1. Program on data wrangling: Combining and merging datasets, Reshaping and Pivoting.

Data wrangling is a critical process in data analysis, where data is transformed into a structured, usable format. This program demonstrates key operations in data wrangling: combining and merging datasets, reshaping, pivoting, handling missing data, and generating summary statistics.

- Combining and Merging Datasets

  Combining and merging datasets are essential for integrating multiple data sources.
  Merging: Combines datasets based on a common key using an inner join, retaining only matching rows. This ensures focused analysis on shared data points.
  Concatenation: Stacks datasets vertically, appending new records to create a unified dataset.

- Reshaping Data with Melt

Reshaping is used to change the layout of a dataset to suit specific analytical needs. The melt operation converts wide-format data into long format, turning columns into rows. This format is ideal for grouping, filtering, and visualizing data across variables.

- Pivoting Data

Pivoting reverses the melting process, converting long-format data back into wide format. It summarizes data for easier interpretation by using one column as the index and another as columns. This transformation is particularly useful for summarizing data in a matrix-like format, which is easier to interpret for certain statistical analyses.

- Handling Missing Data

Missing values are replaced with column means to ensure completeness. This maintains data integrity for further analysis.

- Summary Statistics

The program concludes by calculating summary statistics (e.g., mean, standard deviation, min, max) for the filled dataset. Summary statistics provide insights into the dataset's central tendency, dispersion, and overall distribution.

Source Code :

```
import pandas as pd
import numpy as np

# Create two sample DataFrames
sales_data_1 = pd.DataFrame({
   'OrderID': [1, 2, 3, 4],
   'Product': ['Laptop', 'Tablet', 'Smartphone', 'Headphones'],
   'Sales': [100, 200, 2000, 800]
})

sales_data_2 = pd.DataFrame({
   'OrderID': [3, 4, 5, 6],
   'Product': ['Headphones', 'Laptop', 'Smartwatch', 'Tablet'],
   'Sales': [500, 300, 200, 900]
})

# Display the DataFrames
print("Sales Data 1:\n", sales_data_1)
print("\nSales Data 2:\n", sales_data_2)

# Merge DataFrames based on 'OrderID' using an inner join
merged_data = pd.merge(sales_data_1, sales_data_2, on='OrderID', how='inner', suffixes=('_left',
'_right'))
print("\nMerged Data (Inner Join):\n", merged_data)

# Concatenate the DataFrames vertically
combined_data = pd.concat([sales_data_1, sales_data_2], ignore_index=True)
print("\nCombined Data (Concatenated Vertically):\n", combined_data)

# 2. Reshaping Data with Melt
# Create a sample DataFrame for reshaping
reshaping_data = pd.DataFrame({
   'Month': ['Jan', 'Feb', 'Mar'],
   'Product_A': [100, 150, 130],
   'Product_B': [90, 80, 120]
})

print("\nReshaping Data (Original):\n", reshaping_data)
```

```
# Melt the DataFrame to reshape it from wide to long format
melted_data = pd.melt(reshaping_data, id_vars=['Month'], var_name='Product',
value_name='Sales')
print("\nMelted Data (Long Format):\n", melted_data)

# 3. Pivoting Data

# Create a sample DataFrame for pivoting
pivot_data = pd.DataFrame({
   'Month': ['Jan', 'Jan', 'Feb', 'Feb', 'Mar', 'Mar'],
    'Product': ['Product_A', 'Product_B', 'Product_A', 'Product_B', 'Product_A',
            'Product_B'],
    'Sales': [100, 90, 150, 80, 130, 120]
})

print("\nPivot Data (Original):\n", pivot_data)

# Pivot the DataFrame to reshape it back to wide format
pivoted_data = pivot_data.pivot(index='Month', columns='Product', values='Sales')
print("\nPivoted Data (Wide Format):\n", pivoted_data)

# 4. Handling Missing Data
# Introduce some missing values
pivoted_data.loc['Feb', 'Product_A'] = np.nan
pivoted_data.loc['Mar', 'Product_B'] = np.nan
print("\nPivoted Data with Missing Values:\n", pivoted_data)
# Fill missing values with the mean of each column
filled_data = pivoted_data.fillna(pivoted_data.mean())
print("\nFilled Data (Missing Values Handled):\n", filled_data)

# 5. Summary Statistics
print("\nSummary Statistics of Filled Data:\n", filled_data.describe())
```

Output :

Sales Data 1:

	Order ID	Product	Sales
0	1	Laptop	100
1	2	Tablet	200
2	3	Smartphone	2000
3	4	Headphones	800

Sales Data 2:

	Order ID	Product	Sales
0	3	Headphones	500
1	4	Laptop	300
2	5	Smartwatch	200
3	6	Tablet	900

Merged Data (Inner Join):

	Order ID	Product_left	Sales_left	Product_right	Sales_right
0	3	Smartphone	2000	Headphones	500
1	4	Headphones	800	Laptop	300

Combined Data (Concatenated Vertically):

	OrderID	Product	Sales
0	1	Laptop	100
1	2	Tablet	200
2	3	Smartphone	2000
3	4	Headphones	800
4	3	Headphones	500
5	4	Laptop	300
6	5	Smartwatch	200
7	6	Tablet	900

Reshaping Data (Original):

	Month	Product_A	Product_B
0	Jan	100	90
1	Feb	150	80
2	Mar	130	120

Melted Data (Long Format):

	Month	Product	Sales
0	Jan	Product_A	100
1	Feb	Product_A	150
2	Mar	Product_A	130
3	Jan	Product_B	90
4	Feb	Product_B	80
5	Mar	Product_B	120

Pivot Data (Original):

	Month	Product	Sales
0	Jan	Product_A	100
1	Jan	Product_B	90
2	Feb	Product_A	150
3	Feb	Product_B	80
4	Mar	Product_A	130
5	Mar	Product_B	120

Pivoted Data (Wide Format):

Product	Product_A	Product_B
Month		
Feb	150	80
Jan	100	90
Mar	130	120

Pivoted Data with Missing Values:

Product	Product_A	Product_B
Month		
Feb	NaN	80.0
Jan	100.0	90.0
Mar	130.0	NaN

Filled Data (Missing Values Handled):

Product	Product_A	Product_B
Month		
Feb	115.0	80.0
Jan	100.0	90.0
Mar	130.0	85.0

Summary Statistics of Filled Data:

Product	Product_A	Product_B
count	3.0	3.0
mean	115.0	85.0
std	15.0	5.0
min	100.0	80.0
25%	107.5	82.5
50%	115.0	85.0
75%	122.5	87.5
max	130.0	90.0

2. Program on Data Transformation: String Manipulation, Regular Expressions

Data transformation is an essential step in preprocessing text data for analysis. The program demonstrates two critical techniques: string manipulation and regular expressions (regex).

1. String Manipulation:

String manipulation involves performing operations on text data to clean or reformat it for easier analysis. Common operations demonstrated include:

- Trimming Spaces: Removes leading and trailing spaces for cleaner text.
- Changing Case: Converts text to uppercase or lowercase to maintain consistency.
- Counting Substrings: Counts occurrences of specific characters or words.
- Replacing Text: Replaces specific words or patterns with desired text.
- Finding and Splitting: Locates words in a string and splits the text into individual words.
- Checking Prefix/Suffix: Verifies if a string starts or ends with specific content.

These operations are fundamental in cleaning and reformatting raw textual data.

2. Regular Expressions (Regex):

Regex is a powerful tool for pattern matching and text extraction. Key operations include:

- Removing Special Characters: Cleans text by removing unwanted symbols while retaining meaningful content like emails.
- Converting Case: Ensures uniformity by converting text to lowercase.
- Replacing Spaces: Replaces multiple spaces with a single space for better readability.
- Pattern Matching: Finds specific patterns like words starting with vowels or extracting emails.
- Masking Sensitive Information: Replaces email addresses with placeholders to anonymize data.
- Applications: These techniques are widely used for cleaning, structuring, and processing textual datasets.

