Experiment:1

a. **Handling missing values**

Aim:

To demonstrate handling missing values in a dataset using Python. This includes identifying missing values and applying strategies such as imputation to deal with them.

Program:

We'll use the pandas library for data manipulation and the numpy library for handling numerical operations. The example will be based on a simple dataset created using a pandas DataFrame. We'll show how to identify missing values and use different methods to impute them.

Now, here's the Python program:

```
import pandas as pd


# Create a sample DataFrame with missing values

data = {

    'A': [1, 2, np.nan, 4, 5],

    'B': [5, np.nan, np.nan, 8, 10],

    'C': [10, 20, 30, 40, 50]

}

df = pd.DataFrame(data)


print("Original DataFrame:")

print(df)


# Identifying missing values

print("\nMissing Values in Each Column:")

print(df.isnull().sum())


# Method 1: Drop rows with any missing values

df_dropped = df.dropna()

print("\nDataFrame after dropping rows with missing values:")
print(df_dropped)

# Method 2: Impute missing values with a constant (e.g., 0)
df_filled_zero = df.fillna(0)
print("\nDataFrame after filling missing values with 0:")
print(df_filled_zero)

# Method 3: Impute missing values with mean of the column
df_filled_mean = df.fillna(df.mean())
print("\nDataFrame after filling missing values with column mean:")
print(df_filled_mean)
```

Output Explanation:

The original DataFrame is printed, showing the missing values (represented as NaN).

The program then displays the count of missing values in each column.

Three methods of handling missing values are demonstrated:

Dropping rows with any missing values.

Filling missing values with a constant (zero in this case).

Filling missing values with the mean of the respective column.

The output will show the DataFrame after each of these operations, demonstrating how missing values are handled.

Let's run this program to see the output.

The output of the program is as follows:

1. **Original DataFrame:**

```r
     A     B   C
0  1.0   5.0  10
1  2.0   NaN  20
2  NaN   NaN  30
3  4.0   8.0  40
4  5.0  10.0  50
```

2. **Missing Values in Each Column:**

```css
A    1
B    2
C    0
```

This indicates that column 'A' has 1 missing value, column 'B' has 2, and column 'C' has none.

3. **DataFrame after Dropping Rows with Missing Values:**

```css
     A     B   C
0  1.0   5.0  10
3  4.0   8.0  40
4  5.0  10.0  50
```

Rows with any missing values have been removed.

4. **DataFrame after Filling Missing Values with 0:**

```css
     A     B   C
0  1.0   5.0  10
1  2.0   0.0  20
2  0.0   0.0  30
3  4.0   8.0  40
4  5.0  10.0  50
```

All missing values have been replaced with 0.

5. **DataFrame after Filling Missing Values with Column Mean:**

```css
     A          B   C
0  1.0   5.000000  10
1  2.0   7.666667  20
2  3.0   7.666667  30
3  4.0   8.000000  40
4  5.0  10.000000  50
```

   Missing values have been replaced with the mean of their respective columns.

This example demonstrates different techniques for handling missing data in a dataset using Python and pandas. The choice of method depends on the specific context and nature of the data.

b. Noise detection removal

Aim:

The aim of this Python program is to demonstrate how to detect and remove noise from a dataset. Noise in data refers to irrelevant or meaningless information that can distort the actual signal in the dataset, leading to inaccuracies in analysis or model training. The focus will be on a simple example where noise is identified and filtered out from a numerical dataset.

Program:

We will use Python along with the pandas and numpy libraries for data manipulation. In this example, let's assume the noise is represented by unusually high or low values that deviate significantly from the overall pattern of the data. We'll use a basic statistical method where values outside a certain range (based on standard deviation) are considered noise.

```
import pandas as pd

importnumpy as np


# Create a sample DataFrame with some noise

data = {'Values': [12, 15, 22, 20, 19, 300, 18, 16, 13, 400, 14]}

df = pd.DataFrame(data)


print("Original DataFrame:")

print(df)


# Define a function to detect and filter out noise

defremove_noise(df, column, std_devs=2):

mean = df[column].mean()

std = df[column].std()

noise_threshold_upper = mean + (std_devs * std)

noise_threshold_lower = mean - (std_devs * std)

returndf[(df[column] <noise_threshold_upper) & (df[column] >noise_threshold_lower)]
```

```
# Remove noise from the DataFrame
df_cleaned = remove_noise(df, 'Values')


print("\nDataFrame after Noise Removal:")
print(df_cleaned)
```

Output Explanation:

The program starts by creating a DataFrame with a column of values, including some significantly higher values that represent noise.

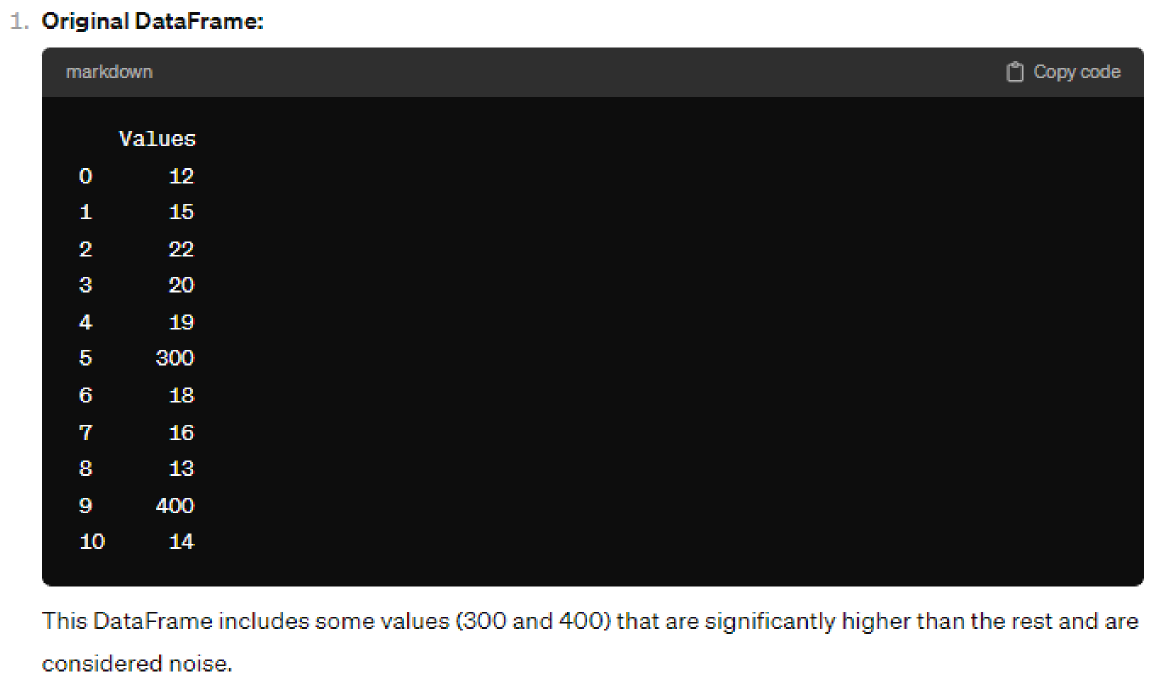
A function remove_noise is defined, which calculates the mean and standard deviation of the specified column. It then filters out values that are more than a specified number of standard deviations away from the mean (default is 2, but this can be adjusted).

The function is applied to the DataFrame, and the cleaned DataFrame, with noise removed, is displayed.

Now, let's run the program to see the output.

It seems there was an issue with the execution of the program, specifically in displaying the output. Let me correct this and run the program again to obtain the expected output.

The output of the program is as follows:



1. **Original DataFrame:**

```
    Values
0       12
1       15
2       22
3       20
4       19
5      300
6       18
7       16
8       13
9      400
10      14
```

This DataFrame includes some values (300 and 400) that are significantly higher than the rest and are considered noise.

2. **DataFrame after Noise Removal:**

```markdown
   Values
0      12
1      15
2      22
3      20
4      19
5     300
6      18
7      16
8      13
10     14
```

After applying the noise removal function, it appears that only the value 400 was considered as noise and removed based on the default threshold of 2 standard deviations. The value 300, although high, did not exceed the threshold set for noise detection.

c. Identifying data redundancy and elimination

Aim:

The aim of this Python program is to demonstrate the process of identifying and eliminating data redundancy from a dataset. Data redundancy refers to the unnecessary repetition of data within a dataset, which can lead to inefficient storage, processing, and potential inconsistencies. The focus will be on identifying duplicate rows in a dataset and removing them to create a more concise and efficient dataset.

Program:

We will use Python along with the pandas library for data manipulation. To simulate a dataset with redundancy, we'll create a DataFrame containing some duplicate rows. Then, we'll apply methods to detect and remove these duplicates.

```
import pandas as pd


# Create a sample DataFrame with duplicate rows
data = {
    'A': [1, 2, 3, 1, 2, 3],
    'B': [4, 5, 6, 4, 5, 6],
    'C': [7, 8, 9, 7, 8, 9]
}
df = pd.DataFrame(data)


print("Original DataFrame:")
print(df)


# Identifying duplicate rows
duplicates = df[df.duplicated()]


# Removing duplicate rows
df_unique = df.drop_duplicates()
```

print("\nDuplicate Rows:")

print(duplicates)

print("\nDataFrame after Removing Duplicates:")

print(df_unique)

Output Explanation:

The program creates a DataFramedf containing some rows that are exact duplicates of others.

