1.)

import pandas as pd

* **Purpose:** Import the Pandas library.
* **Explanation:** This line allows you to use the Pandas library in your code. Pandas is commonly used for data manipulation and analysis, providing data structures like Data Frames.

import numpy as np

**Purpose:** Import the NumPy library.

**Explanation:** This line allows you to use the NumPy library in your code. NumPy is used for numerical computing, especially for handling large, multi-dimensional arrays and matrices.

import matplotlib.pyplot as plt

* **Purpose:** Import the Pyplot module from the Matplotlib library.
* **Explanation:** This line allows you to create various types of plots and visualizations using Matplotlib. The **plt** alias is a common convention.

import seaborn as sns

**Purpose:** Import the Seaborn library.

**Explanation:** Seaborn is a data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphic

import random

* **Purpose:** Import the random module.
* **Explanation:** This module provides functions for generating random numbers. However, in the provided code snippet, it seems that the **random** module is imported but not used. If it's not necessary for your script, you can consider removing it.

**2.**)

from sklearn.model\_selection import train\_test\_split

* **Purpose:** Import the **train\_test\_split** function from Scikit-learn.
* **Explanation:** This function is used to split datasets into training and testing sets. It helps in evaluating the performance of a machine learning model.

from sklearn.linear\_model import LinearRegression

* **Purpose:** Import the **LinearRegression** class from Scikit-learn.
* **Explanation:** This class is used to perform linear regression, a popular method for modeling the relationship between a dependent variable and one or more independent variables.

from sklearn.metrics import mean\_squared\_error

* **Purpose:** Import the **mean\_squared\_error** function from Scikit-learn.

**Explanation:** This function is commonly used to evaluate the performance of a regression model by calculating the mean squared error between predicted and actual values

from sklearn.preprocessing import MinMaxScaler, StandardScaler, MaxAbsScaler, Normalizer

**Purpose:** Import various scalers from Scikit-learn for data preprocessing.

**Explanation:** These scalers are used to scale or normalize input data before feeding it into a machine learning model. Different scalers have different normalization strategies.

3.)

data=pd.read\_csv('BigmartSales.csv')

* **Purpose:** Reads a CSV file named 'BigmartSales.csv' and loads its contents into a Pandas DataFrame.
* **Inference:** Assumes that the dataset is stored in a CSV file format, and the data is loaded into the variable **data**.

df=pd.DataFrame(data)

* **Purpose:** Converts the loaded data into a Pandas DataFrame named **df**.

**Inference:** The variable **df** is now a DataFrame containing the data from the CSV file

df.head()

* **Purpose:** Displays the first few rows of the DataFrame to get a quick overview of the data.
* **Inference:** The **head()** method is used to show the top rows of the DataFrame. This can help in understanding the structure and contents of the dataset

**4.**)

columns\_to\_drop=["Item\_Identifier","Outlet\_Identifier"]

* **Purpose:** Defines a list of column names ("Item\_Identifier" and "Outlet\_Identifier") to be dropped from the DataFrame.
* **Inference:** The columns specified in **columns\_to\_drop** are marked for removal from the DataFrame.

df=df.drop(columns\_to\_drop,axis=1)

* **Purpose:** Drops the specified columns from the DataFrame.
* **Inference:** The **drop()** method is used to remove columns specified in **columns\_to\_drop** from the DataFrame **df**. The **axis=1** argument indicates that columns are being dropped.

df.info()

* **Purpose:** Prints a concise summary of the DataFrame, including information about the data types and non-null values.

**Inference:** The **info()** method is called on the DataFrame to display information such as the number of non-null entries, data types of each column, and memory usage

5.)

* **Purpose:** Extracts the 'Item\_Outlet\_Sales' column from the DataFrame and assigns it to the variable **target\_labels**.
* **Inference:** This line isolates the target variable ('Item\_Outlet\_Sales') from the DataFrame, suggesting that you want to predict or analyze the values in this column.

In machine learning, the target variable is typically the variable that you want to predict or understand. It represents the output or dependent variable in your analysis or model.

Now, the **target\_labels** variable holds the values of the target variable, and you can use it for tasks such as training a machine learning model, performing statistical analysis, or any other relevant analysis that involves the target variable.

6.)

* **Purpose:** Replaces values in the 'Item\_Fat\_Content' column with numerical equivalents.
* **Inference:** This line uses the **replace** method to replace the values in the 'Item\_Fat\_Content' column. 'Low Fat' is replaced with '0', and 'Regular' is replaced with '1'. This type of transformation is commonly done to convert categorical data into a format suitable for certain machine learning algorithms that require numerical inputs.

