Comparison

Model	RMSE	R^2 Score	Tuned RMSE	Tuned R^2 Score
Random Forest	1051	0.52	1003	0.56
Gradient Boosting	996	0.57	996	0.57

TABLE 3.2: Model Performance Comparison after Tuning

- n is the total number of observations.

Random Forest (RF) and Gradient Boosting (GBR) performance comparisons before and after hyperparameter adjustment are shown in this table. The assessment measures that are employed are R^2 Score and Root Mean Squared Error (RMSE):

- When Random Forest was tuned, its R^2 Score increased from 0.52 to 0.56 and its RMSE decreased from 1051 to 1003.
- After adjusting, gradient boosting showed no change and remained steady, suggesting ideal starting conditions.

Performance Analysis Using Graphs

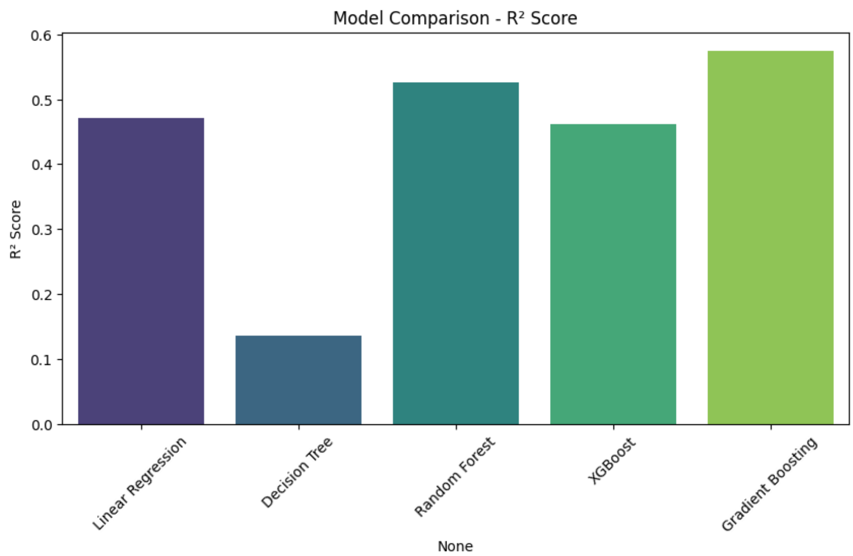


FIGURE 3.7: Model Comparison - R^2 Score

Figure 3.7 depicts the comparative between analysis of R^2 scores. In this graph, Gradient Boosting has the highest R^2 scores.

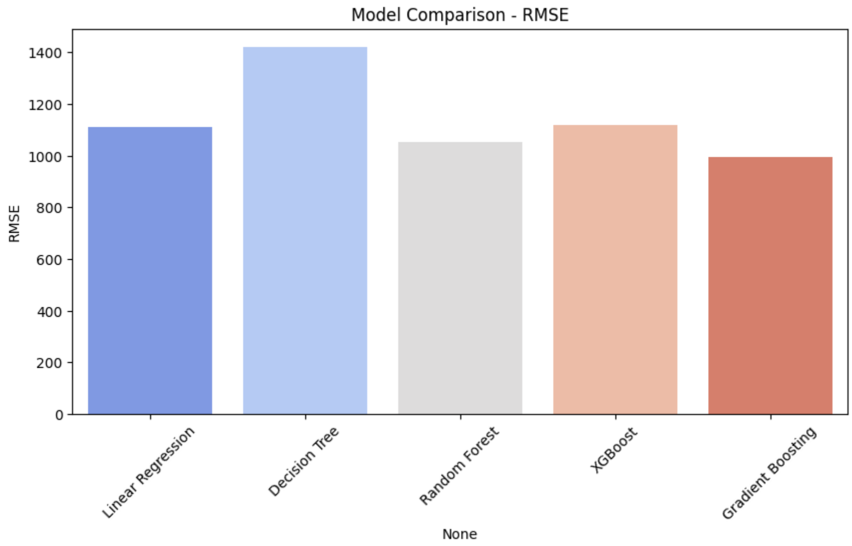


FIGURE 3.8: Model Comparison - RMSE

Figure 3.8 illustrates the comparative between analysis of RMSE values. In this graph, Gradient Boosting has the lowest RMSE value.