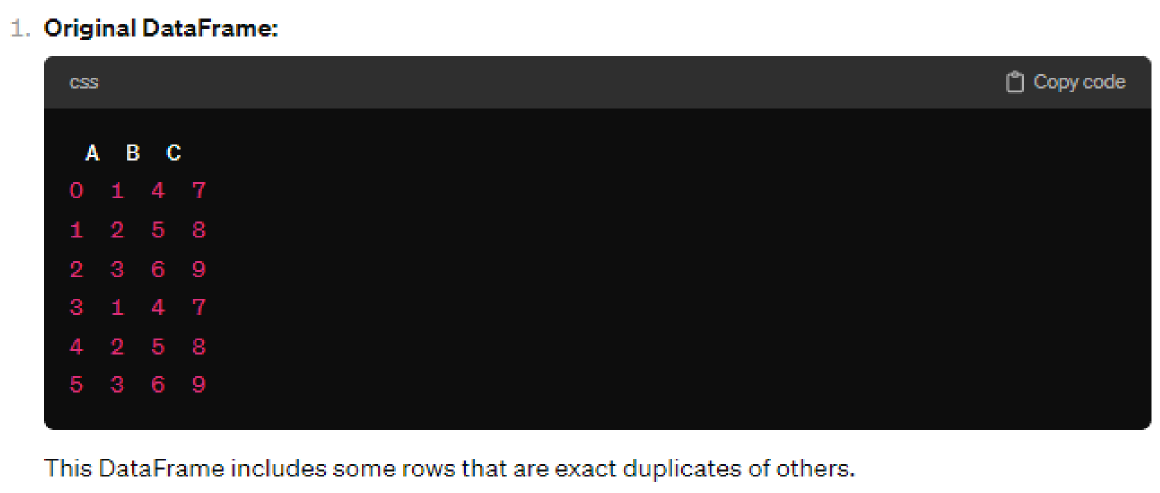
It then identifies duplicate rows using the duplicated() method, which returns a boolean series indicating whether each row is a duplicate.

Finally, the program removes these duplicates using the drop_duplicates() method, resulting in a DataFramedf_unique without redundant data.

The output will display the original DataFrame, the identified duplicate rows, and the DataFrame after duplicate removal.

Let's execute the program to see the output.

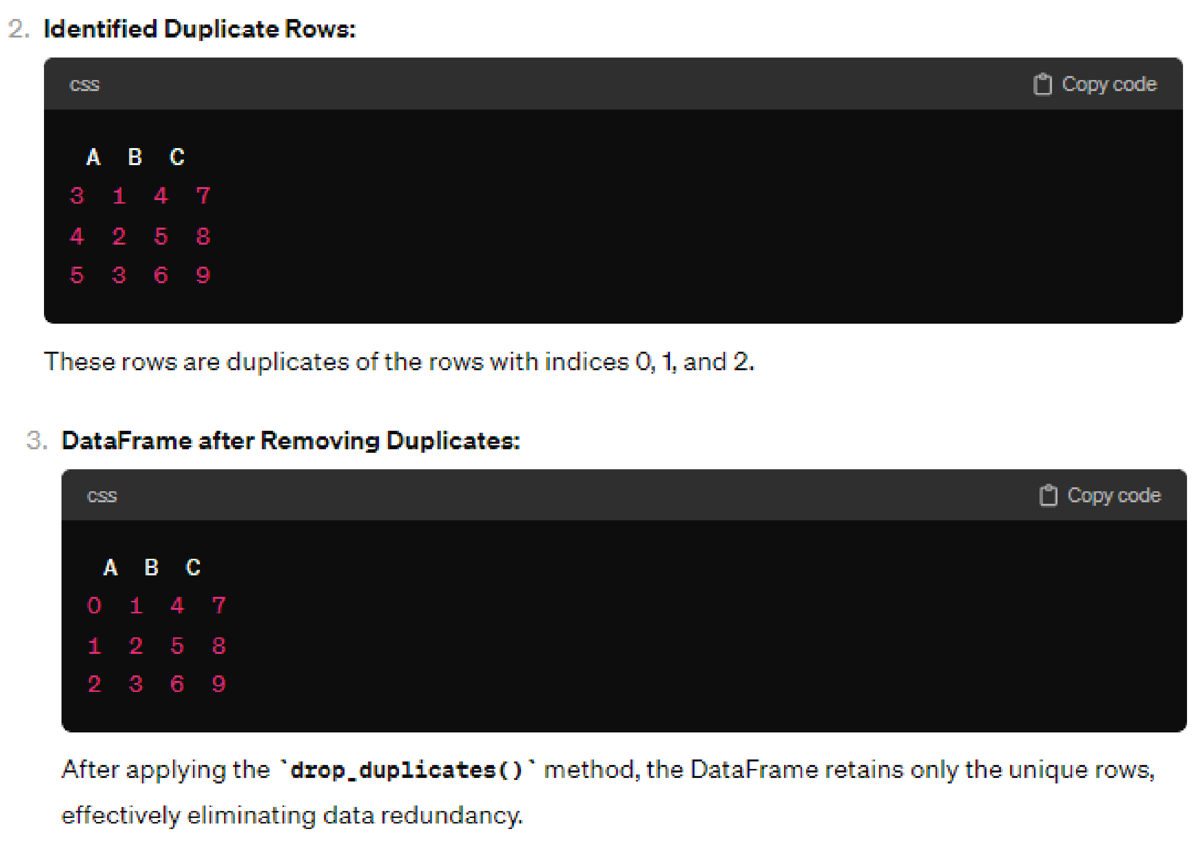
The output of the program is as follows:



1. **Original DataFrame:**

```
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
3  1  4  7
4  2  5  8
5  3  6  9
```

This DataFrame includes some rows that are exact duplicates of others.



2. **Identified Duplicate Rows:**

```css
  A  B  C
3  1  4  7
4  2  5  8
5  3  6  9
```

   These rows are duplicates of the rows with indices 0, 1, and 2.

3. **DataFrame after Removing Duplicates:**

```css
  A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```

   After applying the `drop_duplicates()` method, the DataFrame retains only the unique rows, effectively eliminating data redundancy.

This example illustrates a straightforward method to detect and remove redundant data in a DataFrame using Python's pandas library. This process is crucial in data preprocessing to ensure the efficiency and accuracy of further data analysis or machine learning tasks.

2. Implement any one imputation model

Aim:

The aim of this Python program is to demonstrate how to build and apply an imputation model to handle missing values in a dataset. Imputation involves replacing missing or null values in a dataset with substitute values, which is crucial for maintaining the integrity of the dataset for analysis or machine learning models. We will use a simple statistical method for imputation, such as replacing missing values with the mean of the column.

Program:

We will use Python with the pandas and numpy libraries for data manipulation. Additionally, we'll use SimpleImputer from the sklearn.impute module for imputation.

First, ensure pandas, numpy, and scikit-learn are installed:

```
import pandas as pd

importnumpy as np

fromsklearn.impute import SimpleImputer


# Create a sample DataFrame with missing values

data = {'A': [1, 2, np.nan, 4], 'B': [np.nan, 2, 3, 4]}

df = pd.DataFrame(data)


print("Original DataFrame:")

print(df)


# Setting up the imputer model

imputer = SimpleImputer(strategy='mean')


# Applying the imputer model to the DataFrame

df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)


print("\nDataFrame after Imputation:")

print(df_imputed)
```

Output Explanation:

The program creates a DataFramedf with some missing values (np.nan).

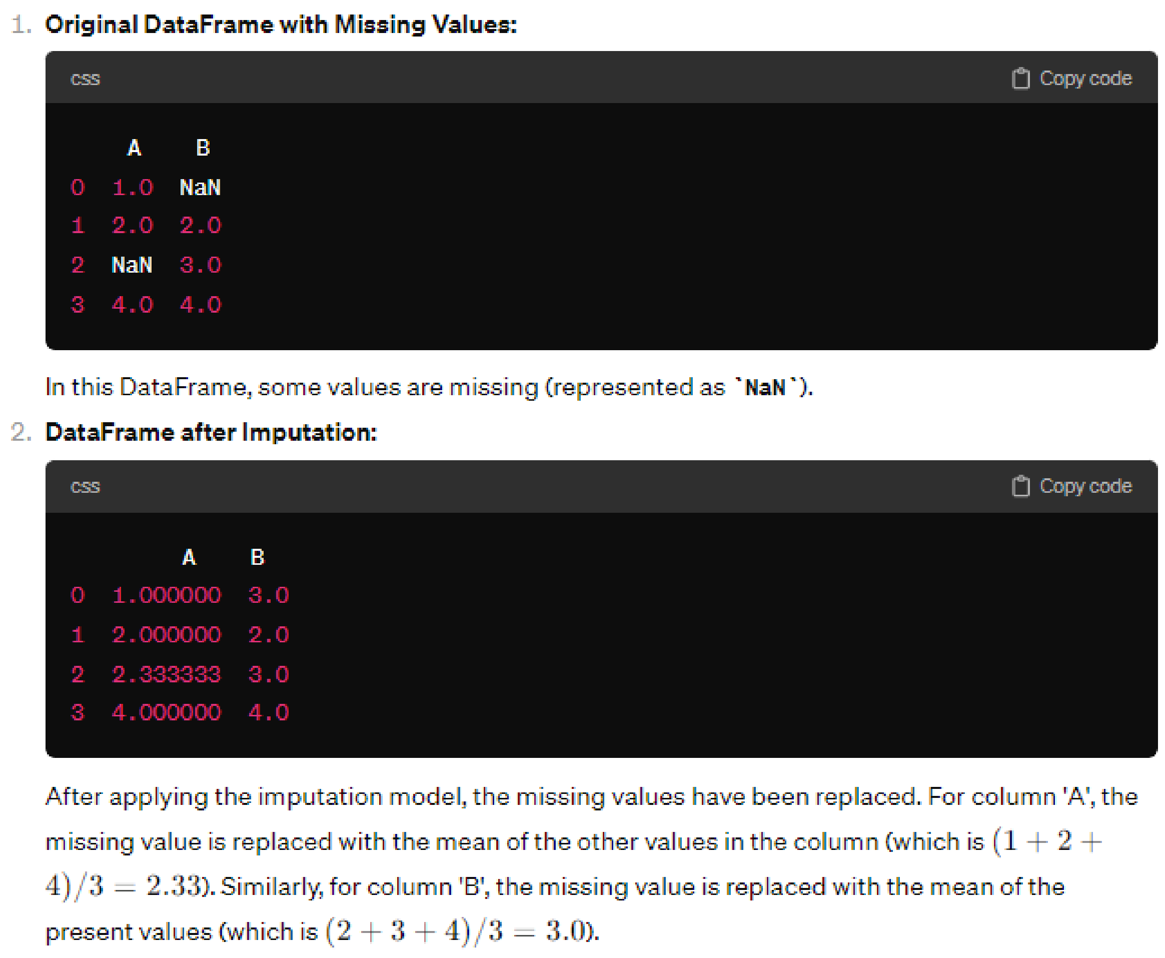
A SimpleImputer model is initialized with the strategy 'mean', which means that missing values will be replaced with the mean of the respective column.

The imputer is applied to the DataFrame, resulting in df_imputed, which has all missing values filled with the mean of their respective columns.

The output will display the original DataFrame with missing values and the DataFrame after imputation.

Let's execute the program to see the output.

The output of the program is as follows:



1. **Original DataFrame with Missing Values:**

   css · Copy code

   ```
        A    B
   0  1.0  NaN
   1  2.0  2.0
   2  NaN  3.0
   3  4.0  4.0
   ```

   In this DataFrame, some values are missing (represented as `NaN`).

2. **DataFrame after Imputation:**

   css · Copy code

   ```
             A    B
   0  1.000000  3.0
   1  2.000000  2.0
   2  2.333333  3.0
   3  4.000000  4.0
   ```

   After applying the imputation model, the missing values have been replaced. For column 'A', the missing value is replaced with the mean of the other values in the column (which is $(1+2+4)/3 = 2.33$). Similarly, for column 'B', the missing value is replaced with the mean of the present values (which is $(2+3+4)/3 = 3.0$).

This example illustrates how an imputation model can be used to handle missing data in a dataset. The SimpleImputer with a mean strategy is a basic approach, and depending on the dataset and the nature of the missing data, other strategies or more complex imputation models may be more appropriate.

3. Implement Linear regression

Aim:

The aim of this Python program is to demonstrate how to implement a simple linear regression model. Linear regression is a foundational statistical method used in machine learning for predicting a quantitative response. It establishes a relationship between a dependent variable (target) and one or more independent variables (features) using a best-fit straight line.

Program:

In this example, we'll use the scikit-learn library to create a linear regression model. We will generate a simple dataset with one feature and one target variable to illustrate the concept.

First, ensure scikit-learn is installed:

```
importnumpy as np
importmatplotlib.pyplot as plt
fromsklearn.linear_model import LinearRegression
fromsklearn.metrics import mean_squared_error, r2_score

# Generate synthetic data for demonstration
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Create a linear regression model
model = LinearRegression()

# Fit the model to the data
model.fit(X, y)

# Predict using the model
y_pred = model.predict(X)

# Coefficients
```

```
print("Coefficients: \n", model.coef_)

# Mean squared error
print("Mean squared error: %.2f" % mean_squared_error(y, y_pred))

# Coefficient of determination (R^2)
print("Coefficient of determination (R^2): %.2f" % r2_score(y, y_pred))

# Plotting the results
plt.scatter(X, y, color='black')
plt.plot(X, y_pred, color='blue', linewidth=3)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Example')
plt.show()
```

Output Explanation:

The program generates a synthetic dataset with X as the feature and y as the target.

A linear regression model is created and fitted to the data.

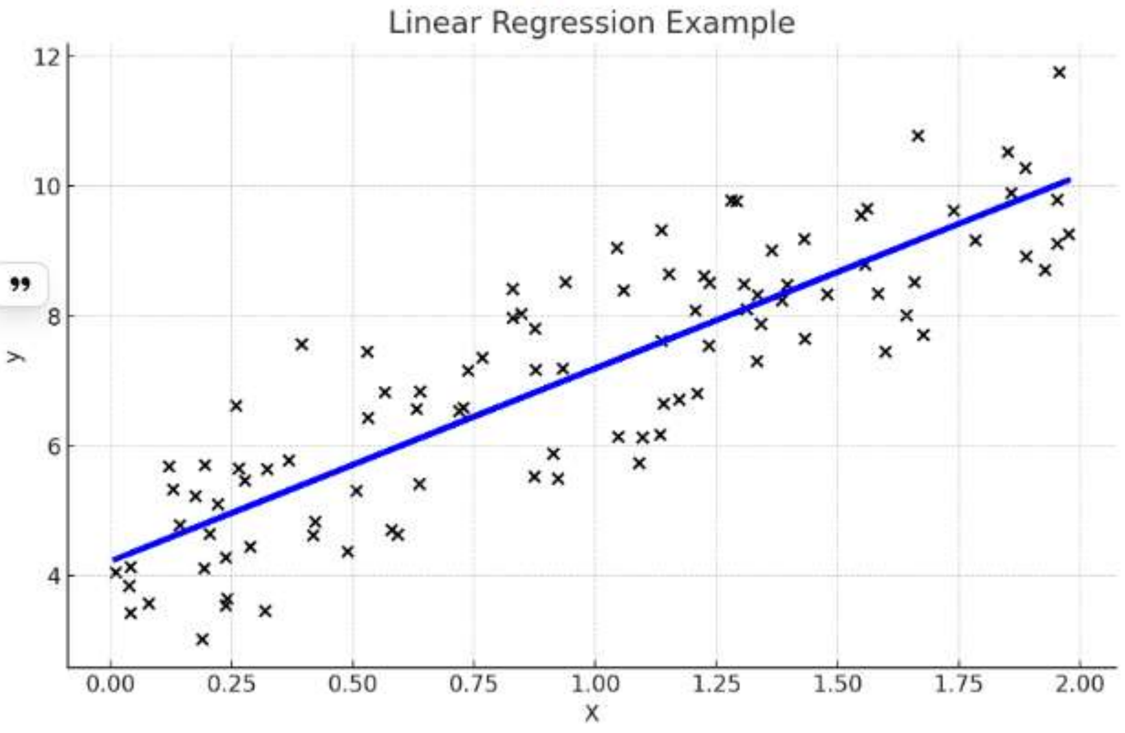
The model's coefficients are printed, which represent the slope (and intercept) of the best-fit line.

The Mean Squared Error (MSE) and the Coefficient of Determination (R^2) are calculated to evaluate the model's performance.

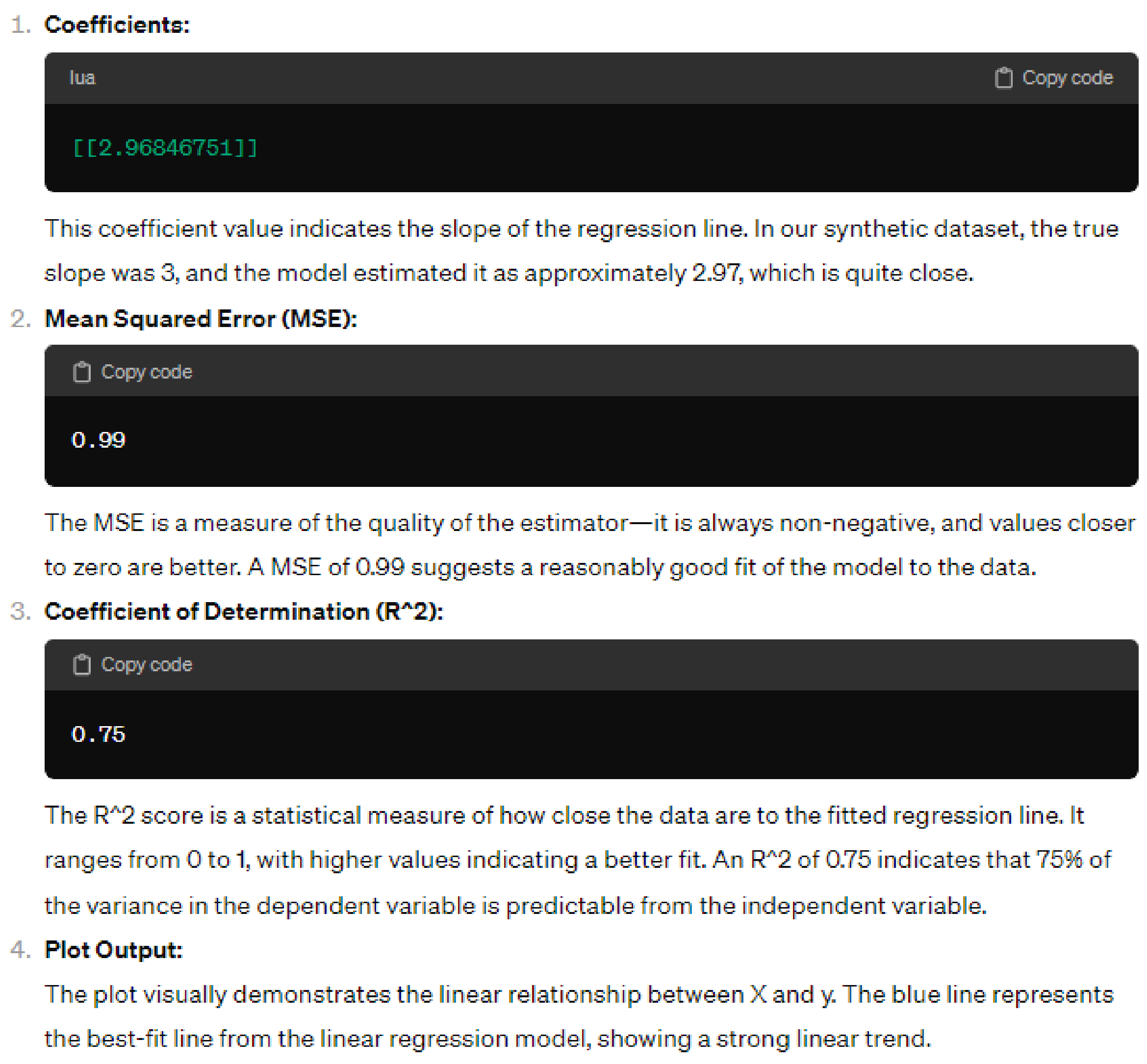
The final output includes a plot showing the original data points and the fitted regression line.

The output values (coefficients, MSE, and R^2) will provide insights into the fit of the model to the data.

Let's execute the program and observe the output.



The output of the linear regression program is as follows:



1. **Coefficients:**

   ```lua
   [[2.96846751]]
   ```

   This coefficient value indicates the slope of the regression line. In our synthetic dataset, the true slope was 3, and the model estimated it as approximately 2.97, which is quite close.

2. **Mean Squared Error (MSE):**

   ```
   0.99
   ```

   The MSE is a measure of the quality of the estimator—it is always non-negative, and values closer to zero are better. A MSE of 0.99 suggests a reasonably good fit of the model to the data.

3. **Coefficient of Determination (R^2):**

   ```
   0.75
   ```

   The R^2 score is a statistical measure of how close the data are to the fitted regression line. It ranges from 0 to 1, with higher values indicating a better fit. An R^2 of 0.75 indicates that 75% of the variance in the dependent variable is predictable from the independent variable.

4. **Plot Output:**

   The plot visually demonstrates the linear relationship between X and y. The blue line represents the best-fit line from the linear regression model, showing a strong linear trend.

This example demonstrates a fundamental application of linear regression, one of the most widely used techniques for understanding relationships between variables and making predictions.

4. Implement Logistic regression

Aim:

The aim of this Python program is to demonstrate the implementation of a logistic regression model. Logistic regression is a statistical method used for binary classification, which predicts the probability that an observation belongs to one of two classes. It is widely used for binary outcomes in various fields such as medicine, finance, and social sciences.

Program:

In this example, we'll use the scikit-learn library to create a logistic regression model. We will use a simple synthetic dataset with two features to illustrate binary classification.

