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 5 : Documentation

Problem Definition

The core problem we aim to address is predicting product demand based on historical sales data and related features. Accurate demand prediction is crucial for efficient inventory management and planning. By leveraging machine learning techniques, we intend to build a model that forecasts future product demand, allowing businesses to optimize their operations.

Objective

Our primary objective is to develop a robust predictive model that accurately forecasts product demand. The model will utilize features such as historical sales, pricing information, and store details to generate predictions. By achieving accurate demand forecasts, we aim to assist businesses in making informed decisions regarding inventory, pricing strategies, and resource allocation.

This documentation will elaborate on the steps planned to enhance the product demand prediction model. The goal is to improve accuracy and efficiency in forecasting product demand by utilizing innovative methodologies and leveraging the provided dataset's columns, including ID, Store ID, Base Price, Total Price, and Unit Sold.

Dataset Overview

The dataset used for this project is sourced from Kaggle, a popular platform for sharing and discovering datasets related to various domains. The dataset titled "Product Demand Prediction with Machine Learning" provides essential information necessary for predicting product demand, including data columns such as ID, Store ID, Base Price, Total Price, and Unit Sold.

- **Dataset Title**: Product Demand Prediction with Machine Learning
- **Source**: Kaggle
- **Dataset Link**: https://www.kaggle.com/datasets/chakradharmattapalli/product-demand-prediction-with-machine-learning

The dataset we are working with contains crucial information related to product demand. The provided columns are as follows:

Details about columns (columns which you gonna use)

1. **Store ID:** This column can help you differentiate and analyze product demand by store location. It's useful if you have multiple stores and want to understand demand variations across different locations.
2. **Total Price:** The total price of products sold. This can be a valuable feature as it can influence product demand. Seasonal variations in prices can be important for forecasting.

3. **Base Price:** The base price of products. Base price fluctuations can affect demand, and including this information can help identify price-demand relationships.
4. **Units Sold:** This is your target variable, representing the number of units of a product sold during a specific time period.

This dataset serves as the foundation for our product demand prediction model, enabling us to analyze historical sales data and forecast future demand accurately.

DESIGN THINKING :

Design Thinking is a methodology that provides a solution-based approach to solving problems. It involves understanding human needs, challenging assumptions, re-framing problems, creating innovative solutions, and testing those solutions. For our project, this approach is crucial as it guides us through the problem-solving process, ensuring a well-structured and effective solution.

Objective and Approach

- **Objective:** The primary objective is to implement a structured design thinking approach, ensuring a comprehensive understanding and solution to the problem.

- **Approach:** We will meticulously follow each step of the design thinking approach, aiming for a deep understanding of the problem and a well￾structured solution. Regular evaluations and reviews will be conducted to maintain the integrity and effectiveness of our approach.

Data Collection

The foundation of building an accurate prediction model lies in the data we collect. Gathering comprehensive historical sales data and external influencing factors is crucial. The dataset needs to be rich and diverse, incorporating various parameters that could influence product demand. External factors such as marketing campaigns, holidays, and economic indicators play a significant role in demand prediction.

Data Preprocessing

Data preprocessing is a critical step in preparing the collected data for analysis. The main objectives are to handle missing values, convert categorical features into numerical representations, and normalize numerical data to prevent biases during model training. Preprocessing ensures the data is in a suitable format for accurate model training and prediction.

Objectives and Approach

- **Objective**: To ensure the data is in a suitable format for accurate model training and prediction by preprocessing it effectively.

- **Approach:** We will employ various techniques like imputation for missing values, one-hot encoding for categorical features, and scaling for numerical data. Additionally, we will design features that encapsulate temporal patterns and other relevant information, optimizing the dataset for model training.

Feature Engineering

Feature engineering is a crucial step to enhance the dataset by creating additional features that capture seasonal patterns, trends, and external influences on product demand. These features enrich the dataset, providing valuable insights for the machine learning model.

Model Selection

The selection of an appropriate regression algorithm is paramount for demand forecasting. Linear Regression, Random Forest, XGBoost, and other regression algorithms will be evaluated to determine the most suitable model based on the dataset and prediction requirements.

Objectives and Approach

- **Objective:** To enhance the dataset and choose the most suitable regression algorithm for accurate demand prediction.
- **Approach:** We will employ domain knowledge and experimentation to engineer features that enhance the dataset's predictive power.

Subsequently, we will evaluate multiple regression algorithms and select the one that best fits the project's objectives.

Model Training

This step involves splitting the dataset into training and testing sets. The selected machine learning model is then trained using the training set, allowing it to learn the patterns and relationships within the data.

