

**TITLE: *MULTIPLE LINEAR REGRESSION USING
PYTHON WITH SCIKIT-LEARN LIBRARY***

INDEX

1.1 OBJECTIVES

1.2 EQUIPMENT/SETUP

1.3 BACKGROUND

1.4 MULTIPLE LINEAR REGRESSION STRATEGY

1.5 PROBLEM DESCRIPTION

1.6 PROCEDURE

1.7 LAB PRACTICE/EXPERIMENT

1.7.1 Getting the dataset

1.7.2 Understanding the dataset

1.7.3 Import Libraries in Google Colab

1.7.4 Fetching data from google drive

1.7.5 Plotting the data for understanding

1.7.6 Plotting the data in different fashion

1.7.7 Data Pre-processing

1.7.7.1 Checking the NULL values

1.7.7.2 Define dependent (target) and independent (predictor) features

1.7.7.3 Splitting Train and Test data

1.7.7.4 Feature Scaling for Multiple Linear Regression

1.7.8 Model Building and Training

1.7.8.1 Training the model

1.7.8.2 Predicting for test data

1.7.9 Model Evaluation

1.7.9.1 Calculating Metrics

1.7.10 Visualizing Predictions

1.7.10.1 Visualize predicted vs. actual values

1.7.10.2 Visualizing Residuals

1.7.10.3 Visualizing Coefficients

1.8 RESULT/COMMENTS

1.9 INSTRUCTIONS

ATTACHMENT

REPORT FORM

NOTE: The report form must be filled up based on your understanding after completion of the experiment and have to be submitted after showing the results.

1.1 OBJECTIVE(S):

- In this tutorial project, you will build and evaluate multiple linear regression models using Python. You will use the scikit-learn library for calculating the regression, pandas for data management, and seaborn for visualization. The project aims to predict sales revenue based on advertising spending through different media channels such as TV, radio, and newspaper.

1.2 EQUIPMENT/SETUP:

- A Computer (PC) with Internet connection
- Operating system
- Any browser to use Google Colab (<https://colab.research.google.com>)
- To follow this tutorial, you will need a Jupyter notebook environment (e.g., Google Colab) with the following libraries installed: numpy, pandas, matplotlib, seaborn, and scikit-learn.

1.3 BACKGROUND:

Multiple linear regression is a statistical technique used to model the relationship between multiple independent variables and a dependent variable. In this project, you'll apply it to predict sales revenue using advertising spending data. Scikit-learn is a powerful library for machine learning, while pandas and seaborn provide efficient data management and visualization capabilities.

1.4 MULTIPLE LINEAR REGRESSION STRATEGY:

This project will follow a step-by-step approach:

- Data Loading and Exploration
- Data Preprocessing
- Model Building and Training
- Model Evaluation
- Visualization of Results

1.5 PROBLEM DESCRIPTION:

The Advertising dataset contains information about advertising spending on TV, radio, and newspaper, along with corresponding sales revenue. Your goal is to build a multiple linear regression model to predict sales revenue based on advertising spending through these media channels.

1.6 PROCEDURE:

- Get the 'Advertising.csv' and upload in your Google Drive.
- Open Google Colab (<https://colab.research.google.com/>).

- Create/rename your project as ‘*Multiple Linear Regression.ipynb*’.
- Follow the steps of next section (read and understand) and execute all the codes.

1.7 LAB PRACTICE/EXPERIMENT:

1.7.1 Getting the dataset

Download the dataset ‘**Advertising.csv**’ from <https://github.com/TITHI-KHAN/MULTIPLE-LINEAR-REGRESSION> and upload in your Google Drive. Please open the CSV file to see and understand the data.

1.7.2 Understanding the dataset

I've provided a dataset with four columns: TV, radio, newspaper, and sales. Each row represents a data point with corresponding values for TV advertising budget, radio advertising budget, newspaper advertising budget, and sales.

It is a marketing dataset where you have information about advertising budgets on different media (TV, radio, newspaper) and the resulting sales.

Display the first few rows, statistics, and data types using pandas.

```
import pandas as pd

data = pd.read_csv('Advertising.csv')
print(data.head())
print(data.describe())
print(data.info())
```

The CSV file contains 200 records (small dataset) in total. The **output** shows:

```

      TV  radio  newspaper  sales
0  230.1   37.8         69.2   22.1
1   44.5   39.3         45.1   10.4
2   17.2   45.9         69.3    9.3
3  151.5   41.3         58.5   18.5
4  180.8   10.8         58.4   12.9

      TV      radio  newspaper      sales
count  200.000000  200.000000  200.000000  200.000000
mean   147.042500   23.264000   30.554000   14.022500
std     85.854236   14.846809   21.778621    5.217457
min      0.700000    0.000000    0.300000    1.600000
25%     74.375000    9.975000   12.750000   10.375000
50%    149.750000   22.900000   25.750000   12.900000
75%    218.825000   36.525000   45.100000   17.400000
max    296.400000   49.600000  114.000000   27.000000
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0    TV          200 non-null    float64
1   radio        200 non-null    float64
2  newspaper    200 non-null    float64
3   sales        200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
None

```

Explanation:

Let's go through the code step by step and explain each part:

import pandas as pd

- This line imports the pandas library, which is a powerful data manipulation and analysis library in Python. It's commonly used to handle structured data, such as CSV files.

data = pd.read_csv('Advertising.csv')

- This line reads the CSV file named 'Advertising.csv' and loads its contents into a pandas DataFrame called `data`. A DataFrame is a two-dimensional labeled data structure with rows and columns, similar to a table in a database or a spreadsheet.

print(data.head())

- This line prints the first few rows of the DataFrame using the `.head()` method. By default, `.head()` shows the first five rows. It's a quick way to get an overview of the dataset's structure and the type of data it contains.

print(data.describe())

- This line computes summary statistics of the numerical columns in the DataFrame using the `.describe()` method. It provides information such as mean,

standard deviation, minimum, maximum, and quartiles. This summary helps you understand the central tendency and spread of the numerical data.

print(data.info())

- This line provides information about the DataFrame's structure using the `.info()` method. It includes the number of non-null entries, the data types of each column, and memory usage. This is useful for understanding the completeness of the dataset and identifying any missing values.