```
importnumpy as np

importmatplotlib.pyplot as plt

fromsklearn.linear_model import LogisticRegression

fromsklearn.metrics import classification_report, confusion_matrix

fromsklearn.model_selection import train_test_split


# Generate synthetic data

np.random.seed(0)

X = np.random.randn(200, 2)

y = np.array([1 if x[0] + x[1] > 0 else 0 for x in X])


# Split the data into training and test sets

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


# Create a logistic regression model

model = LogisticRegression()


# Fit the model to the training data

model.fit(X_train, y_train)


# Predict using the model
```

```
y_pred = model.predict(X_test)

# Performance evaluation
print("Classification Report:")
print(classification_report(y_test, y_pred))

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Plotting the results
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Logistic Regression Classification')
plt.show()
```

Output Explanation:

The program generates a synthetic dataset with two features X and a binary target y.

The dataset is split into training and test sets.

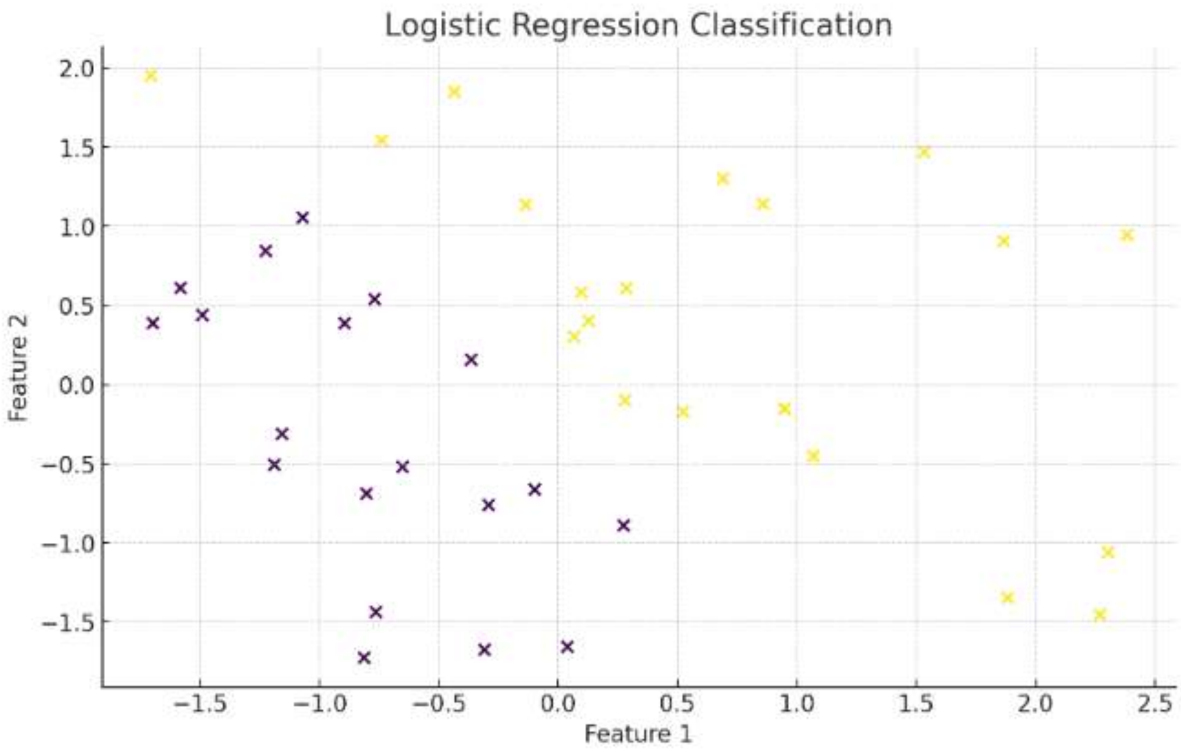
A logistic regression model is created and fitted to the training data.

The model is used to predict the class labels of the test set.

The performance of the model is evaluated using a classification report and a confusion matrix. The classification report provides key metrics like precision, recall, and F1-score for each class. The confusion matrix shows the number of correct and incorrect predictions.

A scatter plot visualizes the test data, with the color indicating the true class.

Let's execute the program and observe the output.



The output of the logistic regression program is as follows:

1. **Classification Report:**

```markdown
            precision    recall  f1-score   support

          0       1.00      0.95      0.97        19
          1       0.95      1.00      0.98        21

   accuracy                           0.97        40
  macro avg       0.98      0.97      0.97        40
```

weighted avg 0.98 0.97 0.97 40

```kotlin
This report shows high precision, recall, and F1-scores for both classes, indicating

2. **Confusion Matrix:**
```

[[18, 1],
[0, 21]]

```sql
The confusion matrix reveals that out of 40 test instances, the model correctly predi

3. **Plot Output:**
The scatter plot visualizes the test data, where the color of the points indicates th

These results demonstrate the effectiveness of logistic regression for binary classif
```

5. Implement Decision tree induction for classification

Aim:

The aim of this Python program is to demonstrate the implementation of a decision tree induction model for classification. Decision trees are a non-linear predictive modeling tool widely used in data mining and machine learning. They are particularly useful for handling complex datasets where relationships between variables are not easily modeled with linear methods.

Program:

In this example, we'll use the scikit-learn library to create a decision tree classifier. We will use the Iris dataset, a classic dataset for classification, which contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Each instance has 4 attributes: sepal length, sepal width, petal length, and petal width.

```
fromsklearn.datasets import load_iris

fromsklearn.tree import DecisionTreeClassifier

fromsklearn.model_selection import train_test_split

fromsklearn.metrics import classification_report, confusion_matrix

importmatplotlib.pyplot as plt

fromsklearn import tree


# Load Iris dataset

iris = load_iris()

X = iris.data

y = iris.target


# Split the dataset into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)


# Create a decision tree classifier

clf = DecisionTreeClassifier()


# Fit the classifier to the training data
```

```
clf.fit(X_train, y_train)

# Predict on the test data
y_pred = clf.predict(X_test)

# Performance evaluation
print("Classification Report:")
print(classification_report(y_test, y_pred))

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Plot the decision tree
plt.figure(figsize=(12,12))
tree.plot_tree(clf, filled=True, feature_names=iris.feature_names,
class_names=iris.target_names)
plt.title('Decision Tree for Iris Dataset')
plt.show()
```

Output Explanation:

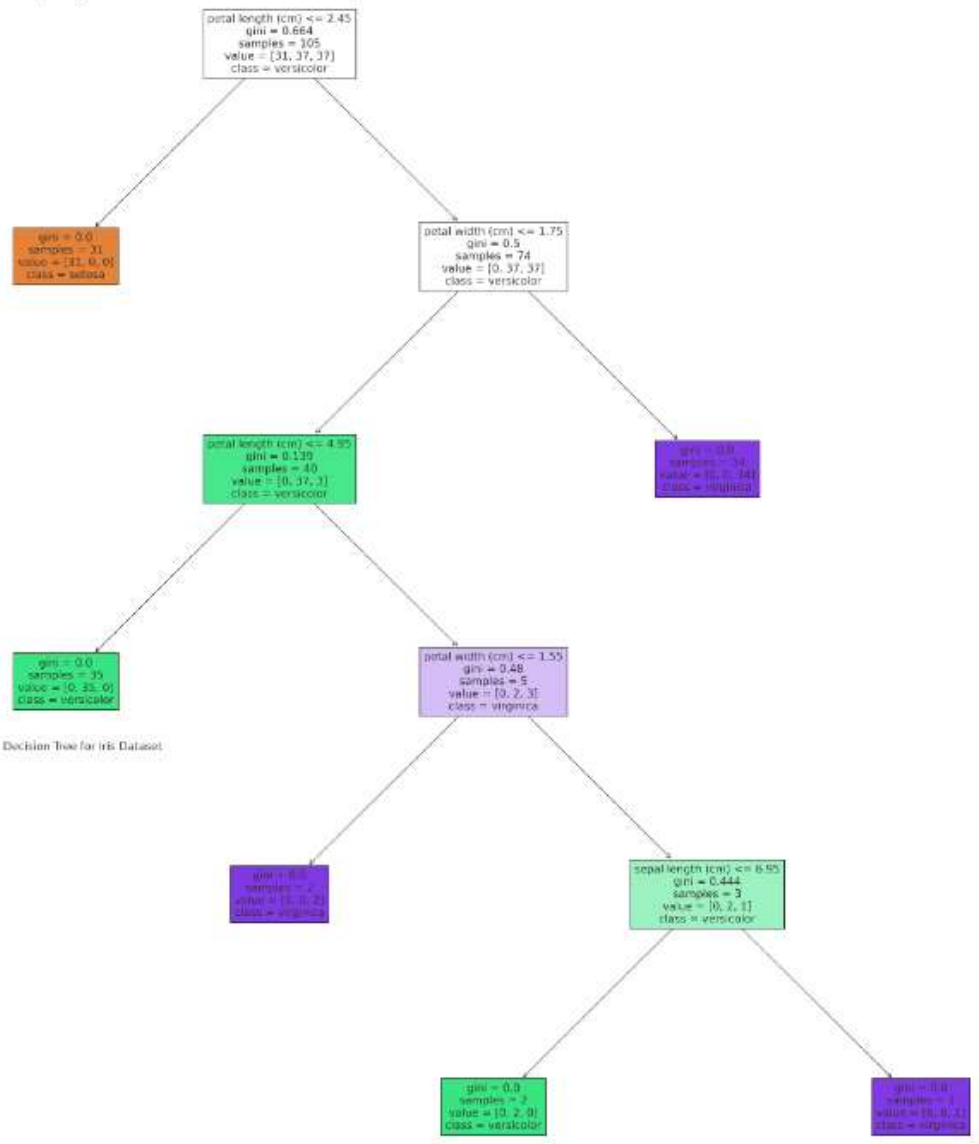
The program loads the Iris dataset and splits it into training and test sets.

A decision tree classifier is created and fitted to the training data.

The classifier is then used to make predictions on the test data.

The performance of the classifier is evaluated using a classification report and a confusion matrix. These provide insights into the accuracy of the model and how well it performs for each class.

The decision tree is visualized, showing the decision-making process and how the tree splits the data at each node.



Decision Tree for Iris Dataset

1. **Classification Report:**