Evaluation

Evaluation is critical to measure the model's performance accurately. Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and other regression metrics will be utilized to assess the model's ability to predict product demand.

Objectives and Approach

- **Objective:** To train the selected machine learning model and evaluate its predictive performance.
- **Approach:** We will use a portion of the dataset to train the model, fine�tuning its parameters to achieve the best results. Subsequently, we will

evaluate the model's performance using appropriate regression metrics

to ensure its accuracy and effectiveness.

Installing python

This section outlines the process of downloading and installing Python 3.12.0 on the [Your Operating System] system. Python 3.12.0 is the latest release of the Python programming language, offering numerous enhancements and features.

The first step in the installation process is to download Python 3.12.0 from the official Python website.

- Open your web browser and navigate to the official Python website at https://www.python.org/downloads/.

Selecting the Python 3.12.0 Version

- On the website, you will notice that Python 3.12.0 is the latest stable version. You should automatically see Python 3.12.0 as the default version on the website.
- Click on the "Download" button to initiate the download of the Python 3.12.0 installer appropriate for your operating system.
- The installer file will be saved to your system. The filename typically follows the format "python-3.12.0.exe" for Windows.

Locate the downloaded installer file and double-click it to run the installation.

Adding Python 3.12.0 to PATH

- During the installation, it is recommended to select the option "Add Python 3.12.0 to PATH." This addition makes Python easily accessible from the command line.
- Click "Install" to start the installation process.

Completing Installation

- The installer will copy Python 3.12.0 files to your system. Upon successful installation, a confirmation screen will appear, indicating "Setup was successful."

Verifying the Installation

To ensure the successful installation of Python 3.12.0, open a command prompt or terminal and enter the following command:

python --version

The displayed output should indicate the installed Python 3.12.0 version.

Python 3.12.0

Python 3.12.0 has been successfully downloaded and installed on your system. You have followed the steps to access the official Python website, select the Python 3.12.0 version, and complete the installation process.

Details of libraries to be used and way to download

For this project, we will utilize several Python libraries for data analysis, machine learning, and time series forecasting. Below are the libraries used:

- **Pandas:** Used for data manipulation and analysis.
- **NumPy:** Provides support for numerical operations and array manipulation.
- **Scikit-Learn:** Used for machine learning tasks.
- **Matplotlib:** A popular library for data visualization.
- **Seaborn:** Another visualization library for enhancing data presentation.

```
!pip install pandas
!pip install numpy
!pip install scikit-learn
!pip install matplotlib
!pip install seaborn
```

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.

```python
import pandas as pd
```

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.

```python
import numpy as np
```

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.

```python
 import plotly.express as px
```

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.

```python
 import seaborn as sns
```

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.

```python
import matplotlib.pyplot as plt
```

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.

```python
from sklearn.model_selection import train_test_split
```

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
```

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"]]
```

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:

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

9.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

data = data.dropna()

# Fit and transform the imputer to fill missing values in
```

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

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:

```
xtrain, xtest, ytrain, ytest = train_test_split(x, y,

                                                test_size=0.2,

                                                random_state=42)

from sklearn.tree import DecisionTreeRegressor
```

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 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)
```

Model

In this analysis, we have employed a Decision Tree Regressor, a machine learning algorithm used for regression tasks, to model the relationships between our preprocessed data variables. This algorithm is particularly well-suited for capturing non-linear patterns and relationships in the data, making it a valuable tool for our analysis

Why Not ARIMA or Prophet?

The exclusion of ARIMA or Prophet for time series forecasting may raise questions. These methods are widely recognized for their utility in such tasks, but the decision to not utilize them for this specific project is rooted in the characteristics of our dataset. Notably, our dataset lacks a pivotal element for time series forecasting: time.

ARIMA and Prophet excel when applied to time series data, where observations are systematically recorded at regular intervals. These methods inherently account for the chronological order of data points, enabling them to make informed predictions. Given the absence of temporal data in our dataset, it is more prudent to adopt a regression-based model like Random Forest for predicting product demand based on the available features.

What metrics used for the accuracy check

To evaluate model performance, we will undertake these steps:

- Splitting the dataset into training and validation sets, allocating 80% for training and 20% for validation.

- Utilizing Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) as evaluation metrics.

Model evaluation and comparison will be vital in this phase. By comparing the performance of ARIMA, Prophet, LSTM, and XGBoost models, we aim to identify the most accurate approach for demand prediction. This comparison will guide us in selecting the best-performing model for fine-tuning and optimization.