1.7.3 Import Libraries in Google Colab

In Google Colab, import the necessary libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

Explanation:

Let's go through each import statement and explain the purpose of each library:

import numpy as np

- This line imports the NumPy library, which provides support for arrays and matrices, along with mathematical functions to operate on these arrays. NumPy is essential for numerical computations in Python.

import pandas as pd

- This line imports the pandas library, which is widely used for data manipulation and analysis. It offers data structures like DataFrame and Series, making it easier to work with structured data.

import matplotlib.pyplot as plt

- This line imports the `pyplot` module from the Matplotlib library. Matplotlib is a popular plotting library that allows you to create various types of graphs and visualizations. The `pyplot` module provides a simple interface for creating and customizing plots.

import seaborn as sns

- This line imports the Seaborn library, which is built on top of Matplotlib and provides a higher-level interface for creating visually appealing statistical

graphics. Seaborn simplifies the process of creating complex visualizations and provides various styles and color palettes.

from sklearn.model_selection import train_test_split

- This line imports the `train_test_split` function from scikit-learn's `model_selection` module. This function is used to split datasets into training and testing subsets. It's a fundamental step in machine learning to ensure that your model is evaluated on unseen data.

from sklearn.linear_model import LinearRegression

- This line imports the `LinearRegression` class from scikit-learn's `linear_model` module. Linear regression is a simple and widely used machine learning algorithm for modeling the relationship between dependent and independent variables.

from sklearn.metrics import mean_squared_error, r2_score

- This line imports the `mean_squared_error` and `r2_score` functions from scikit-learn's `metrics` module. These functions are used to evaluate the performance of regression models. `mean_squared_error` calculates the mean squared error between actual and predicted values, while `r2_score` computes the coefficient of determination, indicating how well the model fits the data.

In summary, the code snippet imports the necessary libraries for data manipulation, visualization, model training and evaluation, and metrics calculation. These libraries will be used throughout your analysis and modeling process.

1.7.4 Fetching data from google drive

If using Google Drive, mount it and read the dataset.

```
from google.colab import drive

drive.mount('/content/drive')

data = pd.read_csv('/content/drive/My Drive/advertising.csv')
```

Explanation:

from google.colab import drive

- This line imports the `drive` module from the `google.colab` package. This module allows you to connect and interact with your Google Drive directly within a Google Colab notebook.

drive.mount('/content/drive')

- This line mounts your Google Drive to the '/content/drive' directory in the Colab environment. After running this line, you'll need to authenticate with your Google account and grant permission to access your Google Drive.

data = pd.read_csv('/content/drive/My Drive/advertising.csv')

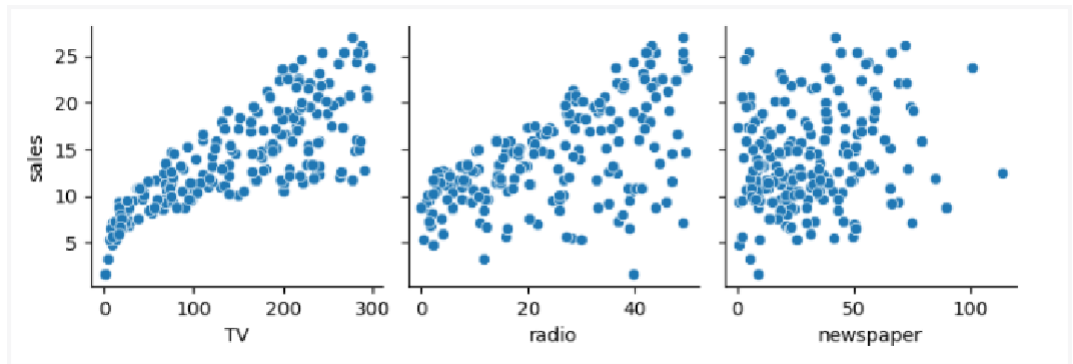
- This line reads a CSV file named 'advertising.csv' from your Google Drive. The file is located in the 'My Drive' directory. You need to provide the correct path to the file based on its location in your Google Drive.

1.7.5 Plotting the data for understanding

Visualize relationships between features and the target using seaborn.

```
sns.pairplot(data, x_vars=['TV', 'radio', 'newspaper'], y_vars='sales', kind='scatter')
plt.show()
```

Output:



Explanation:

The code I provided is creating a pair plot using the Seaborn library. A pair plot is a matrix of scatter plots that allows you to visualize relationships between multiple pairs of variables. Let's break down the code:

sns.pairplot(data, x_vars=['TV', 'radio', 'newspaper'], y_vars='sales', kind='scatter')

- `sns.pairplot`: This function from Seaborn is used to create a pair plot.
- `data`: The DataFrame containing the data you want to visualize.
- `x_vars=['TV', 'radio', 'newspaper']`: This parameter specifies the variables that will be plotted on the x-axis.
- `y_vars='sales'`: This parameter specifies the variable that will be plotted on the y-axis.
- `kind='scatter'`: This parameter specifies the type of plot to create. In this case, it's a scatter plot.

The pair plot will create scatter plots for each combination of the specified variables (x_vars) against the sales variable (y_vars).

plt.show()

- This line displays the generated pair plot. `plt.show()` is necessary to show the plot in the notebook or a separate window, depending on your environment.