After this line of code is executed, the 'Item\_Fat\_Content' column in the DataFrame **df** will have numerical values ('0' and '1') instead of the original categorical values ('Low Fat' and 'Regular'). This is often a necessary step in preparing categorical data for machine learning models.

7.)

* **Purpose:** Iterates through the specified columns, creates a mapping of unique values to ordinal values, and applies the mapping to the DataFrame.
* **Inference:** This loop processes each column specified in **ordinal\_encoding**. For each column, it creates a dictionary **list\_data\_ordinal** where the unique values are mapped to ordinal values starting from 1. It then uses the **map** function to replace the original values in the DataFrame with their corresponding ordinal values.

After running this code, the specified columns ('Item\_Type', 'Outlet\_Type', 'Outlet\_Location\_Type') in the DataFrame **df** will be ordinal-encoded, where each unique value is replaced with an ordinal value based on its position in the unique values list. The ordinal values start from 1 and increase incrementally. This transformation is helpful when certain machine learning algorithms require numerical representations for categorical data while preserving the ordinal relationship between categories.

8.)

**Purpose:** The purpose of the provided code is to perform ordinal encoding on categorical variables within a DataFrame. Ordinal encoding is used to represent categorical data with an inherent order as integers. This is commonly done when preparing data for machine learning models that require numerical input, as it allows preserving the ordinal relationships between categories.

**Inference:** The code sequentially encodes three categorical columns - "Item\_Type," "Outlet\_Type," and "Outlet\_Location\_Type" - in the DataFrame. It creates unique lists of categories for each column, assigns ordinal values to each unique category, and then maps these ordinal values back to the corresponding column in the DataFrame. The result is a modified DataFrame where these categorical columns are represented by their ordinal counterparts, potentially making the data suitable for certain machine learning algorithms that expect numerical input.

9.)

**Purpose:**

The purpose of the provided code is to handle missing values in the "Outlet\_Size" column of a DataFrame (**df**). Specifically, it imputes (fills in) the missing values with the mode (most frequent value) of the "Outlet\_Size" column. Imputation is a common data preprocessing step, and filling missing values with the mode is a strategy often used for categorical data.

**Inference:**

The line of code first calculates the mode value for the "Outlet\_Size" column using **df['Outlet\_Size'].mode()[0]**. Then, it replaces any missing values in the "Outlet\_Size" column with this mode value using the **fillna()** method with **inplace=True**. The **inplace=True** parameter modifies the original DataFrame without the need for reassignment.

This imputation strategy is reasonable when dealing with categorical data like "Outlet\_Size" because it ensures that missing values are replaced with the most frequent category, thereby maintaining the integrity of the existing distribution of values in that column. The subsequent commented-out **#print(df['Outlet\_Size'])** line suggests that there might have been a verification step to observe the changes in the "Outlet\_Size" column after imputation. If uncommented, this line would print the updated "Outlet\_Size" column with missing values replaced by the mode.

10.)

**Purpose:**

The purpose of the provided code is to identify and count the number of null (missing) values in each column of the DataFrame (**df**). This is a crucial step in data exploration and preprocessing to understand the extent of missing data in the dataset.

**Inference:**

The output of the code reveals the following information:

1. **Item\_Weight:** There are 1463 null values in the "Item\_Weight" column. This suggests that a significant portion of the "Item\_Weight" data is missing.
2. **Item\_Fat\_Content, Item\_Visibility, Item\_Type, Item\_MRP, Outlet\_Establishment\_Year, Outlet\_Size, Outlet\_Location\_Type, Outlet\_Type, Item\_Outlet\_Sales:** These columns have no null values, as indicated by the counts of 0.

The inference allows the data analyst or scientist to make informed decisions about how to handle missing values in the dataset. The presence of a substantial number of missing values in the "Item\_Weight" column may prompt further investigation into the reasons for the missing data and consideration of appropriate strategies for imputation or handling missing values during analysis or modeling

11.)

**Purpose:**

The purpose of the provided code is to handle missing values in the "Item\_Weight" column of the DataFrame (**df**). Specifically, it imputes (fills in) the missing values with the mode (most frequent value) of the "Item\_Weight" column. Imputation is a common data preprocessing step, and filling missing values with the mode is a strategy often used for numerical data.

