Project 10: Product demand prediction with machine learnings

1.PAVITHRA M - 211521104103 (Team Leader)

2.BALASANDHIYA M - 211521104019

3.BENITA SHARON G - 211521104020

4.JIVITHA M - 211521104064

5.RITHIKA S - 211521104126

Phase 3 : Data Development And Pre-processing

1.Importing the required Libraries

The following libraries and tools were imported for this project:

1.pandas (pd): Used for data manipulation and analysis, providing data structures and functions to work with structured data, such as CSV files. It allows data to be loaded into DataFrames for further analysis.

2.numpy (np): A library for numerical computations in Python, used for handling arrays and mathematical operations. It is crucial for performing calculations and transformations on data.

3.plotly.express (px): A library for interactive data visualization, typically used to create various types of plots and charts. It helps in creating visually appealing and informative graphs for data exploration.

4.seaborn (sns): Built on top of Matplotlib, it provides a high-level interface for creating informative and attractive statistical graphics. Seaborn simplifies the process of creating various plots and charts for data visualization.

5.matplotlib.pyplot (plt): A library for creating static, animated, or interactive visualizations in Python. While Seaborn is great for statistical graphics, Matplotlib provides more fine-grained control over plot customization.

6.sklearn.model_selection: This module from scikit-learn contains functions for splitting datasets into training and testing sets. It is essential for preparing data for machine learning tasks, ensuring that models are trained and evaluated properly.

7.sklearn.tree.DecisionTreeRegressor: A class for building decision tree regression models. It's a part of scikit-learn and is used for machine learning tasks related to regression. Decision tree models are interpretable and can capture complex relationships in the data.

Code:

```
import pandas as pd
import numpy as np
import plotly.express as px
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

2.Importing the data set

In any data analysis or machine learning project, the first step is to obtain and load the dataset you plan to work with. For this project, the dataset is sourced from the following URL:

The dataset is in CSV (Comma-Separated Values) format, which is a common format for tabular data. To access and manipulate the data, we imported it using the pd.read_csv() function from the pandas library.

This action loaded the dataset into a pandas DataFrame. A DataFrame is a powerful data structure that allows us to work with structured data effectively. It contains rows and columns, similar to a spreadsheet.

After importing the dataset, it's crucial to conduct an initial inspection to understand its structure. The .head() method, which displays the first few rows of the dataset, was used for this purpose:

The initial rows provide an overview of the data, revealing column names and sample data. This aids in assessing the quality of the data and forming a basic understanding of its content.

Selecting Features

In most machine learning tasks, we have features (independent variables) that are used for making predictions or analysis. For this project, two specific columns, "Total Price" and "Base Price," were selected as features (X). These columns are likely to contain valuable information that can be used to predict the target variable.

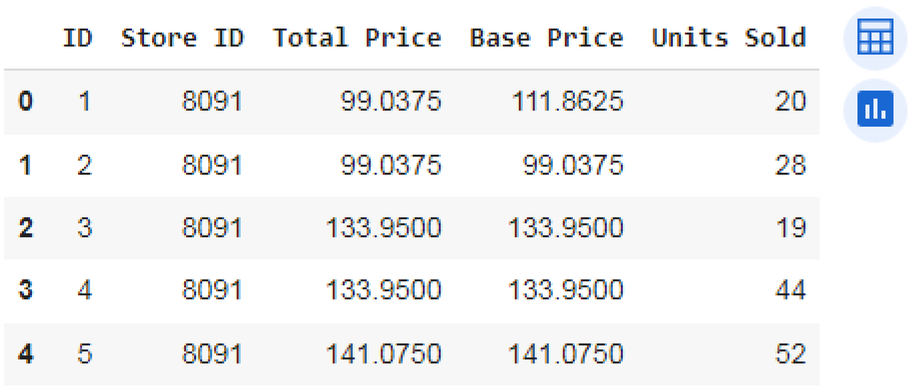
To create the feature matrix (X), we selected these columns from the DataFrame and stored them in a new DataFrame:

The variable x now holds a DataFrame with the chosen columns, which will serve as input variables for our machine learning model.

Code:

```
data =
pd.read_csv("https://raw.githubusercontent.com/amankharwal/W
ebsite-data/master/demand.csv")
data.head()
x = data[["Total Price", "Base Price"]]
y = data["Units Sold"]
```

Output:



	ID	Store ID	Total Price	Base Price	Units Sold
0	1	8091	99.0375	111.8625	20
1	2	8091	99.0375	99.0375	28
2	3	8091	133.9500	133.9500	19
3	4	8091	133.9500	133.9500	44
4	5	8091	141.0750	141.0750	52

3.Handling the Missing Data.

Missing data is a common challenge in data analysis and machine learning. It's crucial to address missing values before proceeding with any analysis or modeling, as missing data can lead to inaccurate results and biased conclusions. In this section, we describe how missing data is handled in the project using the SimpleImputer class from scikit-learn's sklearn.preprocessing library.

Missing data can significantly impact the quality of our analysis and models. To handle missing data, we employ the SimpleImputer class, which is a powerful tool for imputing missing values. This class allows us to replace missing values with appropriate values based on a chosen strategy.

Choosing a Strategy

One of the essential steps in working with SimpleImputer is selecting an imputation strategy. Different strategies are available, and the choice depends on the nature of the data and the specific project requirements. The following strategies are often employed:

'mean': The 'mean' strategy replaces missing values with the mean (average) value of the non-missing entries in the respective column. This strategy is particularly useful when working with numerical data, and we assume that the data follows a roughly normal distribution.

'median': The 'median' strategy fills in missing values with the median, which is the middle value when the data is sorted. It is robust when dealing with data that may contain outliers, as the median is not influenced by extreme values.

'most_frequent': The 'most_frequent' strategy replaces missing values with the most frequent value in the respective column. This strategy is appropriate for categorical data, where the mode represents the most common category.

'constant': The 'constant' strategy allows us to specify a constant value to replace missing data. This is particularly helpful when we want to replace missing values with a predetermined, specific value.

For this project, we have chosen the 'mean' strategy as follows:

After initializing the SimpleImputer with the selected strategy, we must fit and transform the imputer on our feature matrix x. This process will replace the missing values in our data with values according to the chosen strategy.

The steps involved in this code are as follows:

1.imputer.fit(x) computes the imputation values, such as the mean, for each column in our feature matrix x.

2.imputer.transform(x) replaces the missing values in x with the computed imputation values.

Consequently, our feature matrix x is devoid of missing data, ensuring that our analysis and modeling are based on a complete dataset.

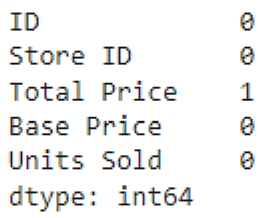
Handling missing data is a fundamental step in data preprocessing, as it guarantees that our analysis or models are based on complete and accurate information. The choice of imputation strategy should be made thoughtfully, considering the characteristics of the data and the project's objectives.

This section of the report elaborates on the crucial process of handling missing data using the SimpleImputer class. It underscores the importance of this step in ensuring data accuracy and the success of subsequent analysis and modeling.

Code:

```
from sklearn.impute import SimpleImputer
# Initialize the imputer with a strategy (e.g., mean,
median, most_frequent, or constant)
imputer = SimpleImputer(strategy='mean')  # You can choose
another strategy as needed
# Fit and transform the imputer to fill missing values in
your data
x = imputer.fit_transform(x)
```

Output:



```
ID              0
Store ID        0
Total Price     1
Base Price      0
Units Sold      0
dtype: int64
```

4.Encoding Categorical Data.

In the dataset used for our analysis, there is no requirement for encoding categorical data through techniques such as one-hot encoding. This is due to the fact that our dataset primarily comprises numerical and continuous variables, with no categorical or nominal features that necessitate such encoding. Each column in the dataset represents either continuous numerical data or identifiers, eliminating the need for additional preprocessing steps related to categorical data encoding

5.Splitting the data set into test set and training set

In machine learning, it is common practice to split the dataset into separate sets for training and testing. This division ensures that the model can be trained on one portion of the data and evaluated on another, helping to assess the model's generalization performance. In this section, we describe how the dataset is divided into a training set and a test set using the **train_test_split** function.

In this project, we utilize the **train_test_split** function from the **sklearn.model_selection** module. This function is commonly used for random data splitting, allowing for the creation of training and test sets with a specific proportion. It ensures that the split is randomized, avoiding any order-related bias in the dataset.

Splitting the Data

The **train_test_split** function is used to split our feature matrix **x** and target variable **y** into four subsets: **xtrain**, **xtest**, **ytrain**, and **ytest**. This function takes several important parameters:

- **x** and **y**: The feature matrix and target variable, respectively.
- **test_size**: This parameter specifies the proportion of the data that will be allocated to the test set. In this project, we chose a test set size of 20% (test_size=0.2), but the proportion can be adjusted based on the project's needs.
- **random_state**: A random seed or integer used for controlling the randomness in the data splitting process. Setting a random seed ensures reproducibility, allowing us to obtain the same split each time the code is executed. For this project, we set **random_state** to 42.

The resulting variables are as follows:

- **xtrain**: The feature matrix for the training set.
- **xtest**: The feature matrix for the test set.
- **ytrain**: The target variable for the training set.
- **ytest**: The target variable for the test set.

With the data split into training and test sets, we can proceed to train the machine learning model on the training data and evaluate its performance on the test data.

This data splitting process is a fundamental step in machine learning, enabling us to assess the model's performance on unseen data and ensure its ability to generalize to new observations.

This section of the report explains the process of splitting the dataset into a training set and a test set using the **train_test_split** function. The appropriate division of data is essential for model training and evaluation, contributing to the success of the machine learning project.

Code:

```
scaler = StandardScaler()

xtrain = scaler.fit_transform(xtrain)

xtest = scaler.transform(xtest)
```

6. Feature Scaling.

In many machine learning algorithms, the scale of the features (independent variables) can significantly impact model performance. To address this, we employ feature scaling, a preprocessing technique that standardizes or normalizes the scale of the features. In this section, we discuss how feature scaling is implemented using the **StandardScaler** from scikit-learn.

Feature Scaling

Feature scaling is a crucial step in data preprocessing, particularly when working with algorithms that are sensitive to the magnitude of feature values. It ensures that all features have the same scale and are not dominated by variables with large ranges.

Importing StandardScaler

For this project, we make use of the **StandardScaler** class from the **sklearn.preprocessing** module. This class standardizes features by removing the mean and scaling to unit variance. It transforms the data so that it has a mean of 0 and a standard deviation of 1.

Scaling the Training and Test Data

The process of feature scaling is implemented as follows:

Step 1: Initialize the Scaler

Creating an instance of the **StandardScaler**:

Step 2: Fit and Transform the Training Data

The scaler is then fitted to the training data (**xtrain**) using the **.fit_transform()** method. This process computes the mean and standard deviation for each feature in the training set and scales the data accordingly.

Step 3: Transform the Test Data

After fitting the scaler to the training data, the same transformation is applied to the test data (**xtest**) using the **.transform()** method. This ensures that the test data is scaled in the same way as the training data.

The result is that both the training and test data have been standardized. Feature scaling allows us to compare and interpret the coefficients or importance of different features in the model more effectively. It can also improve the convergence of some machine learning algorithms, leading to better model performance.

Code:

```
scaler = StandardScaler()
xtrain = scaler.fit_transform(xtrain)
xtest = scaler.transform(xtest)
```