1.7.6 Plotting the data in different fashion

Explore other plot types like histograms, box plots, and correlation matrices.

```
# Histograms
sns.histplot(data=data, x='TV', bins=20, kde=True)
plt.title('TV Advertisement Histogram')
plt.xlabel('TV Advertisement')
plt.ylabel('Frequency')
plt.show()

sns.histplot(data=data, x='radio', bins=20, kde=True)
plt.title('Radio Advertisement Histogram')
plt.xlabel('Radio Advertisement')
plt.ylabel('Frequency')
plt.show()

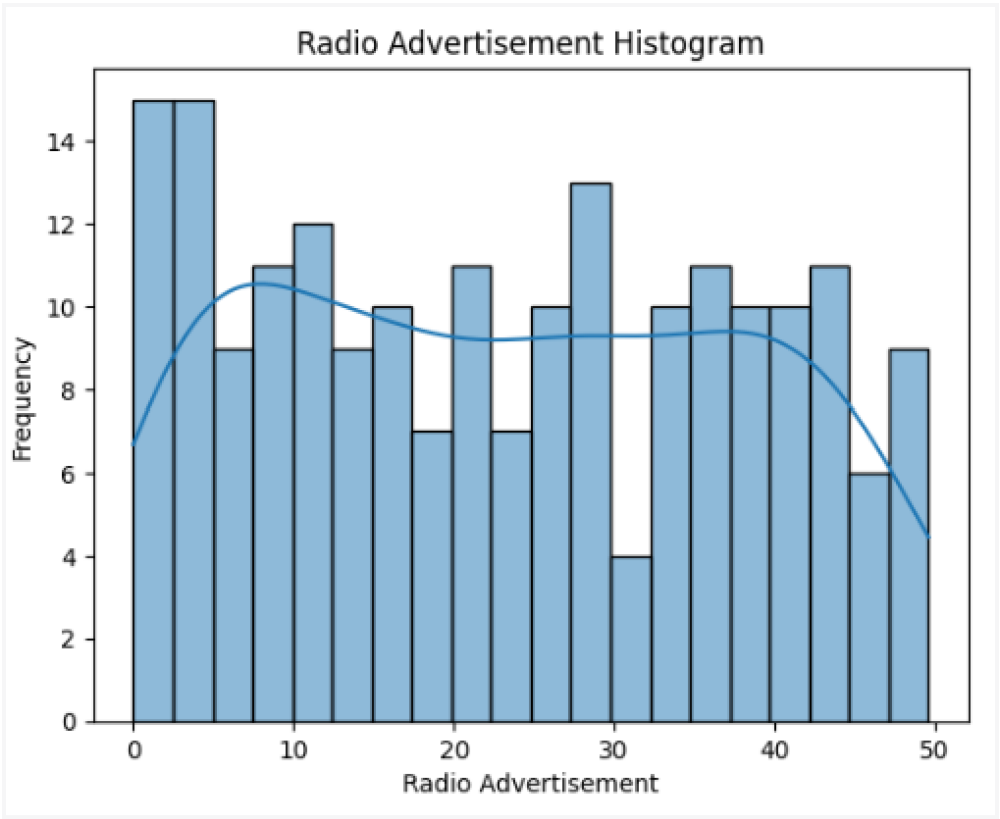
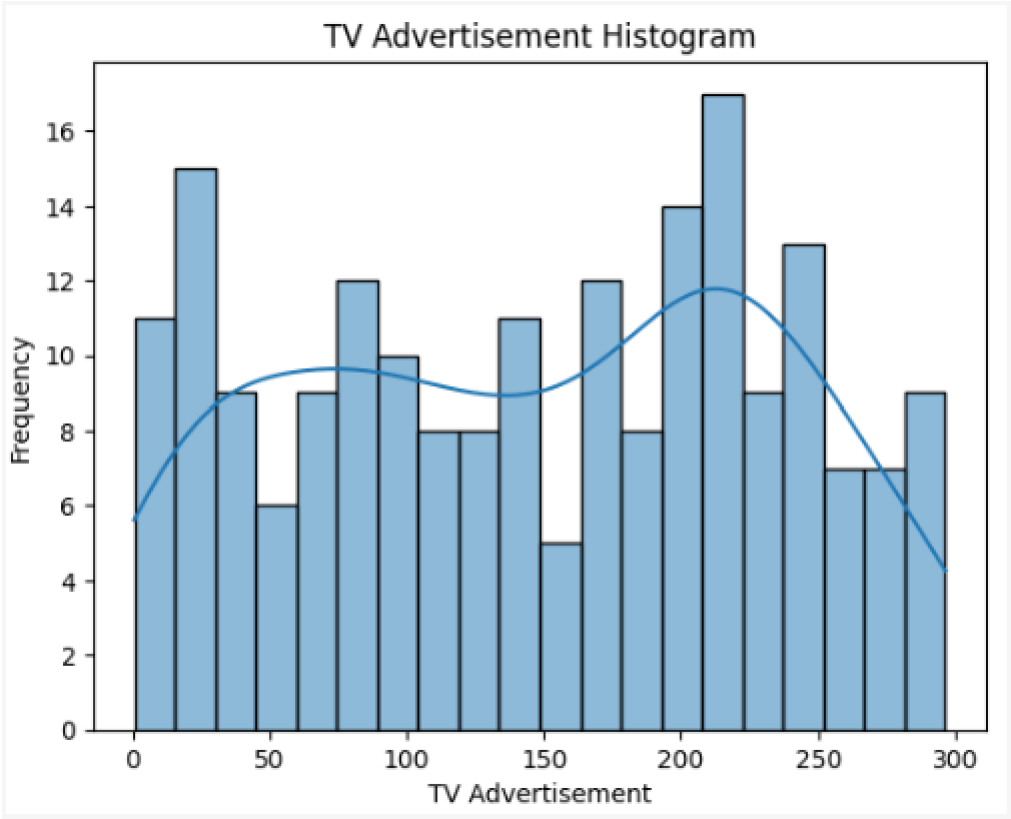
sns.histplot(data=data, x='newspaper', bins=20, kde=True)