CODE

```
xtrain,xtest,ytrain,ytest=train_test_split(x,y,test_size=0.2,
random_state=42)

model = DecisionTreeRegressor()

model.fit(xtrain, ytrain)

DecisionTreeRegressor()
```

Data Visualization

This section explores the visualizations created before preprocessing:

1.Scatter Plot:Total price vs Units sold

The Preprocessed Total Price vs. Units Sold Scatter Plot explores the relationship between the preprocessed total price and units sold. It reveals the correlation after data preprocessing and scaling, offering a clearer perspective on their connection.

CODE

```
fig = px.scatter(data, x="Units Sold", y="Total Price",

                    size='Units Sold')
```

```
fig.show()
```

2.Histogram: Total price vs Units sold

This histogram illustrates the relationship between "Total Price" and "Units Sold," providing insights into the distribution of this relationship.

CODE

```
fig=plt.hist(data)

plt.xlabel("Units Sold")

plt.ylabel("Total Price")

plt.show()
```

2.Histogram: Total price vs Units sold

This histogram illustrates the relationship between "Total Price" and "Units Sold," providing insights into the distribution of this relationship.

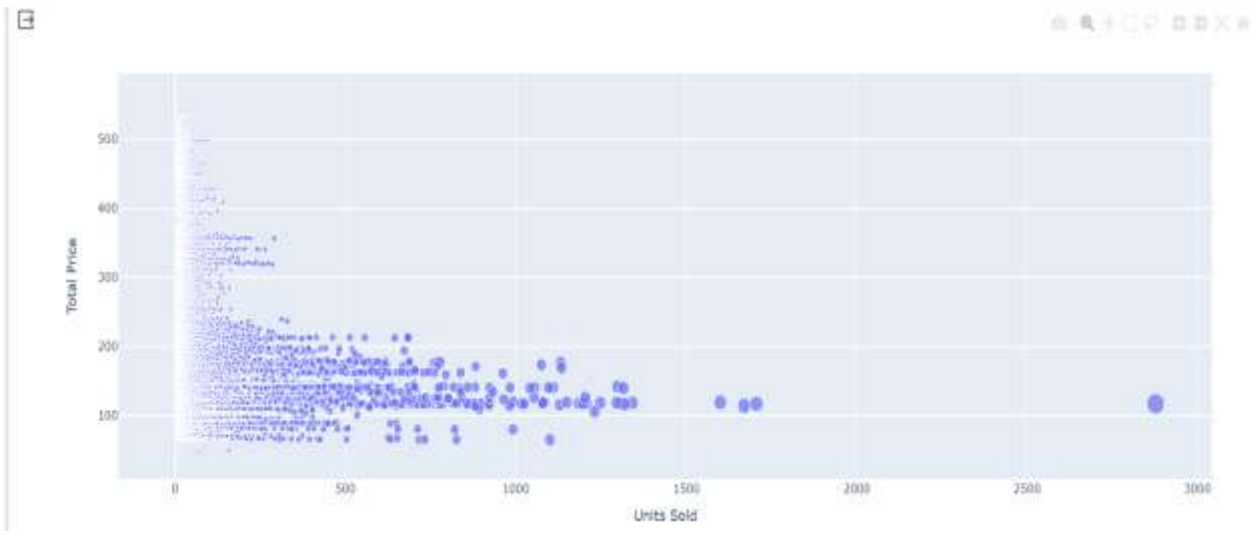
CODE

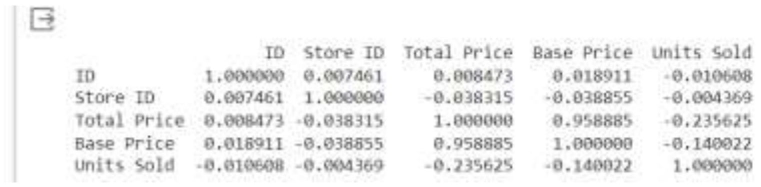
```
#prediction after training

y_pred = model.predict(xtest)

plt.hist(y_pred, bins=20, color='blue')

plt.xlabel('Units sold')

plt.ylabel('Base price')

plt.tight_layout()

plt.show()
```