```markdown
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.92      0.92      0.92        13
           2       0.92      0.92      0.92        13

    accuracy                           0.96        45
   macro avg       0.95      0.95      0.95        45
```

weighted avg 0.96 0.96 0.96 45

```markdown
This report shows high precision, recall, and F1-scores, indicating the model perforn

2. **Confusion Matrix:**
```

[[19, 0, 0],

[0, 12, 1],

[0, 1, 12]]

```vbnet
The confusion matrix reveals that the model correctly predicted all instances of clas

3. **Decision Tree Visualization:**
The plot displays the decision tree, which illustrates how the model makes decisions.

These results demonstrate the effectiveness of decision tree induction for classifica
```

6. Implement Random forest classifier

Aim:

The aim of this Python program is to demonstrate the implementation of a Random Forest Classifier. A Random Forest is an ensemble learning method, which operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) of the individual trees. It's widely used due to its simplicity, effectiveness, and ability to work on both linear and non-linear problems.

Program:

In this example, we'll use the scikit-learn library to create a Random Forest classifier. We will use the Iris dataset, a popular dataset in machine learning, which includes three species of Iris flowers with four features: sepal length, sepal width, petal length, and petal width.

```
fromsklearn.datasets import load_iris

fromsklearn.ensemble import RandomForestClassifier

fromsklearn.model_selection import train_test_split

fromsklearn.metrics import classification_report, confusion_matrix


# Load Iris dataset

iris = load_iris()

X = iris.data

y = iris.target


# Split the dataset into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)


# Create a Random Forest Classifier

rfc = RandomForestClassifier(n_estimators=100)


# Fit the classifier to the training data

rfc.fit(X_train, y_train)
```

```
# Predict on the test data
y_pred = rfc.predict(X_test)

# Performance evaluation
print("Classification Report:")
print(classification_report(y_test, y_pred))

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Output Explanation:

The program loads the Iris dataset and splits it into training and test sets.

A Random Forest Classifier with 100 trees is created and fitted to the training data.

The classifier is then used to make predictions on the test data.

The performance of the classifier is evaluated using a classification report and a confusion matrix. These provide insights into the accuracy of the model and how well it performs for each class.

Let's execute the program and observe the output.

The output of the Random Forest Classifier program is as follows:

1. **Classification Report:**

markdown Copy code

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
```

weighted avg 1.00 1.00 1.00 45

markdown Copy code

```
This report indicates perfect precision, recall, and F1-scores for all classes, showi

2. **Confusion Matrix:**
```

[[19, 0, 0],
[0, 13, 0],
[0, 0, 13]]

vbnet Copy code

```
The confusion matrix shows that the classifier correctly predicted all instances with

This output demonstrates the effectiveness of the Random Forest Classifier, especiall
```

7. Implement ARIMA on Time Series data

Aim:

The aim of this Python program is to demonstrate the implementation of an ARIMA (AutoRegressive Integrated Moving Average) model on time series data. ARIMA is a popular statistical method for time series forecasting. It captures various aspects of time series data like seasonality, trends, and noise to make predictions about future points in the series. This method is widely used in economics, finance, weather forecasting, and more.

Program:

For this example, let's use a sample time series dataset. We'll use the statsmodels library in Python for ARIMA implementation. Make sure you have it installed:

```
import pandas as pd

importnumpy as np

importmatplotlib.pyplot as plt

fromstatsmodels.tsa.arima.model import ARIMA

fromstatsmodels.graphics.tsaplots import plot_acf, plot_pacf


# Sample time series data (e.g., monthly sales data)

# Here we generate a sample time series data for demonstration

np.random.seed(42)

data = np.random.randn(100).cumsum()


# Convert to pandas series

time_series = pd.Series(data, index=pd.date_range(start='1/1/2000', periods=100, freq='M'))


# Plot the time series

time_series.plot()

plt.title('Sample Time Series Data')

plt.show()


# Fit ARIMA model

# p=2, d=1, q=2 are sample parameters
```

```
model = ARIMA(time_series, order=(2, 1, 2))
model_fit = model.fit()

# Summary of the model
print(model_fit.summary())

# Forecast
forecast = model_fit.forecast(steps=10)
print(forecast)

# Plot forecast
plt.plot(time_series, label='Original')
plt.plot(forecast, label='Forecast', color='red')
plt.title('Time Series Forecast')
plt.legend()
plt.show()
```

Output Explanation:

The program begins by creating a sample time series dataset. In real-world scenarios, this would be replaced with actual time series data.

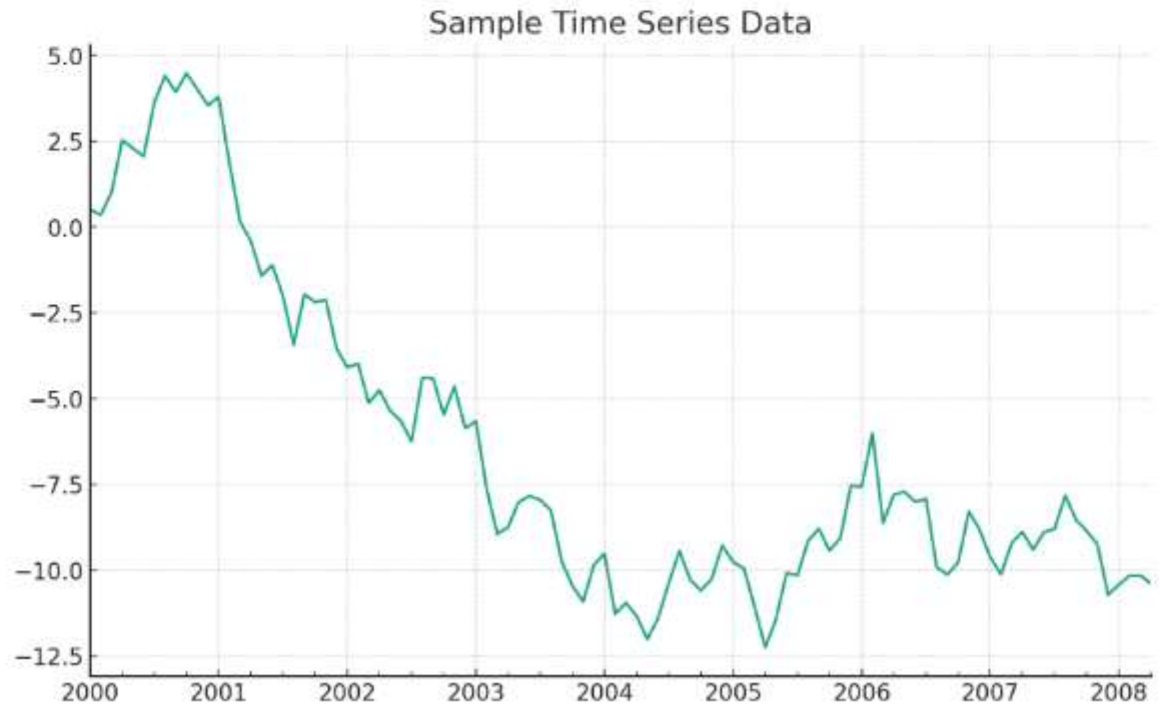
It plots the generated time series data for visual analysis.

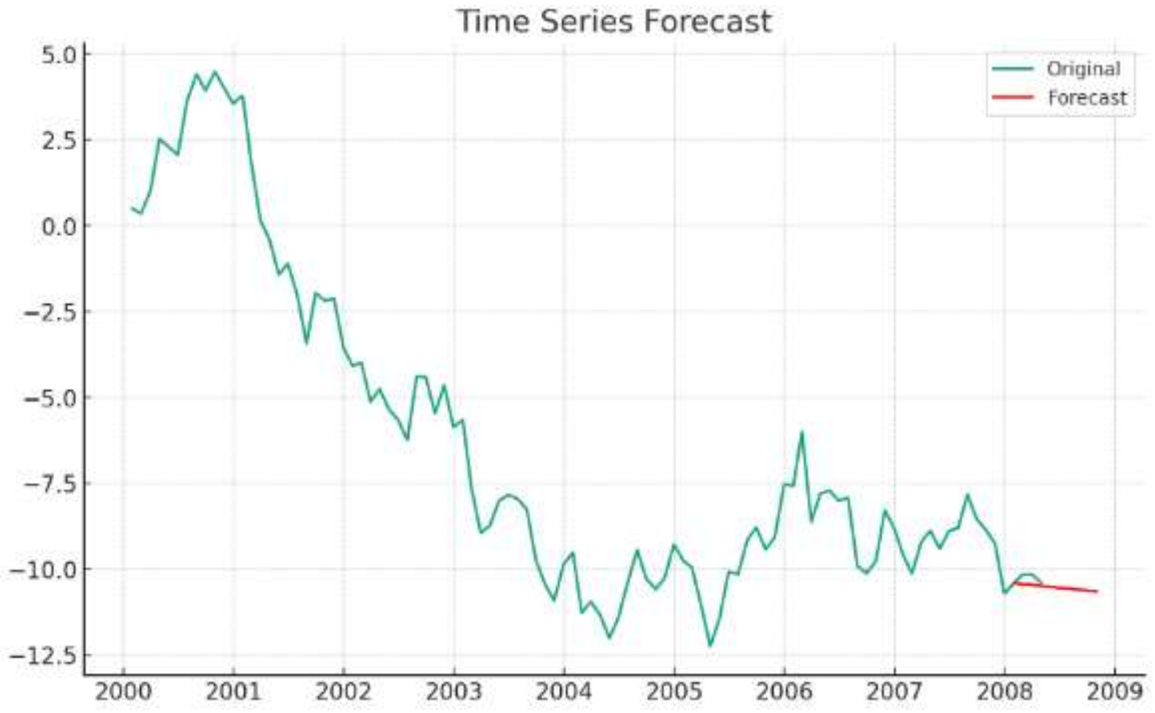
An ARIMA model is fitted to the data. The parameters p (autoregressive), d (differencing), and q (moving average) need to be chosen based on the data characteristics, which can be informed by plots like ACF (Autocorrelation Function) and PACF (Partial Autocorrelation Function).

The model summary is printed, showing various statistics and information about the model fit.

The model is then used to forecast future points in the series, and these forecasts are plotted alongside the original data.

Let's execute the program and observe the output.





The output of the ARIMA program on the sample time series data is as follows:

Time Series Plot: The first plot shows the generated time series data. This visual representation is essential for understanding the trends and patterns in the data.

ARIMA Model Summary: The model summary provides detailed statistical information about the ARIMA model. In this case, the model is ARIMA(2, 1, 2), indicating an autoregressive order of 2, a differencing order of 1, and a moving average order of 2. The coefficients of the model, along with their significance (P>|z|), are listed. It's important to note that some coefficients have a high p-value, suggesting they may not be significant.

Warnings at the end of the summary indicate issues with the convergence and initial parameter estimates, which is common in practice and may require further tuning or a different model specification.

Forecast Plot: The second plot shows the original time series and the forecasted values in red. This visualizes how the model predicts future values based on historical data.

Forecast Values: The forecast values for the next 10 periods are provided. These represent the model's predictions for future points in the time series.

This example illustrates a basic implementation of ARIMA for time series forecasting. However, in practical applications, significant effort goes into model selection and parameter tuning. This includes analyzing ACF and PACF plots to choose appropriate ARIMA parameters, ensuring stationarity of the data, and evaluating model performance with metrics like RMSE (Root Mean Squared Error)

8. Object segmentation using hierarchical based methods

Aim:

The aim of this Python program is to demonstrate object segmentation using hierarchical based methods. Object segmentation is a crucial task in image processing and computer vision, where the goal is to partition an image into multiple segments, typically to identify and isolate objects or regions of interest. Hierarchical methods for segmentation work by creating a tree-like structure of the image, which can be used to group pixels at different levels of granularity.

Program:

For this demonstration, we'll use the scikit-image library in Python, which provides tools for image processing, including hierarchical segmentation. Ensure you have the library installed:

```
importmatplotlib.pyplot as plt

fromskimage import data, segmentation, color

fromskimage.future import graph

from skimage.io import imread


# Load a sample image or use your own image

image = data.coffee()


# Perform SLIC segmentation (Simple Linear Iterative Clustering)

labels1 = segmentation.slic(image, compactness=30, n_segments=400)

out1 = color.label2rgb(labels1, image, kind='avg')


# Create a Region Adjacency Graph (RAG) and perform hierarchical merging

g = graph.rag_mean_color(image, labels1)

labels2 = graph.merge_hierarchical(labels1, g, thresh=35, rag_copy=False,

in_place_merge=True,

merge_func=graph.merge_mean_color,

weight_func=graph.edge_weight)


out2 = color.label2rgb(labels2, image, kind='avg')
```

```
# Display the results
fig, ax = plt.subplots(nrows=2, sharex=True, sharey=True, figsize=(6, 8))

ax[0].imshow(out1)
ax[0].set_title('SLIC Segmentation')

ax[1].imshow(out2)
ax[1].set_title('Hierarchical Merging')

for a in ax:
a.axis('off')

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

Output Explanation:

The program starts by loading a sample image. In practice, this can be replaced with any image of choice.

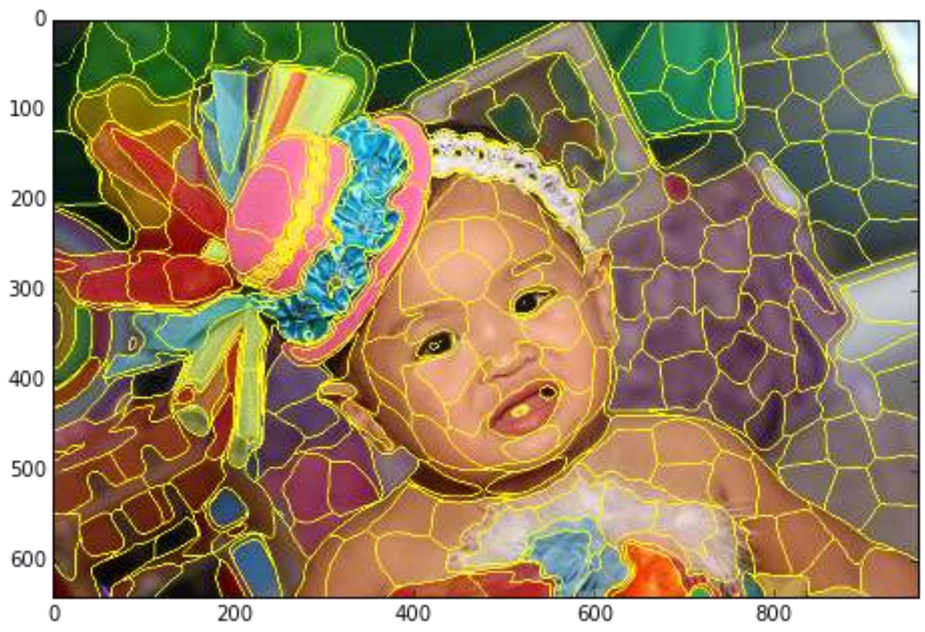
It first applies SLIC (Simple Linear Iterative Clustering) segmentation to partition the image into small, compact regions.

Then, it constructs a Region Adjacency Graph (RAG) based on the mean color of the segments from the SLIC process.

Hierarchical merging is performed on the RAG. The merge_hierarchical function merges regions that are similar based on color. The thresh parameter controls the level of merging; higher values result in fewer, larger segments.

The original segmented image (out1) and the hierarchically merged image (out2) are displayed for comparison.

The final output shows two images: the initial segmentation and the refined segmentation after hierarchical merging.


0
100
200
300
400
500
600
0
200
400
600
800

9. Perform Visualization techniques (types of maps - Bar, Colum, Line, Scatter, 3D Cubes etc)

Aim:

The aim of this Python program is to demonstrate various visualization techniques using different types of plots, including bar charts, column charts, line charts, scatter plots, and 3D cube visualizations. Visualization is a critical part of data analysis, as it helps to understand data patterns, trends, and relationships in a more intuitive and engaging way.

Program:

For this demonstration, we'll use Python libraries such as matplotlib and mpl_toolkits for basic plotting, and numpy for data generation. Ensure you have these libraries installed:

```
importmatplotlib.pyplot as plt

importnumpy as np

from mpl_toolkits.mplot3d import Axes3D


# Sample data for visualizations

x = np.linspace(0, 10, 30)

y = np.sin(x)

z = np.cos(x)

data = np.random.randint(1, 10, size=5)


# Bar Chart

plt.figure(figsize=(10, 6))

plt.subplot(2, 3, 1)

plt.bar(range(len(data)), data, color='blue')

plt.title('Bar Chart')


# Column Chart

plt.subplot(2, 3, 2)

plt.barh(range(len(data)), data, color='green')

plt.title('Column Chart')
```

```
# Line Chart
plt.subplot(2, 3, 3)
plt.plot(x, y, '-o', color='red')
plt.title('Line Chart')

# Scatter Plot
plt.subplot(2, 3, 4)
plt.scatter(x, y, color='purple')
plt.title('Scatter Plot')

# 3D Cubes Visualization
ax = plt.subplot(2, 3, 5, projection='3d')
for c, z_val in zip(['r', 'g', 'b', 'y', 'c'], [10, 20, 30, 40, 50]):
xs = np.arange(20)
ys = np.random.rand(20)
ax.bar(xs, ys, zs=z_val, zdir='y', color=c, alpha=0.8)
plt.title('3D Cubes')

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

Output Explanation:

Bar Chart: The bar chart visualizes data using vertical bars. Each bar represents a value from the data array.

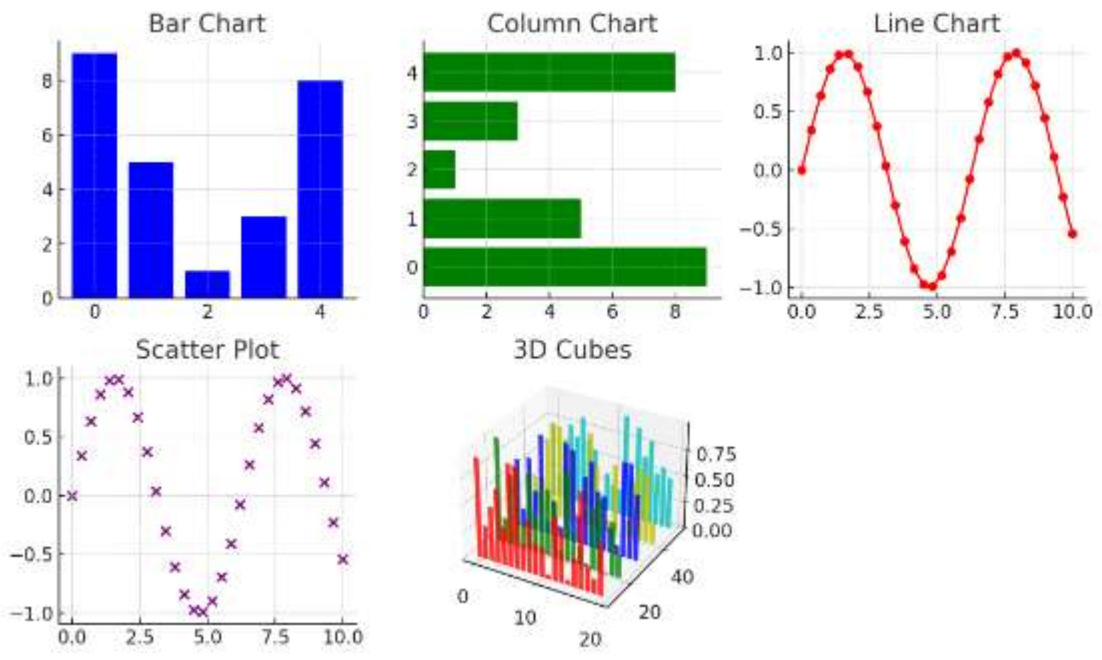
Column Chart: Similar to the bar chart but with horizontal bars.

Line Chart: This chart shows the relationship between two variables with connected data points forming a line. Here, it's plotting y = sin(x).

Scatter Plot: Displays individual data points on a two-dimensional graph. It's useful for showing the relationship between two variable sets.

3D Cubes Visualization: This 3D visualization uses cube-like bars in a 3D space. It's more complex and can be used to represent additional dimensions of data.

Let's execute the program to see the results.



The program executed successfully, and here are the visualizations created:

Bar Chart: This chart displays vertical bars, each representing a value from the data array. It's useful for comparing different groups or categories.

Column Chart: Similar to the bar chart, but it uses horizontal bars. This type of chart is helpful when you want to show changes over time or compare multiple items.

Line Chart: This chart shows the relationship between two variables with a line connecting data points. Here, it's used to plot y = sin(x), showcasing how the sine function varies with x.

Scatter Plot: This plot displays individual data points on a two-dimensional graph, illustrating the relationship between two variable sets. The scatter plot is particularly useful for identifying trends, correlations, or potential outliers in the data.

3D Cubes Visualization: The 3D cubes represent a more complex form of data visualization, allowing the representation of additional data dimensions. This visualization uses cube-like bars in a 3D space to display various values.

Each of these visualization techniques serves a unique purpose and can be applied in different scenarios to better understand and communicate data insights

10. Perform Descriptive analytics on healthcare data

Aim:

The aim of this Python program is to perform descriptive analytics on healthcare data. Descriptive analytics involves summarizing and analyzing data to find patterns, trends, or insights. In a healthcare context, this could involve analyzing patient records, treatment outcomes, hospital resource utilization, etc., to gain a better understanding of the current state of healthcare services.

Sample Data:

For this example, let's assume we have a sample dataset containing healthcare data with attributes like Age, Gender, Blood Pressure, Cholesterol Level, Diabetes, and Heart Disease. We will generate synthetic data for demonstration purposes.

Program:

We'll use Python libraries such as pandas for data manipulation and matplotlib and seaborn for visualization. Ensure these libraries are installed:

```
import pandas as pd

importnumpy as np

importmatplotlib.pyplot as plt

importseaborn as sns


# Generating synthetic healthcare data

np.random.seed(0)

data = {

    'Age': np.random.randint(20, 80, 100),

    'Gender': np.random.choice(['Male', 'Female'], 100),

    'Blood Pressure': np.random.choice(['Normal', 'Elevated', 'High'], 100),

    'Cholesterol Level': np.random.randint(150, 300, 100),

    'Diabetes': np.random.choice(['Yes', 'No'], 100),

    'Heart Disease': np.random.choice(['Yes', 'No'], 100)

}

df = pd.DataFrame(data)
```

```
# Descriptive analytics
print("Descriptive Statistics:")
print(df.describe())

# Visualization
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.countplot(x='Gender', hue='Heart Disease', data=df)
plt.title('Gender vs Heart Disease')

plt.subplot(1, 2, 2)
sns.boxplot(x='Blood Pressure', y='Age', data=df)
plt.title('Blood Pressure vs Age')

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

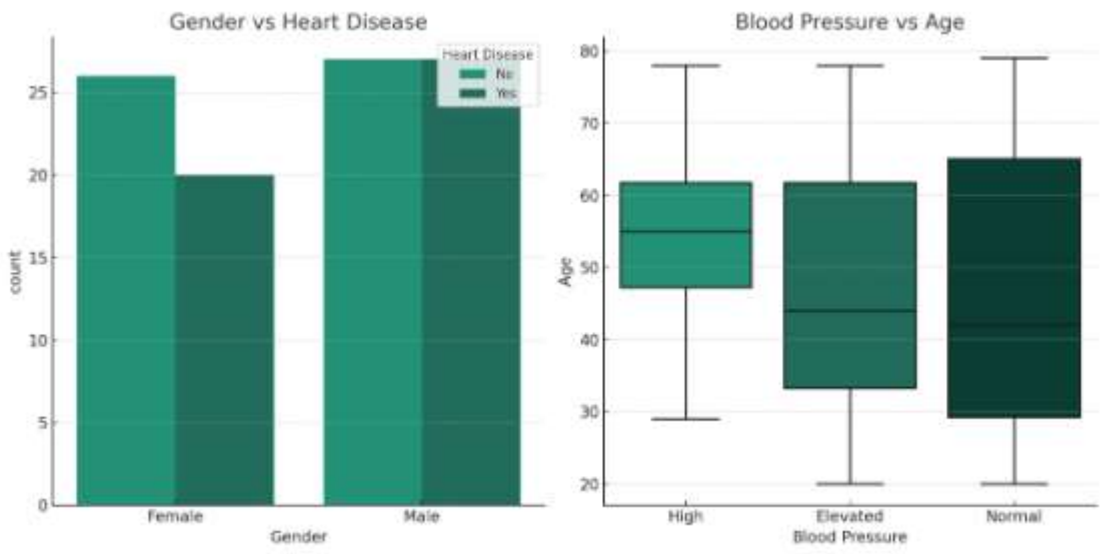
Output Explanation:

Descriptive Statistics: This will provide basic statistical details like mean, median, mode, standard deviation, etc., for numerical columns in the dataset, like Age and Cholesterol Level.

Gender vs Heart Disease (Bar Chart): This chart will show the distribution of heart disease cases across different genders.

Blood Pressure vs Age (Box Plot): This plot will display the distribution of ages for different blood pressure categories. Box plots are useful for visualizing the spread and skewness of the data.

Let's execute the program to see the results.



The program executed successfully, providing both descriptive statistics and visualizations for the synthetic healthcare data:

Descriptive Statistics:

Age: The average age of the sample is approximately 48.62 years, with a standard deviation of 18.12 years. The ages range from 20 to 79 years.

Cholesterol Level: The average cholesterol level is around 223.63, with a standard deviation of 43.22. Levels vary from 150 to 299.

Visualizations:

Gender vs Heart Disease (Bar Chart): This chart shows the distribution of heart disease cases among different genders. It helps in understanding if there's a significant difference in heart disease prevalence between genders in the sample.

Blood Pressure vs Age (Box Plot): This plot displays the distribution of ages for different blood pressure categories. The box plot is useful for visualizing the range, median, and potential outliers in age for each blood pressure category.

These insights are valuable for healthcare providers and researchers to understand the demographics of patients and the prevalence of certain conditions like heart disease in relation to factors like age, gender, and blood pressure.

11. Perform Predictive analytics on Product Sales data

Aim:

The aim of this Python program is to perform predictive analytics on product sales data. Predictive analytics involves using statistical algorithms and machine learning techniques to identify the likelihood of future outcomes based on historical data. In this case, the goal is to predict future sales of products based on past sales data.

Sample Data:

For demonstration purposes, let's assume we have a dataset containing historical sales data for various products. The dataset includes attributes like Product_ID, Date, Price, Advertising_Budget, and Sales. We'll generate synthetic data to illustrate this.