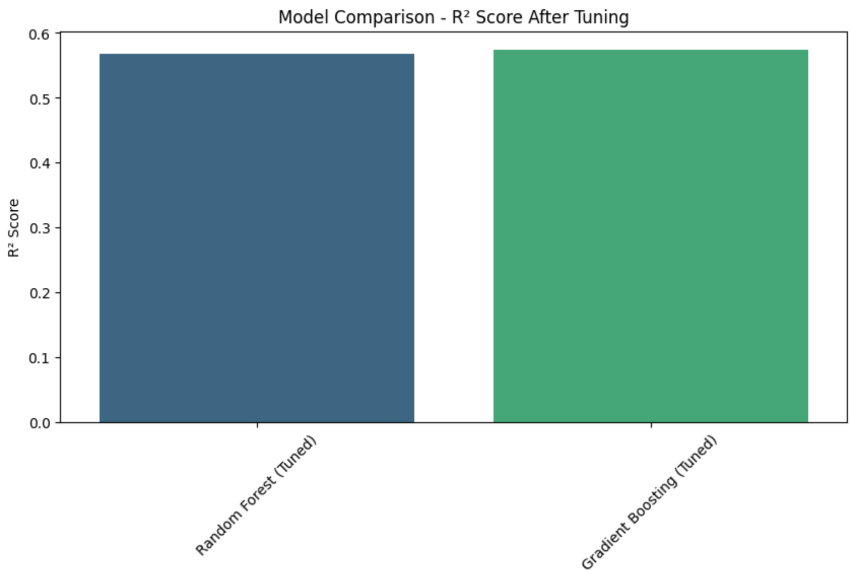


FIGURE 3.9: Model Comparison - R^2 Score After Tuning

Figure 3.9 illustrates the comparative between analysis of RMSE values. In this graph, Gradient Boosting still has the highest R^2 scores.

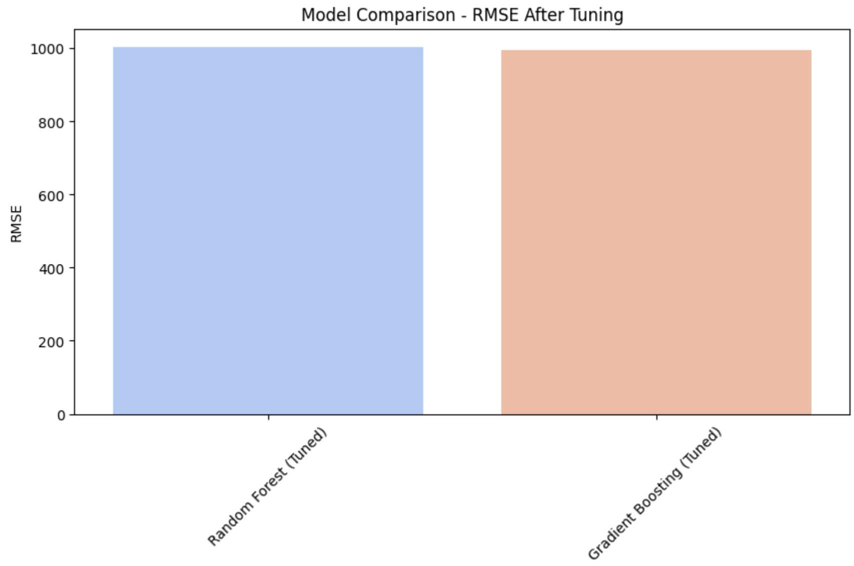


FIGURE 3.10: Model Comparison - RMSE After Tuning

Figure 3.10 illustrates the comparative between analysis of RMSE values. In this graph, Gradient Boosting still has the lowest RMSE value.