Source Code :

```
import re

#String Manupulation :

# Sample text to work with
text = "  Hello, World! Welcome to Python programming Language.  "

# 1. Strip leading and trailing spaces
clean_text = text.strip()
print(f"Original Text: '{text}'")
print(f"Text after stripping spaces: '{clean_text}'")

# 2. Convert the text to uppercase
upper_text = clean_text.upper()
print(f"\nText in uppercase: '{upper_text}'")

# 3. Convert the text to lowercase
lower_text = clean_text.lower()
print(f"\nText in lowercase: '{lower_text}'")

# 4. Count occurrences of a substring (e.g., "o")
count_o = clean_text.count("o")
print(f"\nNumber of occurrences of 'o': {count_o}")

# 5. Replace a word in the string
replaced_text = clean_text.replace("Python", "HTML")
print(f"\nText after replacing 'Python' with 'HTML': '{replaced_text}'")

# 6. Find the position of a word in the string
position_world = clean_text.find("World")
print(f"\nPosition of 'World' in the text: {position_world}")

# 7. Split the text into words (by default on spaces)
words = clean_text.split()
print(f"\nList of words in the text: {words}")

# 8. Join the words back into a single string
joined_text = " ".join(words)
```

```
print(f"\nText after joining words: '{joined_text}'")

# 9. Check if the text starts with "Hello"
starts_with_hello = clean_text.startswith("Hello")
print(f"\nDoes the text start with 'Hello'? {starts_with_hello}")

# 10. Check if the text ends with a specific word (e.g., "programming.")
ends_with_programming = clean_text.endswith("programming.")
print(f"\nDoes the text end with 'programming.'? {ends_with_programming}")
```

OUTPUT :

```
Original Text: '  Hello, World! Welcome to Python programming Language.  '
Text after stripping spaces: 'Hello, World! Welcome to Python programming Language.'

Text in uppercase: 'HELLO, WORLD! WELCOME TO PYTHON PROGRAMMING LANGUAGE.'

Text in lowercase: 'hello, world! welcome to python programming language.'

Number of occurrences of 'o': 6

Text after replacing 'Python' with 'HTML': 'Hello, World! Welcome to Data HTML programming
Language.'

Position of 'World' in the text: 7

List of words in the text: ['Hello,', 'World!', 'Welcome', 'to', 'Python', 'programming', 'Language.']

Text after joining words: 'Hello, World! Welcome to Python programming Language.'

Does the text start with 'Hello'? True

Does the text end with 'programming.'? False
```

#Regular Expressions :

```
# Sample text
text = """
John's email is [warrnerjhon@gmail.com]. He said, "Python is awesome!!"    It's a     great
language.
Another email: [xyz@gmail.com].
"""

# 1. Remove special characters except for spaces and email-related characters.
# Using regex to remove non-alphabetic characters and non-email symbols
clean_text = re.sub(r"[^a-zA-Z0-9@\.\s]", "", text)
print("Text after removing special characters:")
print(clean_text)

# 2. Convert the text to lowercase
clean_text = clean_text.lower()
print("\nText after converting to lowercase:")
print(clean_text)

# 3. Replace multiple spaces with a single space
clean_text = re.sub(r"\s+", " ", clean_text)
print("\nText after replacing multiple spaces:")
print(clean_text)

# 4. Extract all words starting with a vowel (a, e, i, o, u)
vowel_words = re.findall(r"\b[aeiouAEIOU]\w+", clean_text)
print("\nWords starting with a vowel:")
print(vowel_words)

# 5. Replace email addresses with '[abc@gmail.com]'
masked_text = re.sub(r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b",
"[abc@gmail.com]", clean_text)
print("\nText after replacing emails:")
print(masked_text)
```

Output :

```
Text after removing special characters:

Johns email is warrnerjhon@gmail.com. He said Python is awesome    Its a     great  language.
Another email xyz@gmail.com.
```

Text after converting to lowercase:

johns email is warrnerjhon@gmail.com. he said python is awesome its a great language. another email xyz@gmail.com.

Text after replacing multiple spaces:
johns email is warrnerjhon@gmail.com. he said python is awesome its a great language. another email xyz@gmail.com.

Words starting with a vowel:
['email', 'is', 'is', 'awesome', 'its', 'another', 'email']

Text after replacing emails:
johns email is [abc@gmail.com]. he said python is awesome its a great language. another email [abc@gmail.com].

3. Program on Time series: GroupBy Mechanics to display in data vector, multivariate time series and forecasting formats

Time series analysis involves working with data collected over time, helping in understanding patterns and making forecasts. The program demonstrates three key aspects: GroupBy mechanics, data formats, and forecasting.

- GroupBy Mechanics:

Time series data can be grouped to summarize and analyze trends over specific intervals (e.g., months). The program groups daily data by month using the resample method and calculates the monthly mean. This helps identify patterns or trends at a higher granularity, such as seasonal or monthly variations.

- Data Formats:

Vector Format: Displays a single variable (e.g., Value_A) as a sequence of values over time, useful for analyzing one aspect of the dataset. Multivariate Time Series: Includes multiple variables (e.g., Value_A and Value_B), allowing for the analysis of relationships between variables over time.

- Time Series Forecasting:

Uses the Holt-Winters Exponential Smoothing model to predict future values based on historical data. The program splits data into training and testing sets, fits the model to the training data, and forecasts for the test period. Results are visualized to compare actual values and predictions, aiding in decision-making.

Applications: Time series analysis is widely used in fields like finance, economics, and weather forecasting.

Source Code :

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Create sample time series data
np.random.seed(42)
date_range = pd.date_range(start="2023-04-12", end="2023-04-12", freq="D")
data = pd.DataFrame({
    "Date": date_range,
    "Value_A": np.random.normal(100, 10, len(date_range)),
    "Value_B": np.random.normal(200, 20, len(date_range)),
})

# Set Date as the index
data.set_index("Date", inplace=True)

# GroupBy Mechanics
def groupby_mechanics(data):
    print("\n--- GroupBy Mechanics ---")
    # Group data by month and calculate mean
    grouped = data.resample('M').mean()
    print(grouped)
    return grouped

# Data Formats: Vector and Multivariate
def data_formats(data):
    print("\n--- Data Formats ---")
    # Display data as vector
    print("\nVector Format:")
    print(data["Value_A"].head())

    # Display multivariate time series
    print("\nMultivariate Time Series:")
    print(data.head())

# Forecasting Example
def time_series_forecasting(data):
```

```
    print("\n--- Forecasting ---")
    # Select a single column for forecasting
    ts = data["Value_A"]

    # Train-Test Split
    train = ts[:int(0.8 * len(ts))]
    test = ts[int(0.8 * len(ts)):]

    # Fit the Holt-Winters Exponential Smoothing model
    model = ExponentialSmoothing(train, seasonal="add", seasonal_periods=30).fit()

    # Forecast for the test period
    forecast = model.forecast(len(test))

    # Plot results
    plt.figure(figsize=(12, 6))
    plt.plot(train, label="Train")
    plt.plot(test, label="Test")
    plt.plot(forecast, label="Forecast")
    plt.legend()
    plt.title("Time Series Forecasting")
    plt.show()

# Main function
if __name___== "__main__":
    print("--- Time Series Data ---")
    print(data.head())

    # Grouping Mechanics
    monthly_data = groupby_mechanics(data)

    # Data Formats
    data_formats(data)

    # Time Series Forecasting
     time_series_forecasting(data)
```

Output :

```
--- Time Series Data ---
                Value_A        Value_B
Date
2023-04-12    104.967142    200.251848
2023-04-13    98.617357     201.953522
2023-04-14    106.476885    184.539804
2023-04-15    115.230299    200.490203
2023-04-16    97.658466     209.959966

--- GroupBy Mechanics ---
c:\Users\Lenovo\Jhon\Statistics Lab\p3.py:22: FutureWarning: 'M' is deprecated and will be
removed in a future version, please use 'ME' instead.
  grouped = data.resample('M').mean()
            Value_A     Value_B
Date
2023-04-30   98.940175  201.469137
2023-05-31   97.012894  201.349620
2023-06-30  100.455494  201.436198
2023-07-31   99.067553  196.571382
2023-08-31  100.828514  199.414531
2023-09-30  100.318000  193.094705
2023-10-31  101.007366  197.896876
2023-11-30  101.699258  201.755240
2023-12-31   99.284726  205.322539
2024-01-31   99.887451  197.749011
2024-02-29  103.252043  197.832125
2024-03-31   98.903094  198.593193
2024-04-30  100.869866  193.921756