**Inference:**

1. **item\_weight\_mode\_value:** The mode value of the "Item\_Weight" column is calculated using **df['Item\_Weight'].mode()[0]**.
2. **df['Item\_Weight'].fillna(item\_weight\_mode\_value, inplace=True):** This line of code replaces any missing values in the "Item\_Weight" column with the calculated mode value (**item\_weight\_mode\_value**). The **inplace=True** parameter modifies the original DataFrame without the need for reassignment.
3. **#print(df['Item\_Weight']):** This commented-out line suggests that there might have been a print statement to check the values of the "Item\_Weight" column after the imputation. If uncommented, it would display the updated "Item\_Weight" column with missing values imputed using the mode.

**12.**)

**Purpose:**

The purpose of the provided code is to display the first few rows of the DataFrame (**df**) using the **head()** method. This is a common operation to quickly inspect the structure and content of the dataset

The dataset (**df**) consists of various features such as item weight, fat content, visibility, type, maximum retail price, outlet establishment year, size, location type, outlet type, and item outlet sales. The data seems to include both numerical and categorical variables. Initial inspection of the first few rows indicates a diverse set of information, including product details and outlet-related information. Further exploration and preprocessing may be needed for analysis or machine learning tasks.

**13.**)

**Purpose:**

The purpose of the provided code is to create a boxplot for all numeric columns in the DataFrame (**df**). This visualization is useful for identifying the distribution, central tendency, and presence of outliers in the numerical data.

**Inference:**

1. **#numeric\_columns = df.select\_dtypes(include=['int', 'float']).columns:**
   * This line is commented out but suggests an attempt to select only numeric columns in the DataFrame. However, the specific code is not executed, so the variable **numeric\_columns** is not used in the subsequent code.
2. **df.boxplot(figsize=(20, 10)):**
   * This line generates a boxplot for all numeric columns in the DataFrame. The **figsize=(20, 10)** parameter specifies the size of the figure.
3. **plt.title("Boxplot of dataset Original data"):**
   * Sets the title of the boxplot to "Boxplot of dataset Original data."
4. **plt.show():**
   * Displays the boxplot.

14.)

**Purpose:**

The purpose of the provided code is to create a scatter plot for two variables from the DataFrame (**df**): "Item\_Type" on the x-axis and "Item\_Outlet\_Sales" on the y-axis. A scatter plot is a visualization technique used to examine the relationship between two continuous variables and identify patterns, trends, or potential correlations in the data.

**Inference:**

1. **plt.scatter(df['Item\_Type'], df['Item\_Outlet\_Sales']):**
   * This line creates a scatter plot with "Item\_Type" on the x-axis and "Item\_Outlet\_Sales" on the y-axis. Each point in the plot represents a combination of these two variables for individual observations in the dataset.
2. **plt.title('Scatter Plot of original DataFrame'):**
   * Sets the title of the scatter plot to "Scatter Plot of original DataFrame."
3. **plt.xlabel('Item\_Type') and plt.ylabel('Item\_Outlet\_Sales'):**
   * Labels the x-axis as "Item\_Type" and the y-axis as "Item\_Outlet\_Sales," providing context for the variables being plotted.
4. **plt.show():**
   * Displays the scatter plot.

15.)

**Purpose:**

The purpose of the provided code is to prepare the data for a machine learning model. It selects specific columns from the DataFrame (**df**) and separates them into two sets: **X** (independent variables or features) and **y** (dependent variable or target). This is a common step in supervised learning, where you have features (X) that are used to predict the target variable (y).

**Inference:**

1. **#numeric\_columns = df.select\_dtypes(include=['int', 'float']).columns:**
   * This line is commented out, suggesting an attempt to select only numeric columns in the DataFrame. However, this specific code is not executed in the provided snippet.
2. **X = df[['Item\_Weight', 'Item\_Visibility', 'Item\_Type', 'Item\_MRP', 'Outlet\_Location\_Type', 'Outlet\_Type']]:**
   * This line creates a DataFrame **X** that includes specific columns: 'Item\_Weight', 'Item\_Visibility', 'Item\_Type', 'Item\_MRP', 'Outlet\_Location\_Type', and 'Outlet\_Type'. These columns are presumably selected as features for the machine learning model.
3. **y = df[['Item\_Outlet\_Sales']]:**
   * This line creates a DataFrame **y** that includes the target variable 'Item\_Outlet\_Sales.' This variable is what the model will aim to predict based on the features in **X**.
4. **print("\nX Variables :",X.columns):**
   * Prints the column names of the features in **X**.
5. **print("\ny variable :",y.columns):**
   * Prints the column name of the target variable in **y**.