3.4 Final Model Selection

The optimal model is chosen based on RMSE after the results from many models are compared, even with hyperparameter tuning used for optimization. The Gradient Boosting model was the recommended option for deployment as it showed the highest accuracy in forecasting Big Mart sales.

Chapter 4

Web Application for BigMart Sales Prediction

To enhance accessibility and usability, a web application has been developed for predicting BigMart sales. This application allows users to input product attributes and receive sales predictions instantly. The backend is powered by a Flask web framework, while the frontend provides a user-friendly interface.

4.1 Technology Stack

The web application is built using the following technologies:

- Flask: A lightweight Python web framework used to handle HTTP requests and integrate the machine learning model.
- Scikit-learn: For model training and predictions.
- NumPy: Used for numerical transformations of input features.
- HTML, CSS, Bootstrap: For designing a responsive and user-friendly web interface.
- Pickle: For saving and loading the trained machine learning model.

4.2 System Architecture

The web application follows a client-server architecture:

- The client sends input data through the web interface.
- The server processes this input, transforms it into a suitable format, and passes it to the trained machine learning model.
- The model predicts sales values based on input features.
- The predicted sales figure is sent back to the client and displayed on the webpage.

4.3 System Architecture

The application follows a modular architecture comprising:

- **Data Preprocessing Module:** Handles data cleaning and feature engineering.

- **Model Training Module:** Trains and saves machine learning models.
- **Flask Application:** Serves as the web interface, handling user interactions and displaying predictions.
- **Templates:** HTML files rendered by Flask to present data and visualizations.

4.4 Implementation Details

4.4.1 Backend Development

The backend is developed using Flask, a lightweight WSGI web application framework in Python. The core functionalities include:

Model Loading

Pre-trained models are loaded using the `pickle` module:

```
import pickle

models = {
    "RandomForest": pickle.load(open("models/RandomForest.pkl", "rb")),
    "Ridge": pickle.load(open("models/Ridge.pkl", "rb")),
    "DecisionTree": pickle.load(open("models/DecisionTree.pkl", "rb")),
    "XGBoost": pickle.load(open("models/Xgboost.pkl", "rb")),
    "Linear": pickle.load(open("models/Linear.pkl", "rb")),
    "Ada": pickle.load(open("models/Ada.pkl", "rb"))
}
```

Flask Routes

The application defines several routes to handle different user interactions:

- `"/"`: Renders the homepage.
- `"/predict"`: Accepts user input for prediction and displays results.
- `"/view_csv"`: Displays samples of preprocessed and raw data.
- `"/dataset_insight"`: Provides insights into the dataset, including summary statistics and label mappings.
- `"/analysis"`: Presents analysis results with visualizations.

Prediction Function

User inputs are processed and passed to the prediction function:

```
@app.route("/predict", methods=['GET', 'POST'])
def predict():
    if request.method == 'POST':
        try:
            features = [
                float(request.form["Item_Identifier"]),
                float(request.form["Item_weight"]),
                float(request.form["Item_Fat_Content"]),
                float(request.form["Item_visibility"]),
                float(request.form["Item_Type"]),
                float(request.form["Item_MPR"]),
```

```
                float(request.form["Outlet_identifier"]),
                float(request.form["Outlet_established_year"]),
                float(request.form["Outlet_size"]),
                float(request.form["Outlet_location_type"]),
                float(request.form["Outlet_type"])
            ]

            predictions, filtered_items, sales_plot, heatmap, correlation_plot, strong_factors = prediction.predict_sales(features)

            return render_template(
                'predict.html',
                predictions=predictions,
                sales_plot=sales_plot,
                filtered_items=filtered_items.to_html(classes="table table-striped"),
                heatmap=heatmap,
                correlation_plot=correlation_plot,
                strong_factors=strong_factors
            )

        except Exception as e:
            return render_template('predict.html', error=f"Error: {str(e)}")

    return render_template('predict.html')
```

4.4.2 Frontend Development

The frontend is built using HTML and CSS, with Jinja2 templates for dynamic content rendering. Key templates include:

`index.html`

Serves as the homepage, providing an overview of the application.