--- Data Formats ---

Vector Format:
Date
2023-04-12    104.967142
2023-04-13     98.617357
2023-04-14    106.476885
2023-04-15    115.230299
2023-04-16     97.658466
Name: Value_A, dtype: float64
```

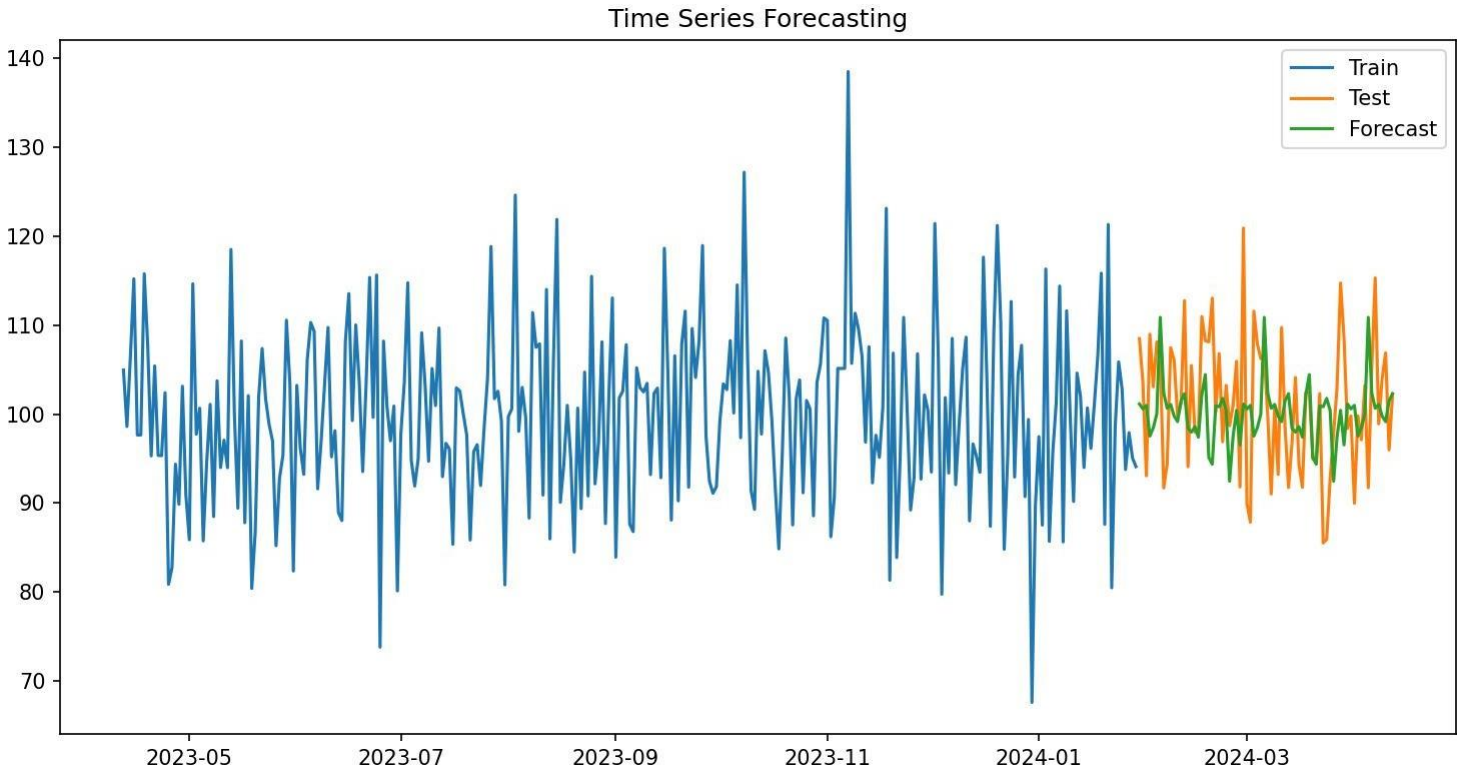
Multivariate Time Series:

Date	Value_A	Value_B
2023-04-12	104.967142	200.251848
2023-04-13	98.617357	201.953522
2023-04-14	106.476885	184.539804
2023-04-15	115.230299	200.490203
2023-04-16	97.658466	209.959966

--- Forecasting ---

C:\Users\Lenovo\AppData\Local\Programs\Python\Python312\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred frequency D will be used.

self._init_dates(dates, freq)



4. Program to measure central tendency and measures of dispersion: Mean, Median, Mode, Standard Deviation, Variance, Mean deviation and Quartile deviation for a frequency distribution/data.

The measures are essential for understanding the distribution and variability of data in a systematic way.

1. Central Tendency: These measures help identify the "center" or typical value of a dataset:

- Mean: The average of the data values, showing the overall central value.
- Median: The middle value when the data is arranged in order, representing the midpoint of the dataset.
- Mode: The most frequently occurring value in the data, showing the most common observation.

2. Dispersion: These measures describe how spread out the data is:

- Variance: Shows how much the data values differ from the mean on average.
- Standard Deviation: The square root of variance, indicating the average distance of data from the mean.
- Mean Deviation: The average of the absolute differences between data values and the mean.
- Quartile Deviation: Focuses on the variability of the middle 50% of the data.

Program Working:

- Input: The program takes two inputs: data values and their frequencies.
- Processing: It calculates the measures of central tendency (mean, median, mode) and dispersion (variance, standard deviation, etc.) using Python libraries like NumPy and pandas.
- Output: The program provides all the computed measures, giving insights into the dataset's characteristics.

Advantages of Computational Statistics:

- Efficiency: Automates complex calculations, saving time.
- Accuracy: Reduces human error in computations.

Source Code :

```
import numpy as np
import pandas as pd

def cal_sta(data , freq):
   df = pd.DataFrame({'Value': data, 'Frequency' :  freq})
   total = df['Frequency'].sum()

   df['Weighted_Value'] = df['Value'] * df['Frequency']
   mean = df['Weighted_Value'].sum() / total


   cumulative_frequency = df['Frequency'].cumsum()
   median_index = cumulative_frequency.searchsorted(total / 2)
   median = df['Value'][median_index]

   mode = df['Value'][df['Frequency'].idxmax()]


   variance = np.average((df['Value'] - mean) ** 2, weights=df['Frequency'])
   std_deviation = np.sqrt(variance)

   mean_deviation = np.average(np.abs(df['Value'] - mean), weights=df['Frequency'])


   q1 = np.percentile(data, 25)
   q3 = np.percentile(data, 75)
   quartile_deviation = (q3 - q1) / 2
   return {
      'Mean': mean,
      'Median': median,
      'Mode': mode,
      'Variance': variance,
      'Standard Deviation': std_deviation,
      'Mean Deviation': mean_deviation,
      'Quartile Deviation': quartile_deviation
   }


data_input = input("Enter the data values separated by commas (e.g., 10, 20, 30): ")
```

```
frequencies_input = input("Enter the corresponding frequencies separated by commas (e.g., 1, 2, 3): ")

data = list(map(int, data_input.split(',')))
frequencies = list(map(int, frequencies_input.split(',')))

statistics = cal_sta(data, frequencies)

for stat, value in statistics.items():
    print(f"{stat}: {value:.2f}")
```

Output :

```
Enter the data values separated by commas (e.g., 10, 20, 30): 10,11,12,13,14
Enter the corresponding frequencies separated by commas (e.g., 1, 2, 3): 1,2,1,3,2
Mean: 12.33
Median: 13.00
Mode: 13.00
Variance: 1.78
Standard Deviation: 1.33
Mean Deviation: 1.19
Quartile Deviation: 1.00
```

5. Program to perform cross validation for a given dataset to measure Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and R2 Error using Validation Set, Leave One Out Cross-Validation(LOOCV) and K-fold Cross-Validation approaches.

Cross-validation is a method to evaluate a model's performance by testing it on different subsets of data. It ensures that the model generalizes well to unseen data. The program calculates three key metrics for model evaluation:

- 1. Root Mean Squared Error (RMSE): Measures the average prediction error, emphasizing larger errors.
- 2. Mean Absolute Error (MAE): Measures the average prediction error without emphasizing outliers.
- 3. R^2 Score: Indicates how well the model explains the variability in the data.

Cross-Validation Techniques

- 1. Validation Set Approach: Splits the data into training (80%) and validation (20%). Tests the model on the validation set after training.
- 2. Leave-One-Out Cross-Validation (LOOCV): Uses one sample as the test set and the rest for training. Repeats this process for all samples.
- 3. K-Fold Cross-Validation: Divides the data into k equal parts (folds). Trains on folds and tests on the remaining fold, repeated times.

Purpose: The program evaluates a linear regression model using these techniques and calculates RMSE, MAE, and R^2 to compare performance. It ensures reliable and unbiased model evaluation.

Source Code :

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, KFold, LeaveOneOut
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.datasets import fetch_california_housing

# Load the California housing dataset
data = fetch_california_housing()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target)

# Function to calculate and display metrics
def display_metrics(y_true, y_pred):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)
    print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
    print(f"Mean Absolute Error (MAE): {mae:.4f}")
    print(f"R² Score: {r2:.4f}")
    return rmse, mae, r2

# Validation Set Approach
def validation_set_approach(X, y):
    print("Validation Set Approach:")
    # Split the dataset into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

    # Initialize and train the model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Make predictions on the validation set
    y_pred = model.predict(X_val)

    # Display metrics
    display_metrics(y_val, y_pred)
```

```
# Leave-One-Out Cross-Validation (LOOCV) Approach
def loocv_approach(X, y):
    print("Leave-One-Out Cross-Validation (LOOCV):")
    loo = LeaveOneOut()
    y_true, y_pred = [], []

    # Loop through each sample using LOOCV
    for train_index, test_index in loo.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        # Initialize and train the model
        model = LinearRegression()
        model.fit(X_train, y_train)

        # Make prediction for the single test sample
        y_pred.append(model.predict(X_test)[0])
        y_true.append(y_test.iloc[0])

    # Display metrics
    display_metrics(y_true, y_pred)

# K-Fold Cross-Validation Approach
def kfold_approach(X, y, k=5):
    print(f"{k}-Fold Cross-Validation Approach:")
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    y_true, y_pred = [], []

    # Loop through each fold
    for train_index, test_index in kf.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        # Initialize and train the model
        model = LinearRegression()
        model.fit(X_train, y_train)

        # Make predictions on the test set
        y_pred.extend(model.predict(X_test))
        y_true.extend(y_test)
```

```
    # Display metrics
    display_metrics(y_true, y_pred)

# Main function to run all approaches
def main():
    print("Cross-Validation for RMSE, MAE, and R²:\n")
    validation_set_approach(X, y)
    print("\n")
    loocv_approach(X, y)
    print("\n")
    kfold_approach(X, y, k=5)  # You can change k for different K-Fold Cross-Validation

# Execute the main function
if __name__== "__main__":
    main()
```