16.)

**Purpose:**

The purpose of the provided code is to perform the following tasks:

1. Split the dataset into training and testing sets using the train\_test\_split function.
2. Apply Linear Regression on the training set.
3. Make predictions on the test set using the trained model.
4. Calculate the Root Mean Squared Error (RMSE) as an evaluation metric for the regression model.

**Inference:**

1. **Splitting the dataset:**
   * The dataset is split into training and testing sets using the **train\_test\_split** function. 80% of the data is used for training (**X\_train** and **y\_train**), and 20% is reserved for testing (**X\_test** and **y\_test**).
2. **Model Training:**
   * Linear Regression is applied to the training set using **LinearRegression()** from scikit-learn (**model = LinearRegression()**). The model is fitted using **model.fit(X\_train, y\_train)**.
3. **Features of X\_train, y\_train:**
   * The features used in training the model are printed, providing transparency about the features considered in the linear regression model.
4. **Making Predictions:**
   * Predictions (**y\_pred**) are made on the test set using the trained linear regression model.
5. **Root Mean Squared Error (RMSE):**
   * The RMSE is calculated to evaluate the performance of the linear regression model on the test set. The RMSE is a measure of the average magnitude of the errors between predicted and actual values.
6. **Displaying Results:**
   * The shape of the training and testing sets is printed, showing the number of rows and columns in each set.
   * The name of the linear regression model is printed (**Model Name: LinearRegression()**).
   * The features used in training the model are listed.

17.)

**Purpose:**

The purpose of the provided code is to standardize the features using the StandardScaler and then split the dataset into training and testing sets. Standardization is a preprocessing step that transforms the features to have a mean of 0 and a standard deviation of 1, which can be beneficial for certain machine learning algorithms, especially those that rely on distance metrics.

**Inference:**

1. **Applying StandardScaler:**
   * The StandardScaler from scikit-learn is applied to the feature matrix **X** using **scaler.fit\_transform(X)**. This standardizes the features, making them have a mean of 0 and a standard deviation of 1.
2. **Splitting the Dataset:**
   * The standardized features (**X\_standardized**) are then used to split the dataset into training and testing sets using **train\_test\_split**. 80% of the data is used for training (**X\_train** and **y\_train**), and 20% is reserved for testing (**X\_test** and **y\_test**).

**18.)**

**Purpose:**

The purpose of the provided code is to create and display boxplots for all numeric fields in the DataFrame (**df**). Boxplots are a visualization technique used to summarize the distribution of numerical data and identify potential outliers.

**Inference:**

1. **plt.figure(figsize=(20, 3)):**
   * This line sets the figure size for the boxplots, specifying a larger width (20) and a shorter height (3).
2. **sns.boxplot(data=df, orient='v'):**
   * The **sns.boxplot** function from the Seaborn library is used to create vertical boxplots for all numeric fields in the DataFrame. Each boxplot represents the distribution of values for a specific numeric column.
3. **plt.title('Boxplots for All Numeric Fields'):**
   * Sets the title of the boxplot to "Boxplots for All Numeric Fields."
4. **plt.show():**
   * Displays the boxplots.

**19)**

**Apply Linear Regression and calculate RMSE value**

**Steps:**

Import Libraries:

The code begins by importing necessary libraries, including LinearRegression from scikit-learn (sklearn), and metrics for evaluation (mean\_squared\_error and numpy).

Define and Fit the Model:

A Linear Regression model is instantiated using LinearRegression() and trained on the training data (X\_train and y\_train) using the fit method.

Make Predictions:

The trained model is used to predict the target variable (y\_pred) on the test set (X\_test).

Calculate RMSE:

The Root Mean Squared Error (RMSE) is computed using the mean\_squared\_error function from scikit-learn and numpy to take the square root. The result is stored in the variable standard\_scaler\_rmse.

Print RMSE:

The calculated RMSE value is printed to the console.

**Inference:**

Linear Regression Model:

The Linear Regression model has been successfully trained on the provided training data.

Predictions:

Predictions have been made on the test set, and the results are stored in y\_pred.

RMSE Calculation:

The RMSE is a measure of the average deviation between the predicted and actual values. A lower RMSE indicates better model performance.

Results:

The RMSE value for the Linear Regression model on the test set is printed to the console.