plt.title('Newspaper Advertisement Histogram')
plt.xlabel('Newspaper Advertisement')
plt.ylabel('Frequency')
plt.show()

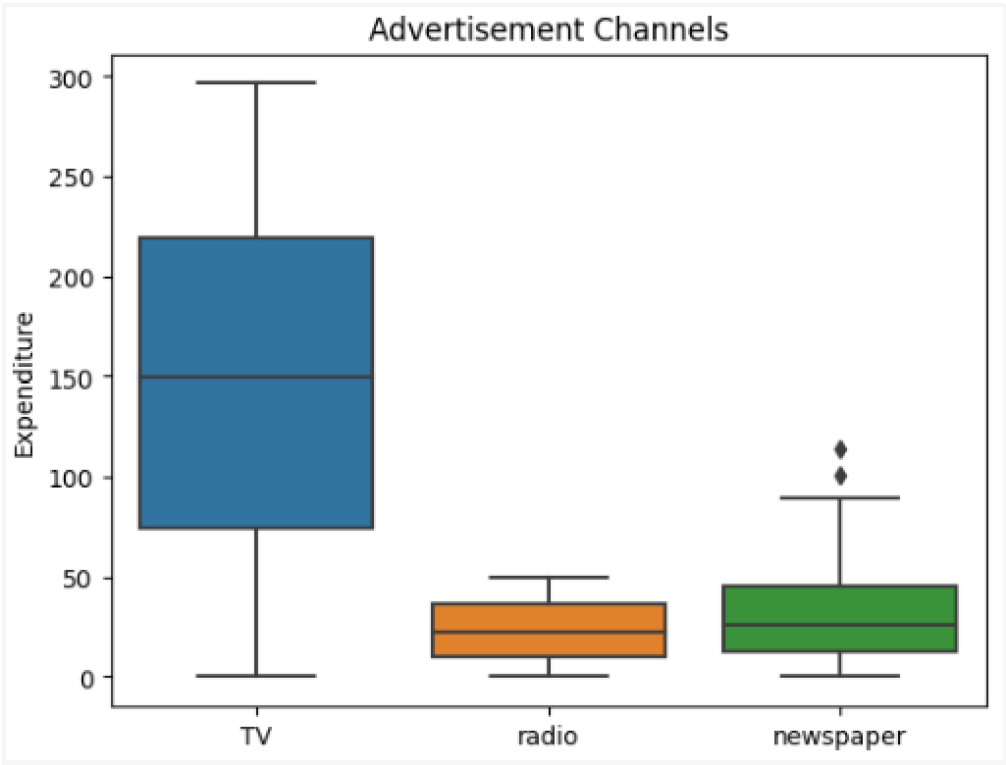
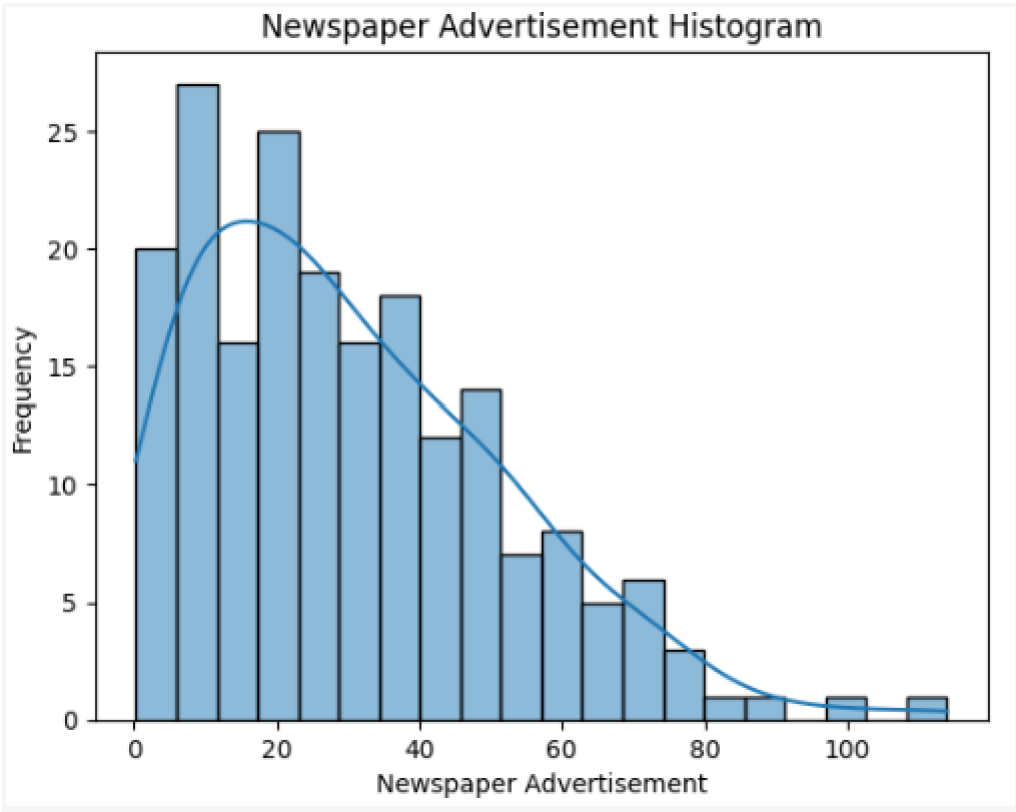
# Box plots
sns.boxplot(data=data[['TV', 'radio', 'newspaper']])
plt.title('Advertisement Channels')
plt.ylabel('Expenditure')
plt.show()

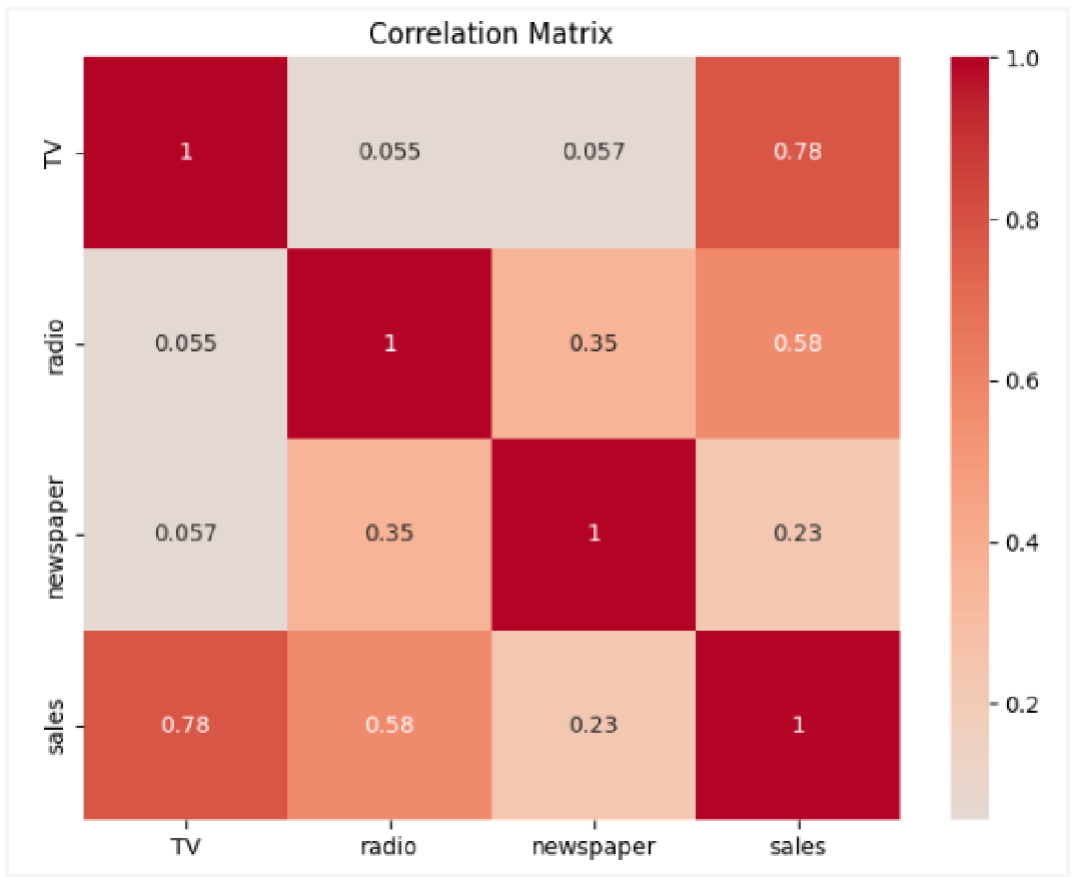
# Correlation matrix
correlation_matrix = data[['TV', 'radio', 'newspaper', 'sales']].corr()

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix')
plt.show()
```