MODEL

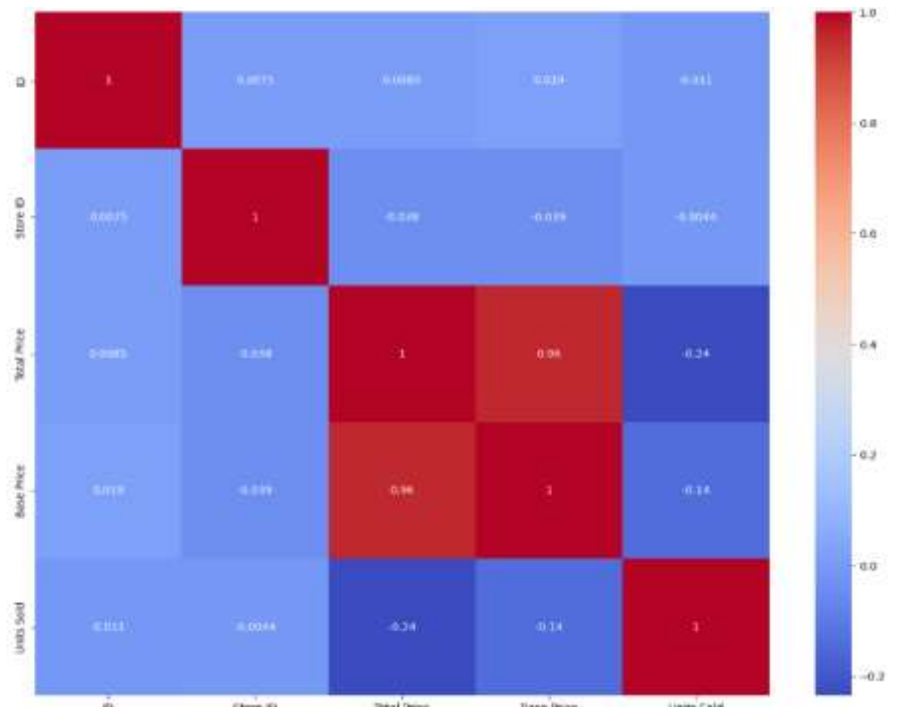
```python
import pandas as pd
import numpy as np
import plotly.express as px
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor

data = pd.read_csv("https://raw.githubusercontent.com/amankharwal/Website-
data/master/demand.csv")
data.head()
data.isnull().sum()
data = data.dropna()
fig = px.scatter(data, x="Units Sold", y="Total Price",
                 size='Units Sold')
fig.show()
print(data.corr())

correlations = data.corr(method='pearson')
plt.figure(figsize=(15, 12))
sns.heatmap(correlations, cmap="coolwarm", annot=True)
plt.show()

x = data[["Total Price", "Base Price"]]
y = data["Units Sold"]
xtrain, xtest, ytrain, ytest = train_test_split(x, y,
                                                test_size=0.2,
                                                random_state=42)
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor()
model.fit(xtrain, ytrain)

#features = [["Total Price", "Base Price"]]
features = np.array([[133.00, 140.00]])
model.predict(features)

#prediction after training
y_pred = model.predict(xtest)
fig=plt.hist(data)
plt.xlabel("Units Sold")
plt.ylabel("Total Price")
plt.show()
plt.hist(y_pred, bins=20, color='blue')
plt.xlabel('Units sold')
plt.ylabel('Base price')
plt.tight_layout()
plt.show()
```

OUTPUT:

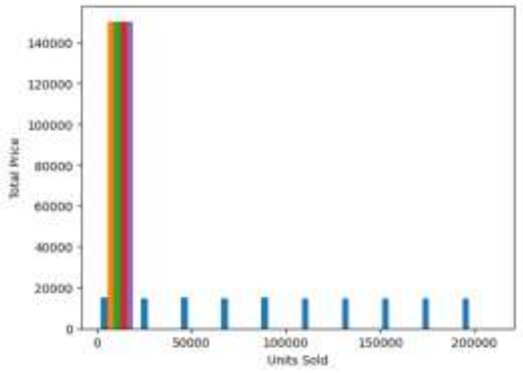


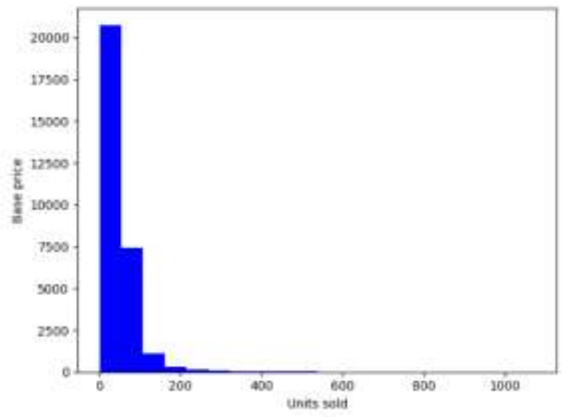


```
                    ID  Store ID  Total Price  Base Price  Units Sold
ID            1.000000  0.007461     0.008473    0.018911   -0.010608
Store ID      0.007461  1.000000    -0.038315   -0.038855   -0.004369
Total Price   0.008473 -0.038315     1.000000    0.958885   -0.235625
Base Price    0.018911 -0.038855     0.958885    1.000000   -0.140022
Units Sold   -0.010608 -0.004369    -0.235625   -0.140022    1.000000
```