20) **Apply MinMaxScaler, split the dataset into train and test(20%), apply LinearRegression and calculate RMSE**

**Steps:**

Import Libraries:

Necessary libraries are imported, including MinMaxScaler, LinearRegression, and metrics for evaluation (mean\_squared\_error and numpy). Also, train\_test\_split is used to split the dataset.

Apply MinMaxScaler:

The MinMaxScaler is applied to normalize the features of the dataset, and the result is stored in X\_scaled.

Split Dataset:

The dataset is split into training and testing sets using train\_test\_split. 80% of the data is used for training, and 20% is reserved for testing.

Apply Linear Regression:

A Linear Regression model is instantiated and trained on the scaled training data (X\_train and y\_train).

Make Predictions:

The trained model is used to predict the target variable (y\_pred) on the test set (X\_test).

Calculate RMSE:

The Root Mean Squared Error (RMSE) is computed using the mean\_squared\_error function from scikit-learn and numpy. The result is printed to the console.

**Inference:**

MinMaxScaler:

Features in the dataset have been normalized using MinMaxScaler, ensuring that all features are on a similar scale.

Dataset Splitting:

The dataset has been split into training and testing sets to evaluate the model's performance on unseen data.

Linear Regression Model:

The Linear Regression model has been successfully trained on the scaled training data.

Predictions:

Predictions have been made on the test set, and the results are stored in y\_pred.

RMSE Calculation:

The RMSE is a measure of the average deviation between the predicted and actual values. A lower RMSE indicates better model performance.

Results:

The RMSE value for the Linear Regression model on the test set, after applying MinMaxScaler, is printed to the console.

21) **Apply RobustScaler,Split the dataset into train and test(20%), apply LinearRegression and calculate RMSE**

**Steps:**

Import Libraries:

Necessary libraries are imported, including RobustScaler, LinearRegression, and metrics for evaluation (mean\_squared\_error and numpy). Also, train\_test\_split is used to split the dataset.

Apply RobustScaler:

The RobustScaler is applied to scale the features of the dataset, and the result is stored in X\_scaled. RobustScaler is resilient to outliers.

Split Dataset:

The dataset is split into training and testing sets using train\_test\_split. 80% of the data is used for training, and 20% is reserved for testing.

Apply Linear Regression:

A Linear Regression model is instantiated and trained on the scaled training data (X\_train and y\_train).

Make Predictions:

The trained model is used to predict the target variable (y\_pred) on the test set (X\_test).

Calculate RMSE:

The Root Mean Squared Error (RMSE) is computed using the mean\_squared\_error function from scikit-learn and numpy. The result is printed to the console.

**Inference:**

RobustScaler:

Features in the dataset have been scaled using RobustScaler, which is resilient to outliers.

Dataset Splitting:

The dataset has been split into training and testing sets. The model was trained on 80% of the data and tested on the remaining 20%.

Linear Regression Model:

The Linear Regression model has been successfully trained on the scaled training data.

Predictions:

Predictions have been made on the 20% test set (X\_test). The results are stored in y\_pred.

RMSE Calculation:

The RMSE is a measure of the average deviation between the predicted and actual values on the test set. A lower RMSE indicates better model performance.

Results:

The RMSE value for the Linear Regression model on the 20% test set, after applying RobustScaler, is printed to the console.

22) **Apply MaxAbsScaler, split the dataset into train and test(20%), apply LinearRegression and calculate RMSE**

**Steps:**

Import Libraries:

Necessary libraries are imported, including MaxAbsScaler, LinearRegression, and metrics for evaluation (mean\_squared\_error and numpy). Also, train\_test\_split is used to split the dataset.

Apply MaxAbsScaler:

The MaxAbsScaler is applied to scale the features of the dataset, and the result is stored in X\_scaled. MaxAbsScaler scales each feature by its maximum absolute value, ensuring that the features lie within the range [-1, 1].

Split Dataset:

The dataset is split into training and testing sets using train\_test\_split. 80% of the data is used for training, and 20% is reserved for testing.

Apply Linear Regression:

A Linear Regression model is instantiated and trained on the scaled training data (X\_train and y\_train).

Make Predictions:

The trained model is used to predict the target variable (y\_pred) on the test set (X\_test).

Calculate RMSE:

The Root Mean Squared Error (RMSE) is computed using the mean\_squared\_error function from scikit-learn and numpy. The result is printed to the console.

**Inference:**

MaxAbsScaler:

Features in the dataset have been scaled using MaxAbsScaler, ensuring that each feature lies within the range [-1, 1].

Dataset Splitting:

The dataset has been split into training and testing sets. The model was trained on 80% of the data and tested on the remaining 20%.

Linear Regression Model:

The Linear Regression model has been successfully trained on the scaled training data.

Predictions:

Predictions have been made on the 20% test set (X\_test). The results are stored in y\_pred.

RMSE Calculation:

The RMSE is a measure of the average deviation between the predicted and actual values on the test set. A lower RMSE indicates better model performance.

Results:

The RMSE value for the Linear Regression model on the 20% test set, after applying MaxAbsScaler, is printed to the console.

23)

**Steps:**

Import Libraries:

Necessary libraries are imported, including Normalizer, LinearRegression, and metrics for evaluation (mean\_squared\_error and numpy). Also, train\_test\_split is used to split the dataset.

Apply Normalizer:

The Normalizer is applied to normalize the features of the dataset, and the result is stored in X\_normalized. Normalizer normalizes each sample (row) independently to have unit norm.

Split Dataset:

The dataset is split into training and testing sets using train\_test\_split. 80% of the data is used for training, and 20% is reserved for testing.

Apply Linear Regression:

A Linear Regression model is instantiated and trained on the normalized training data (X\_train and y\_train).

Make Predictions:

The trained model is used to predict the target variable (y\_pred) on the test set (X\_test).

Calculate RMSE:

The Root Mean Squared Error (RMSE) is computed using the mean\_squared\_error function from scikit-learn and numpy. The result is printed to the console.

**Inference**:

Normalizer:

Features in the dataset have been normalized using the Normalizer, ensuring that each sample has unit norm.

Dataset Splitting:

The dataset has been split into training and testing sets. The model was trained on 80% of the data and tested on the remaining 20%.

Linear Regression Model:

The Linear Regression model has been successfully trained on the normalized training data.

Predictions:

Predictions have been made on the 20% test set (X\_test). The results are stored in y\_pred.

RMSE Calculation:

The RMSE is a measure of the average deviation between the predicted and actual values on the test set. A lower RMSE indicates better model performance.

Results:

The RMSE value for the Linear Regression model on the 20% test set, after applying Normalizer, is printed to the console.

24) **Define a function valuelabel to place the legend of each bar in the histogram**

**Steps:**

Import Libraries:

Necessary libraries, including Matplotlib and NumPy, are imported for data visualization and array operations.

Define valuelabel Function:

The function valuelabel is defined to add value labels on top of each bar in a histogram.

Generate Example Data:

An example dataset (data) is generated using NumPy's random.randn() function. Replace this with your actual dataset.

Create Histogram:

A histogram is created using Matplotlib's hist function, displaying the distribution of the dataset.

Add Value Labels:

The valuelabel function is called to add value labels on top of each bar in the histogram.

Customize Plot:

The plot is customized with a title, x-axis label, and y-axis label.

Show Plot:

The plot is displayed using plt.show().

**Inference:**

Histogram Visualization:

The code generates a histogram to visualize the distribution of a dataset using blue bars.

Value Labels:

The valuelabel function is employed to add precise value labels on top of each bar in the histogram, showing the height of each bar.

Customization:

The plot is further customized with a title, x-axis label ('Values'), and y-axis label ('Frequency').

Example Data:

An example dataset (data) is used for demonstration. Replace it with your actual dataset.

25) **Plot a histogram to display the RMSE value of each scaler**

**Steps:**

Import Libraries:

Necessary libraries, including Matplotlib and random, are imported for data visualization and random color generation.

Generate Random Colors:

Random colors are generated for each bar in the bar chart using the random library.

Plot Bar Chart:

A bar chart is plotted using Matplotlib's bar function, with each bar representing the RMSE value for a specific preprocessing scaler.

Customize the Plot:

The plot is customized with appropriate labels for the x-axis, y-axis, and title.

Add Value Labels:

The valuelabel function is called to add value labels on top of each bar in the bar chart.

Display the Plot:

The final bar chart is displayed using plt.show().

**Inference:**

Bar Chart Visualization:

The code generates a bar chart to visually compare RMSE values for different preprocessing techniques.

Random Color Assignment:

Random colors are assigned to each bar, providing a visually appealing representation.

Customization:

The plot is customized with appropriate labels for the x-axis ('Preprocessing Techniques'), y-axis ('RMSE Values'), and title ('Bar Chart for RMSE values').

Value Labels:

The valuelabel function is used to add precise RMSE values on top of each bar in the chart.