Output :

Cross-Validation for RMSE, MAE, and R^2:

Validation Set Approach:
Root Mean Squared Error (RMSE): 0.7456
Mean Absolute Error (MAE): 0.5332
R^2 Score: 0.5758

Leave-One-Out Cross-Validation (LOOCV):
Root Mean Squared Error (RMSE): 0.7268
Mean Absolute Error (MAE): 0.5317
R^2 Score: 0.6033

5-Fold Cross-Validation Approach:
Root Mean Squared Error (RMSE): 0.7284
Mean Absolute Error (MAE): 0.5317
R^2 Score: 0.6015

6. Program to display Normal, Binomial Poisson, Bernoulli disrtibutions for a given frequency distribution and analyze the results.

Probability distributions describe how the values of a random variable are distributed. They help in understanding the behavior of data and are essential in statistics and data analysis. The program visualizes four key probability distributions for a given frequency distribution.

Distributions Covered

- Normal Distribution:

A continuous distribution forming a bell-shaped curve. It is symmetric about the mean, and most data points cluster around the mean. Useful for modeling natural phenomena.

- Binomial Distribution:

A discrete distribution representing the number of successes in a fixed number of trials. It depends on two parameters: the number of trials () and the probability of success (). Common in scenarios like flipping a coin or rolling a die.

- Poisson Distribution:

A discrete distribution that models the number of events in a fixed interval of time or space. It is characterized by the average rate () of occurrence. Useful for modeling rare events like system failures or call arrivals.

- Bernoulli Distribution:

A discrete distribution representing a single trial with two outcomes: success or failure. It is defined by the probability of success (). Used in binary events like yes/no or true/false.

Purpose: Accepts user input for data values and their frequencies.

Visualizes the probability density function (PDF) or probability mass function (PMF) for each distribution.

Helps users compare how well each distribution fits the data.

Importance: Understanding Data: Helps identify patterns in data.

Modeling Real-World Scenarios: Simulates phenomena like natural variations or rare events.

Source Code :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, binom, poisson, bernoulli

def get_user_data():

    data_input = input("Enter the data values separated by commas (e.g., 10, 20, 30): ")
    frequencies_input = input("Enter the corresponding frequencies separated by commas (e.g., 2,
3, 4): ")

    data = list(map(int, data_input.split(',')))
    freq = list(map(int, frequencies_input.split(',')))

    return data, freq

def plot_normal_distribution(data, freq):

    mean = np.mean(data)
    std_dev = np.std(data)

    x = np.linspace(min(data), max(data), 100)
    pdf = norm.pdf(x, mean, std_dev)

    plt.plot(x, pdf, 'r-', lw=2, label='Normal Distribution')
    plt.title('Normal Distribution')
    plt.xlabel('Value')
    plt.ylabel('Probability Density')
    plt.show()

def plot_binomial_distribution(data, freq):

    n = max(data)
    p = np.mean(data) / n

    x = np.arange(0, n+1)
    pmf = binom.pmf(x, n, p)

    plt.bar(x, pmf, alpha=0.7, color='b', label='Binomial Distribution')
```

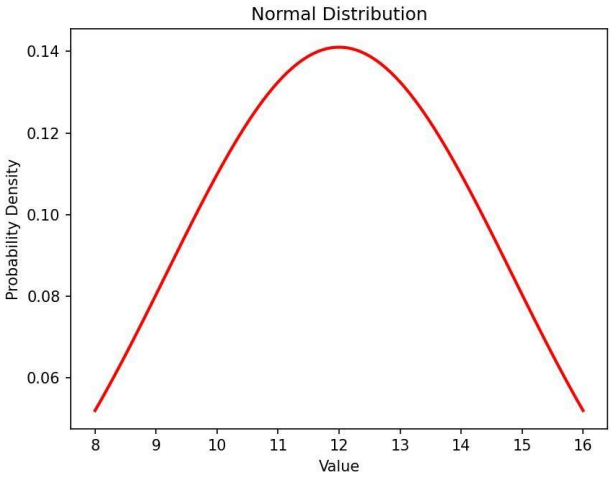
```
    plt.title('Binomial Distribution')
    plt.xlabel('Value')
    plt.ylabel('Probability')
    plt.show()

def plot_poisson_distribution(data, freq):

    lam = np.mean(data)

    x = np.arange(0, max(data)+1)
    pmf = poisson.pmf(x, lam)

    plt.bar(x, pmf, alpha=0.7, color='g', label='Poisson Distribution')
    plt.title('Poisson Distribution')
    plt.xlabel('Value')
    plt.ylabel('Probability')
    plt.show()

def plot_bernoulli_distribution(data, freq):

    success_prob = np.mean(data) / max(data)

    x = [0, 1]
    pmf = bernoulli.pmf(x, success_prob)

    plt.bar(x, pmf, alpha=0.7, color='purple', label='Bernoulli Distribution')
    plt.title('Bernoulli Distribution')
    plt.xlabel('Value')
    plt.ylabel('Probability')
    plt.show()

def analyze_distributions(data, freq):

    print("Analyzing Normal Distribution:")
    plot_normal_distribution(data, freq)

    print("Analyzing Binomial Distribution:")
    plot_binomial_distribution(data, freq)

    print("Analyzing Poisson Distribution:")
```

```
    plot_poisson_distribution(data, freq)

    print("Analyzing Bernoulli Distribution:")
    plot_bernoulli_distribution(data, freq)

data, freq = get_user_data()
analyze_distributions(data, freq)
```

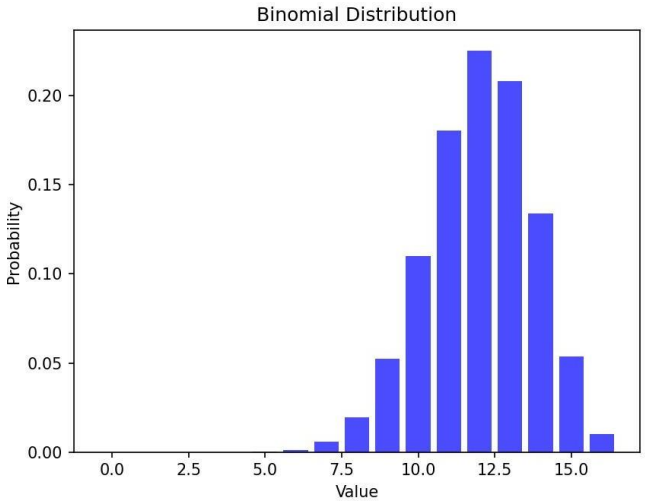
Output :