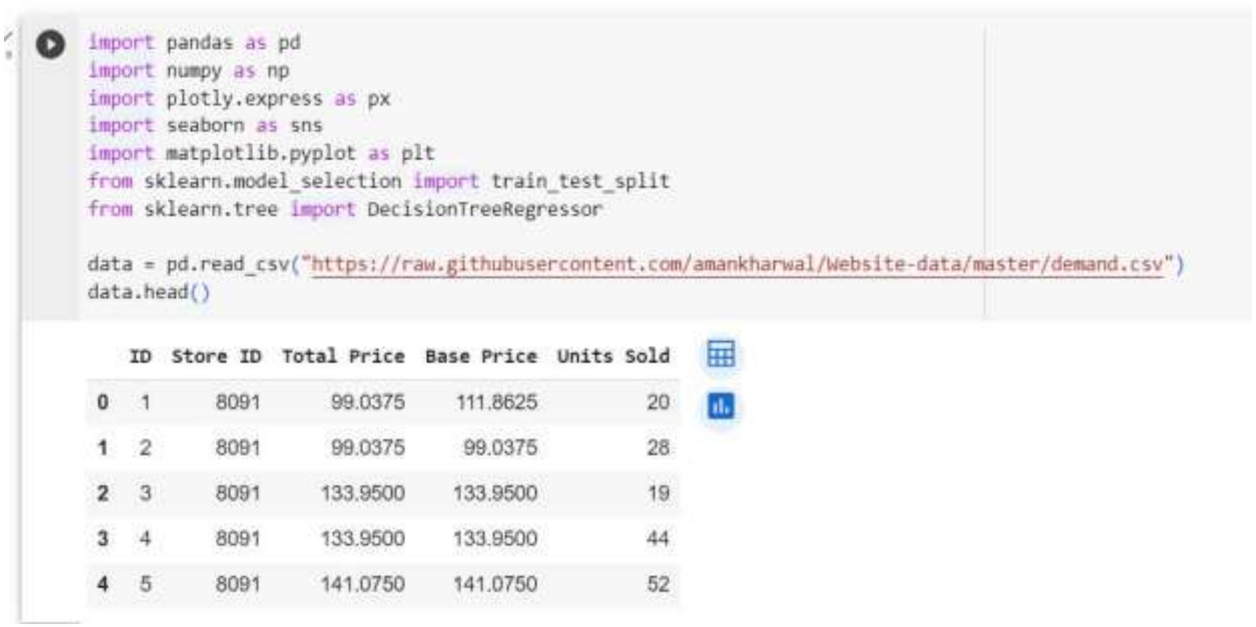


```
X does not have valid feature names, but DecisionTreeRegressor was fitted with feature names
```



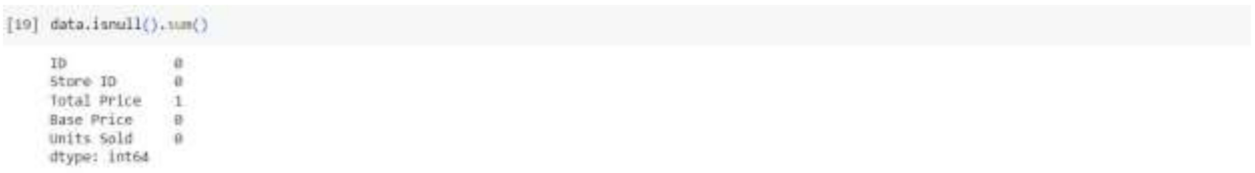


EXECUTING:



```python
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
from sklearn.tree import DecisionTreeRegressor

data = pd.read_csv("https://raw.githubusercontent.com/amankharwal/Website-data/master/demand.csv")
data.head()
```

