MACHINE LEARNING LAB (BCSL606)

LAB MANUAL 2024-25 EVEN

6TH SEMESTER – A section

Department of Information Science and Engineering

Prepared by –

Manasa S M

Assistant Professor

Dept of ISE

JNNCE

The Machine Learning Laboratory (BCSL606) under the Visvesvaraya Technological University (VTU) curriculum is designed to provide hands-on experience with various machine learning algorithms and techniques. The lab exercises aim to enhance understanding by implementing and experimenting with different models and methods.

Machine Learning Lab is designed to provide hands-on experience with fundamental machine learning techniques and algorithms. It typically involves implementing various supervised and unsupervised learning models, understanding data preprocessing techniques, evaluating model performance, and using tools like Python, Scikit-Learn, TensorFlow, or other ML frameworks.

Course outcomes (Course Skill Set):

At the end of the course the student will be able to:

1. Illustrate the principles of multivariate data and apply dimensionality reduction techniques.
2. Demonstrate similarity-based learning methods and perform regression analysis.
3. Develop decision trees for classification and regression problems, and Bayesian models for probabilistic learning.
1. Implement the clustering algorithms to share computing resources.

Introduction to Machine Learning

Machine Learning (ML) is a branch of artificial intelligence (AI) that enables computers to learn from data and make predictions or decisions without being explicitly programmed. It focuses on developing algorithms that can identify patterns, improve performance over time, and make data-driven decisions.

Why Machine Learning?

Traditional programming relies on predefined rules to solve problems, but ML allows computers to learn from past experiences (data) and improve automatically. It is widely used in various domains like healthcare, finance, robotics, natural language processing, and more.

Types of Machine Learning

1. **Supervised Learning –** The model is trained using labeled data, meaning the input comes with the corresponding correct output.Example: Spam detection in emails.

2. **Unsupervised Learning –** The model learns patterns from unlabeled data without explicit outputs. Example: Customer segmentation in marketing.

3. **Reinforcement Learning –** The model learns by interacting with an environment and receiving feedback in the form of rewards or penalties. Example: AlphaGo, self-driving cars.

Applications of Machine Learning

- **Healthcare:** Disease prediction, medical diagnosis

- **Finance:** Fraud detection, stock price prediction
- **E-commerce:** Recommendation systems, customer sentiment analysis
- **Robotics:** Autonomous navigation, industrial automation
- **Natural Language Processing**: Chatbots, language translation**.**

Machine Learning is revolutionizing various industries by making systems smarter and more efficient. As more data becomes available, ML continues to evolve, providing better solutions to complex problems.

Python

Basic of Python:

1. Various Datatypes in Python
   ➔ int, float, complex. Str, list, tuple, range, set, dict, bool etc.
2. Variables in Python
   ➔ A **variable** in Python is a name that refers to a memory location where data is stored. Python is **dynamically typed**, meaning you don't need to declare the data type explicitly—it is assigned automatically based on the value.
   ➔ Ex -

```
x = 10              # Integer
y = 3.14            # Float
name = "Alice"      # String
is_valid = True     # Boolean
```

Why Python?

Python is the most popular language for executing Machine Learning (ML) algorithms due to several key reasons:

1. Extensive Libraries and Frameworks

Python offers a rich ecosystem of ML libraries, making it easy to implement complex algorithms without starting from scratch. Some major libraries include:

- **NumPy** – Efficient numerical computing
- **Pandas** – Data manipulation and preprocessing
- **Matplotlib & Seaborn** – Data visualization
- **Scikit-learn** – Standard ML algorithms(classification, regression, clustering)
- **TensorFlow &PyTorch** – Deep learning frameworks
- **OpenCV** – Computer vision applications

2. Simple and Readable Syntax

Python's easy-to-read syntax allows developers to focus on ML logic rather than complex programming structures.

3. Platform Independence

Python runs on **Windows, Linux, and macOS** without major changes in code, making it flexible for ML development on various systems.

4. Integration with Other Technologies

Python can easily integrate with **C, C++, Java, and other languages**, allowing developers to combine ML models with existing applications.

Python is the preferred language for ML due to its **simplicity, rich libraries, community support, and scalability**. It allows researchers and developers to focus on building and optimizing ML models rather than worrying about complex coding.

Pandas

Pandas is an open source, BSD – licensed library providing high-performance, easy-to-use data structures and data analysis tools for python programming language.Pandas is an essential library in machine learning because it provides data structures (like DataFrames) that are perfect for handling, analyzing, and preprocessing data before applying machine learning algorithms.

#First step is to import pandas

import pandas **as** pd

importnumpy**as** np

What pandas does?

Pandas makes it easy to read data from various file formats (e.g., CSV, Excel, SQL databases) and perform common operations like filtering, grouping, reshaping, and merging datasets.

It provides data structures likeDataFrame and Series:

- **DataFrame**: Two-dimensional, labeled data structure (similar to a table or spreadsheet), which you will use for most tasks in machine learning and data science.
- **Series**: One-dimensional array-like object (similar to a column in a table).

Example –

```
df = pd.read_csv('data.csv')  # Load data from a CSV file into a DataFrame
print(df.head())  # Print the first 5 rows of the DataFrame
```

Some inbuilt functions:

1. **df.info()**
2. **df.describe()** //only int and float values of column will be taken into consideration. Categorical features are skipped.

Matplotlib – Simple Visualization Library

Matplotlib is a ploting library for the python programming language and its numerical mathematics extension Numpy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt or GTK+.

Some of the major Pros of Matplotlib are:

- Generally easy to get started for simple plots.
- Support for custom labels and texts.
- Great control of every element in a figure.
- High quality output in many formats
- Very customizable in general.

```
import matplotlib.pyplot as plt

%matplotlib inline
```

Seaborn – A statistical Data Visualization Library

To get more statistical tools to understand more about the data. Dataset to be divided into independent and dependent feature. Seaborn is a powerful Python visualization library built on top of **Matplotlib** and closely integrated with **Pandas**. It provides high-level interfaces for drawing attractive and informative statistical graphics.

1. **Setting Seaborn styles**
   - darkgrid
   - whitegrid
   - dark
   - white

2. **Basic Seaborn plots**
   - Scatter plot
   - Line plot
   - Histogram
   - Box Plot
   - Violin plot

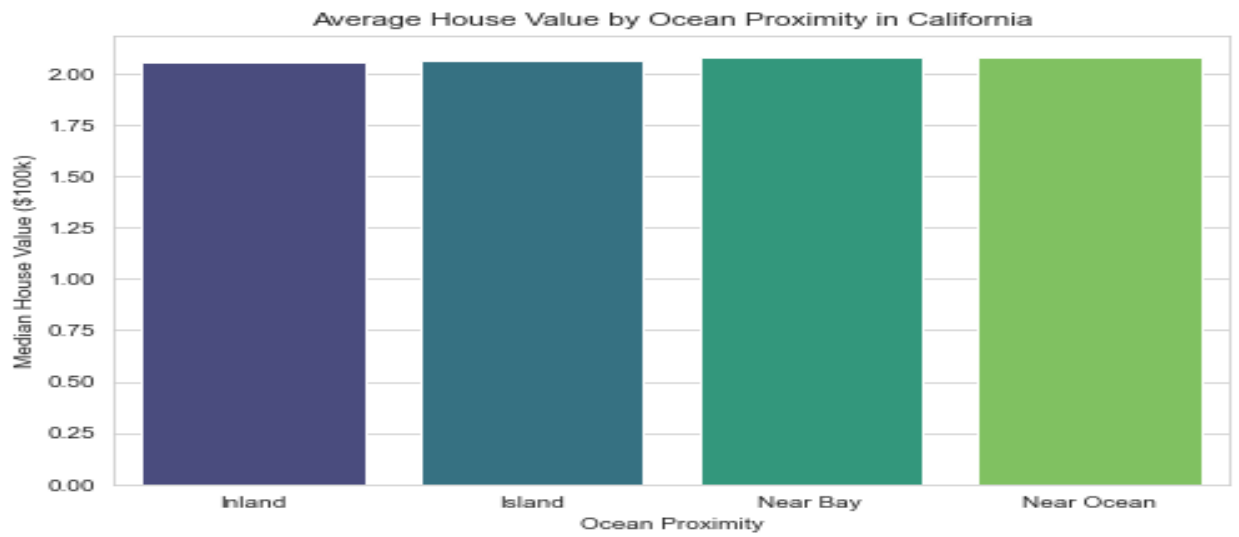
Example:

```
python

sns.set_style("whitegrid")
```

Sample Program
To create a barchart for California Housing Dataset

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing

# Load the California housing dataset
housing = fetch_california_housing(as_frame=True)
df = housing.frame

# Add a categorical "ocean_proximity" column (simulated)
np.random.seed(42)
df["ocean_proximity"] = np.random.choice(["Near Bay", "Inland", "Near Ocean", "Island"], size=len(df))

# Calculate average house value per category
avg_house_value = df.groupby("ocean_proximity")["MedHouseVal"].mean().reset_index()

# Create a bar chart using Seaborn
sns.set_style("whitegrid")
plt.figure(figsize=(8,5))
sns.barplot(x="ocean_proximity", y="MedHouseVal", data=avg_house_value, palette="viridis")

# Add labels and title
plt.xlabel("Ocean Proximity")
plt.ylabel("Median House Value ($100k)")
plt.title("Average House Value by Ocean Proximity in California")

# Show the plot
plt.show()
```

Output:



Program 1

Develop a program to create histograms for all numerical features and analyze the distribution of each feature. Generate box plots for all numerical features and identify any outliers. Use California Housing dataset.