```
Enter the data values separated by commas (e.g., 10, 20, 30): 8, 10, 12, 14, 16
Enter the corresponding frequencies separated by commas (e.g., 2, 3, 4): 1, 2, 1, 3, 1
```

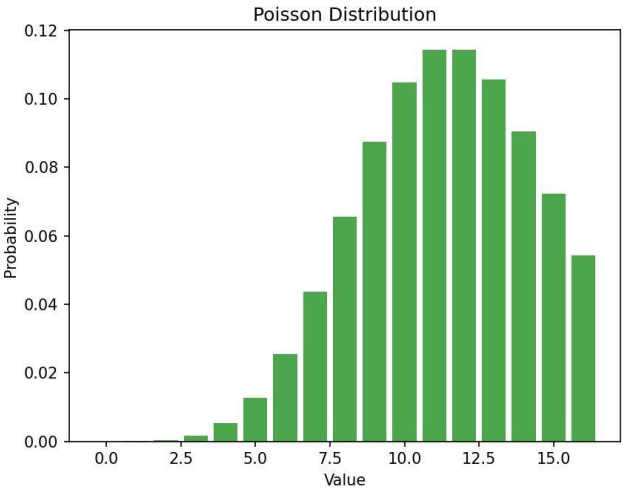
Analyzing Normal Distribution:



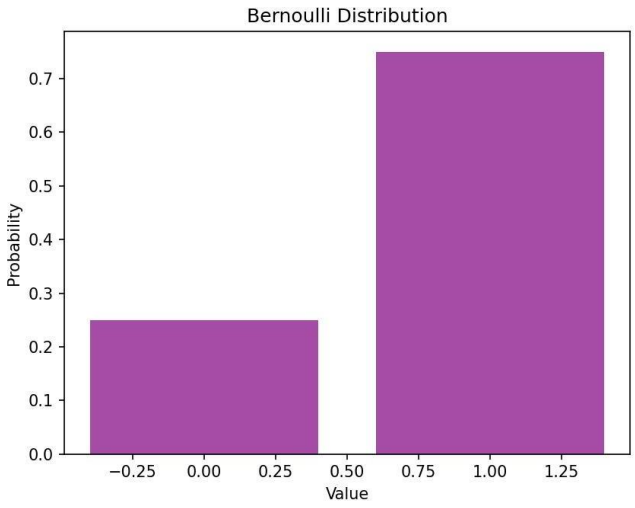
Analyzing Binomial Distribution:



Analyzing Poisson Distribution:



Analyzing Bernoulli Distribution:



7. Program to implement one sample, two sample and paired sample t-tests for a sample data and analyse the results.

T-Tests are commonly used to assess whether there is a statistically significant difference between groups or conditions. These tests help us make inferences about populations based on sample data. Types of t-tests:

1. One-Sample T-Test:

This test compares the mean of a sample to a known value (often a population mean) to determine if the sample mean is significantly different from this reference value. For eg, in the code, we compare the average exam scores of a group of students to a population mean of 85. The null hypothesis assumes there is no difference, and the alternative hypothesis suggests a difference in means.

2. Two-Sample T-Test:

This test is used to compare the means of two independent groups to determine if they differ significantly. In the code, we compare the scores of two groups (Group A and Group B). The null hypothesis suggests that there is no difference between the two groups, while the alternative hypothesis indicates a significant difference.

3. Paired-Sample T-Test:

This test compares the means of two related groups, typically measuring the same subjects before and after an intervention. In the code, we compare the scores of the same group of students before and after a treatment. The null hypothesis assumes no difference between the two sets of scores, while the alternative hypothesis suggests a significant change.

Results are Interpreted as:

- T-Statistic: This value tells us how much the sample mean differs from the hypothesized value (or the mean of the second group in case of two-sample or paired tests) in terms of standard error.
- P-Value: This value indicates the probability of observing the data if the null hypothesis were true. If the p-value is smaller than the chosen significance level (usually 0.05), we reject the null hypothesis and conclude there is a statistically significant difference.

Source Code :

```
import numpy as np
import pandas as pd
from scipy import stats

exam_scores = np.array([85, 87, 90, 78, 88, 95, 82, 79, 94, 91])

group_A = np.array([85, 89, 88, 90, 93, 85, 84, 79, 90, 87])
group_B = np.array([82, 86, 85, 87, 92, 80, 81, 78, 89, 85])

before_treatment = np.array([82, 84, 88, 78, 80, 85, 90, 79, 87, 83])
after_treatment = np.array([85, 87, 89, 81, 83, 88, 92, 82, 89, 86])

def one_sample_ttest(data, population_mean):
    t_stat, p_value = stats.ttest_1samp(data, population_mean)
    return t_stat, p_value

def two_sample_ttest(group1, group2):
    t_stat, p_value = stats.ttest_ind(group1, group2)
    return t_stat, p_value

def paired_sample_ttest(before, after):
    t_stat, p_value = stats.ttest_rel(before, after)
    return t_stat, p_value


def analyze_ttest_results(t_stat, p_value, alpha=0.05):
    print(f"T-statistic: {t_stat}")
    print(f"P-value: {p_value}")
    if p_value < alpha:
        print("Result: The null hypothesis is rejected (statistically significant difference).")
    else:
        print("Result: The null hypothesis cannot be rejected (no statistically significant difference).")


print("One-Sample T-Test:")
t_stat, p_value = one_sample_ttest(exam_scores, 85)
analyze_ttest_results(t_stat, p_value)
print()
```

```
print("Two-Sample T-Test:")
t_stat, p_value = two_sample_ttest(group_A, group_B)
analyze_ttest_results(t_stat, p_value)
print()

print("Paired-Sample T-Test:")
t_stat, p_value = paired_sample_ttest(before_treatment, after_treatment)
analyze_ttest_results(t_stat, p_value)
```

Output :

```
One-Sample T-Test:
T-statistic: 1.0189950494649807
P-value: 0.3348142605778697
Result: The null hypothesis cannot be rejected (no statistically significant difference).

Two-Sample T-Test:
T-statistic: 1.3547090246981803
P-value: 0.19227122007981406
Result: The null hypothesis cannot be rejected (no statistically significant difference).

Paired-Sample T-Test:
T-statistic: -11.758942438532781
P-value:  9.151111215642479e-07
Result: The null hypothesis is rejected (statistically significant difference).
```

8. Program to implement One-way and Two-way ANOVA tests and analyze the results

ANOVA (Analysis of Variance) is a statistical method used to test if there are significant differences between the means of multiple groups.

1. One-Way ANOVA:

Used when comparing the means of more than two groups based on one factor. It checks if the group means are significantly different. Null Hypothesis (H_o): All group means are equal. Alternative Hypothesis (H_1): At least one group mean is different.

2. Two-Way ANOVA:

Used when there are two factors, and it tests the individual effects of each factor and their interaction on the dependent variable. Null Hypothesis(H_o): Neither factor nor their interaction significantly affects response. Alternative Hypothesis (H_1): At least one factor or their interaction significantly affects the response.

Key Results:

- F-statistic: Indicates how much the group means differ.
- P-value: If less than 0.05, we reject the null hypothesis, suggesting a significant difference.

Source Code :

```
import numpy as np
import pandas as pd
from scipy.stats import f_oneway
import statsmodels.api as sm
from statsmodels.formula.api import ols

# Function for One-way ANOVA
def one_way_anova(data, groups, response):
    """
    Perform one-way ANOVA.
    :param data: DataFrame containing the dataset
    :param groups: Column name for grouping variable
    :param response: Column name for response variable
    """
    grouped_data = [group[response].values for _, group in data.groupby(groups)]
    f_stat, p_value = f_oneway(*grouped_data)
    print("\nOne-way ANOVA Results:")
    print(f"F-statistic: {f_stat:.4f}, p-value: {p_value:.4f}")
    if p_value < 0.05:
        print("Reject the null hypothesis: Significant difference among group means.")
    else:
        print("Fail to reject the null hypothesis: No significant difference among group means.")

# Function for Two-way ANOVA
def two_way_anova(data, response, factor1, factor2):
    """
    Perform two-way ANOVA.
    :param data: DataFrame containing the dataset
    :param response: Column name for response variable
    :param factor1: Column name for first factor
    :param factor2: Column name for second factor
    """
    formula = f"{response} ~ C({factor1}) + C({factor2}) + C({factor1}):C({factor2})"
    model = ols(formula, data).fit()
    anova_table = sm.stats.anova_lm(model, typ=2)  # Type II ANOVA
    print("\nTwo-way ANOVA Results:")
    print(anova_table)
```

```
if __name__== "__main__":
    # Example dataset for One-way ANOVA
    data_one_way = pd.DataFrame({
        "Group": np.repeat(['A', 'B', 'C'], 10),
        "Score": np.concatenate([
            np.random.normal(loc=50, scale=5, size=10),
            np.random.normal(loc=55, scale=5, size=10),
            np.random.normal(loc=60, scale=5, size=10)
        ])
    })

    # Perform One-way ANOVA
    one_way_anova(data_one_way, groups="Group", response="Score")

    # Example dataset for Two-way ANOVA
    data_two_way = pd.DataFrame({
        "Factor1": np.repeat(['Low', 'Medium', 'High'], 6),
        "Factor2": np.tile(['Type1', 'Type2'], 9),
        "Response": np.concatenate([
            np.random.normal(loc=50, scale=5, size=6),
            np.random.normal(loc=55, scale=5, size=6),
            np.random.normal(loc=60, scale=5, size=6)
        ])
    })

    # Perform Two-way ANOVA
    two_way_anova(data_two_way, response="Response", factor1="Factor1", factor2="Factor2")
```

Output :

```
One-way ANOVA Results:
F-statistic: 13.9602, p-value: 0.0001
Reject the null hypothesis: Significant difference among group means.