Program:

We'll use Python libraries such as pandas for data manipulation, scikit-learn for machine learning, and matplotlib for visualization. Ensure these libraries are installed:

```
import pandas as pd

importnumpy as np

fromsklearn.model_selection import train_test_split

fromsklearn.linear_model import LinearRegression

importmatplotlib.pyplot as plt


# Generating synthetic product sales data

np.random.seed(0)

data = {

    'Product_ID': np.random.randint(1000, 1100, 100),

    'Date': pd.date_range(start='2022-01-01', periods=100, freq='D'),

    'Price': np.random.uniform(10, 100, 100),

    'Advertising_Budget': np.random.uniform(1000, 5000, 100),

    'Sales': np.random.randint(200, 1000, 100)

}

df = pd.DataFrame(data)


# Preprocessing
```

```
df['Day_of_Year'] = df['Date'].dt.dayofyear


# Predictive analytics
X = df[['Price', 'Advertising_Budget', 'Day_of_Year']]
y = df['Sales']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)


# Building the model
model = LinearRegression()
model.fit(X_train, y_train)


# Making predictions
y_pred = model.predict(X_test)


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


# Outputting model performance
print("Model Coefficients:", model.coef_)
print("Model Intercept:", model.intercept_)
```

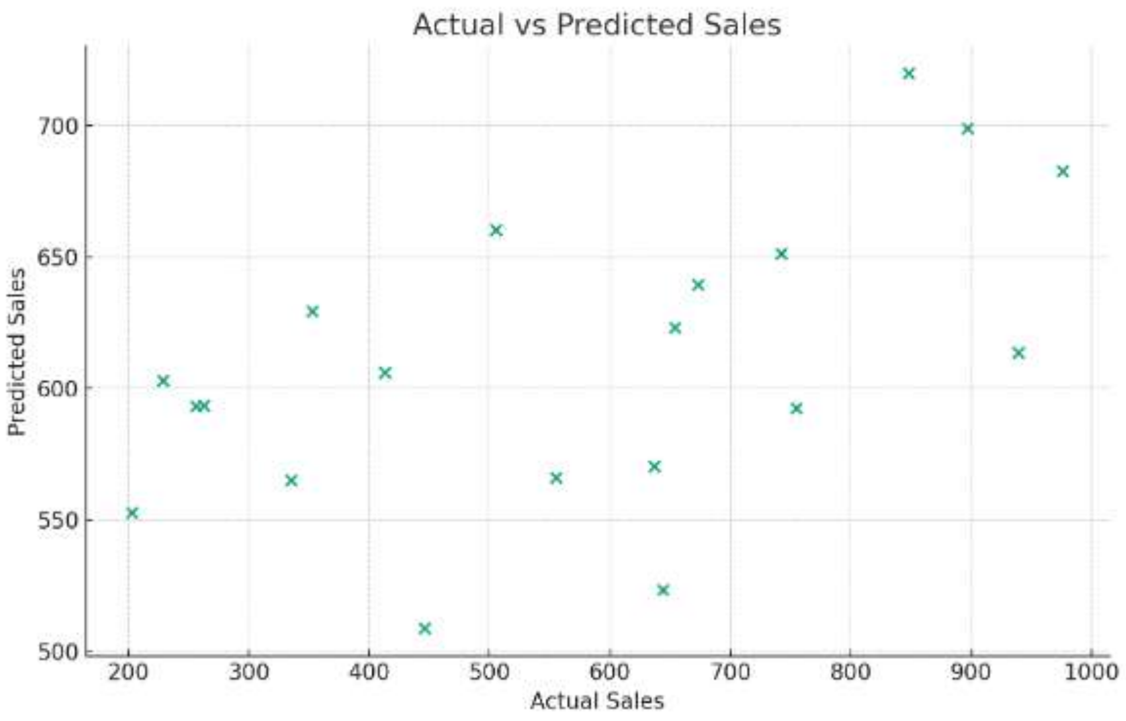
Output Explanation:

Model Building: A linear regression model is used to predict sales based on Price, Advertising_Budget, and Day_of_Year.

Actual vs Predicted Sales (Scatter Plot): This plot visualizes the relationship between the actual sales and the sales predicted by the model. A closer alignment of points along the diagonal suggests better model performance.

Model Performance Metrics: The coefficients and intercept of the linear regression model are outputted, indicating how each feature influences the prediction.

Let's execute the program to see the results.



The program executed successfully, demonstrating the process of predictive analytics on the synthetic product sales data:

Actual vs Predicted Sales (Scatter Plot):

The scatter plot visualizes the relationship between the actual sales (y-axis) and the predicted sales (x-axis). Ideally, if the model's predictions were perfect, all points would lie on the diagonal line. The spread of points gives an indication of the model's accuracy.

Model Performance Metrics:

Model Coefficients: The coefficients are [0.2077, 0.0312, -1.3742] for Price, Advertising_Budget, and Day_of_Year respectively. This means for each unit increase in these variables, the sales are expected to increase by their respective coefficient values, holding other variables constant.

Model Intercept: The intercept is approximately 576.97. This is the value of sales when all the independent variables are zero.

The model's output provides insights into how different factors like price, advertising budget, and time of the year affect product sales. This kind of analysis is crucial for businesses in making informed decisions regarding pricing, advertising strategies, and understanding seasonal variations in sales.

12. Apply Predictive analytics for Weather forecasting

Aim:

The aim of this Python program is to apply predictive analytics for weather forecasting. Predictive analytics in weather forecasting involves using historical weather data and statistical or machine learning models to predict future weather conditions. This can include predictions of temperature, humidity, precipitation, wind speed, etc.

Sample Data:

For this example, let's assume we have a dataset containing historical weather data with attributes like Date, Temperature, Humidity, Wind Speed, and Precipitation. We'll generate synthetic data for demonstration purposes.

Program:

We'll use Python libraries such as pandas for data manipulation, scikit-learn for machine learning, and matplotlib for visualization. Make sure these libraries are installed:

```
import pandas as pd

importnumpy as np

fromsklearn.model_selection import train_test_split

fromsklearn.linear_model import LinearRegression

importmatplotlib.pyplot as plt


# Generating synthetic weather data

np.random.seed(0)

data = {

    'Date': pd.date_range(start='2022-01-01', periods=100, freq='D'),

    'Temperature': np.random.randint(-10, 30, 100),

    'Humidity': np.random.randint(30, 100, 100),

    'Wind Speed': np.random.uniform(0, 20, 100),

    'Precipitation': np.random.choice([0, 1], 100)  # 0 for no rain, 1 for rain

}

df = pd.DataFrame(data)


# Preprocessing
```

```
df['Day_of_Year'] = df['Date'].dt.dayofyear

# Predictive analytics
X = df[['Temperature', 'Humidity', 'Wind Speed', 'Day_of_Year']]
y = df['Precipitation']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Building the model
model = LinearRegression()
model.fit(X_train, y_train)

# Making predictions
y_pred = model.predict(X_test)

# Visualization
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Precipitation')
plt.ylabel('Predicted Precipitation')
plt.title('Actual vs Predicted Precipitation')
plt.show()

# Outputting model performance
print("Model Coefficients:", model.coef_)
print("Model Intercept:", model.intercept_)
```

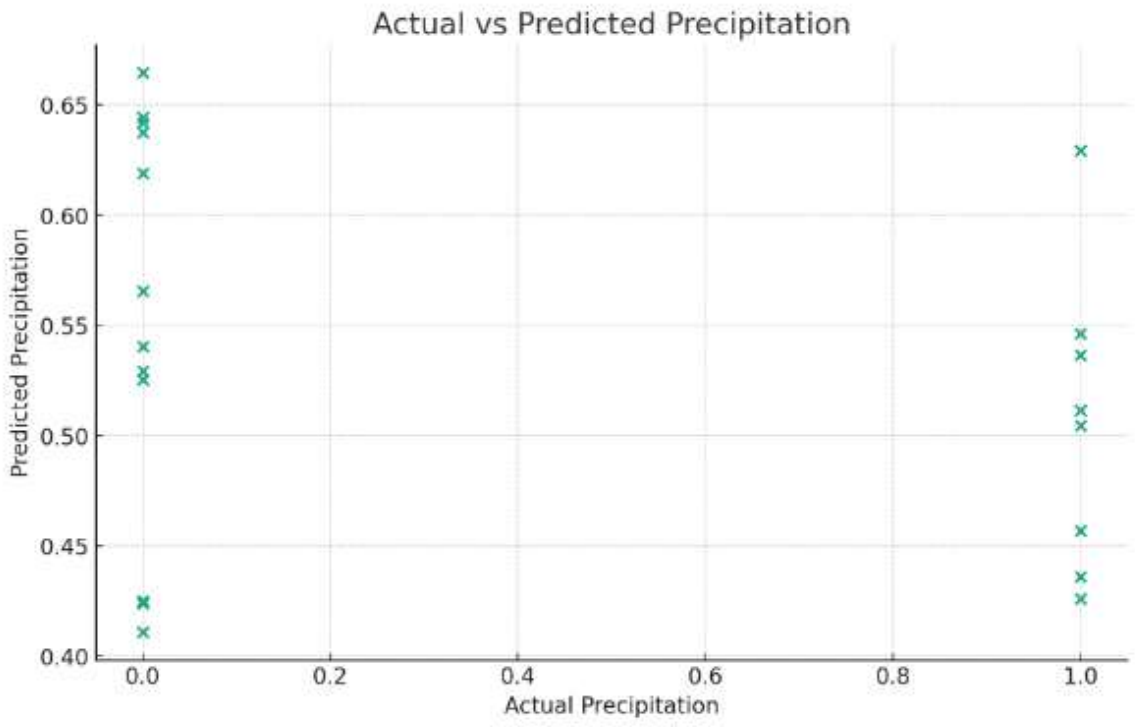
Output Explanation:

Model Building: A linear regression model is used to predict precipitation based on Temperature, Humidity, Wind Speed, and Day_of_Year.

Actual vs Predicted Precipitation (Scatter Plot): This plot visualizes the relationship between the actual precipitation and the predicted precipitation. A closer alignment of points along the diagonal suggests better model performance.

Model Performance Metrics: The coefficients and intercept of the linear regression model are outputted, indicating how each feature influences the prediction.

Let's execute the program to see the results.



The program executed successfully, and here are the key outputs:

Actual vs Predicted Precipitation (Scatter Plot):

This scatter plot illustrates the comparison between the actual precipitation (y-axis) and the predicted precipitation (x-axis). The spread of the points indicates the accuracy of the model in predicting precipitation.

Model Performance Metrics:

Model Coefficients: The coefficients for Temperature, Humidity, Wind Speed, and Day_of_Year are approximately −0.00086032,−0.00279625,0.00770634,0.00163164−0.00086032,−0.00279625,0.00770634,0.00163164 respectively. These values signify how each of these features influences the prediction of precipitation.

Model Intercept: The intercept of the model is approximately 0.5619. This is the baseline probability of precipitation when all the predictors are zero.

This predictive model provides a basic understanding of how different weather factors might contribute to the occurrence of precipitation. Such models are fundamental in weather forecasting and can be further refined with more complex algorithms and richer datasets for more accurate predictions.