```
import matplotlib.pyplotas plt
import seaborn as sns
import pandas as pd

# Load the California Housing dataset
data = pd.read_csv('housing.csv')
df = data

numerical_features = df.columns

# Create histograms for all numerical features
df.hist(bins=30, figsize=(12, 10), layout=(4, 2), edgecolor='black')
plt.suptitle('Histograms of Numerical Features', fontsize=16)
plt.show()

# Create box plots for all numerical features
plt.figure(figsize=(12, 10))
for i, col in enumerate(df.columns):
      plt.subplot(4, 2, i + 1)
      sns.boxplot(y=df[col])
      plt.title(f'Box Plot of {col}')
      plt.tight_layout()
plt.show()

# Identify outliers using IQR method
outliers = {}
for col in df.columns:
      Q1 = df[col].quantile(0.25)
      Q3 = df[col].quantile(0.75)
      IQR = Q3 - Q1
      lower_bound = Q1 - 1.5 * IQR
      upper_bound = Q3 + 1.5 * IQR
      outliers[col] = df[(df[col] <lower_bound) | (df[col] >upper_bound)][col]

# Print outliers for each column
for col, outlier_valuesin outliers.items():
      print(f'Outliers in {col}:')
      print(outlier_values.describe())
      print('\n')
```

Outliers:

Outliers are data points that deviate significantly from the typical distribution of a feature. They can arise from various causes, including:

1. **Data Entry Errors**
2. **Measurement Errors**
3. **Sampling Issues**
4. **Natural Variability**

Identifying and understanding outliers is crucial, as they can significantly impact statistical analyses and model performance. Depending on their cause, outliers might indicate errors that need correction or genuine variability that offers valuable insights into the data.

- We calculate the first quartile (Q1) and third quartile (Q3) for each feature.
- The interquartile range (IQR) is computed as IQR = Q3 - Q1.
- Outliers are defined as data points below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR.
- Finally, we print the summary statistics of the outliers for each feature that contains them.

Histograms

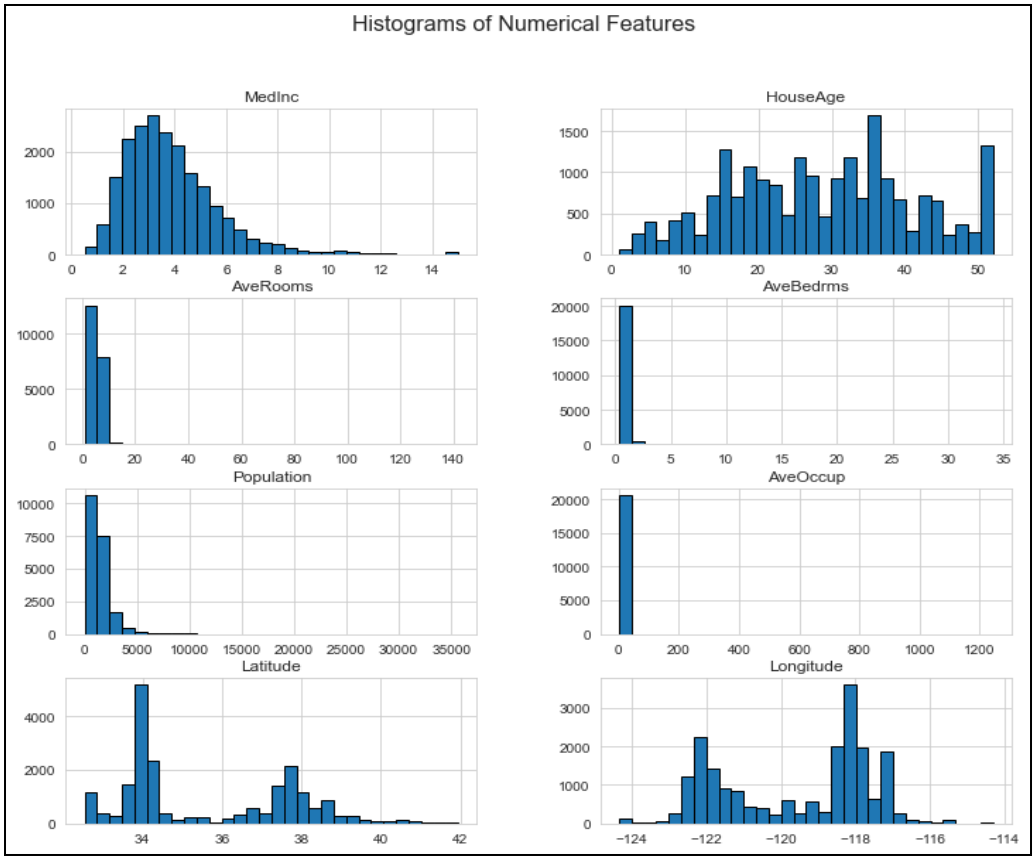
- A histogram divides a dataset into intervals (bins) and counts how many values fall into each bin.
- The x-axis represents the bins (ranges of values).
- The y-axis represents the frequency (count of values in each bin).

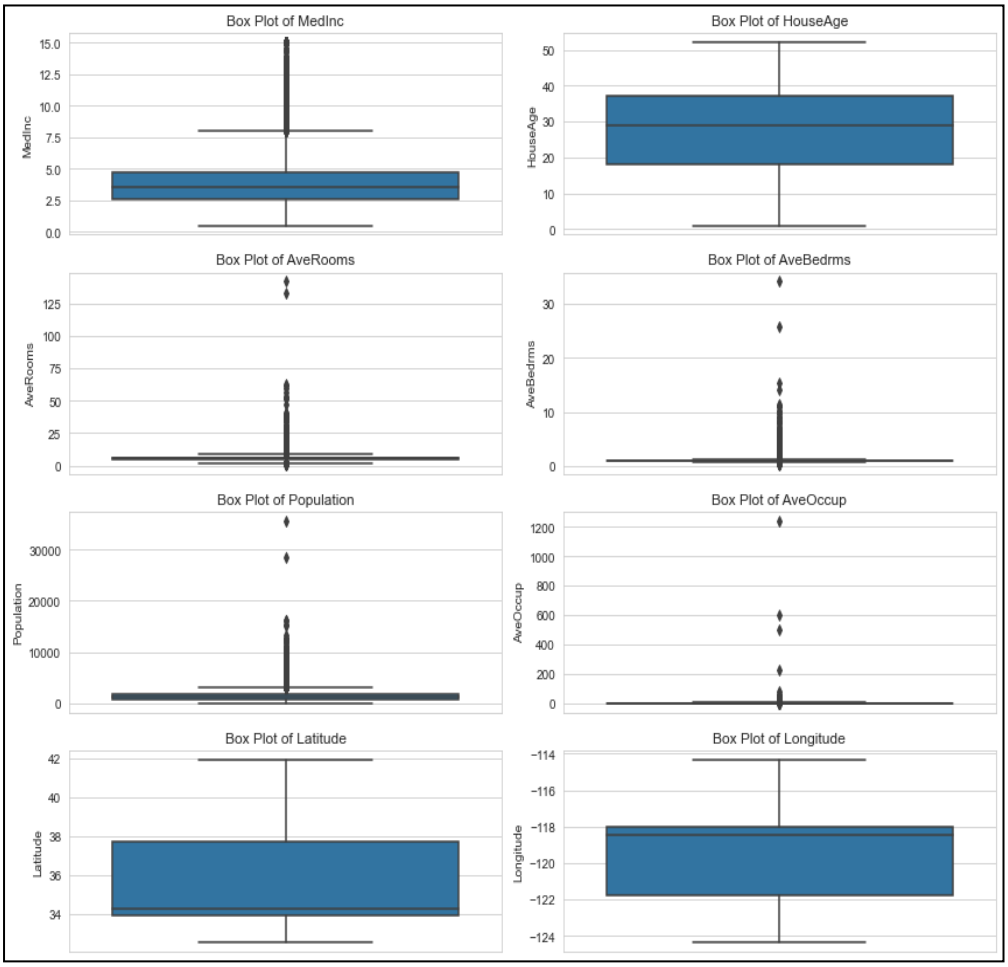
Boxplots

A boxplot (box-and-whisker plot) is a graphical representation used to show the distribution, variability, and outliers in a dataset. It provides a summary using five key statistics:

- Minimum (smallest value, excluding outliers)
- First Quartile (Q1) – 25th percentile
- Median (Q2) – 50th percentile
- Third Quartile (Q3) – 75th percentile
- Maximum (largest value, excluding outliers)

Output:


Histograms of Numerical Features
MedInc
HouseAge
AveRooms
AveBedrms
Population
AveOccup
Latitude
Longitude


Box Plot of MedInc
Box Plot of HouseAge
Box Plot of AveRooms
Box Plot of AveBedrms
Box Plot of Population
Box Plot of AveOccup
Box Plot of Latitude
Box Plot of Longitude

Program-2

Develop a program to Compute the correlation matrix to understand the relationships between pairs offeatures. Visualize the correlation matrix using a heatmap to know which variables have strongpositive/negative correlations. Create a pair plot to visualize pairwise relationships between features. Use California Housing dataset.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplotas plt

# Step 1: Load the California Housing Dataset
california_data = pd.read_csv('housing.csv')
data = california_data

# Step 2: Compute the correlation matrix
correlation_matrix = data.corr()

# Step 3: Visualize the correlation matrix using a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title('Correlation Matrix of California Housing Features')
plt.show()

# Step 4: Create a pair plot to visualize pairwise relationships
sns.pairplot(data, diag_kind='kde', plot_kws={'alpha': 0.5})
plt.suptitle('Pair Plot of California Housing Features', y=1.02)
plt.show()
```

Explanation:

Correlation Matrix: A correlation matrix is a **quick and effective tool** to analyze relationships between multiple features simultaneously.

Example: Correlation Matrix

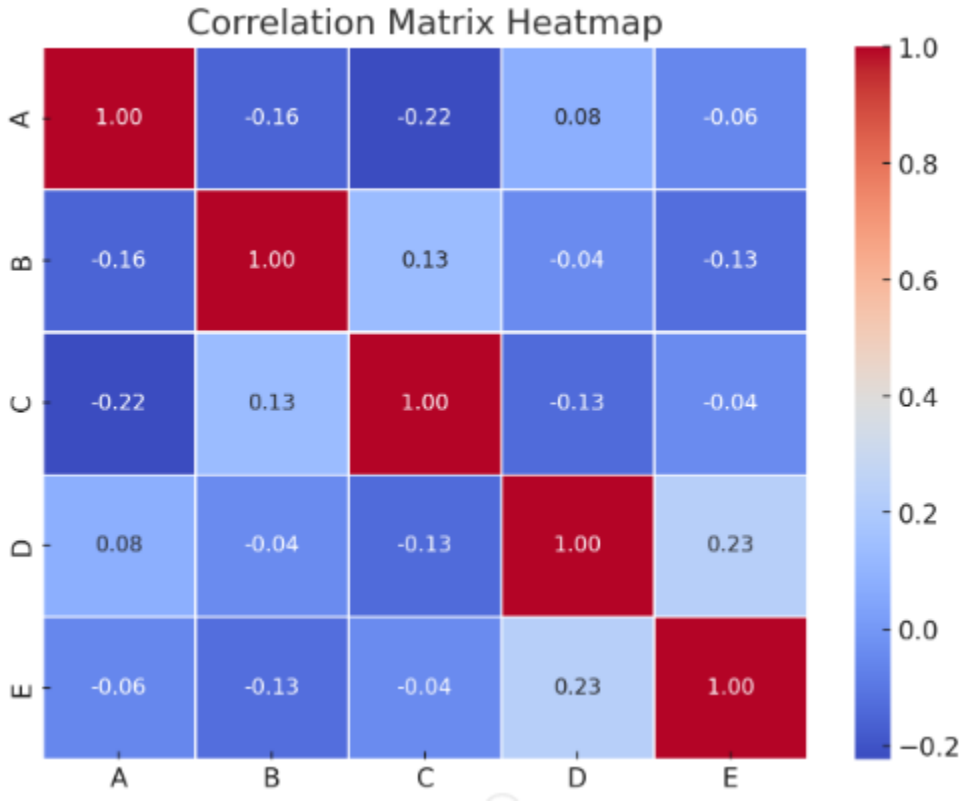
Feature	X1	X2	X3
X1	1.0	0.8	0.2
X2	0.8	1.0	-0.5
X3	0.2	-0.5	1.0

- X1 and X2 (0.8) → Strong positive correlation.
- X2 and X3 (-0.5) → Moderate negative correlation.
- X1 and X3 (0.2) → Weak correlation.

To visualize the correlation matrix, we can use seaborn's heatmap in Python. Here's a simple example:

Steps:

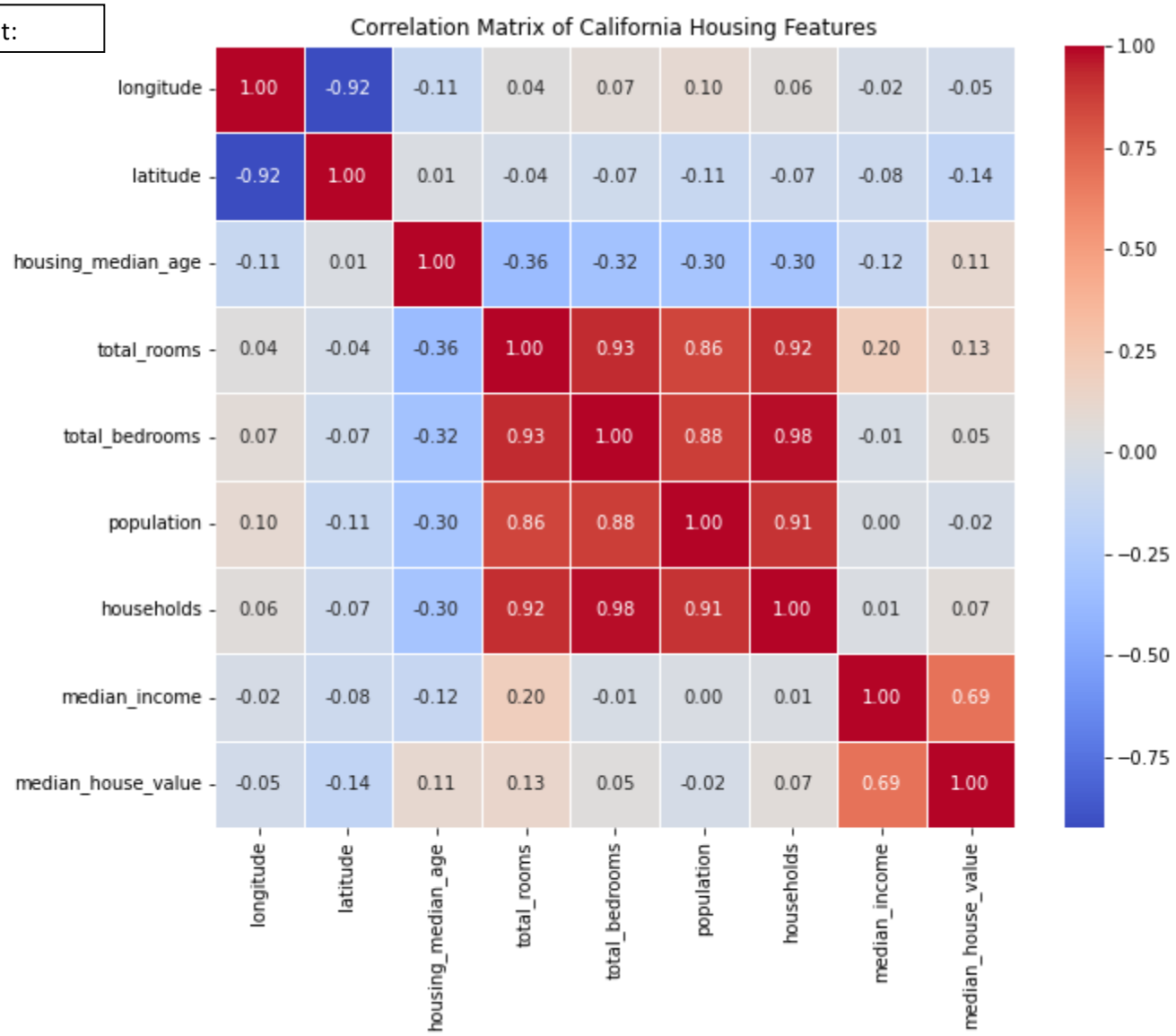
1. Generate some random data.
2. Compute the correlation matrix.
3. Visualize it using a heatmap.
4. The values represent the Pearson correlation coefficient (ranging from -1 to 1).
5. A value close to 1 means a strong positive relationship.
6. A value close to -1 means a strong negative relationship.
7. A value near 0 means no correlation.

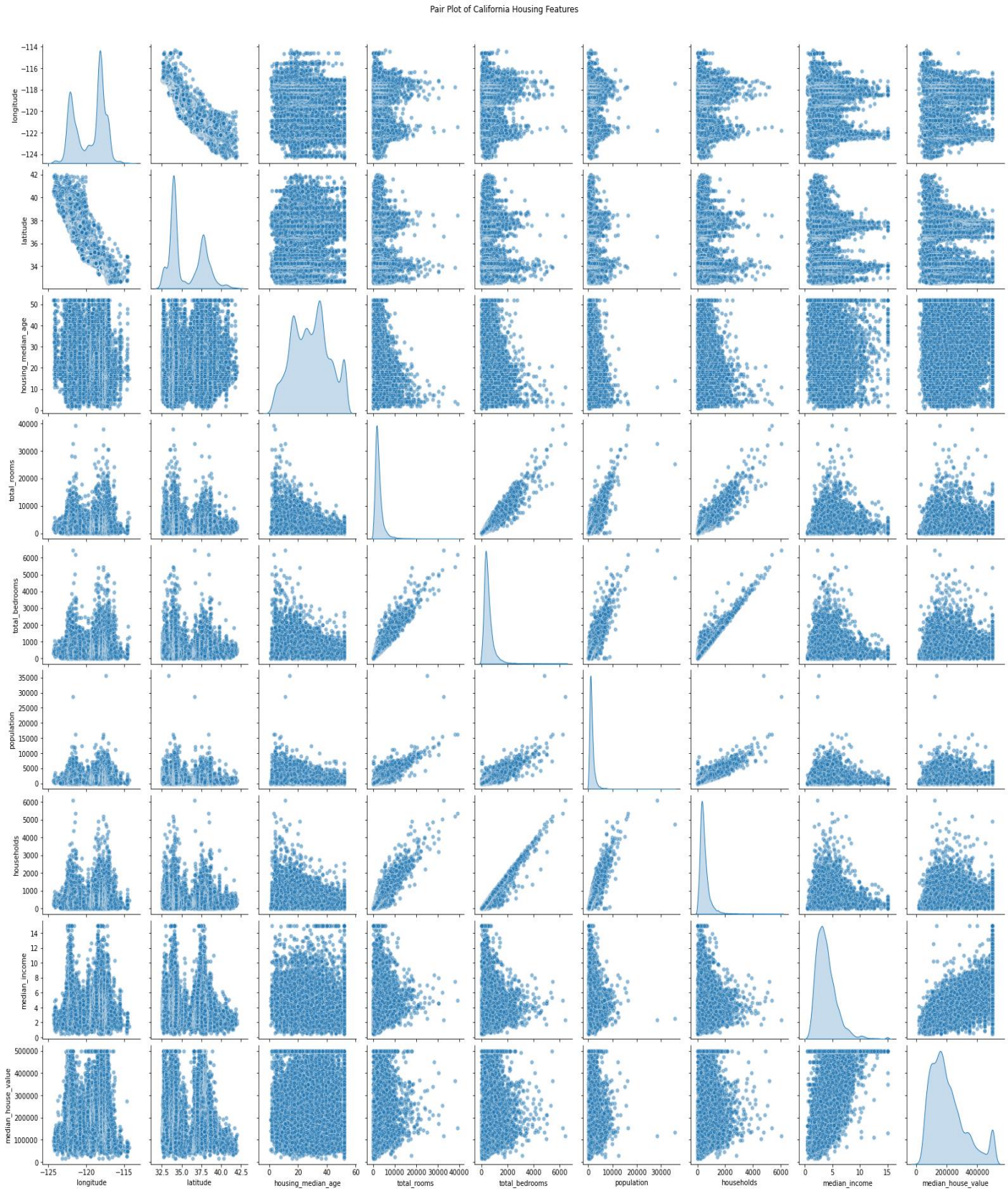


What is a Pairplot?

A pairplot (from Seaborn's sns.pairplot) is a powerful visualization that shows pairwise relationships between multiple numerical features in a dataset. It is especially useful for exploratory data analysis (EDA).

Output:


Correlation Matrix of California Housing Features
longitude
latitude
housing_median_age
total_rooms
total_bedrooms
population
households
median_income
median_house_value
1.00 -0.92 -0.11 0.04 0.07 0.10 0.06 -0.02 -0.05
-0.92 1.00 0.01 -0.04 -0.07 -0.11 -0.07 -0.08 -0.14
-0.11 0.01 1.00 -0.36 -0.32 -0.30 -0.30 -0.12 0.11
0.04 -0.04 -0.36 1.00 0.93 0.86 0.92 0.20 0.13
0.07 -0.07 -0.32 0.93 1.00 0.88 0.98 -0.01 0.05
0.10 -0.11 -0.30 0.86 0.88 1.00 0.91 0.00 -0.02
0.06 -0.07 -0.30 0.92 0.98 0.91 1.00 0.01 0.07
-0.02 -0.08 -0.12 0.20 -0.01 0.00 0.01 1.00 0.69
-0.05 -0.14 0.11 0.13 0.05 -0.02 0.07 0.69 1.00
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75


Pair Plot of California Housing Features
longitude
latitude
housing_median_age
total_rooms
total_bedrooms
population
households
median_income
median_house_value

Program – 3

Develop a program to implement Principal Component Analysis (PCA) for reducing the dimensionality of theIris dataset from 4 features to 2.

```
import pandas as pd
import matplotlib.pyplotas plt
from sklearn.datasetsimport load_iris
from sklearn.decompositionimport PCA
from sklearn.preprocessingimport StandardScaler

# Load the Iris dataset
data = load_iris()
iris_df = pd.DataFrame(data.data, columns=data.feature_names)
iris_df['target'] = data.target

# Standardize the features
scaler = StandardScaler()
scaled_data = scaler.fit_transform(iris_df[data.feature_names])

# Apply PCA to reduce dimensions from 4 to 2
pca = PCA(n_components=2)
principal_components = pca.fit_transform(scaled_data)

# Create a DataFrame with the principal components
pca_df = pd.DataFrame(data=principal_components, columns=['PC1', 'PC2'])
pca_df['target'] = iris_df['target']

# Visualize the PCA result
plt.figure(figsize=(8, 6))
colors = ['red', 'green', 'blue']
for target, color in zip(data.target_names, colors):
      indices = pca_df['target'] == list(data.target_names).index(target)
      plt.scatter(pca_df.loc[indices, 'PC1'],
      pca_df.loc[indices, 'PC2'],
      color=color,
      label=target)

plt.title('PCA of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid()
plt.show()

# Explained variance ratio
print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Explanation:

Dimensionality Reduction in Machine Learning

Dimensionality reduction is the process of reducing the number of input variables (features) in a dataset while retaining as much relevant information as possible. It helps simplify models, improve efficiency, and reduce overfitting.

Principal Component Analysis:

PCA is a technique that reduces the dimensionality of data while preserving as much important information as possible. It transforms high-dimensional data into a new set of features called principal components.

Why is PCA Needed in Machine Learning?

- Reduces Dimensionality
- Removes Redundancy & Correlation
- Speeds Up Training
- Improves Model Performance
- Enhances Data Visualization

When to Use PCA?

- When the dataset has many features and suffers from the curse of dimensionality.
- When features are highly correlated.
- When the goal is to visualize data in 2D or 3D.
- When computational efficiency needs to be improved.

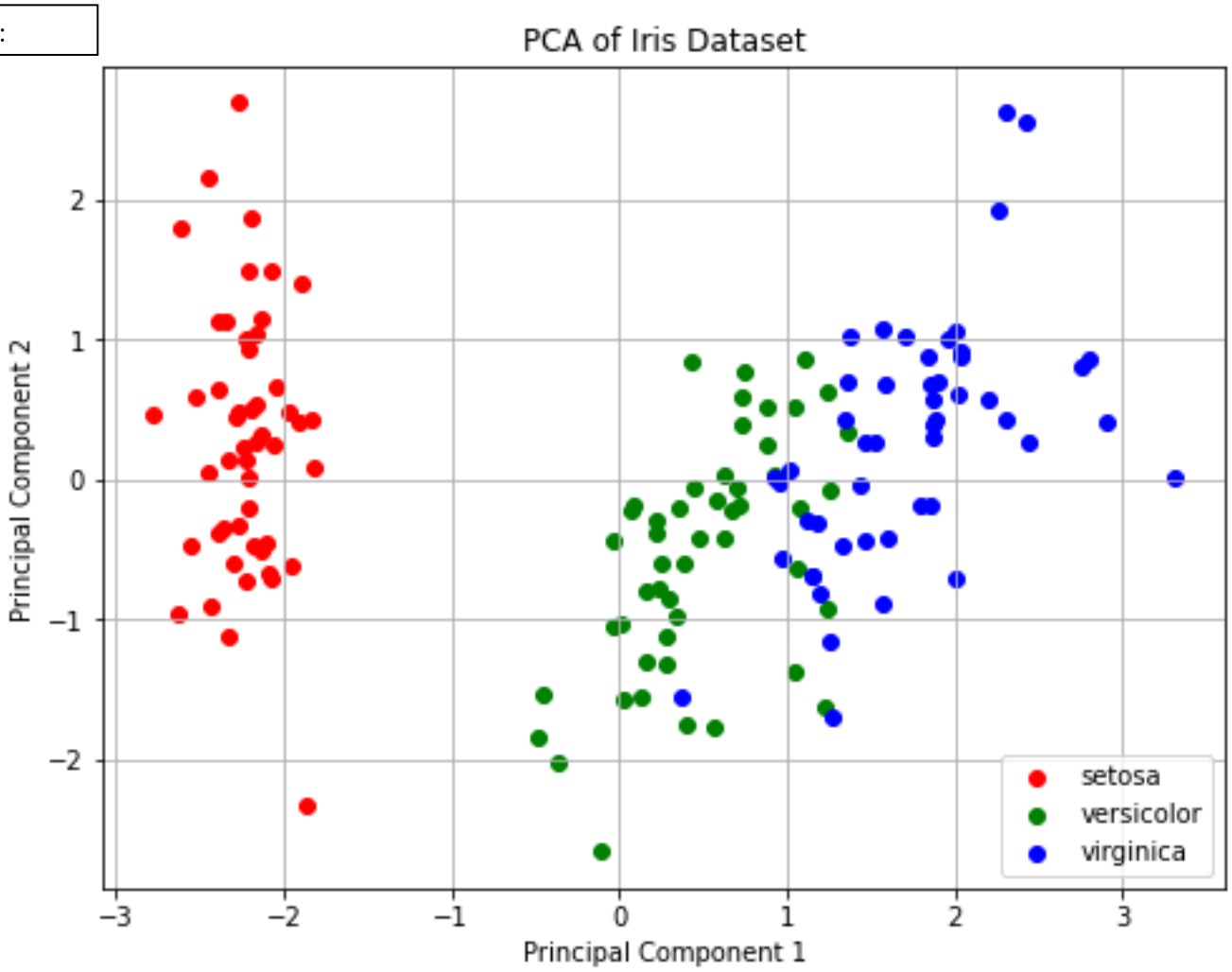
Steps to apply PCA to the Iris dataset:

1. Load the Iris dataset.
2. Standardize the features.
3. Apply PCA to reduce dimensions (e.g., from 4D to 2D for visualization).
4. Visualize the transformed data.

The Iris dataset is a well-known dataset in machine learning, containing 150 samples of iris flowers categorized into three species:

1. Setosa, 2. Versicolor 3.Virginica

Output:



Explained variance ratio: [0.72962445 0.22850762]

Program – 4

For a given set of training data examples stored in a .CSV file, implement and demonstrate the Find-Salgorithm to output a description of the set of all hypotheses consistent with the training examples.

```
import pandas as pd

training_data= pd.read_csv("dataset.csv")
print(training_data)
h=list()

for iin training_data.iloc[0]:
      h.append(i)
h.pop()

print('\n Initial value of hypotheses is\n',h)

for j in range(1,training_data.shape[0]):
      if training_data.iloc[j,training_data.shape[1]-1]=='Yes':
            for iin range(0,training_data.shape[1]-1):
                  if h[i]!=training_data.iloc[j,i]:
                        h[i]='?'

print('\n Final value of hypotheses (G) is\n ', h)
```

Explanation:

Key Concepts of Find-S Algorithm:

- The Find-S algorithm is a simple machine learning algorithm used to find the most specific hypothesis that fits all the positive examples in a given dataset.
- It particularly focuses on positive training examples while ignoring negative ones.
- The algorithm tries to identify essential patterns in features that lead to positive outcomes.
- Gradually generalizes this hypothesis as it processes positive examples
- Uses three types of values in hypothesis:
    - Specific values (required conditions)
    - '?' (any value allowed)
    - 'ϕ' (null/initial state)

Example:

Let's say we have a dataset for determining whether a day is good for playing tennis, based on weather conditions.

Outlook	Temperature	Humidity	Wind	PlayTennis
Sunny	Warm	Normal	Strong	Yes
Sunny	Warm	High	Strong	Yes
Rainy	Cold	High	Strong	No
Sunny	Warm	High	Weak	Yes

1. **Initialize the most specific hypothesis:**

   h=("ϕ", ϕ, ϕ, ϕ) (empty values for attributes)

2. **Process positive examples:**

   - First example (Sunny, Warm, Normal, Strong → **Yes**):

     h=("Sunny", "Warm", "Normal", "Strong")

   - Second example (Sunny, Warm, High, Strong →***Yes):***

     h=("Sunny", "Warm", "?" , "Strong")

   - Third example is negative → ignored.

   - Fourth example (Sunny, Warm, High, Weak → **Yes**):

     h=("Sunny", "Warm", "?", "?")

   - Final hypothesis:

     ("Sunny", "Warm", "?", "?")

This means the algorithm has generalized the concept of a good day for playing tennis as a Sunny and Warm day, with humidity and wind conditions being irrelevant.

Output:

```
  Outlook Temperature Humidity    Wind Play Tennis
0   Sunny        Warm   Normal  Strong         Yes
1   Sunny        Warm     high  Strong         Yes
2   Rainy        Cold     high  Strong          No
3   Sunny        Warm     high    Weak         Yes

 Initial value of hypotheses is
 ['Sunny', 'Warm', 'Normal', 'Strong']

 Final value of hypotheses (G) is
  ['Sunny', 'Warm', '?', '?']
```

Program – 5

Develop a program to implement k-Nearest Neighbour algorithm to classify the randomly generated 100 values of x in the range of [0,1]. Perform the following based on dataset generated.

1. **Label the first 50 points $\{x_1,\ldots\ldots,x_{50}\}$ as follows: if $(x_i \leq 0.5)$, then $x_i \in Class_1$, else $x_i \in Class_2$**
2. **Classify the remaining points, $x_{51},\ldots\ldots,x_{100}$ using KNN. Perform this for k=1,2,3,4,5,20,30**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

# Generate 100 random values of x in the range [0, 1]
x = np.random.rand(100)
print(x)

# Label the first 50 points
labels = np.array([1 if xi <= 0.5 else 2 for xi in x[:50]])

# Prepare the dataset for KNN
X_train = x[:50].reshape(-1, 1) # First 50 points (training data)
y_train = labels
X_test = x[50:].reshape(-1, 1) # Remaining 50 points (test data)

# Perform KNN classification for different values of k
k_values = [1, 2, 3, 4, 5, 20, 30]
results = {}

for k in k_values:
      knn = KNeighborsClassifier(n_neighbors=k)
      print(knn)
      knn.fit(X_train, y_train)
      predictions = knn.predict(X_test)
      print(predictions)
      results[k] = predictions

# Visualize the results
plt.figure(figsize=(12, 8))
for k, predictions in results.items():
      plt.scatter(x[50:], [k] * 50, c=predictions, cmap='coolwarm',
      label=f'k={k}', alpha=0.7)

plt.xlabel('x (Test Data)')
plt.ylabel('k Value')
plt.title('KNN Classification of Randomly Generated Data')
plt.colorbar(label='Class')
plt.legend()
plt.grid()
plt.show()
```

```
# Print predictions for each k
for k, predictions in results.items():
      print(f"k={k}: {predictions}")
```

Explanation:

- This program demonstrates how to perform K-Nearest Neighbor (K-NN) classification on a randomly generated dataset.
- The dataset consists of a set of 100 random values (between 0 and 1), and these values are labeled as either 1 or 2 based on certain conditions.
- We use the K-NN algorithm to classify the second half of the points and visualize the results for different values of k.

Key Concepts of K-NN Algorithm:

- In the K-NN algorithm, we classify a new data point by looking at the classes of the **k nearest neighbors** (based on distance).
- **Training Data**: The data on which the model is trained.
- **Test Data:** The data used to evaluate how well the model performs after being trained.
- **Distance Calculation:** The distance metric used in K-NN (like Euclidean distance) helps to find the nearest neighbors.
- Different K Values: The k value influences the model's behavior:
    - A small value of k (like 1) can lead to a model that is very sensitive to noise and can overfit.
    - A larger value of k results in a smoother model that is less sensitive to noise but might underfit if k is too large.
- The plot visualizes how the K-NN algorithm classifies the test data points for different values of k. Each k gives a different set of predictions, and you can observe how the decision boundaries change with different values of k.
- The steps followed in this program are:
    - **Importing Libraries**:
        - numpy (np)
        - matplotlib.pyplot (plt)

- KNeighborsClassifier from sklearn: The K-NN classifier that we'll use to perform classification.

- **Generate Random Data**:
  - We generate 100 random values between 0 and 1 using np.random.rand(100).
  - This creates an array x of 100 floating-point numbers. These numbers will serve as our input features.
- **Labeling the Data**:
  - This creates a label array labels for the first 50 points where each point gets a class label 1 or 2 based on the condition.
- **Preparing Training and Testing Data**:
  - X_train: We take the first 50 points (from the x array) and reshape it into a 2D array (-1, 1 reshapes it into a column vector). This will be used as the training data.
  - y_train: This is the corresponding label for each of the 50 training points.
  - X_test: The remaining 50 points (from index 50 to 99) will be used as test data to predict the labels based on the training data.
- **Performing K-NN for Different K Values**:
  - We create a list k_values that contains different values of K
  - For each k in k_values, Initialize a KNeighborsClassifier with the current k value.
  - Fit the K-NN model using X_train and y_train
  - Use the model to **predict** the labels for the test data X_test.
- **Visualizing the Results**:
  - It creates a **scatter plot** to visualize how K-NN performs for different values of k.
  - For each k, we plot the test data points (x[50:]), where the x-axis is the test data and the y-axis represents the k value used in the K-NN algorithm

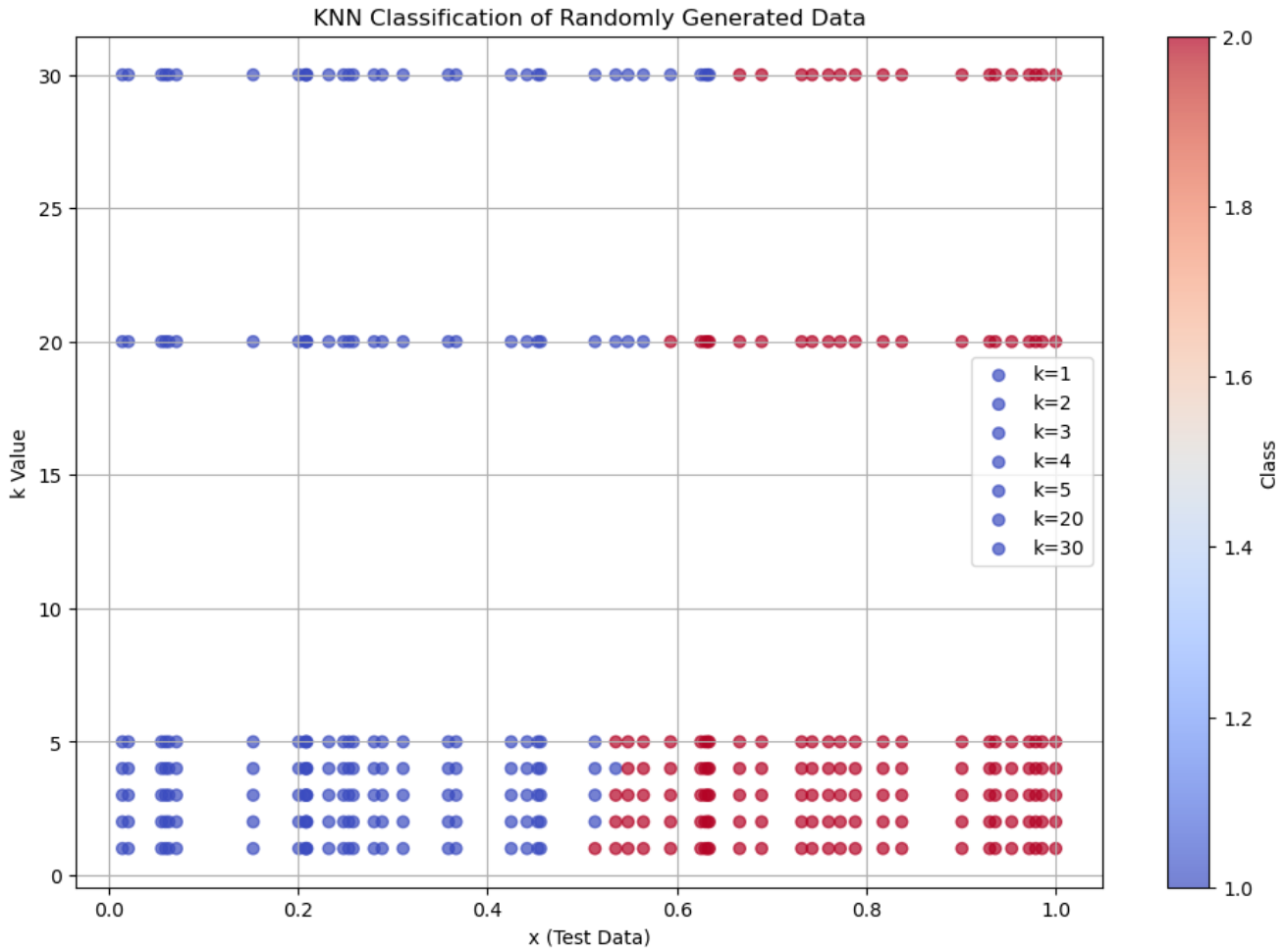
- **Printing the Predictions:**
  - This prints the predicted labels for the test data for each value of k.
  - It shows how the predictions change as the number of nearest neighbors (k) changes.

Output:

Display of 100 randomly generated data

```
[0.71283195 0.4383753  0.58132208 0.3636362  0.38548475 0.33021979
 0.09672451 0.42878053 0.25789024 0.36822185 0.67095876 0.97944448
 0.10619639 0.07288359 0.15178481 0.49976332 0.33633884 0.47451458
 0.03012453 0.02187612 0.96831355 0.02303831 0.82567617 0.32265233
 0.41774645 0.55911857 0.50534198 0.63033585 0.49458131 0.294335
 0.36594394 0.82054683 0.7515052  0.36152738 0.28191983 0.40610883
 0.0092991  0.1711654  0.05462658 0.60256673 0.2884183  0.17382127
 0.41049488 0.61952656 0.94024801 0.03166984 0.44174096 0.72500128
 0.64630229 0.3357605  0.20929809 0.59326147 0.73160957 0.53538658
 0.25827435 0.28038274 0.78822853 0.51351526 0.24812398 0.44177322
 0.54837034 0.63244622 0.76007015 0.93009753 0.62970932 0.74261461
 0.97869346 0.36690414 0.06011774 0.23256194 0.83728267 0.02097622
 0.98540811 0.66614464 0.45323671 0.42503363 0.56461055 0.15269286
 0.25361919 0.2008622  0.81753127 0.97187026 0.77242059 0.20907754
 0.62520762 0.90078647 0.99976421 0.07177898 0.35875266 0.28889799
 0.68937949 0.01466853 0.05601201 0.31095987 0.6341138  0.20752202
 0.93596276 0.45602696 0.06365237 0.9532899 ]
```

```
KNeighborsClassifier(n_neighbors=1)
[1 2 2 2 1 1 2 2 1 1 2 2 2 2 2 2 2 1 1 1 2 1 2 2 1 1 2 1 1 1 2 2 2 1 2 2 2
 1 1 1 2 1 1 1 2 1 2 1 1 2]
KNeighborsClassifier(n_neighbors=2)
[1 2 2 2 1 1 2 1 1 1 2 2 2 2 2 2 2 1 1 1 2 1 2 2 1 1 2 1 1 1 2 2 2 1 2 2 2
 1 1 1 2 1 1 1 2 1 2 1 1 2]
KNeighborsClassifier(n_neighbors=3)
[1 2 2 2 1 1 2 1 1 1 2 2 2 2 2 2 2 1 1 1 2 1 2 2 1 1 2 1 1 1 2 2 2 1 2 2 2
 1 1 1 2 1 1 1 2 1 2 1 1 2]
KNeighborsClassifier(n_neighbors=4)
[1 2 2 1 1 1 2 1 1 1 2 2 2 2 2 2 2 1 1 1 2 1 2 2 1 1 2 1 1 1 2 2 2 1 2 2 2
 1 1 1 2 1 1 1 2 1 2 1 1 2]
KNeighborsClassifier()
[1 2 2 2 1 1 2 1 1 1 2 2 2 2 2 2 2 1 1 1 2 1 2 2 1 1 2 1 1 1 2 2 2 1 2 2 2
 1 1 1 2 1 1 1 2 1 2 1 1 2]
KNeighborsClassifier(n_neighbors=20)
[1 2 2 1 1 1 2 1 1 1 1 2 2 2 2 2 2 1 1 1 2 1 2 2 1 1 1 1 1 1 2 2 2 1 2 2 2
 1 1 1 2 1 1 1 2 1 2 1 1 2]
KNeighborsClassifier(n_neighbors=30)
[1 1 2 1 1 1 2 1 1 1 1 1 2 2 1 2 2 1 1 1 2 1 2 2 1 1 1 1 1 1 2 2 2 1 1 2 2
 1 1 1 2 1 1 1 1 1 2 1 1 2]
```


KNN Classification of Randomly Generated Data
k Value
x (Test Data)
Class
k=1
k=2
k=3
k=4
k=5
k=20
k=30
0
5
10
15
20
25
30
0.0
0.2
0.4
0.6
0.8
1.0
1.0
1.2
1.4
1.6
1.8
2.0

Program – 6

Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

```
import matplotlib.pyplot as plt # plot function is used
from scipy.interpolate import interp1d # plotting for fixed points
import statsmodels.api as sm #statistical tool testing

x=[i/5.0 for i in range(30)] # x=0-30

y=[1,2,1,2,3,4,2,4,3,5,7,8,7,8,9,10,9,10,9,10,11,13,14,13,13,15,16,17,18,20]


lowess=sm.nonparametric.lowess(y,x) # locally weighted scatter point smoothing
lowess_x=list(zip(*lowess))[0]
lowess_y=list(zip(*lowess))[1]

f=interp1d(lowess_x,lowess_y,bounds_error=False)

xnew=[i/10.0 for i in range(100)]
ynew=f(xnew)
plt.plot(x,y,'o')
plt.plot(lowess_x,lowess_y,'*')
plt.plot(xnew,ynew,'-')
plt.show()
```

Explanation:

Locally Weighted Regression:

Locally Weighted Regression (LWR) is a **non-parametric** algorithm, which means it **does not make strong assumptions** about the shape of the data (like linear or polynomial).

Instead, it fits the model **locally** — meaning it focuses on a small portion of the data around the point where you want a prediction.

Working:

- Pick a point (x) where you want to predict the output (y).
- Look around nearby data points.
- It gives more weight to data points that are closer to x.
- Farther points get less weight.

- Fit a simple model (like linear regression) using these weighted points.
- Use that local model to predict the value at x.

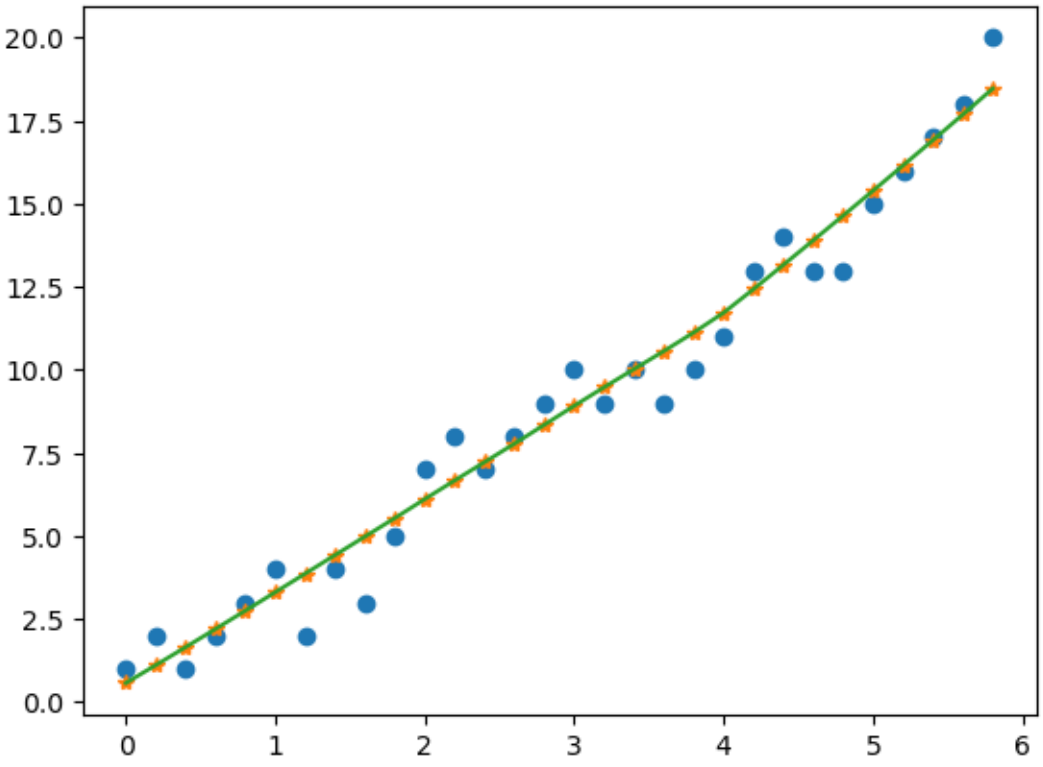
Steps of the program explained:

1. Importing necessary libraries
2. Creating the data (x and y)
3. Applying LOWESS (Locally Weighted Scatterplot Smoothing)
   → lowess(y, x) smooths the noisy data by fitting small local regressions.
4. Separating smoothed x and y
   → zip(*lowess) separates the paired points lowess_x and lowess_y.
5. Creating an interpolating function
   → interp1d creates a function f() that can give **interpolated y-values** for **any x** within the smoothed range.
6. Create new x-values for smooth plotting
7. Compute new y-values using interpolation
8. Plotting everything.

Summary:

- You're smoothing noisy data using LOWESS.
- Then you make the smoothed data even smoother using interpolation for plotting.
- This helps you see trends clearly in data that’s not linear.

Output:



Program – 7

Develop a program to demonstrate the working of Linear Regression and Polynomial Regression. Use Boston Housing Dataset for Linear Regression and Auto MPG Dataset (for vehicle fuel efficiency prediction) for Polynomial Regression.

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline

# Load the California Housing dataset as a DataFrame

#data = fetch_california_housing(as_frame=True)

data=pd.read_csv('BostonHousing.csv')
housing_df = data  # The dataset as a Pandas DataFrame
print(housing_df)

# List of numerical features

numerical_features = housing_df.columns


#Compute Correlation & Select
Feature corr_matrix_boston =
housing_df.corr()
top_feature_boston = corr_matrix_boston['medv'].drop('medv').idxmax()

#Prepare Data
X_Boston = housing_df[[top_feature_boston]] #Selecting the most correlated
feature y_Boston = housing_df['medv']
X_train_Boston, X_test_Boston, y_train_Boston, y_test_Boston =
    train_test_split( X_Boston, y_Boston, test_size=0.2,
    random_state=42)

#Train Linear Regression Model
lin_reg_Boston = LinearRegression()
lin_reg_Boston.fit(X_train_Boston, y_train_Boston)

#Make Predictions and Compute Error
y_pred_Boston = lin_reg_Boston.predict(X_test_Boston)
mse_Boston = mean_squared_error(y_test_Boston, y_pred_Boston)
print(f'Boston Housing Linear Regression MSE: {mse_Boston:.4f}')

#Plot results
plt.figure(figsize=(10,5))
sns.scatterplot(x=X_test_Boston[top_feature_boston], y=y_test_Boston,
label='Actual')
sns.lineplot(x=X_test_Boston[top_feature_boston].values.flatten(),
y=y_pred_Boston, color='red', label = 'Predicted')
```

```
plt.xlabel(top_feature_boston)
plt.ylabel('medval')
plt.title('Boston Housing Linear Regression')
plt.legend()
plt.show()

#Auto MPG Dataset: Polynomial
Regression
data1=pd.read_csv('autompg.csv')

m_df=data1


#numerical_features1 =
m_df.columns corr_matrix_mpg =
m_df.corr()
top_feature_mpg = corr_matrix_mpg['mpg'].drop('mpg').idxmax()

#Prepare Data
X_mpg = m_df[[top_feature_mpg]]
y_mpg = m_df['mpg']
X_train_mpg, X_test_mpg, y_train_mpg, y_test_mpg = train_test_split(X_mpg,
y_mpg, test_size=0.2, random_state = 42)

#Train Ploynomial Regression Model
poly_reg_mpg = make_pipeline(PolynomialFeatures(degree=2), StandardScaler(),
LinearRegression())
poly_reg_mpg.fit(X_train_mpg, y_train_mpg)

#Make Predictions and Compute Error
y_pred_mpg = poly_reg_mpg.predict(X_test_mpg)
mse_mpg = mean_squared_error(y_test_mpg, y_pred_mpg)
print(f'Auto MPG Polynomial Regression MSE: {mse_mpg:.4f}')

# Plot Auto MPG Polynomial Regression results

plt.figure(figsize=(10, 5))
sns.scatterplot(x=X_test_mpg[top_feature_mpg], y=y_test_mpg, label='Actual')
sns.scatterplot(x=X_test_mpg[top_feature_mpg], y=y_pred_mpg,  color='red',
label='Predicted')
plt.xlabel(top_feature_mpg)
plt.ylabel('MPG')
plt.title('Auto MPG Polynomial Regression')
plt.legend()
plt.show()
```

Explanation:

Part 1: Boston Housing – Linear Regression:

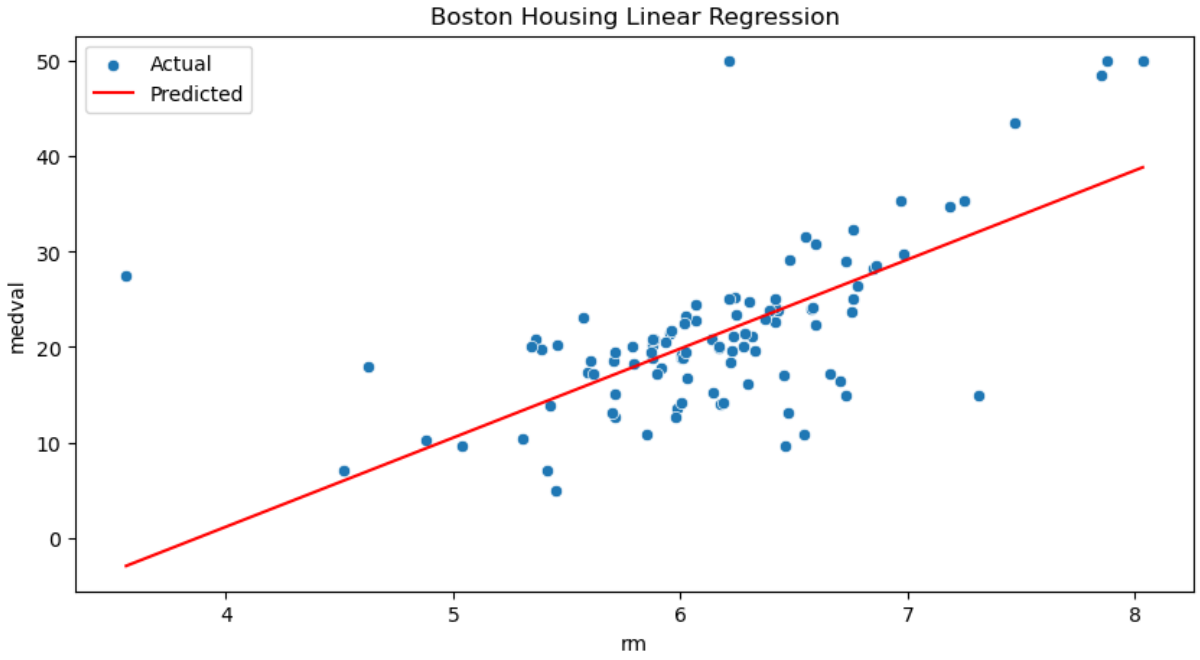
1. Importing Libraries
2. Load the Boston Housing Dataset
3. Find the Most Correlated Feature
   - You calculate correlation between all features and medv (median house value).
   - Then you find the feature most correlated with medv.
4. Prepare the Data
   - Splits data: 80% for training, 20% for testing
5. Train the Linear Regression Model
6. Make Predictions & Calculate Error
   - Predict house prices on the test set.
   - Compute Mean Squared Error (MSE) — lower is better.
7. Plot Results
   - Scatter plot: Actual prices
   - Line plot: Predicted prices (best-fit line)

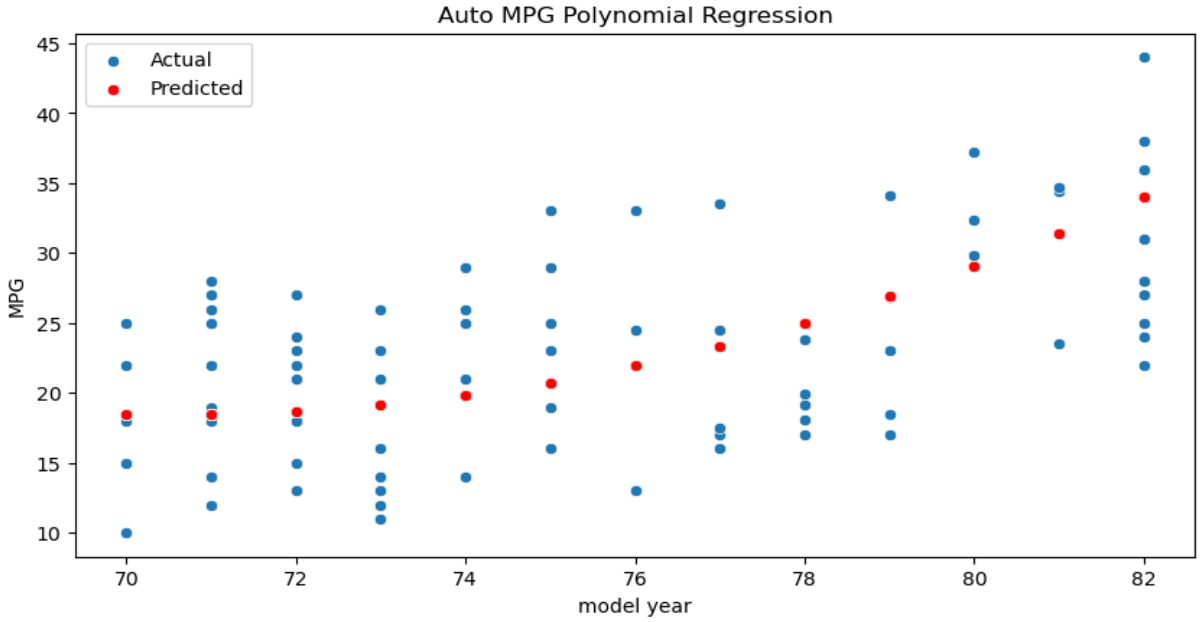
Part 2: Auto MPG – Polynomial Regression

1. Load the Dataset
2. Find Most Correlated Feature
   - Calculate correlation between all features and mpg.
   - Pick the one most correlated with mpg.
3. Prepare Data
   - Select the best feature as X, and mpg as target y.
   - Split into training and testing sets.
4. Train Polynomial Regression Model
   - Create a Polynomial Regression pipeline (degree=2).
   - Fit the model to the training data.
5. Predict and Calculate MSE
6. Plot Results

Output:

```
        crim    zn  indus  chas    nox  ...  tax  ptratio       b  lstat  medv
0    0.00632  18.0   2.31     0  0.538  ...  296     15.3  396.90   4.98  24.0
1    0.02731   0.0   7.07     0  0.469  ...  242     17.8  396.90   9.14  21.6
2    0.02729   0.0   7.07     0  0.469  ...  242     17.8  392.83   4.03  34.7
3    0.03237   0.0   2.18     0  0.458  ...  222     18.7  394.63   2.94  33.4
4    0.06905   0.0   2.18     0  0.458  ...  222     18.7  396.90   5.33  36.2
..       ...   ...    ...   ...    ...  ...  ...      ...     ...    ...   ...
501  0.06263   0.0  11.93     0  0.573  ...  273     21.0  391.99   9.67  22.4
502  0.04527   0.0  11.93     0  0.573  ...  273     21.0  396.90   9.08  20.6
503  0.06076   0.0  11.93     0  0.573  ...  273     21.0  396.90   5.64  23.9
504  0.10959   0.0  11.93     0  0.573  ...  273     21.0  393.45   6.48  22.0
505  0.04741   0.0  11.93     0  0.573  ...  273     21.0  396.90   7.88  11.9

[506 rows x 14 columns]
Boston Housing Linear Regression MSE: 46.1448
```





Program – 8

Develop a program to demonstrate the working of the decision tree algorithm. Use Breast Cancer Data set for building the decision tree and apply this knowledge to classify a new sample.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import numpy as np

# ---- 1. Load and prepare the Breast Cancer dataset ----
# Load the Breast Cancer dataset
data = load_breast_cancer()
X = data.data  # Features (30 features)
y = data.target  # Target variable (0 = malignant, 1 = benign)

# 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)

# ---- 2. Train a Decision Tree Classifier ----
# Initialize and train the Decision Tree classifier
classifier = DecisionTreeClassifier(random_state=42)
classifier.fit(X_train, y_train)

# ---- 3. Evaluate the model ----
# Predict on the test set
y_pred = classifier.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of the Decision Tree model: {accuracy * 100:.2f}%")

# ---- 4. Classify a new sample ----
# Example new sample: feature values of a new breast cancer sample
# The features must be in the same order and format as the training data.
new_sample = np.array([[15.68, 19.53, 124.5, 929.6, 0.0930, 0.0638, 0.0747, 0.0592, 0.226, 0.0393,
                        0.0318, 0.0718, 0.0197, 0.0204, 0.0326, 0.0934, 0.0425, 0.0475, 0.0741, 0.07
                        0.0405, 0.0421, 0.0808, 0.0578, 0.0451, 0.0658, 0.0749, 0.0593, 0.0331, 0.06

# Use the model to classify the new sample
new_prediction = classifier.predict(new_sample)

# Output the classification result
if new_prediction == 0:
    print("The new sample is classified as 'Malignant' (0).")
else:
    print("The new sample is classified as 'Benign' (1).")
```

Explanation:

Decision Tree Algorithm

- A **Decision Tree** is a popular **supervised machine learning** algorithm used for both **classification** and **regression** tasks.
- It works by **splitting data** into smaller and smaller groups based on certain conditions, forming a **tree-like structure**.
- Each node in the tree:
  - Represents a **decision** based on a feature (input variable).
  - The branches represent the **outcome** of the decision.
- The leaves (end points) represent the **final output** (either a class label or a value)

Output:

```
In [1]: runfile('C:/Users/JNNCE-ISE11/Downloads/programm8.py', wdir='C:/Users/JNNCE-ISE11/Downloads')
Accuracy of the Decision Tree model: 94.74%
The new sample is classified as 'Benign' (1).

In [2]: runfile('C:/Users/JNNCE-ISE11/Downloads/program9ml.py', wdir='C:/Users/JNNCE-ISE11/Downloads')
Accuracy of the Naive Bayes classifier: 77.50%
```

Program – 9

Develop a program to implement the Naive Bayesian classifier considering Olivetti Face Data set for training. Compute the accuracy of the classifier, considering a few test data sets.

```
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
data = fetch_olivetti_faces(shuffle=True, random_state=42)

X = data.data
y = data.target
#print(f"Dataset contains {X.shape[0]} samples with {X.shape[1]} features (flattened images).")
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
naive_bayes_classifier = GaussianNB()
naive_bayes_classifier.fit(X_train, y_train)
y_pred = naive_bayes_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of the Naive Bayes classifier: {accuracy * 100:.2f}%")

# Plotting the confusion matrix
fig, axes = plt.subplots(3, 5, figsize=(12, 8))
for ax, image, label, prediction in zip(axes.ravel(), X_test, y_test, y_pred):
    ax.imshow(image.reshape(64, 64), cmap=plt.cm.gray)
    ax.set_title(f"True: {label}, Pred: {prediction}")
    ax.axis('off')
plt.show()
```

Explanation:

Naïve Bayes Classifier:

- **Naive Bayes** is a **supervised machine learning** algorithm based on applying **Bayes' Theorem** with a **strong (naive) assumption**:
  All features are independent of each other given the class.
- It is mainly used for **classification tasks**, especially when the dataset is large.

How it Works:

- It calculates the **probability** of each class for a given data point.
- It **predicts** the class with the **highest probability**.

Bayes' Theorem formula:

$$P(Class|Data) = \frac{P(Data|Class) \times P(Class)}{P(Data)}$$

- P(Class|Data)P(Class | Data)P(Class|Data) = Posterior Probability (what we want)
- P(Data|Class)P(Data | Class)P(Data|Class) = Likelihood (how likely the data is, given the class)
- P(Class)P(Class)P(Class) = Prior Probability (how common the class is)
- P(Data)P(Data)P(Data) = Evidence (how common the data is overall)

Output:


True: 18, Pred: 18
True: 0, Pred: 0
True: 5, Pred: 5
True: 22, Pred: 22
True: 22, Pred: 2
True: 27, Pred: 27
True: 16, Pred: 16
True: 18, Pred: 18
True: 31, Pred: 31
True: 35, Pred: 35
True: 12, Pred: 16
True: 5, Pred: 5
True: 22, Pred: 22
True: 0, Pred: 0
True: 25, Pred: 25

Program – 10

Develop a program to implement k-means clustering using Wisconsin Breast Cancer data set and visualize the clustering result.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import seaborn as sns

# ---- 1. Load and prepare the Wisconsin Breast Cancer dataset ----
# Load the Breast Cancer dataset from sklearn
data = load_breast_cancer()

# Extract the features (X) and the target (y)
X = data.data  # Features (30 features)
y = data.target  # Target variable (0 = malignant, 1 = benign)

# Convert the dataset into a DataFrame for better visualization
df = pd.DataFrame(X, columns=data.feature_names)
df['target'] = y

# ---- 2. Standardize the features ----
# Standardize the features (important for K-means clustering)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# ---- 3. Apply K-means clustering ----
# Apply K-means with 2 clusters (malignant vs benign)
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X_scaled)

# Get the predicted clusters
df['cluster'] = kmeans.labels_
print(df.head())
# ---- 4. Visualize the Clusters ----
# We will visualize the data using the first two principal components (PCA)
from sklearn.decomposition import PCA

# Apply PCA to reduce the dimensionality to 2D
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Create a DataFrame with the PCA results
pca_df = pd.DataFrame(X_pca, columns=['PC1', 'PC2'])
pca_df['target'] = y
pca_df['cluster'] = kmeans.labels_

# Plotting the clusters
plt.figure(figsize=(10,7))
sns.scatterplot(x='PC1', y='PC2', hue='cluster', palette='viridis', data=pca_df, s=100, marker='o',
plt.title("K-means Clustering of Breast Cancer Data (PCA-reduced)", fontsize=16)
plt.xlabel("Principal Component 1", fontsize=14)
```

```
plt.ylabel("Principal Component 2", fontsize=14)
plt.legend(title='Cluster')
plt.show()

# Plot the actual target labels for comparison
plt.figure(figsize=(10,7))
sns.scatterplot(x='PC1', y='PC2', hue='target', palette='coolwarm', data=pca_df, s=100, marker='o',
plt.title("Actual Labels (Malignant vs Benign)", fontsize=16)
plt.xlabel("Principal Component 1", fontsize=14)
plt.ylabel("Principal Component 2", fontsize=14)
plt.legend(title='Target')
plt.show()
```

Explanation:

k-means Clustering:

- **K-Means** is an **unsupervised machine learning** algorithm used for **clustering** tasks. It groups similar data points together into **K different clusters** based on feature similarity.
- Partition the data into **K clusters** where each data point belongs to the cluster with the **nearest mean** (center).

How it Works (Step-by-Step):

1. **Choose K**: Decide the number of clusters (K).
2. **Initialize**: Randomly pick K points as **initial cluster centers** (called centroids).
3. **Assign**: For each point, assign it to the **nearest centroid** (using distance measures like Euclidean distance).
4. **Update**: Recalculate the **centroids** by taking the mean of all points assigned to each cluster.
5. **Repeat**:
    - Assign points to the nearest centroid.
    - Update centroids.
    - Until centroids **don't change much** (or after a fixed number of iterations).

- **Centroid**: Center point of a cluster (average of points in the cluster).
- **Inertia**: Sum of squared distances from each point to its centroid (used to measure how tight the clusters are).

Advantages:

- **Simple** and easy to understand.
- **Fast** and **efficient** for large datasets.
- Works well when clusters are **well-separated** and **spherical**.

Disadvantages:

- Must **choose K** manually.
- Sensitive to **initial centroid placement**.
- Struggles with **non-spherical** or **overlapping** clusters.
- Not good if clusters have very **different sizes** or **densities**.