Output:







Explanation:

This code snippet is performing various types of visualization using the Seaborn library. Let's break down each section:

Histograms

```
sns.histplot(data=data, x='TV', bins=20, kde=True)
```

```
plt.title('TV Advertisement Histogram')
```

```
plt.xlabel('TV Advertisement')
```

```
plt.ylabel('Frequency')
```

```
plt.show()
```

- This code creates a histogram of the 'TV' advertising spending data using Seaborn's `histplot` function.
- `data=data`: Specifies the DataFrame containing the data.
- `x='TV'`: Specifies the variable to plot on the x-axis.
- `bins=20`: Sets the number of bins for the histogram.
- `kde=True`: Adds a kernel density estimate plot on top of the histogram.

- `plt.title`, plt.xlabel`, plt.ylabel`: Sets the title, x-axis label, and y-axis label for the plot.`

- `plt.show()`: Displays the plot.

The code for the histograms of 'radio' and 'newspaper' advertisements is similar to the above code.

Box Plots

```
sns.boxplot(data=data[['TV', 'radio', 'newspaper']])
```

```
plt.title('Advertisement Channels')
```

```
plt.ylabel('Expenditure')
```

```
plt.show()
```

- This code creates a box plot to visualize the distribution of advertising expenditures for 'TV', 'radio', and 'newspaper'.

- `data=data[['TV', 'radio', 'newspaper']]`: Specifies the DataFrame and columns to include in the box plot.

- `plt.title`, plt.ylabel`: Sets the title and y-axis label for the plot.`

- `plt.show()`: Displays the plot.

Correlation Matrix Heatmap

```
correlation_matrix = data[['TV', 'radio', 'newspaper', 'sales']].corr()
```

```
plt.figure(figsize=(8, 6))
```

```
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
```

```
plt.title('Correlation Matrix')
```

```
plt.show()
```

- This code calculates the correlation matrix for the 'TV', 'radio', 'newspaper', and 'sales' variables using the `.corr()` method.

- `plt.figure(figsize=(8, 6))`: Sets the figure size for the heatmap.

- `sns.heatmap`: Creates a heatmap of the correlation matrix.

- `annot=True`: Displays the correlation values within the cells.

- `cmap='coolwarm'`: Sets the color map for the heatmap.

- `center=0`: Sets the center value for the color map.

- `plt.title`, plt.show(): Sets the title for the heatmap and displays it.`

These visualization techniques help you explore and understand the distribution, spread, and relationships among the variables in your dataset.

1.7.7 Data Pre-processing

Prepare the data for modeling.

1.7.7.1 Checking the NULL values

Check for and handle NULL values.

```

null_counts = data.isnull().sum()
data_cleaned = data.dropna()
null_counts

```

Output:

```

TV          0
radio       0
newspaper   0
sales       0
dtype: int64

```

Explanation:

The code snippet I provided is used to handle missing values in the dataset and count the number of missing values in each column. Let's break down each part:

null_counts = data.isnull().sum()

- This line calculates the sum of missing values in each column of the DataFrame `data` using the `.isnull().sum()` method chain. The result is a Series where each index corresponds to a column name, and the value represents the number of missing values in that column.

data_cleaned = data.dropna()

- This line creates a new DataFrame called `data_cleaned` by removing rows that contain any missing values. The `.dropna()` function drops rows with missing values, and the resulting DataFrame contains only rows with complete data.

null_counts

- This line is used to display the `null_counts` Series that contains the count of missing values in each column.

Here's an example of how this code works:

Assume your original `data` DataFrame looks like this:

```
| TV | Radio | Newspaper | Sales |
|-----|-----|-----|-----|
| 230 | 37 | 22 | 25 |
| NaN | 24 | 18 | 24 |
| 170 | 43 | NaN | 28 |
| 100 | NaN | 35 | 18 |
```

After running the code, the `null_counts` Series would show:

```
TV      1
Radio    1
Newspaper 1
Sales    0
dtype: int64
```

And the `data_cleaned` DataFrame would contain:

```
| TV | Radio | Newspaper | Sales |
|-----|-----|-----|-----|
| 230 | 37 | 22 | 25 |
```

This process helps you identify and handle missing values in your dataset, ensuring that you work with complete and accurate data for analysis and modeling.

1.7.7.2 Define dependent (target) and independent (predictor) features

```
x = data_cleaned[['TV', 'radio', 'newspaper']]
y = data_cleaned['sales']
```

Explanation:

The code snippet I provided is used to separate the dataset into predictor variables (features) and the target variable. Let's break down each part:

X = data_cleaned[['TV', 'radio', 'newspaper']]

- This line creates a DataFrame `X` containing the predictor variables (features). It includes the columns 'TV', 'radio', and 'newspaper' from the `data_cleaned` DataFrame. Each row in `X` corresponds to an observation, and each column

represents a different feature.

y = data_cleaned['sales']

- This line creates a Series `y` containing the target variable. It extracts the 'sales' column from the `data_cleaned` DataFrame. Each value in `y` corresponds to the target value for a specific observation.