`predict.html`

Contains a form for users to input features and displays prediction results along with visualizations.

`view_csv.html`

Displays tables showcasing samples from the datasets.

`dataset_insight.html`

Presents dataset summaries and label mappings.

`analysis.html`

Showcases analysis results with embedded images of plots.

Chapter 5

Installation and Usage Guide

This chapter provides step-by-step instructions for installing and running the BigMart Sales Prediction web application from the GitHub repository.

5.1 Prerequisites

Before proceeding, ensure that the following prerequisites are met:

- **Python Installed:** Verify that Python (version 3.6 or higher) is installed on your system. You can download it from the official Python website: https://www.python.org/downloads/
- **Git Installed:** Ensure that Git is installed to clone the repository. Download it from: https://git-scm.com/downloads

5.2 Cloning the Repository

To obtain the latest version of the application, clone the GitHub repository using the following command:

```
git clone https://github.com/TonyVoo/Project.git
```

5.3 Setting Up the Virtual Environment

It is recommended to use a virtual environment to manage the project's dependencies. Follow these steps:

1. **Navigate to the Project Directory:**

   ```
   cd Project
   ```

2. **Create a Virtual Environment:**

   ```
   python -m venv env
   ```

3. **Activate the Virtual Environment:**

   - **On Windows:**

```
.\env\Scripts\activate
```

- **On macOS and Linux:**

```
source env/bin/activate
```

5.4 Installing Dependencies

With the virtual environment activated, install the required dependencies using the `requirements.txt` file:

```
pip install -r requirements.txt
```

5.5 Running the Application

After installing the dependencies, start the Flask application with the following command:

```
python app.py
```

By default, the application will run on `http://127.0.0.1:5000/`.

5.6 Accessing the Application

Open a web browser and navigate to `http://127.0.0.1:5000/` to interact with the BigMart Sales Prediction web application.

5.7 Deactivating the Virtual Environment

After using the application, you can deactivate the virtual environment by executing:

```
deactivate
```

By following these steps, you will have the BigMart Sales Prediction web application up and running on your local machine.

Bibliography

[1] Nayana R, Chaithanya G, Meghana T, Narahari KS, Sushma M. *Predictive Analysis for Big Mart Sales using Machine Learning Algorithms*. International Journal of Engineering Research & Technology (IJERT), 2021. Available at: https://www.ijert.org/research/predictive-analysis-for-big-mart-sales-using-machine-learning-algorithms-IJERTCONV1 pdf

[2] Mallipeddi Vineeth Guptha, Gande Abhilash. *Prediction of Big Mart Sales using Machine Learning Algorithms*. Bachelor of Engineering Thesis, Sathyabama Institute of Science and Technology, 2022. Available at: https://sist.sathyabama.ac.in/sist_naac/documents/1.3.4/1822-b.e-cse-batchno-149.pdf

[3] Vidya Chitre, Shruti Mahishi, Sharvari Mhatre, Shreya Bhagwat. *Big Mart Sales Analysis*. International Journal of Innovative Technology and Exploring Engineering (IJITEE), 2022. Available at: https://www.researchgate.net/publication/360285933_Big_Mart_Sales_Analysis

[4] Ruiyun Kang. *Sales Prediction of Big Mart based on Linear Regression, Random Forest, and Gradient Boosting*. Advances in Economics Management and Political Sciences, 2023. Available at: https://www.researchgate.net/publication/373896945_Sales_Prediction_of_Big_Mart_based_on_Linear_Regression_Random_Forest_and_Gradient_Boosting

[5] Uzzivirus. *Big Mart Sales Prediction using Neural Networks*. Kaggle, 2021. Available at: https://www.kaggle.com/code/uzzivirus/big-mart-sales-prediction-using-neural-networks#Model-Prediction