Two-way ANOVA Results:
                    sum_sq    df         F    PR(>F)
C(Factor1)          445.793486   2.0  17.604091  0.000270
C(Factor2)            8.818186   1.0   0.696449  0.420283
C(Factor1):C(Factor2)  145.092949   2.0   5.729625  0.017914
Residual            151.939738  12.0        NaN       NaN
```

9. Program to implement correlation, rank correlation and regression and plot x-y plot and heat maps of correlation matrices.

Correlation:

- Pearson Correlation: Measures the linear relationship between two variables (X and Y). A value close to 1 means a strong positive relationship, -1 means a strong negative relationship, and 0 means no linear relationship. The program calculates this correlation using the corr function in Pandas.

Rank Correlation (Spearman's Rank Correlation):

- This measures the strength of a monotonic (ordered) relationship between two variables, using their ranks rather than actual values. It can detect non-linear relationships, and values close to 1 or -1 indicate strong positive or negative relationships.

Linear Regression:

- Linear regression fits a straight line to the data, modeling the relationship between a dependent variable (Y) and an independent variable (X). The program uses scikit-learn to fit a regression line and calculates the Mean Squared Error (MSE) to evaluate the fit.

Visualizations:

- X-Y Scatter Plot: Displays the data points, with a red regression line showing the fitted model.
- Heatmap: Visualizes the correlation matrix, showing the strength of relationships between variables. This program helps to understand relationships between variables using correlation, regression, and visual tools.

Source Code :

```
# Import required libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import spearmanr
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Generate sample data (or load your dataset here)
np.random.seed(42)  # For reproducibility
x = np.random.rand(100) * 100  # Random values for x
y = 2.5 * x + np.random.normal(0, 25, 100)  # Linear relation with noise

# Convert data into a DataFrame
data = pd.DataFrame({'X': x, 'Y': y})

# Compute Correlation
pearson_corr = data.corr(method='pearson')  # Pearson Correlation
spearman_corr, _ = spearmanr(data['X'], data['Y'])  # Spearman Rank Correlation

# Linear Regression
X = data['X'].values.reshape(-1, 1)  # Reshape for sklearn
Y = data['Y'].values
model = LinearRegression()
model.fit(X, Y)
Y_pred = model.predict(X)
regression_coeff = model.coef_[0]  # Slope
regression_intercept = model.intercept_  # Intercept
mse = mean_squared_error(Y, Y_pred)

# Print statistical results
print("Pearson Correlation Coefficient Matrix:")
print(pearson_corr)
print("\nSpearman Rank Correlation Coefficient:", spearman_corr)
print("\nLinear Regression Equation: Y = {:.2f}X + {:.2f}".format(regression_coeff,
regression_intercept))
print("Mean Squared Error (MSE):", mse)
```

```
# Plot X-Y scatter plot with regression line
plt.figure(figsize=(8, 6))
plt.scatter(data['X'], data['Y'], color='blue', label='Data Points')
plt.plot(data['X'], Y_pred, color='red', label='Regression Line')
plt.title('X-Y Scatter Plot with Regression Line')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()

# Plot heatmap of correlation matrix
plt.figure(figsize=(6, 5))
sns.heatmap(pearson_corr, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Heatmap of Correlation Matrix')
plt.show()
```

Output :

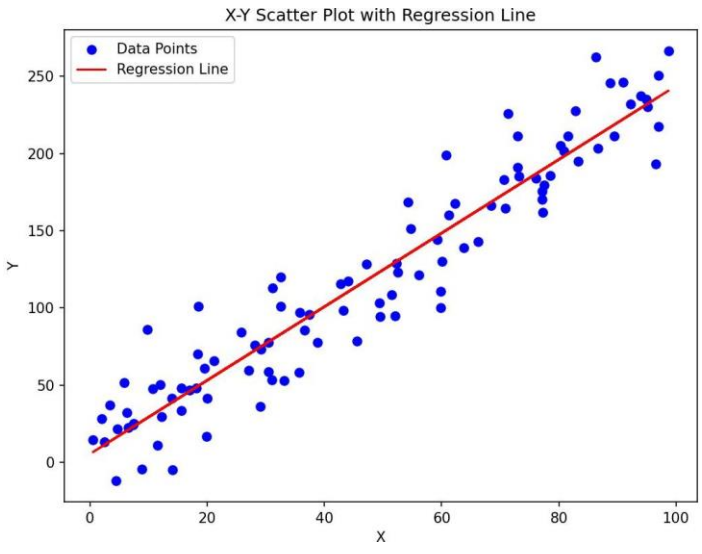
Pearson Correlation Coefficient Matrix:

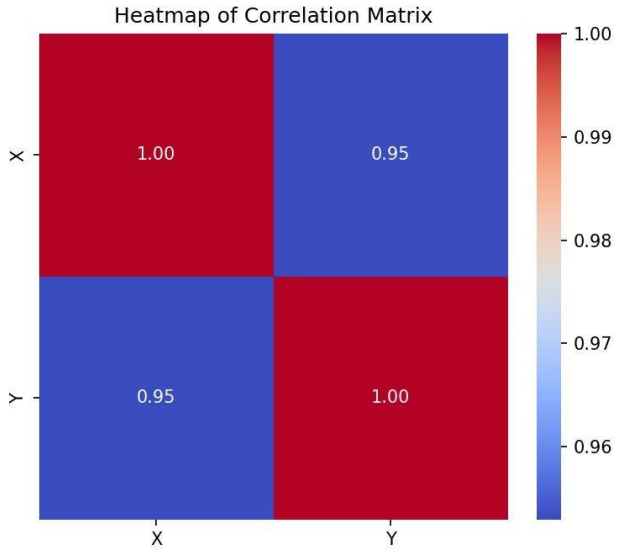
	X	Y
X	1.000000	0.952966
Y	0.952966	1.000000

Spearman Rank Correlation Coefficient: 0.9519351935193517

Linear Regression Equation: Y = 2.39X + 5.38

Mean Squared Error (MSE): 504.11535247940856





10. Program to implement PCA for Wisconsin dataset, visualize and analyze the results.

This program demonstrates Principal Component Analysis (PCA) on the Wisconsin Breast Cancer dataset to reduce the dimensionality of the data, visualize the results, and analyze the explained variance of the components.

Principal Component Analysis (PCA):

PCA is a technique used to reduce the dimensionality of large datasets while preserving as much information as possible. It transforms the original features into new, uncorrelated variables called principal components. The goal is to project the data into fewer dimensions, typically 2 or 3, for easier visualization while retaining most of the data's variance.

Standardization:

Before applying PCA, the data is standardized using StandardScaler to ensure that each feature has zero mean and unit variance. This is important because PCA is sensitive to the scale of the data.

Applying PCA:

PCA is performed to reduce the data to 2 principal components for visualization. The program then calculates the explained variance ratio, which tells us how much variance (information) each principal component captures.

Visualization: PCA Scatter Plot: The program creates a scatter plot of the first two principal components (PCA1 and PCA2) to visualize how the data points are distributed in the reduced space. Points are colored based on the target variable (malignant or benign).

- Explained Variance: A bar plot shows how much variance each of the first two principal components explains.
- Cumulative Variance: A line plot shows how much cumulative variance is explained as more components are added.

Source Code :

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Load the Wisconsin Breast Cancer dataset
data = load_breast_cancer()
X = data.data  # Features
y = data.target  # Target variable (0 = malignant, 1 = benign)
feature_names = data.feature_names
target_names = data.target_names

# Standardize the data (important for PCA)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA(n_components=2)  # Reduce to 2 dimensions for visualization
X_pca = pca.fit_transform(X_scaled)

# Get explained variance ratio for each component
explained_variance_ratio = pca.explained_variance_ratio_

# Create a DataFrame for visualization
pca_df = pd.DataFrame(X_pca, columns=['PCA1', 'PCA2'])
pca_df['Target'] = y

# Plot the PCA results
plt.figure(figsize=(8, 6))
sns.scatterplot(data=pca_df, x='PCA1', y='PCA2', hue='Target', palette='Set1', alpha=0.8)
plt.title('PCA of Wisconsin Breast Cancer Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(target_names)
```

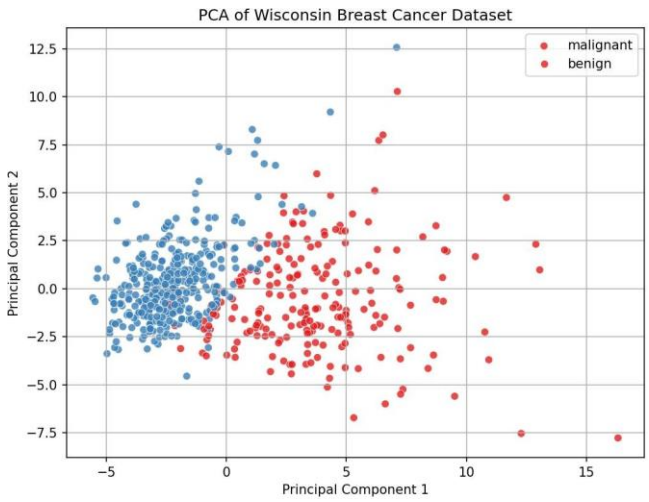
```
plt.grid()
plt.show()

# Plot explained variance ratio
plt.figure(figsize=(8, 5))
plt.bar(range(1, 3), explained_variance_ratio, tick_label=['PCA1', 'PCA2'], color='skyblue')
plt.title('Explained Variance Ratio of PCA Components')
plt.xlabel('Principal Components')
plt.ylabel('Variance Explained')
plt.show()

# Full PCA with all components for analysis
pca_full = PCA()
X_pca_full = pca_full.fit_transform(X_scaled)
cumulative_variance = np.cumsum(pca_full.explained_variance_ratio_)

# Plot cumulative explained variance
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', linestyle='--',
color='b')
plt.title('Cumulative Explained Variance')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.grid()
plt.show()

# Print key insights
print("PCA Analysis of Wisconsin Breast Cancer Dataset")
print("------------------------------------------------------")
print(f"Explained Variance (PCA1): {explained_variance_ratio[0]:.4f}")
print(f"Explained Variance (PCA2): {explained_variance_ratio[1]:.4f}")
print("Cumulative Variance Explained by All Components:")
for i, cum_var in enumerate(cumulative_variance, start=1):
    print(f"  Component {i}: {cum_var:.4f}")
```