This separation of the data into `X` (features) and `y` (target) is a common practice in machine learning and statistical analysis. It allows you to keep the predictor variables separate from the target variable, making it easier to perform modeling, training, and evaluation tasks.

1.7.7.3 Splitting Train and Test data

Split the data into training and testing sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Explanation:

The code is using the `train_test_split` function from scikit-learn to split the dataset into training and testing sets. Let's break down each part:

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

- `train_test_split`: This function is used to split arrays or matrices into random train and test subsets. It's often used for splitting datasets into training and testing sets for machine learning models.

- `X`: The predictor variable DataFrame (features).

- `y`: The target variable Series.

- `test_size=0.2`: This parameter specifies the proportion of the dataset that should be used for testing. In this case, 20% of the data will be used for testing, and the remaining 80% will be used for training.

- `random_state=42`: This parameter sets the random seed for reproducibility. Providing a specific random seed ensures that the same split is generated each time you run the code.

After running this line of code, you'll have:

- `X_train`: The training set of predictor variables.

- `X_test`: The testing set of predictor variables.

- `y_train`: The training set of target variables.
- `y_test`: The testing set of target variables.

These sets are now ready for training and evaluating your machine learning model.

1.7.7.4 Feature Scaling for Multiple Linear Regression

Normalize or standardize features if necessary.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Explanation:

The code snippet I provided demonstrates how to use the `StandardScaler` from scikit-learn to scale the features of your dataset. Scaling is an important preprocessing step that helps ensure that features are on similar scales, which can improve the performance of certain machine learning algorithms. Let's breakdown each part:

from sklearn.preprocessing import StandardScaler

- This line imports the `StandardScaler` class from the `sklearn.preprocessing` module. The `StandardScaler` is used to standardize features by removing the mean and scaling to unit variance.

scaler = StandardScaler()

- This line creates an instance of the `StandardScaler` class. The scaler will be used to transform the features.

X_train_scaled = scaler.fit_transform(X_train)

- This line applies the scaling transformation to the training set of predictor variables, `X_train`. The `fit_transform()` method calculates the mean and standard deviation from the training data and scales the features accordingly. The scaled features are stored in the `X_train_scaled` array.

X_test_scaled = scaler.transform(X_test)

- This line applies the same scaling transformation to the testing set of predictor variables, `X_test`. The `transform()` method uses the mean and standard deviation calculated from the training data to scale the testing features. The scaled features are stored in the `X_test_scaled` array.

After applying the scaling, both `X_train_scaled` and `X_test_scaled` will have features that are standardized to have zero mean and unit variance.

This step is particularly important when you have features with different scales, as it can help prevent certain features from dominating the learning process of machine learning algorithms that are sensitive to the scale of input features.

1.7.8 Model Building and Training

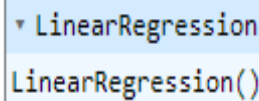
Build and train the multiple linear regression model.

1.7.8.1 Training the model

Train the linear regression model.

```
model = LinearRegression()
model.fit(X_train_scaled, y_train)
```

Output:



Explanation:

The code I provided is used to create an instance of the `LinearRegression` model from scikit-learn, and then fit the model to the scaled training data. Let's break down each part:

model = LinearRegression()

- This line creates an instance of the `LinearRegression` class. The `LinearRegression` class represents a linear regression model. It's a basic regression algorithm used to model the relationship between one or more independent variables (features) and a dependent variable (target).

model.fit(X_train_scaled, y_train)

- This line fits (trains) the linear regression model using the scaled training data. The `fit()` method takes the scaled predictor variables `X_train_scaled` and the target variable `y_train` as arguments. The model will learn the coefficients that best fit the training data based on the linear relationship between the features and the target.

After running this code, the `model` instance will be trained and ready to make predictions using new, unseen data. The learned coefficients represent the weights assigned to each feature in the linear equation used for making predictions.

Keep in mind that the features have been scaled, which is important for linear regression to ensure that each feature contributes fairly to the prediction, regardless of its original scale.

1.7.8.2 Predicting for Test Data

Use the trained model to make predictions.

```
y_pred = model.predict(X_test_scaled)
```

Explanation:

The code I provided is used to make predictions using the trained linear regression model on the scaled testing data. Let's break down the line:

y_pred = model.predict(X_test_scaled)

- This line uses the trained `model` (which is a `LinearRegression` instance) to predict the target variable `y` based on the scaled testing data `X_test_scaled`.
- `model.predict(X_test_scaled)` calculates the predicted values of the target variable using the learned coefficients and the feature values in `X_test_scaled`.
- The predicted values are stored in the array `y_pred`.

After running this line of code, you'll have the predicted values of the target variable for the testing data. These predicted values can be compared with the actual target values (`y_test`) to evaluate the model's performance and accuracy.

1.7.9 Model Evaluation

Evaluate the model's performance.

1.7.9.1 Calculating Metrics

Calculate mean squared error and R-squared.

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("MSE", mse)
print("R2", r2)
```

Output:

```
MSE 3.174097353976106
R2 0.8994380241009119
```

Explanation:

The code I provided is used to calculate and print the Mean Squared Error (MSE) and R-squared (R2) score, which are common metrics for evaluating the performance of regression models. Let's break down each part:

mse = mean_squared_error(y_test, y_pred)

- This line calculates the Mean Squared Error (MSE) between the actual target values `y_test` and the predicted target values `y_pred`. The `mean_squared_error()` function from scikit-learn's `metrics` module is used for this calculation. MSE measures the average squared difference between the actual and predicted values. Lower values of MSE indicate better model performance.

r2 = r2_score(y_test, y_pred)

- This line calculates the R-squared (R2) score between the actual target values `y_test` and the predicted target values `y_pred`. The `r2_score()` function from scikit-learn's `metrics` module is used for this calculation. R2 score measures the proportion of the variance in the target variable that is predictable from the independent variables. It provides an indication of how well the model fits the data.

print("MSE", mse)

print("R2", r2)

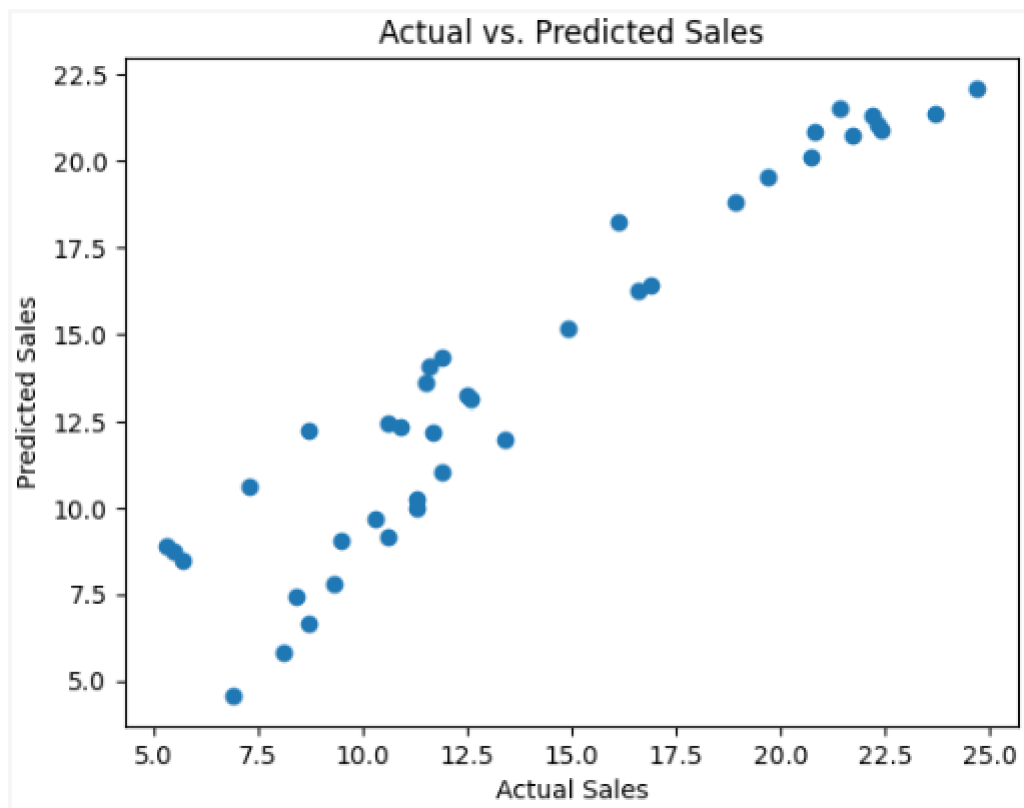
- These lines print the calculated MSE and R2 score to the console.

After running this code, you'll get the values of MSE and R2 score, which provide insights into the model's accuracy and how well it fits the data. Lower MSE and higher R2 values are generally indicative of better model performance.

1.7.10 Visualizing Predictions

1.7.10.1 Visualize predicted vs. actual values

```
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Actual vs. Predicted Sales')
plt.show()
```

Output:**Explanation:**

The code I provided is creating a scatter plot to visually compare the actual sales values (`y_test`) with the predicted sales values (`y_pred`) from your linear regression model. Let's break down each part:

`plt.scatter(y_test, y_pred)`

- This line creates a scatter plot using the Matplotlib library's `scatter` function. It takes `y_test` (actual sales values) as the x-axis values and `y_pred` (predicted sales values) as the y-axis values. Each point on the plot represents an observation, with its actual sales value on the x-axis and its predicted sales value on the y-axis.

`plt.xlabel('Actual Sales')`**`plt.ylabel('Predicted Sales')`**

- These lines set the labels for the x-axis and y-axis, indicating the values being plotted.

`plt.title('Actual vs. Predicted Sales')`

- This line sets the title of the scatter plot.

plt.show()

- This line displays the scatter plot.

By plotting the actual sales values against the predicted sales values, you can visually assess how well your linear regression model's predictions align with the actual data points. If the points on the scatter plot closely follow a diagonal line, it suggests that the model's predictions are accurate. However, if the points are scattered randomly, the model's predictions might not be accurate.

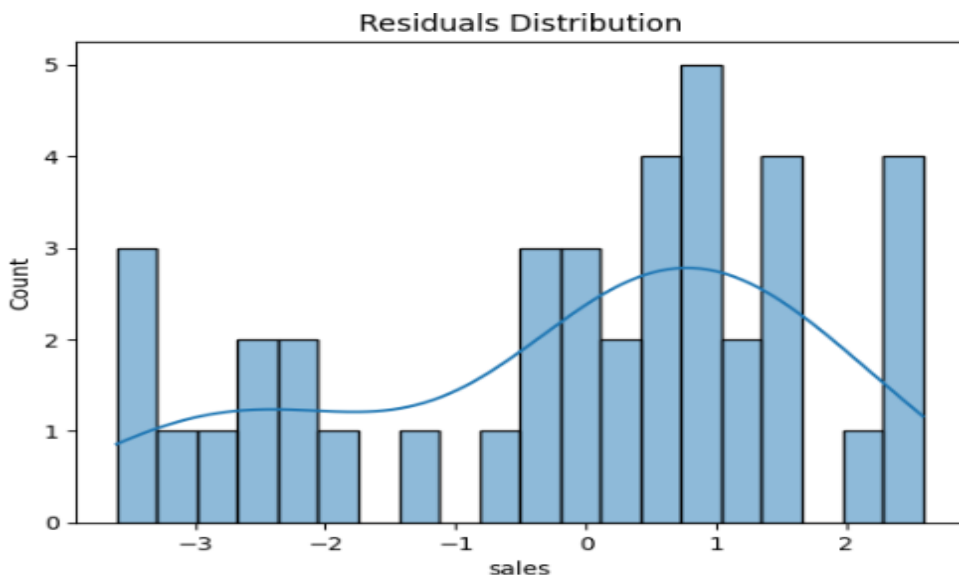
This scatter plot is a useful visualization to evaluate the overall performance and discrepancies between predicted and actual values.

1.7.10.2 Visualizing Residuals

Plot residuals to check for patterns.