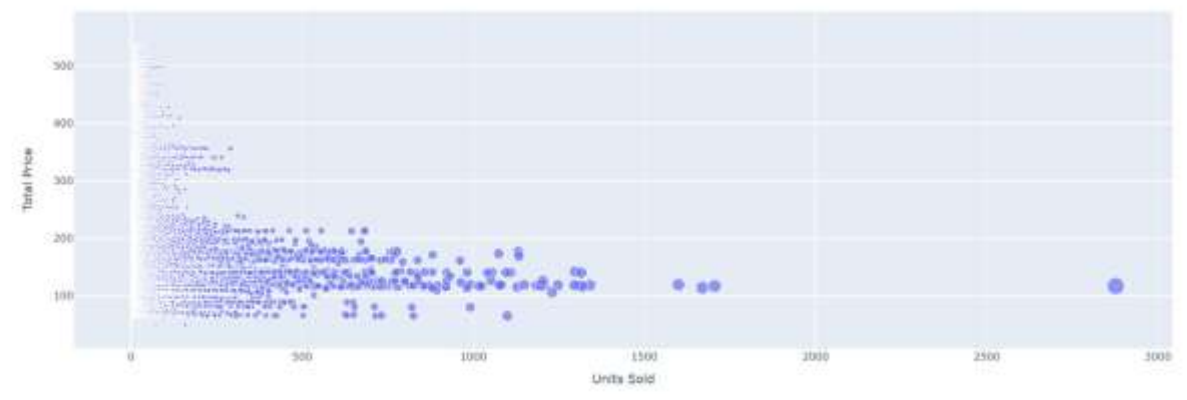
	ID	Store ID	Total Price	Base Price	Units Sold
0	1	8091	99.0375	111.8625	20
1	2	8091	99.0375	99.0375	28
2	3	8091	133.9500	133.9500	19
3	4	8091	133.9500	133.9500	44
4	5	8091	141.0750	141.0750	52

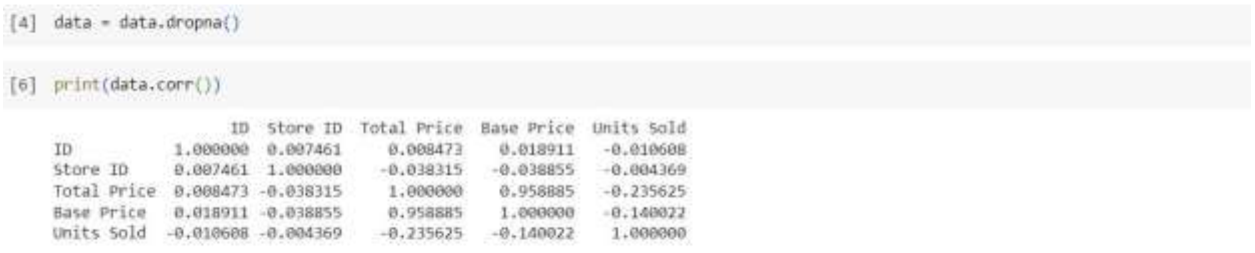


```
[19] data.isnull().sum()
```

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

```python
fig = px.scatter(data, x="Units Sold", y="Total Price",
                 size='Units Sold')
fig.show()
```

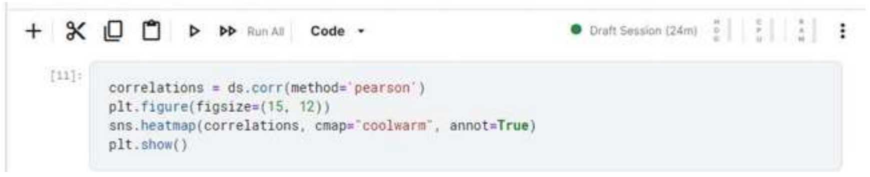




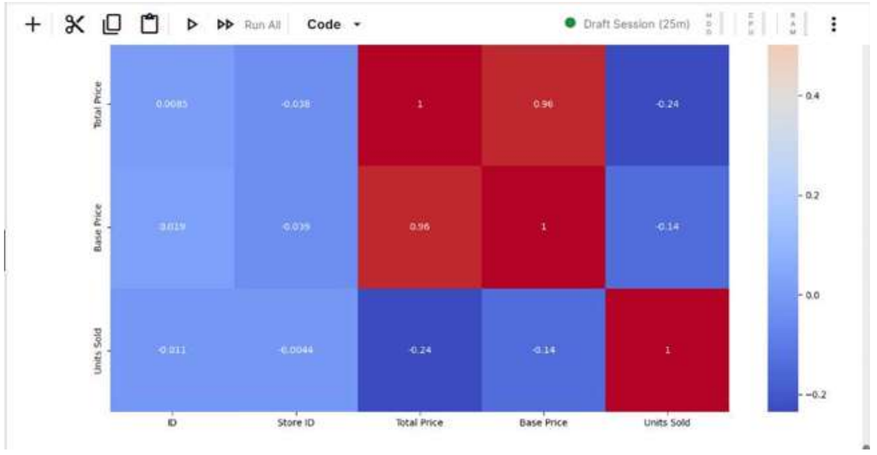
```
[4] data = data.dropna()
```

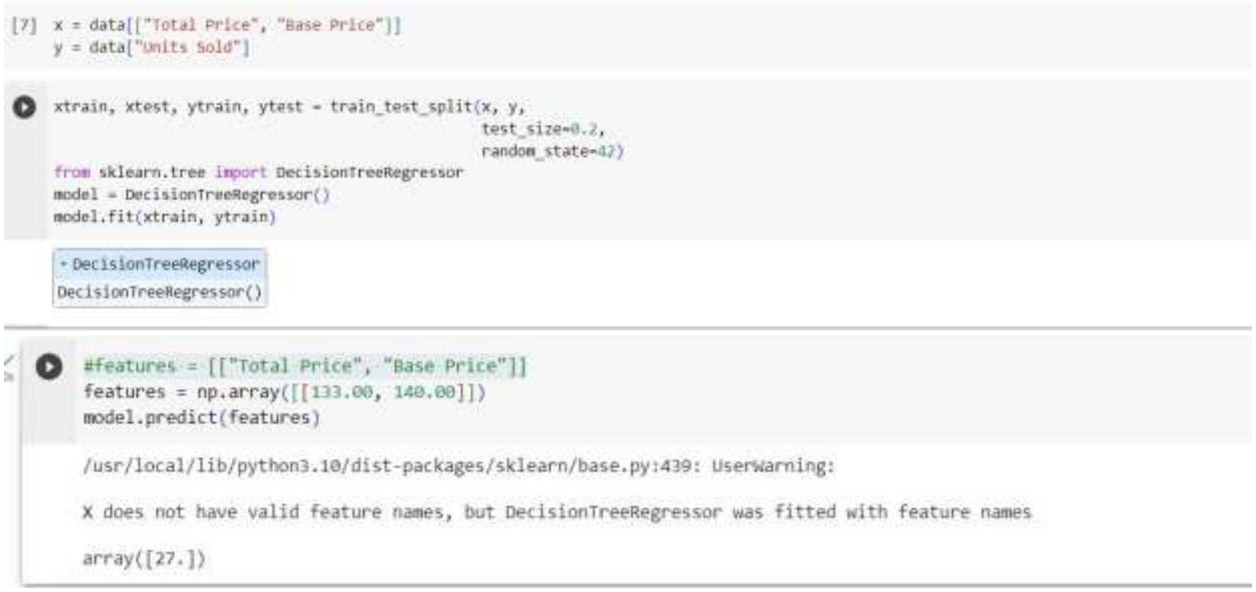
```
[6] print(data.corr())
```

```
                   ID  Store ID  Total Price  Base Price  Units Sold
ID           1.000000  0.007461     0.008473    0.018911   -0.010608
Store ID     0.007461  1.000000    -0.038315   -0.038855   -0.004369
Total Price  0.008473 -0.038315     1.000000    0.958885   -0.235625
Base Price   0.018911 -0.038855     0.958885    1.000000   -0.140022
Units Sold  -0.010608 -0.004369    -0.235625   -0.140022    1.000000
```



```
[11]: correlations = ds.corr(method='pearson')
      plt.figure(figsize=(15, 12))
      sns.heatmap(correlations, cmap="coolwarm", annot=True)
      plt.show()
```





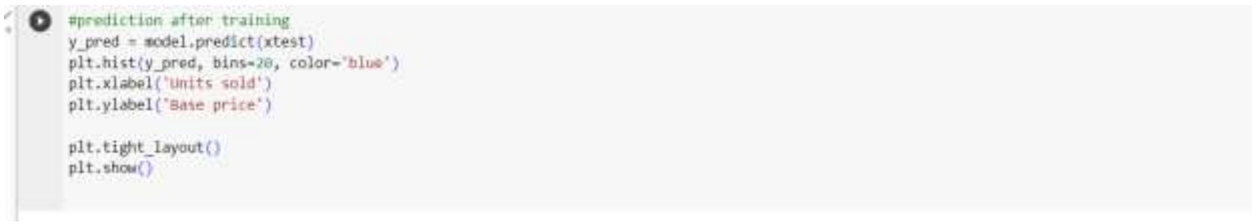
```
[7] x = data[["Total Price", "Base Price"]]
    y = data["Units Sold"]
```

```
xtrain, xtest, ytrain, ytest = train_test_split(x, y,
                                                test_size=0.2,
                                                random_state=42)
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor()
model.fit(xtrain, ytrain)
```

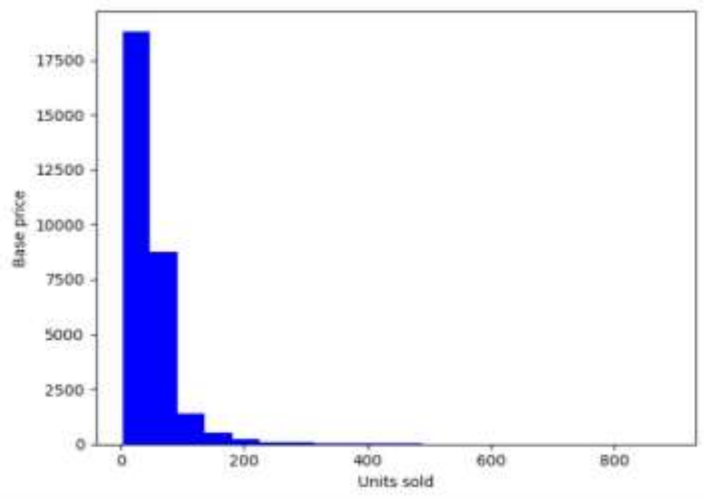
```
▸ DecisionTreeRegressor
DecisionTreeRegressor()
```

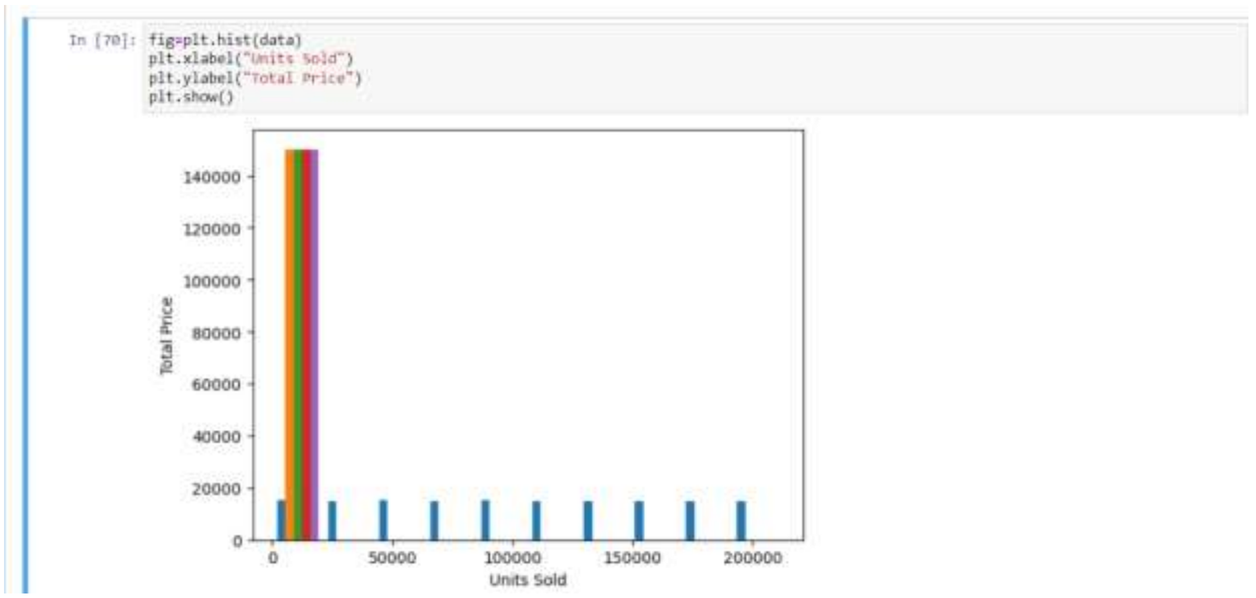
```
#features = [["Total Price", "Base Price"]]
features = np.array([[133.00, 140.00]])
model.predict(features)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning:

X does not have valid feature names, but DecisionTreeRegressor was fitted with feature names

array([27.])
```



```
#prediction after training
y_pred = model.predict(xtest)
plt.hist(y_pred, bins=20, color='blue')
plt.xlabel('Units sold')
plt.ylabel('Base price')

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





```
In [70]: fig=plt.hist(data)
         plt.xlabel("Units Sold")
         plt.ylabel("Total Price")
         plt.show()
```

CONCLUSION

Predicting product demand using machine learning involves leveraging historical data to build predictive models that can forecast future demand. This process typically encompasses data collection, preprocessing, feature engineering, model selection, training, and evaluation. By employing algorithms from libraries like scikit-learn and employing data visualization tools such as Matplotlib and Seaborn, businesses can develop accurate and reliable models that help optimize inventory management, production planning, and supply chain operations. This can lead to more efficient resource allocation and cost savings while ensuring that products are available when customers want them. The final closure for product demand prediction through machine learning is an exciting step towards data-driven decision-making and enhanced operational efficiency for businesses in various industries.