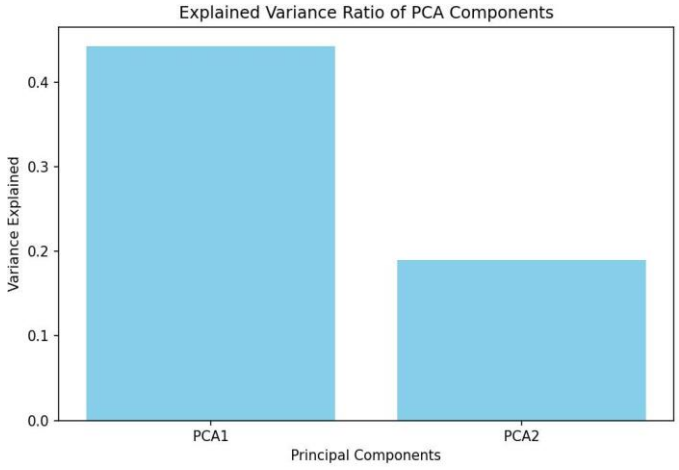
Output :

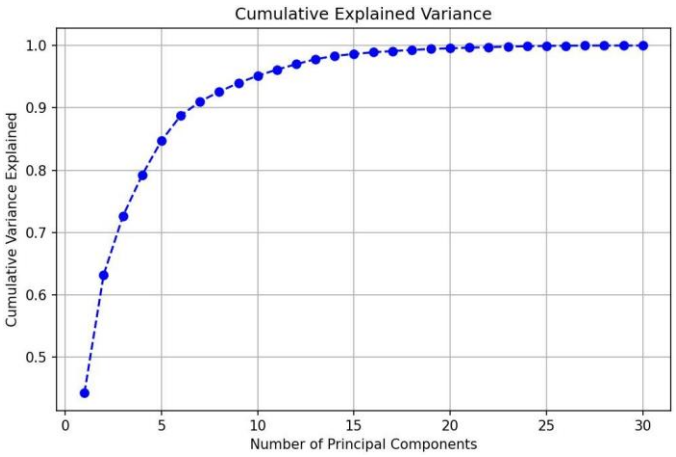
```
PCA Analysis of Wisconsin Breast Cancer Dataset
------------------------------------------------------
Explained Variance (PCA1): 0.4427                    Explained Variance (PCA2): 0.1897
```

Cumulative Variance Explained by All Components:

Component 1: 0.4427
Component 2: 0.6324
Component 3: 0.7264
Component 4: 0.7924
Component 5: 0.8473
Component 6: 0.8876
Component 7: 0.9101
Component 8: 0.9260
Component 9: 0.9399
Component 10: 0.9516
Component 11: 0.9614
Component 12: 0.9701
Component 13: 0.9781
Component 14: 0.9834
Component 15: 0.9865
Component 16: 0.9892
Component 17: 0.9911
Component 18: 0.9929
Component 19: 0.9945
Component 20: 0.9956
Component 21: 0.9966
Component 22: 0.9975
Component 23: 0.9983
Component 24: 0.9989
Component 25: 0.9994
Component 26: 0.9997
Component 27: 0.9999
Component 28: 1.0000
Component 29: 1.0000
Component 30: 1.0000






Cumulative Explained Variance
Cumulative Variance Explained
Number of Principal Components
1.0
0.9
0.8
0.7
0.6
0.5
0
5
10
15
20
25
30

11. Program to implement the working of linear discriminant analysis using iris dataset and visualize the results.

Linear Discriminant Analysis (LDA) is a technique used for dimensionality reduction and classification. It aims to find the linear combinations of features that best separate the classes in the dataset. Unlike Principal Component Analysis (PCA), which maximizes variance, LDA focuses on maximizing class separability.

Key Steps in LDA:

- Data Standardization: Before applying LDA, the data is scaled so that each feature has zero mean and unit variance. This ensures that all features contribute equally to the analysis.
- Compute Discriminants: LDA computes new axes (called discriminants) that maximize the difference between classes.
- Dimensionality Reduction: LDA reduces the dataset to fewer dimensions while preserving as much class separation as possible. In this case, we reduce it to 2 dimensions for easier visualization.

Visualization: The transformed data is plotted in a 2D space, showing how well the classes (species in the Iris dataset) are separated.

Application in the Iris Dataset:

- The Iris dataset has 4 features, and LDA reduces it to 2 components for visualization.
- LDA is useful in classification tasks, where the goal is to predict the class label of new data points based on the transformed features.

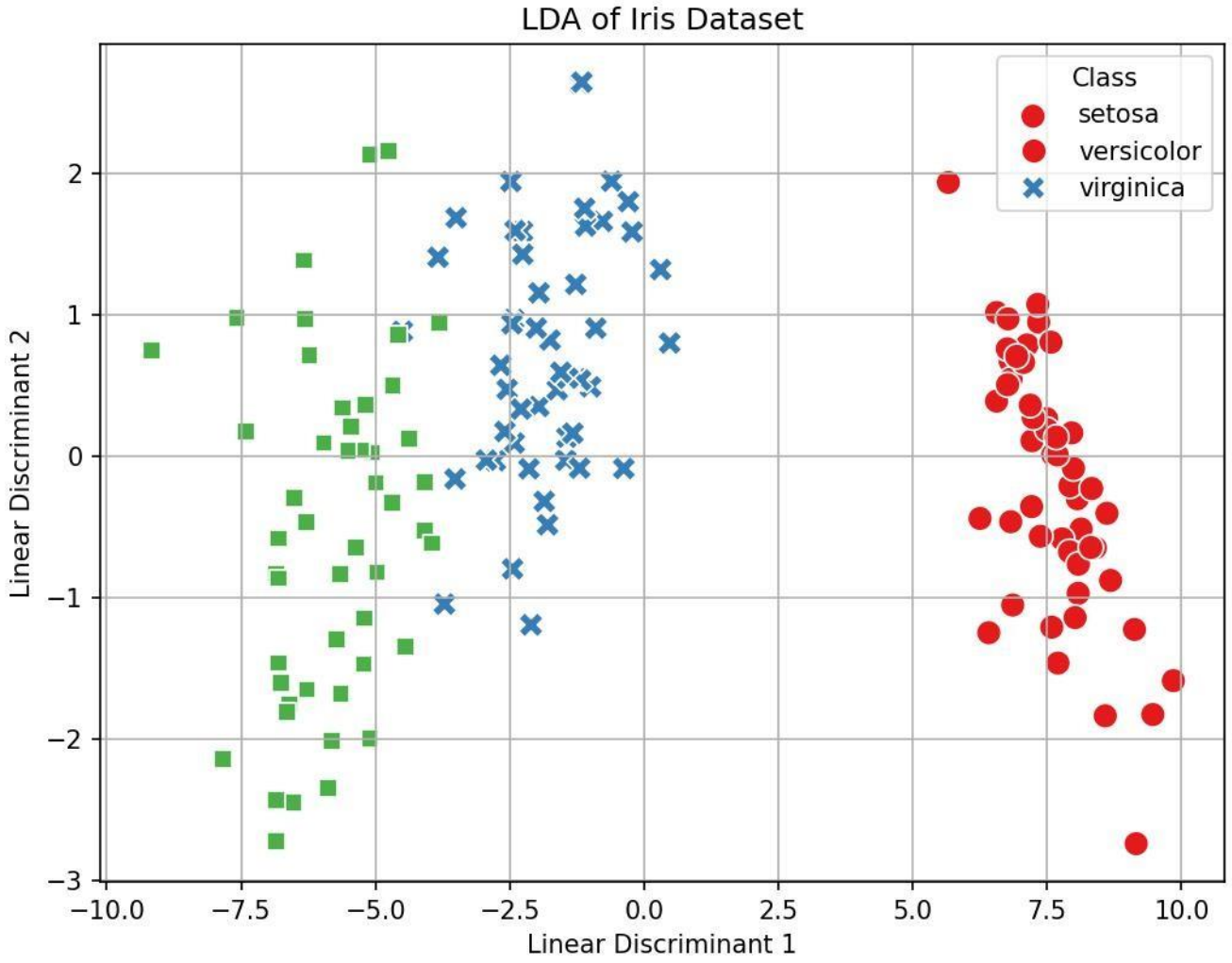
Source Code :

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler

# Load the Iris dataset
data = load_iris()
X = data.data  # Features
y = data.target  # Target variable (0, 1, 2)
target_names = data.target_names  # Class names

# Standardize the data (LDA benefits from scaling)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply Linear Discriminant Analysis (LDA)
lda = LinearDiscriminantAnalysis(n_components=2)  # Reduce to 2 components for visualization
X_lda = lda.fit_transform(X_scaled, y)

# Create a DataFrame for LDA-transformed data
lda_df = pd.DataFrame(X_lda, columns=['LDA1', 'LDA2'])
lda_df['Target'] = y

# Plot the LDA results in 2D space
plt.figure(figsize=(8, 6))
sns.scatterplot(data=lda_df, x='LDA1', y='LDA2', hue='Target', palette='Set1', style='Target',
s=100)
plt.title('LDA of Iris Dataset')
plt.xlabel('Linear Discriminant 1')
plt.ylabel('Linear Discriminant 2')
plt.legend(title='Class', labels=target_names)
plt.grid()
plt.show()
```

```
# Print key insights
print("Linear Discriminant Analysis (LDA) Results")
print("-----------------------------------------------------------")
print("Explained Variance Ratio by LDA Components:")
for i, ratio in enumerate(lda.explained_variance_ratio_, start=1):
    print(f"  LDA{i}: {ratio:.4f}")
```

Output :

```
Linear Discriminant Analysis (LDA) Results
-----------------------------------------------------------
Explained Variance Ratio by LDA Components:
  LDA1: 0.9912
  LDA2: 0.008
```



12. Program to Implement multiple linear regression using iris dataset, visualize and analyze the results.

Multiple Linear Regression (MLR) is a technique used to predict a target variable based on the relationship between multiple input variables. It helps in understanding how different features affect the outcome.

Key Concepts:

- Prediction: MLR creates a model that predicts a target variable using multiple independent variables.
- Training: The model learns from the training data by adjusting coefficients for each feature.
- Evaluation: The model's accuracy is measured using metrics like Mean Squared Error (MSE) and R-squared (R^2).

Application to the Iris Dataset:

In this case, we predict the petal length based on other features like sepal length and petal width. The dataset is split into a training set and a test set. The model is trained on the training set and evaluated on the test set.

Steps:

1. Training: Fit the model using the training data.
2. Prediction: Make predictions on the test data.
3. Evaluation: Use MSE and R^2 to assess model performance.
4. Visualization: Compare the actual vs predicted values using a plot.

MLR is commonly used when there are multiple factors influencing the outcome and helps in making predictions based on them.

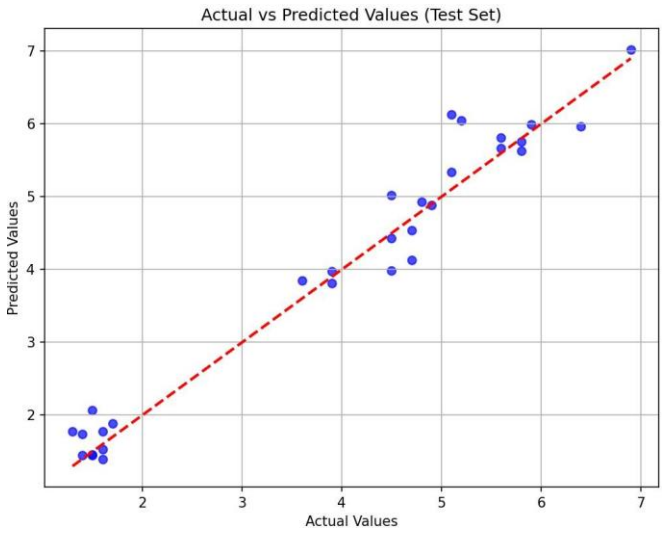
Source Code :

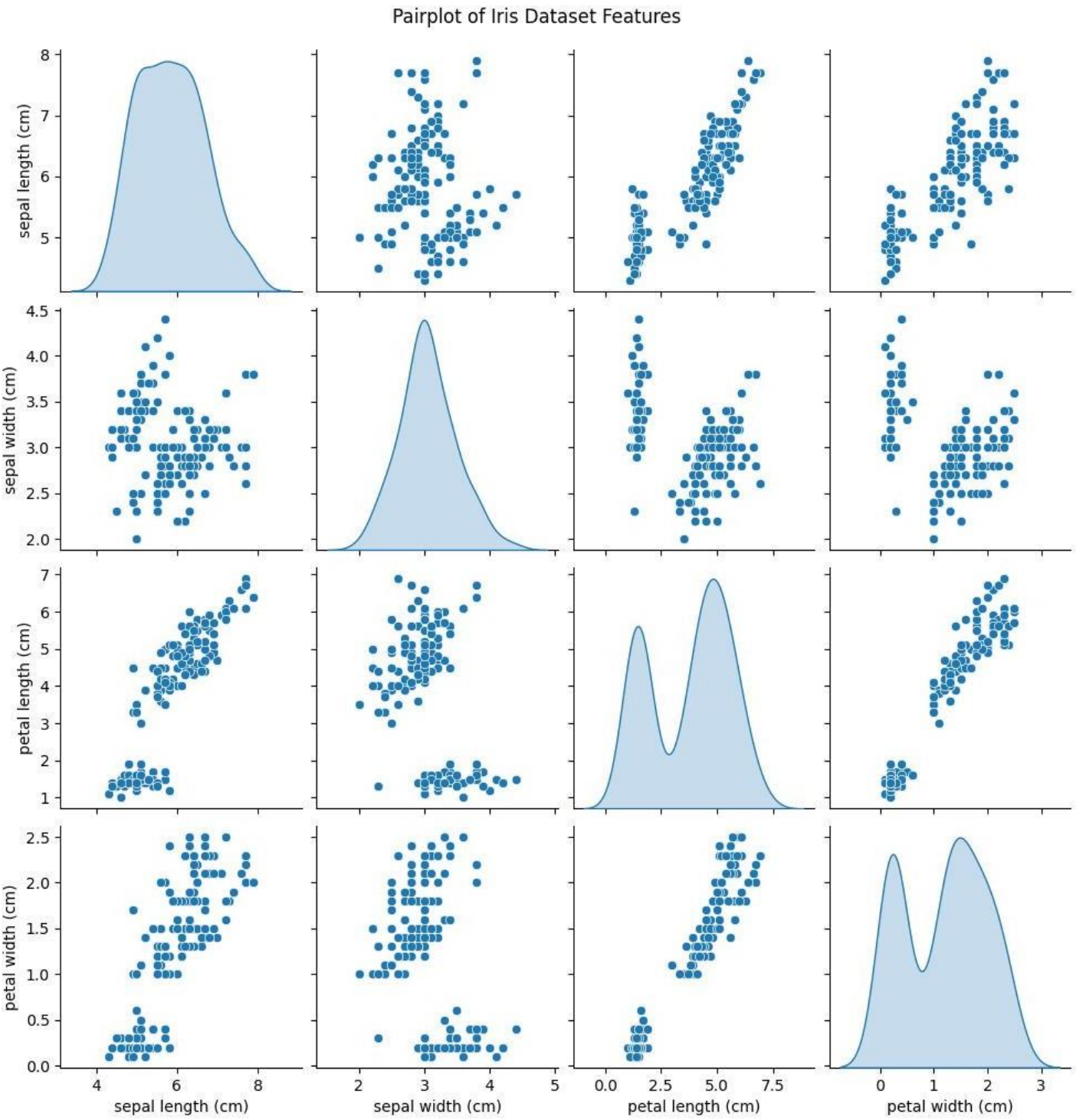
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load the Iris dataset
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Features
y = X['petal length (cm)']  # Let's predict 'petal length' as the dependent variable
X = X.drop(columns=['petal length (cm)'])  # Remove 'petal length' from independent variables

# Split the dataset 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)

# Apply Multiple Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)  # Train the model

# Predict on the test set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Print model performance metrics
print("Multiple Linear Regression Results")
print("---------------------------------------")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R-squared (R²): {r2:.4f}")
print("\nModel Coefficients:")
for feature, coef in zip(X.columns, model.coef_):
    print(f"  {feature}: {coef:.4f}")
print(f"Intercept: {model.intercept_:.4f}")
```

```
# Visualize actual vs predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, color='blue', alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linewidth=2,
linestyle='--')
plt.title('Actual vs Predicted Values (Test Set)')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.grid()
plt.show()

# Pairplot to explore relationships in the dataset
sns.pairplot(pd.DataFrame(data.data, columns=data.feature_names), diag_kind='kde')
plt.suptitle('Pairplot of Iris Dataset Features', y=1.02)
plt.show()
```

Output :

```
Multiple Linear Regression Results
----------------------------------------
Mean Squared Error (MSE): 0.1300
R-squared (R²): 0.9603

Model Coefficients:
 sepal length (cm): 0.7228
 sepal width (cm): -0.6358
 petal width (cm): 1.4675
```




Pairplot of Iris Dataset Features
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)