```
residuals = y_test - y_pred
sns.histplot(residuals, bins=20, kde=True)
plt.title('Residuals Distribution')
plt.show()
```

Output:

**Explanation:**

The code I provided is used to create a histogram of the residuals, which are the differences between the actual target values (`y_test`) and the predicted target values (`y_pred`). Residual analysis helps you understand the distribution and patterns of the prediction errors. Let's break down each part:

residuals = y_test - y_pred

- This line calculates the residuals by subtracting the predicted sales values (`y_pred`) from the actual sales values (`y_test`). The resulting `residuals` array contains the differences between the predicted and actual values for each observation.

sns.histplot(residuals, bins=20, kde=True)

- This line creates a histogram of the residuals using Seaborn's `histplot` function.
- `residuals` is the array of differences calculated above.
- `bins=20` specifies the number of bins for the histogram.
- `kde=True` adds a kernel density estimate plot on top of the histogram.

plt.title('Residuals Distribution')**plt.show()**

- This line sets the title of the histogram plot.
- `plt.show()` displays the histogram plot.

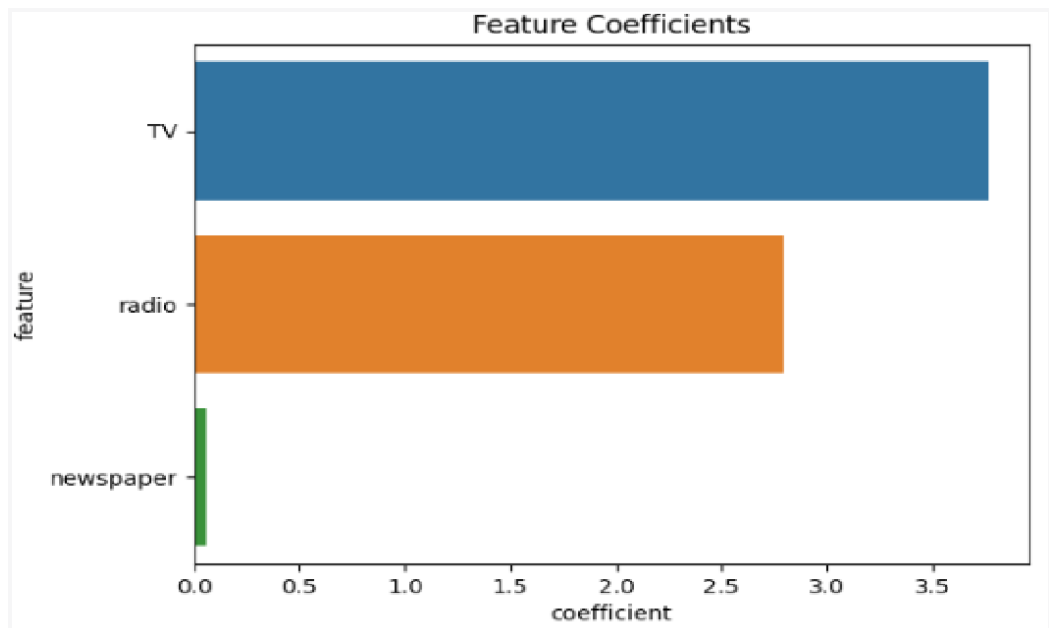
Analyzing the distribution of residuals can provide insights into how well your model is performing. If the residuals are normally distributed around zero and have a constant spread, it suggests that your model's predictions are unbiased and consistent. However, if you observe patterns or non-random behavior in the residuals, it might indicate issues with your model or data.

The histogram of residuals helps you identify potential areas where your model may be making larger errors.

1.7.10.3 Visualizing Coefficients

Visualize the coefficients and their importance.

```
coef_df = pd.DataFrame({'feature': X.columns, 'coefficient': model.coef_})
sns.barplot(x='coefficient', y='feature', data=coef_df)
plt.title('Feature Coefficients')
plt.show()
```

Output:

Explanation:

The code I provided is used to visualize the coefficients of the features in your linear regression model using a bar plot. The coefficients indicate the strength and direction of the relationship between each feature and the target variable. Let's break down each part:

```
coef_df = pd.DataFrame({'feature': X.columns, 'coefficient': model.coef_})
```

- This line creates a DataFrame called `coef_df` to store the feature names and their corresponding coefficients.
- `feature` column: Contains the names of the features from the DataFrame `X`.
- `coefficient` column: Contains the coefficients of the linear regression model, which are stored in `model.coef_`.

```
sns.barplot(x='coefficient', y='feature', data=coef_df)
```

- This line creates a bar plot using Seaborn's `barplot` function.
- `x='coefficient'`: Specifies that the x-axis will represent the coefficients.
- `y='feature'`: Specifies that the y-axis will represent the feature names.
- `data=coef_df`: Specifies the DataFrame containing the data for the plot.

```
plt.title('Feature Coefficients')
```

```
plt.show()
```

- This line sets the title of the bar plot.
- `plt.show()` displays the bar plot.

The bar plot allows you to visualize the impact of each feature on the target variable. Positive coefficients indicate a positive relationship with the target, while negative coefficients indicate a negative relationship. The length of the bars represents the magnitude of the coefficient, indicating the strength of the relationship.

This visualization helps you understand which features have the most influence on the target variable and the direction of their impact.

1.8 RESULT/COMMENTS:

Run all the codes sequentially, observe all the outputs, and write comments in the report form.

Notes: The dataset is small in size and it does not contain real data. Moreover, the Training and Test data selection are random. So, the simulated results could be different from the results shown in this document.

1.9 INSTRUCTIONS:

1. Run all the codes/models and show them to the invigilator/instructor.
1. Observe each of the results and provide your comments in the form.
3. Fill up the report form properly, take a signature from the instructor, and submit.

REFERENCE:

[1]<https://github.com/TITHI-KHAN/MULTIPLE-LINEAR-REGRESSION/tree/main>.

EXPERIMENT NO.: 2.1

**TITLE: MULTIPLE LINEAR REGRESSION USING PYTHON WITH
SCIKIT-LEARN LIBRARY**

DATE:

Batch & Section:

Student ID: _____ Student Name:

Objectives of the experiment:

Some Outputs/Graphs with appropriate title:

Comments (overall understanding):

To fill by the instructor:

Marks from the instructor: ☐ x ☐ y ☐ z

Instructor's Name and Signature:
