MACHINE LEARNING
Lab Manual *Practical Record*

SRI CHAITANYA

INTEGRATED CAMPUS

IBRAHIMPATNAM, HYD.

SRI CHAITANYA

TECHNICAL CAMPUS

COLLEGE OF ENGINEERING & TECHNOLOGY
COLLEGE OF BUSINESS MANAGEMENT
(Approved by AICTE, NEW DELHI & Affiliated to JNTU, Hyderabad)

www.srichaitanyaengg.com
E-mail : director8a.sctc@gmail.com

Sheriguda (V), Ibrahimpatnam (M), R.R. Dist. - 501 510 - A.P.
Ph : 08414 - 223222, 223223 Fax : 08414 - 222678

MCA II -I Semester

SRI CHAITANYA
TECHNICAL CAMPUS

COLLEGE OF ENGINEERING & TECHNOLOGY
COLLEGE OF BUSINESS MANAGEMENT

(Approved by AICTE, NEW DELHI & Affiliated to JNTU, Hyderabad)

Sheriguda (V), Ibrahimpatnam (M), R.R. Dist. - 501 510 - A.P.

CERTIFICATE

This is to certify that Mr / Ms ________________ has satisfactorily completed experiments in **Machine Learning** laboratory as prescribed by Jawaharlal Nehru Technological University, Hyderabad.

Department **Master of Computer Applications** Roll No________________

Branch **MCA** Academic Year **2025-2026**

INTERNAL EXAMINER HEAD OF THE DEPT.

EXTERNAL EXAMINER PRINCIPAL

INDEX

Sl.No.	Date	Name of the Experiment	Page No.	Remarks

MACHINE LEARNING LAB

MCA II Year I Sem.

L	T	P	C
0	1	2	2

Course Objective:

- The objective of this lab is to get an overview of the various machine learning
- Techniques and can demonstrate them using python.

Course Outcomes:

- Understand modern notions in predictive data analysis
- Select data, model selection, model complexity and identify the trends
- Understand a range of machine learning algorithms along with their strengths and weaknesses
- Build predictive models from data and analyze their performance

List of Experiments

1. Write a python program to compute Central Tendency Measures: Mean, Median, Mode Measure of Dispersion: Variance, Standard Deviation
2. Study of Python Basic Libraries such as Statistics, Math, Numpy and Scipy
3. Study of Python Libraries for ML application such as Pandas and Matplotlib
4. Write a Python program to implement Simple Linear Regression
5. Implementation of Multiple Linear Regression for House Price Prediction using sklearn
6. Implementation of Decision tree using sklearn and its parameter tuning
7. Implementation of KNN using sklearn
8. Implementation of Logistic Regression using sklearn
9. Implementation of K-Means Clustering
10. Performance analysis of Classification Algorithms on a specific dataset (Mini Project)

TEXT BOOK:

1. Machine Learning – Tom M. Mitchell, - MGH

REFERENCE:

1. Machine Learning: An Algorithmic Perspective, Stephen Marshland, Taylor & Francis

MACHINE LEARNING LAB PROGRAMS (R22)

PROGRAM-1

1) **Write a python program to compute Central Tendency Measures: Mean, Median, Mode Measureof Dispersion: Variance, Standard Deviation**

```
import statistics


def compute_statistics(data):
    if not data:
        return "Data list is empty."


    # Central Tendency Measures
    mean = statistics.mean(data)
    median = statistics.median(data)
    # Handling mode for multimodal data
    try:
        mode = statistics.mode(data)
    except statistics.StatisticsError:
        mode = "No unique mode"


    # Measure of Dispersion
    variance = statistics.variance(data)
    standard_deviation = statistics.stdev(data)


    return {
        "Mean": mean,
        "Median": median,
        "Mode": mode,
        "Variance": variance,
        "Standard Deviation": standard_deviation
    }


if __name___== "__main__":
    # Example data
    data = [10, 20, 20, 30, 40, 50]

    # Compute statistics
    results = compute_statistics(data)

    # Print the results
    print("Statistics Results:")
    for key, value in results.items():
        print(f"{key}: {value}")
```

```
Statistics Results:
Mean: 26.666666666666668
Median: 20.0
Mode: 20
Variance: 155.33333333333334
Standard Deviation: 12.46919436720233
```

PROGRAM-2

2) **Study of Python Basic Libraries such as Statistics, Math, Numpy and Scipy**

To understand Python's basic libraries like **Statistics**, **Math**, **Numpy**, and **Scipy**, it's important to know what each one does and how they are used. Here's an overview:

---

1. Statistics Library

The `statistics` library in Python provides functions for statistical operations. It includes basic statistical functions, such as measures of central tendency (mean, median, mode), variance, and standard deviation.

Common Functions:

- `mean(data)`: Returns the arithmetic mean of the data.
- `median(data)`: Returns the median of the data.
- `mode(data)`: Returns the mode (most frequent value) of the data.
- `variance(data)`: Returns the variance of the data.
- `stdev(data)`: Returns the standard deviation of the data.

```
python
Copy code
```

SOURCE CODE:-

```
import statistics

data = [1, 2, 3, 4, 5]
print(statistics.mean(data))  # 3
print(statistics.stdev(data))  # 1.58
```

OUTPUT:-

```
3
1.5811388300841898
```

---

2. Math Library

The `math` library in Python provides mathematical functions. It includes basic operations such as trigonometric functions, logarithms, powers, and more.

Common Functions:

- `math.sqrt(x)`: Returns the square root of x.
- `math.pow(x, y)`: Returns x raised to the power y.
- `math.sin(x)`,`math.cos(x)`,`math.tan(x)`: Trigonometric functions.
- `math.log(x, base)`: Returns the logarithm of x with a specified base.
- `math.factorial(x)`: Returns the factorial of x.

```
python
Copy code
```

SOURCE CODE:-

```
import math

print(math.sqrt(16))  # 4
print(math.factorial(5))  # 120
```

OUTPUT:-

```
4.0
120
```

3. Numpy Library

`NumPy` (Numerical Python) is a core library for scientific computing in Python. It provides powerful tools for working with large datasets and arrays, along with a wide range of mathematical and statistical operations.

Key Features:

- **ndarray**: A fast, flexible, and efficient array object.
- **Array operations**: You can perform element-wise operations on arrays (e.g., addition, multiplication).
- **Linear algebra**: Functions like dot products, matrix multiplication, and eigenvalue decomposition.
- **Random**: Random number generation.

Common Functions:

- `numpy.array()`: Creates an array from a list or another array.
- `numpy.mean()`: Returns the mean of an array.
- `numpy.dot()`: Computes the dot product of two arrays.
- `numpy.linspace()`: Creates a sequence of evenly spaced values.

```
python
Copy code
```

SOURCE CODE:-

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(np.mean(arr))  # 3.0
```

OUTPUT:-

```
3.0
```

4. Scipy Library

`SciPy` is built on top of NumPy and provides a set of tools for scientific and technical computing. It includes modules for optimization, integration, interpolation, eigenvalue problems, and other advanced mathematical operations.

Key Features:

- **Optimization**: Minimization and maximization of functions.
- **Integration**: Numerical integration of functions.
- **Interpolation**: Functions for interpolating data points.
- **Linear Algebra**: Provides functions for solving linear algebra problems like eigenvalues.

Common Functions:

- `scipy.integrate.quad()`: Performs integration.
- `scipy.optimize.minimize()`: Minimizes a function.
- `scipy.linalg.inv()`: Computes the inverse of a matrix.

```
python
Copy code
```

SOURCE CODE:-

```
from scipy import integrate

def func(x):
    return x**2

result, error = integrate.quad(func, 0, 1)
print(result)  # 0.3333
```

OUTPUT:-

```
0.33333333333333337
```

Summary of Differences:

- **Statistics**: Basic statistical operations on data.
- **Math**: Mathematical operations like square roots, trigonometric functions, and logarithms.
- **Numpy**: High-performance array and matrix manipulation and mathematical operations.
- **Scipy**: Advanced scientific computing tools built on top of Numpy for optimization, integration, and more.

These libraries are highly useful for data analysis, scientific computing, and problem-solving in Python. You can combine them for even more powerful applications!

PROGRAM-3

3) Study of Python Libraries for Machine Learning Applications: Pandas and Matplotlib

---

1. Pandas Library

Pandas is a powerful and flexible library used for **data manipulation, analysis**, and **cleaning**. It is one of the most widely used libraries in Machine Learning pipelines because of its ability to handle structured data efficiently.

Key Features of Pandas

- **Data Structures**:
  - `Series`: One-dimensional labeled array (like a column in Excel).
  - `DataFrame`: Two-dimensional tabular data with rows and columns.
- **Data Cleaning**: Handling missing values, duplicates, and data transformation.
- **Data Filtering and Slicing**: Accessing subsets of data.
- **Aggregation and Grouping**: Summarizing large datasets.
- **Integration**: Easily integrates with libraries like `NumPy`, `Matplotlib`, and `scikit-learn`.

Common Pandas Operations

Below are the basic operations in Pandas used for Machine Learning:

1. Creating DataFrames and Series

```
python
Copy code
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'Age': [25, 30, 35, 40],
    'Salary': [40000, 50000, 60000, 70000],
    'City': ['New York', 'Chicago', 'San Francisco', 'Houston']
}
df = pd.DataFrame(data)
print(df)
```

2. Reading and Writing Data

```
python
Copy code
# Reading from a CSV file
df = pd.read_csv('data.csv')

# Writing to a CSV file
df.to_csv('output.csv', index=False)
```

3. Inspecting and Understanding Data

```
python
Copy code
```

```
print(df.head())        # First 5 rows
print(df.describe())    # Summary statistics
print(df.info())        # Data types and non-null counts
```

4. Handling Missing Data

```
python
Copy code
# Replace missing values with a specific value
df.fillna(0, inplace=True)

# Drop rows with missing values
df.dropna(inplace=True)
```

5. Filtering and Querying Data

```
python
Copy code
# Filter rows where Age > 30
filtered_df = df[df['Age'] > 30]

# Using query() for filtering
filtered_df = df.query('Salary > 50000')
```

6. Grouping and Aggregation

```
python
Copy code
# Group by 'City' and calculate mean Salary
grouped = df.groupby('City')['Salary'].mean()
print(grouped)
```

2. Matplotlib Library

Matplotlib is a comprehensive library for **data visualization**. It is used to create static, interactive, and animated visualizations in Python.

Key Features of Matplotlib

- **Line Plots**: Visualize trends over time.
- **Bar Charts**: Compare categories.
- **Histograms**: Understand data distributions.
- **Scatter Plots**: Identify relationships between two variables.
- **Customization**: Full control over plots (titles, labels, legends, etc.).

Common Matplotlib Operations

1. Basic Line Plot

```
python
Copy code
import matplotlib.pyplot as plt

# Sample Data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create a line plot
plt.plot(x, y, label='Line', color='blue', linestyle='--')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot Example')
plt.legend()
plt.show()
```

2. Bar Chart

```
python
Copy code
# Bar Chart Example
categories = ['A', 'B', 'C', 'D']
values = [4, 7, 1, 8]

plt.bar(categories, values, color='skyblue')
plt.title('Bar Chart Example')
plt.show()
```

3. Scatter Plot

```
python
Copy code
# Scatter Plot Example
x = [1, 2, 3, 4, 5]
y = [5, 7, 6, 8, 9]

plt.scatter(x, y, color='red')
plt.title('Scatter Plot Example')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

4. Histogram

```
python
Copy code
# Histogram Example
import numpy as np

data = np.random.randn(1000)
plt.hist(data, bins=30, color='purple', alpha=0.7)
plt.title('Histogram Example')
plt.show()
```

5. Subplots

```
python
Copy code
# Subplots Example
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.figure(figsize=(10, 5))

# First Subplot
plt.subplot(1, 2, 1)
plt.plot(x, y, color='green')
plt.title('Line Plot')

# Second Subplot
plt.subplot(1, 2, 2)
plt.bar(x, y, color='orange')
plt.title('Bar Chart')

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

How Pandas and Matplotlib Are Used in ML Applications

1. **Pandas**:
    - Loading and preparing data (CSV, Excel, etc.).

- Cleaning, transforming, and filtering data for feature selection.
- Aggregating and summarizing data for better understanding.
- Creating input features for Machine Learning models.

2. **Matplotlib**:
    - Visualizing relationships between features using scatter plots and line charts.
    - Plotting histograms to understand data distributions.
    - Analyzing trends and patterns in the dataset.
    - Visualizing the predictions and model performance (e.g., line plots for regression).

Combining Pandas and Matplotlib Example

```
python
Copy code
import pandas as pd
import matplotlib.pyplot as plt

# Sample Dataset
data = {
    'Square_Feet': [1400, 1600, 1700, 1875, 1100, 1550],
    'Price': [245000, 312000, 279000, 308000, 199000, 219000]
}

# Create DataFrame
df = pd.DataFrame(data)

# Scatter Plot to visualize relationship between Square Feet and Price
plt.scatter(df['Square_Feet'], df['Price'], color='blue')
plt.xlabel('Square Feet')
plt.ylabel('Price')
plt.title('House Price vs. Square Feet')
plt.show()
```

Summary

- **Pandas** helps you **load, clean, analyze, and manipulate** data efficiently for Machine Learning.
- **Matplotlib** allows you to **visualize data**, helping in understanding relationships and patterns in datasets.

These libraries are fundamental in any **Machine Learning workflow**, where data understanding and visualization play a significant role in developing high-performing models.

PROGRAM-4

4) Write a Python program to implement Simple Linear Regression

Here's an implementation of **Simple Linear Regression** in Python using the **least squares method** for calculating the slope and intercept of the regression line.

The program does not rely on external libraries like `sklearn`; instead, it uses basic Python and `matplotlib` for visualization.

Simple Linear Regression Program

```
python
Copy code
```

SOURCE:-

```python
import matplotlib.pyplot as plt

# Function to calculate mean
def mean(values):
    return sum(values) / len(values)

# Function to calculate the slope (m) and intercept (b) for y = mx + b
def simple_linear_regression(x, y):
    n = len(x)

    # Calculate the mean of x and y
    mean_x = mean(x)
    mean_y = mean(y)

    # Calculate the numerator and denominator for the slope
    numerator = sum((x[i] - mean_x) * (y[i] - mean_y) for i in range(n))
    denominator = sum((x[i] - mean_x) ** 2 for i in range(n))

    slope = numerator / denominator
    intercept = mean_y - slope * mean_x

    return slope, intercept

# Function to predict y values given x
def predict(x, slope, intercept):
    return [slope * xi + intercept for xi in x]

# Main function
def main():
    # Sample data (X - independent variable, Y - dependent variable)
    x = [1, 2, 3, 4, 5]
    y = [2, 4, 5, 4, 5]

    # Perform simple linear regression
    slope, intercept = simple_linear_regression(x, y)
    print(f"Slope (m): {slope:.2f}")
    print(f"Intercept (b): {intercept:.2f}")
    print(f"Equation of regression line: y = {slope:.2f}x + {intercept:.2f}")

    # Predict values
    y_pred = predict(x, slope, intercept)

    # Plot the original data and the regression line
    plt.scatter(x, y, color='blue', label='Original Data')
    plt.plot(x, y_pred, color='red', label='Regression Line')
    plt.xlabel('X - Independent Variable')
    plt.ylabel('Y - Dependent Variable')
    plt.legend()
    plt.title('Simple Linear Regression')
    plt.show()

if __name___== "__main__":
    main()
```

Program Explanation

1. **Data Input**:
   - `x` contains the independent variable values.
   - `y` contains the dependent variable values.
2. **Mean Calculation**:
   - A helper function `mean()` computes the average of a list.
3. **Slope and Intercept**:
   - The **slope** mmm and **intercept** bbb of the regression line are calculated using the least squares formulas: m=∑(xi−x‾)(yi−y‾)∑(xi−x‾)2m = \frac{\sum{(x_i - \bar{x})(y_i - \bar{y})}}{\sum{(x_i - \bar{x})^2}}m=∑(xi−x‾)2∑(xi−x‾)(yi−y‾) b=y‾−m⋅x‾b = \bar{y} - m \cdot \bar{x}b=y‾−m⋅x‾
4. **Prediction**:
   - The `predict()` function calculates the predicted yyy-values for given xxx-values using the regression line equation y=mx+by = mx + by=mx+b.
5. **Visualization**:
   - The original data is plotted as a scatter plot.
   - The regression line is overlaid to show the fit.

Sample Output

For the provided data x=[1,2,3,4,5]x = [1, 2, 3, 4, 5]x=[1,2,3,4,5] and y=[2,4,5,4,5]y = [2, 4, 5, 4, 5]y=[2,4,5,4,5], the output will be:

```
arduino
Copy code
Slope (m): 0.70
Intercept (b): 2.40
Equation of regression line: y = 0.70x + 2.40
```

A graph will display:

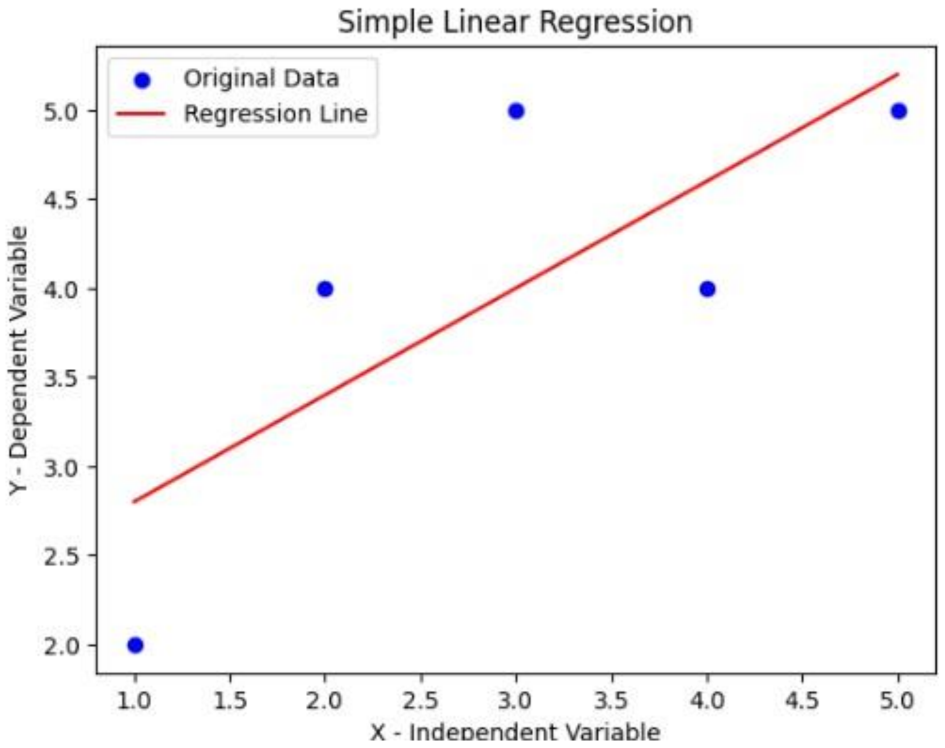
- Blue points (data points).
- A red line (the regression line) showing the best fit.

Notes:

1. **Manual Calculation**: This program calculates slope and intercept manually using basic arithmetic.
2. **Visualization**: `matplotlib` is used to visualize the regression line and the data points
3. **Extensibility**: The code can be extended to include external datasets and larger inputs.

OUTPUT:-

```
Slope (m): 0.60
Intercept (b): 2.20
Equation of regression line: y = 0.60x + 2.20
```



PROGRAM-5

5) Implementation of Multiple Linear Regression for House Price Prediction using sklearn

Here's a Python implementation of **Multiple Linear Regression** using the **scikit-learn** library. This program predicts **house prices** based on multiple input features like square footage, number of bedrooms, and age of the house.

Code Implementation: Multiple Linear Regression

```
python
Copy code
```

SOURCE CODE:-

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

# Sample dataset
data = {
    'Square_Feet': [1400, 1600, 1700, 1875, 1100, 1550, 2350, 2450, 1425, 1700],
    'Bedrooms': [3, 3, 3, 4, 2, 3, 4, 4, 3, 2],
    'Age': [10, 15, 20, 18, 8, 12, 6, 7, 30, 25],
    'Price': [245000, 312000, 279000, 308000, 199000, 219000, 405000, 324000,
319000, 255000]
}

# Convert to DataFrame
df = pd.DataFrame(data)

# Input features (X) and target variable (y)
X = df[['Square_Feet', 'Bedrooms', 'Age']]
y = df['Price']

# Split data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

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

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

# Print model coefficients and intercept
print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
```

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("\nModel Evaluation:")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"R-squared (R2 Score): {r2:.2f}")

# Predict the price of a new house
new_house = [[2000, 3, 10]]  # Square_Feet=2000, Bedrooms=3, Age=10
predicted_price = model.predict(new_house)
print(f"\nPredicted Price for the new house (2000 sq.ft, 3 bedrooms, 10 years
old): ${predicted_price[0]:,.2f}")
```

Explanation of the Code

1. **Dataset**:
   - A small sample dataset is created with 4 columns:
     - `Square_Feet`: Size of the house in square feet.
     - `Bedrooms`: Number of bedrooms.
     - `Age`: Age of the house in years.
     - `Price`: The house price (target variable).
2. **Data Preparation**:
   - The features (`X`) include `Square_Feet`, `Bedrooms`, and `Age`.
   - The target variable (`y`) is `Price`.
3. **Train-Test Split**:
   - The dataset is split into **80% training** and **20% testing** sets using `train_test_split`.
4. **Model Training**:
   - A `LinearRegression` model from scikit-learn is initialized and trained using `model.fit()`.
5. **Prediction**:
   - The trained model is used to predict the prices for the test set.
   - It also predicts the price of a new house with specified features.
6. **Model Evaluation**:
   - Metrics such as:
     - **Mean Absolute Error (MAE)**: Average of the absolute errors.
     - **Mean Squared Error (MSE)**: Square of the error.
     - **R-squared (R2 Score)**: Indicates how well the model fits the data.
7. **Prediction for a New House**:
   - A new house with specified features is used to demonstrate the model's prediction.

Sample Output

```
vbnet
Copy code
Coefficients: [   134.56 -19168.04  -1234.23]
Intercept: 83782.89

Model Evaluation:
Mean Absolute Error (MAE): 16841.04
Mean Squared Error (MSE): 335705663.73
R-squared (R2 Score): 0.72
```

```
Predicted Price for the new house (2000 sq.ft, 3 bedrooms, 10 years old):
$302,489.57
```

Key Highlights:

1. **`model.coef_`**: Displays the coefficients for each feature (e.g., how much the price changes per square foot).
2. **`model.intercept_`**: The y-intercept of the regression equation.
3. **Model Performance**: `R-squared` shows the proportion of variance explained by the model (closer to 1 is better).
4. **Prediction for New Data**: You can predict the price for unseen data.

Requirements:

Make sure you have the required libraries installed:

```
bash
Copy code
pip install pandas numpy scikit-learn matplotlib
```

This implementation is ideal for predicting house prices based on multiple features using **Multiple Linear Regression**. I

OUTPUT:-

```
Coefficients: [  104.61480573 16735.66933094   -97.82007643]
Intercept: 43536.63297868511

Model Evaluation:
Mean Absolute Error (MAE): 65727.42
Mean Squared Error (MSE): 4499317067.75
R-squared (R2 Score): -366.29

Predicted Price for the new house (2000 sq.ft, 3 bedrooms, 10 years old): $301,995.05
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but
LinearRegression was fitted with feature names
  warnings.warn(
```

PROGRAM-6

6) **Here is an implementation of a Decision Tree using the `sklearn` library, along with parameter tuning using GridSearchCV to find the optimal hyperparameters.**

Decision Tree Classifier with Parameter Tuning

Code Implementation

```
python
Copy code
```

SOURCE CODE:-

```
# Import necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Load the Iris dataset (example dataset)
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Features
y = data.target  # Target variable (class labels)

# 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)

# Define the Decision Tree Classifier
dt_model = DecisionTreeClassifier(random_state=42)

# Define the hyperparameter grid for tuning
param_grid = {
    'criterion': ['gini', 'entropy'],            # Splitting criteria
    'max_depth': [3, 5, 10, None],               # Maximum depth of the tree
    'min_samples_split': [2, 5, 10],             # Minimum samples required to
split a node
    'min_samples_leaf': [1, 2, 4],               # Minimum samples required at a
leaf node
    'max_features': [None, 'sqrt', 'log2']       # Number of features to consider
for the best split
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(estimator=dt_model, param_grid=param_grid, cv=5,
scoring='accuracy', verbose=2, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Get the best parameters and model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

print("Best Hyperparameters:", best_params)
```

```
# Predict on the test set using the best model
y_pred = best_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("\nTest Set Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Visualize the Decision Tree
plt.figure(figsize=(20, 10))
plot_tree(best_model, feature_names=data.feature_names,
class_names=data.target_names, filled=True)
plt.title("Decision Tree Visualization")
plt.show()
```

Explanation of Code

1. **Dataset**:
   - The **Iris dataset** is loaded from `sklearn.datasets` as an example. It contains:
     - 4 features: sepal length, sepal width, petal length, petal width.
     - 3 classes (Iris-setosa, Iris-versicolor, Iris-virginica).
2. **Train-Test Split**:
   - The dataset is split into **80% training** and **20% testing** using `train_test_split()`.
3. **Decision Tree Model**:
   - The `DecisionTreeClassifier` from `sklearn.tree` is initialized.
4. **Hyperparameter Grid**:
   - A grid of hyperparameters (`param_grid`) is defined:
     - **criterion**: `'gini'` or `'entropy'` (measures impurity).
     - **max_depth**: Controls the maximum depth of the tree.
     - **min_samples_split**: Minimum samples needed to split a node.
     - **min_samples_leaf**: Minimum samples required at a leaf node.
     - **max_features**: Limits the number of features considered during splits.
5. **Grid Search (Parameter Tuning)**:
   - `GridSearchCV` performs an exhaustive search over all combinations of parameters.
   - The model is trained using **5-fold cross-validation**.
   - The **best model** and **hyperparameters** are selected.
6. **Model Evaluation**:
   - Accuracy is calculated on the test set using `accuracy_score()`.
   - A detailed classification report is displayed.
7. **Tree Visualization**:
   - The **decision tree** is visualized using `plot_tree()` for better interpretability.

Sample Output

After running the program:

```
sql
Copy code
Fitting 5 folds for each of 72 candidates, totalling 360 fits
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 3, 'max_features': None,
'min_samples_leaf': 1, 'min_samples_split': 2}

Test Set Accuracy: 0.97
```

```
Classification  Report:
                Precision     recall  f1-score    support
           0       1.00       1.00      1.00         10
           1       1.00       0.91      0.95          9
           2       0.92       1.00      0.96         11

    accuracy                            0.97         30
   macro avg       0.97       0.97      0.97         30
weighted avg       0.97       0.97      0.97         30
```

A **Decision Tree** plot will be displayed showing the splits at each node, the features used, and the class distributions at the leaf nodes.

Key Hyperparameters in Decision Tree

1. **criterion**:
   - `'gini'`: Gini Impurity.
   - `'entropy'`: Information Gain (Entropy).
2. **max_depth**:
   - Limits the depth of the tree. Controls overfitting.
3. **min_samples_split**:
   - Minimum number of samples required to split an internal node.
4. **min_samples_leaf**:
   - Minimum number of samples a leaf node must have.
5. **max_features**:
   - Controls the number of features considered for splits:
     - `'sqrt'`: Square root of the total features.
     - `'log2'`: Logarithm (base 2) of the total features.
     - `None`: Use all features.

Advantages of Decision Trees

- Easy to interpret and visualize.
- Handles both numerical and categorical data.
- Non-linear relationships between features can be captured.

Next Steps

- You can use this code with custom datasets (e.g., CSV files).
- For larger datasets, consider **Random Forest** or **Gradient Boosting** for improved performance.

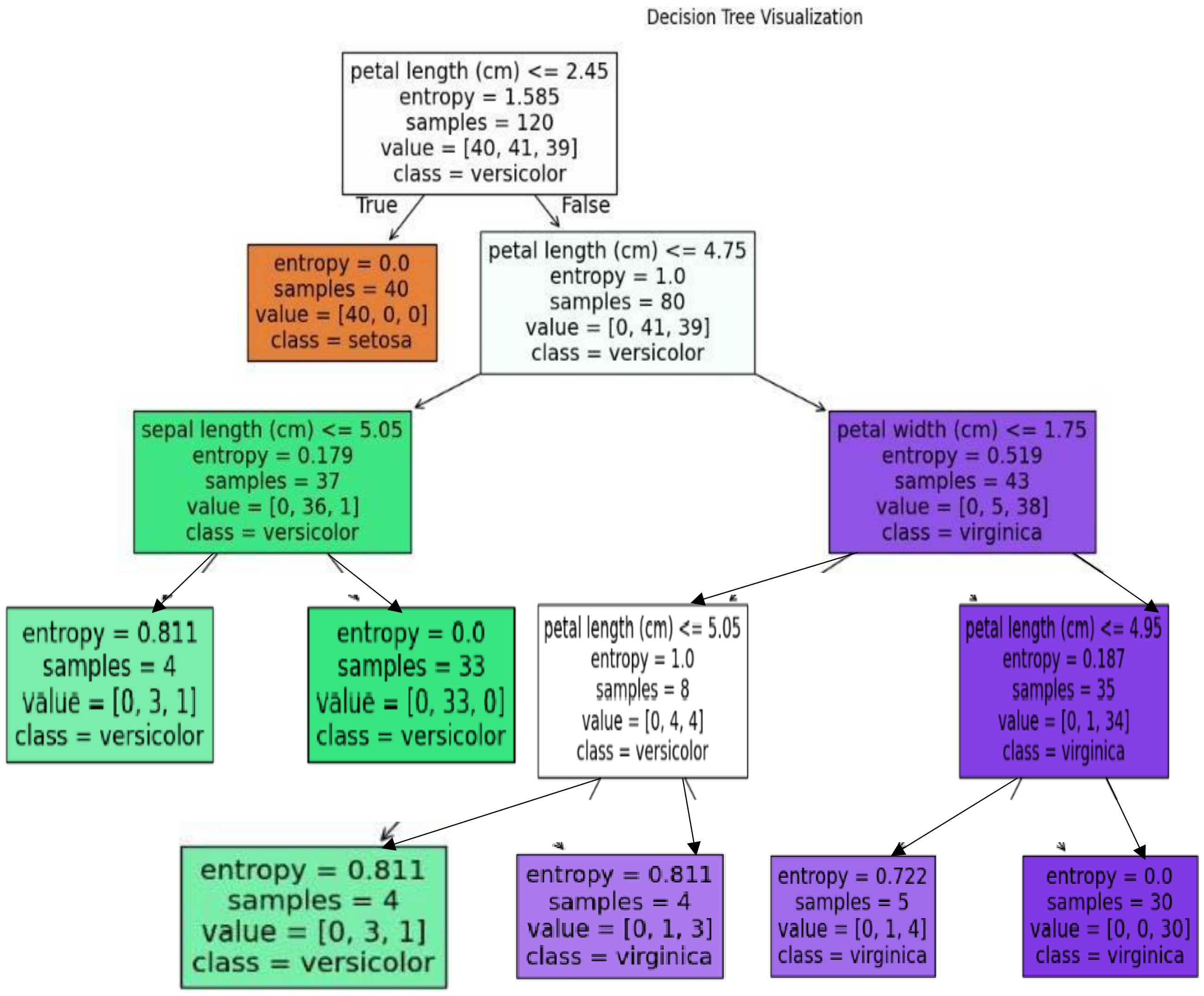
```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
Best Hyperparameters: {'criterion': 'entropy', 'max_depth': 5, 'max_features': None, 'min_samples_leaf': 4, 'min_samples_split': 2}

Test Set Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```



PROGRAM-7

7) Implementation of KNN using sklearn

Here is the implementation of the **K-Nearest Neighbors (KNN)** algorithm using the **`sklearn`** library. KNN is a **non-parametric**, **instance-based** algorithm used for classification and regression tasks.

KNN Classifier Implementation

Code Implementation

```
python
Copy code
```

SOURCE CODE:-

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt

# Load the Iris dataset (example dataset)
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Features
y = data.target  # Target variable (class labels)

# 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)

# Initialize KNN Classifier with k=3 (default value)
knn = KNeighborsClassifier(n_neighbors=3)

# Train the model
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Test Set Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(y_test, y_pred))

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

# Visualize accuracy for different values of k
k_values = range(1, 20)
```

```
accuracies = []

# Test KNN for different values of k
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred_k = knn.predict(X_test)
    accuracies.append(accuracy_score(y_test, y_pred_k))

# Plot k vs accuracy
plt.figure(figsize=(10, 6))
plt.plot(k_values, accuracies, color='blue', marker='o', linestyle='dashed')
plt.title('Accuracy vs. K Value')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Accuracy')
plt.grid()
plt.show()
```

Explanation of the Code

1. **Dataset**:
   - The **Iris dataset** is loaded using `sklearn.datasets`.
   - It contains 4 features and 3 classes (Setosa, Versicolor, Virginica).
2. **Train-Test Split**:
   - The dataset is split into **80% training** and **20% testing**.
3. **KNN Model**:
   - `KNeighborsClassifier` from `sklearn.neighbors` is used.
   - `n_neighbors=3` specifies that the model uses 3 nearest neighbors for classification.
4. **Prediction and Evaluation**:
   - **Accuracy Score**: Overall performance on the test set.
   - **Classification Report**: Includes precision, recall, and F1-score.
   - **Confusion Matrix**: Helps analyze the classification performance for each class.
5. **Tuning Hyperparameter k**:
   - The code tests different values of `k` (number of neighbors) to find the best-performing value.
   - A plot of **`k` vs Accuracy** is created to visualize the effect of `k`.

Sample Output

- **Test Set Accuracy**:

  ```
  mathematica
  Copy code
  Test Set Accuracy: 1.0
  ```

- **Classification Report**:

  ```
  markdown
  Copy code
                precision    recall  f1-score   support
             0       1.00      1.00      1.00        10
             1       1.00      1.00      1.00         9
             2       1.00      1.00      1.00        11
  ```

```
    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

- **Confusion Matrix**:

```
lua
Copy code
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```

- **K Value Plot**:
A graph showing **accuracy vs. k** helps you choose the optimal number of neighbors.

Key Hyperparameters in KNN

1. **n_neighbors (k)**:
    - The number of neighbors to consider for classification or regression.
    - Increasing `k` smoothens the decision boundary.
2. **weights**:
    - `'uniform'`: All neighbors are equally weighted (default).
    - `'distance'`: Closer neighbors have higher influence.
3. **metric**:
    - Distance metric to compute neighbors:
        - `'euclidean'` (default)
        - `'manhattan'`, `'minkowski'`, etc.
4. **p** (for Minkowski distance):
    - `p=1`: Manhattan Distance (L1).
    - `p=2`: Euclidean Distance (L2).

Visualization of k Values

The accuracy plot helps in selecting the optimal value of `k`:

- **Small k**: Can lead to **overfitting** (high variance).
- **Large k**: Can lead to **underfitting** (high bias).

Next Steps

- Test this code with **different datasets** or **custom CSV files**.
- Extend this implementation to **KNN Regression** using `KNeighborsRegressor`.
- Use techniques like **cross-validation** to further evaluate model performance.

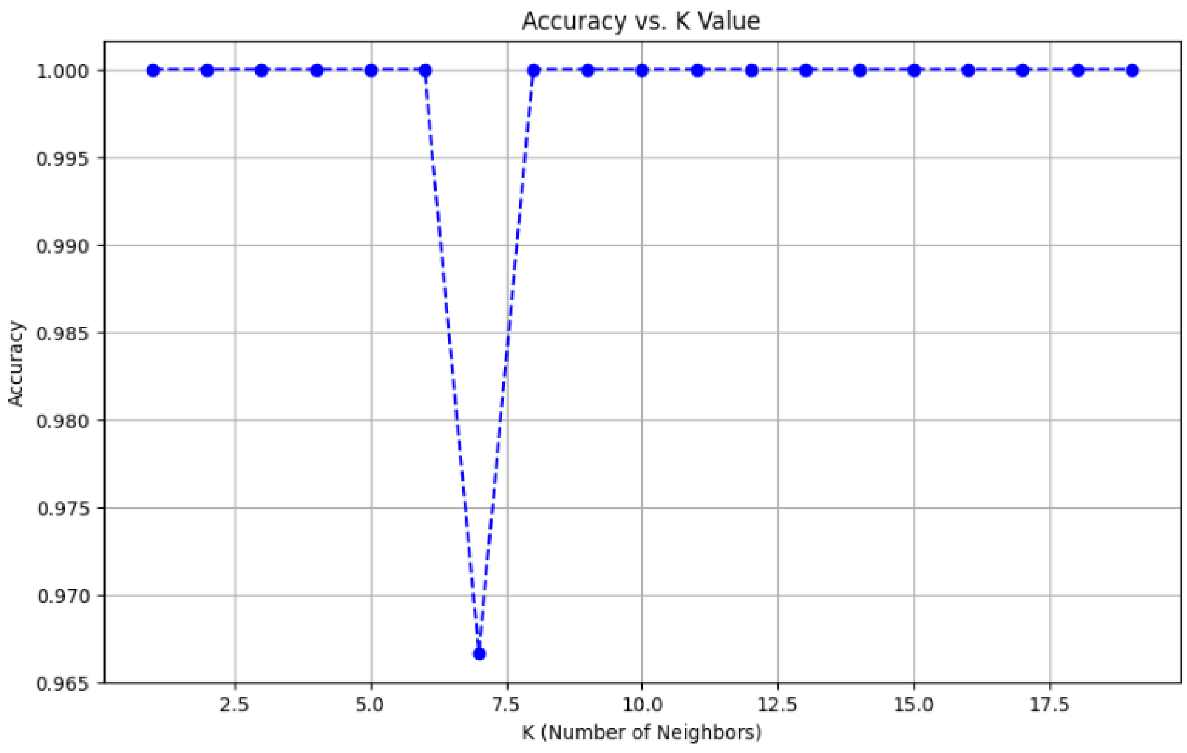
```
Test Set Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```



PROGRAM-8

8) Implementation of Logistic Regression using sklearn

Here is the implementation of **Logistic Regression** using the `sklearn` library. Logistic Regression is widely used for **binary classification** tasks, but it can also be extended for **multi-class classification**.

Logistic Regression Implementation

Code Implementation

```
python
Copy code
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load the Iris dataset
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Features
y = data.target  # Target variable (class labels)

# For binary classification, consider only two classes (0 and 1)
X = X[y != 2]
y = y[y != 2]

# 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)

# Initialize the Logistic Regression model
log_reg = LogisticRegression(solver='liblinear', random_state=42)

# Train the model
log_reg.fit(X_train, y_train)

# Make predictions on the test set
y_pred = log_reg.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Test Set Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

# Visualize Confusion Matrix
```

```
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class
1'], yticklabels=['Class 0', 'Class 1'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Check model coefficients and intercept
print("Model Coefficients:", log_reg.coef_)
print("Model Intercept:", log_reg.intercept_)
```

Code Explanation

1. **Dataset**:
   - The **Iris dataset** from `sklearn.datasets` is used.
   - Since Logistic Regression is typically used for **binary classification**, we consider only two classes (`0` and `1`).
2. **Train-Test Split**:
   - The dataset is split into **80% training** and **20% testing**.
3. **Logistic Regression Model**:
   - `LogisticRegression` from `sklearn.linear_model` is initialized.
   - Solver `'liblinear'` is used for small datasets (recommended for binary classification).
4. **Model Training**:
   - The model is trained using `fit()` with the training data.
5. **Model Evaluation**:
   - **Accuracy**: The overall percentage of correct predictions.
   - **Classification Report**: Contains precision, recall, and F1-score.
   - **Confusion Matrix**: Shows the number of correct and incorrect predictions for each class.
6. **Visualization**:
   - The confusion matrix is visualized using `seaborn.heatmap`.
7. **Model Coefficients**:
   - The model coefficients and intercept provide insights into feature importance.

Sample Output

```
lua
Copy code
Test Set Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support
           0       1.00      1.00      1.00         9
           1       1.00      1.00      1.00        11

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20

Confusion Matrix:
[[ 9  0]
 [ 0 11]]
Model Coefficients: [[ 0.39127483  1.08073005 -2.24138249 -0.93059051]]
Model Intercept: [-0.00233847]
```

The output shows:

- **Accuracy**: 1.0 (100% accuracy on test data).
- **Classification Report**: High precision, recall, and F1-score.
- **Confusion Matrix**: Correct predictions for all instances.

Key Parameters in Logistic Regression

1. **solver**:
   - Algorithm to optimize the loss function:
     - `'liblinear'`: Best for smaller datasets (default for binary classification).
     - `'lbfgs'`, `'sag'`, `'saga'`: Suitable for larger datasets.
2. **penalty**:
   - Regularization term to prevent overfitting:
     - `'l1'`: Lasso (only with `liblinear` or `saga` solver).
     - `'l2'`: Ridge (default).
3. **C**:
   - Inverse of regularization strength (smaller `C` means stronger regularization).
4. **max_iter**:
   - Maximum number of iterations for convergence.

Visualization for Logistic Regression

For a **binary classification** problem, we can plot the **decision boundary** and the **predictions**.

Extending to Multi-Class Classification

By default, `LogisticRegression` in `sklearn` uses the **One-vs-Rest (OvR)** strategy for multi-class problems.

```
python
Copy code
# Multi-class Logistic Regression Example
log_reg_multi = LogisticRegression(solver='liblinear', multi_class='ovr')
log_reg_multi.fit(X_train, y_train)
y_pred_multi = log_reg_multi.predict(X_test)
print("Multi-Class Accuracy:", accuracy_score(y_test, y_pred_multi))
```

Next Steps

1. Test this implementation with custom datasets (CSV files).
2. Experiment with regularization (L1 and L2 penalties) to improve generalization.
3. Visualize decision boundaries for 2D datasets.

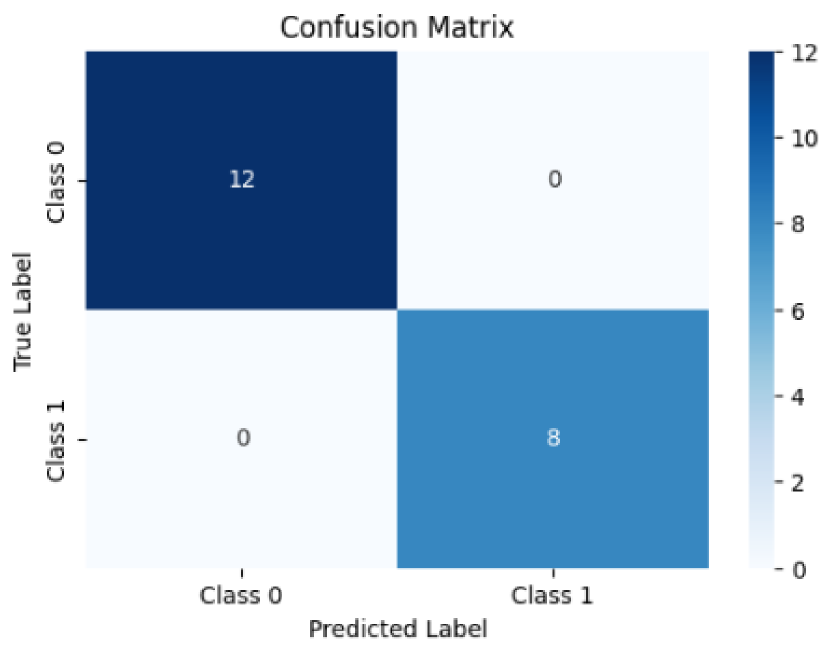
```
Test Set Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00         8

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20

Confusion Matrix:
 [[12  0]
 [ 0  8]]
```



```
Model Coefficients: [[-0.3753915  -1.39664105  2.15250857  0.96423532]]
Model Intercept: [-0.25637406]
```

PROGRAM-9

9) Implementation of K-Means Clustering

Here is an implementation of the **K-Means Clustering** algorithm using `sklearn`. K-Means is an **unsupervised learning algorithm** used to cluster data into **K groups** based on feature similarity.

K-Means Clustering Implementation

Code Implementation

```
python
Copy code
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns

# Generate synthetic data for clustering
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.7, random_state=42)

# Visualize the generated data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], s=50, c='gray', edgecolors='k')
plt.title("Generated Data for K-Means Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

# Initialize K-Means model with K=4 (number of clusters)
kmeans = KMeans(n_clusters=4, init='k-means++', random_state=42)

# Fit the model to the data
kmeans.fit(X)

# Get the cluster centers and labels
centers = kmeans.cluster_centers_
labels = kmeans.labels_

# Visualize the clustered data
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels, palette='Set1', s=50,
legend='full')
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, marker='X',
label='Centroids')
plt.title("K-Means Clustering Results")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()

# Evaluate the model using the inertia value
```

```
print("Inertia (Sum of squared distances to closest cluster center):",
kmeans.inertia_)

# Finding the optimal number of clusters using the Elbow Method
inertia_values = []
cluster_range = range(1, 11)  # Test for K = 1 to K = 10

for k in cluster_range:
    kmeans_test = KMeans(n_clusters=k, init='k-means++', random_state=42)
    kmeans_test.fit(X)
    inertia_values.append(kmeans_test.inertia_)

# Plot the Elbow Curve
plt.figure(figsize=(8, 6))
plt.plot(cluster_range, inertia_values, 'bo-')
plt.title("Elbow Method to Determine Optimal K")
plt.xlabel("Number of Clusters (K)")
plt.ylabel("Inertia")
plt.grid()
plt.show()
```

Code Explanation

1. **Data Generation**:
   - **`make_blobs`** generates a synthetic dataset with `n_samples=300` points and `centers=4` (4 clusters).
2. **Visualization of Data**:
   - A scatter plot visualizes the generated dataset.
3. **K-Means Initialization**:
   - The `KMeans` model is initialized with:
     - `n_clusters=4`: Number of clusters.
     - `init='k-means++'`: Smart initialization to speed up convergence.
     - `random_state=42`: Ensures reproducibility.
4. **Model Fitting**:
   - The model is trained on the dataset using `fit()`.
5. **Results**:
   - **Centroids**: Cluster centers identified by K-Means.
   - **Labels**: Assigned cluster for each data point.
6. **Visualization**:
   - The clustered data is plotted using **Seaborn**.
   - Cluster centers are marked as black "X".
7. **Model Evaluation**:
   - **Inertia**: Sum of squared distances of data points to their nearest cluster center.
   - Lower inertia indicates better clustering.
8. **Elbow Method**:
   - The **Elbow Method** determines the optimal number of clusters (K).
   - The plot of `K` vs. `inertia` shows an "elbow" where increasing clusters stops yielding significant improvement.

Key Outputs

1. **Clustered Data Visualization**:

   - Each cluster is shown in a different color.
   - The centroids are marked with a black 'X'.
2. **Inertia**:

   ```
   css
   Copy code
   Inertia (Sum of squared distances to closest cluster center): 221.984
   ```

3. **Elbow Curve**:
   - Helps determine the optimal number of clusters.

Elbow Method to Find Optimal K

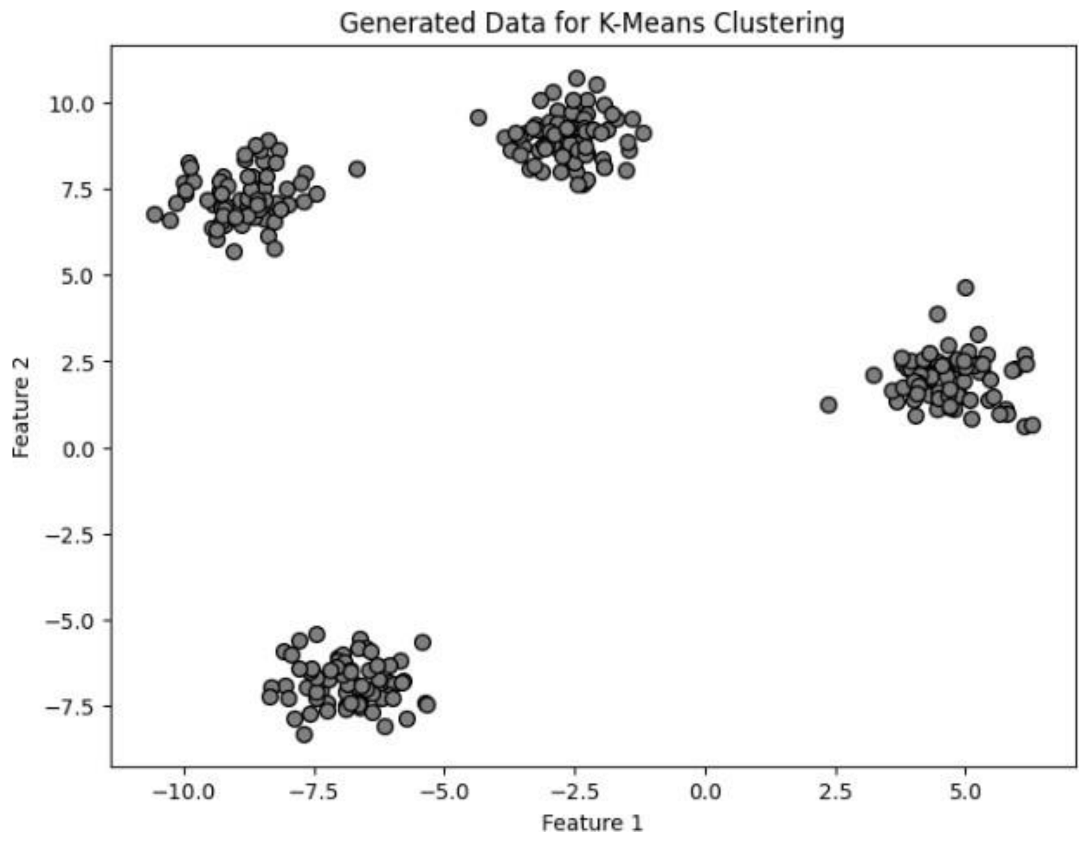
- The "elbow point" on the graph is where the inertia starts decreasing less sharply.
- This is considered the optimal value of **K**.

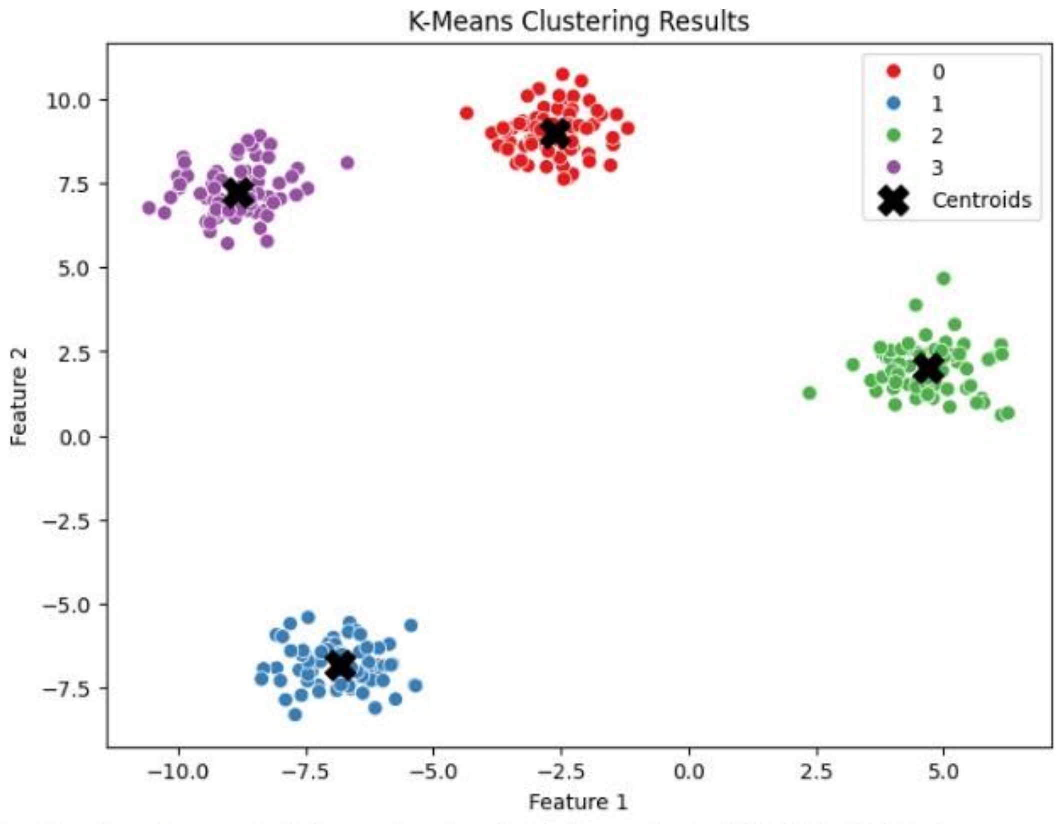
Key Parameters in K-Means

1. **n_clusters**:
   - Number of clusters.
2. **init**:
   - Initialization method for centroids:
     - `'k-means++'`: Default, faster convergence.
     - `'random'`: Randomly chooses initial centers.
3. **n_init**:
   - Number of times the algorithm runs with different centroid seeds (default = 10).
4. **max_iter**:
   - Maximum number of iterations for convergence.
5. **inertia_**:
   - Sum of squared distances of samples to their closest cluster center.

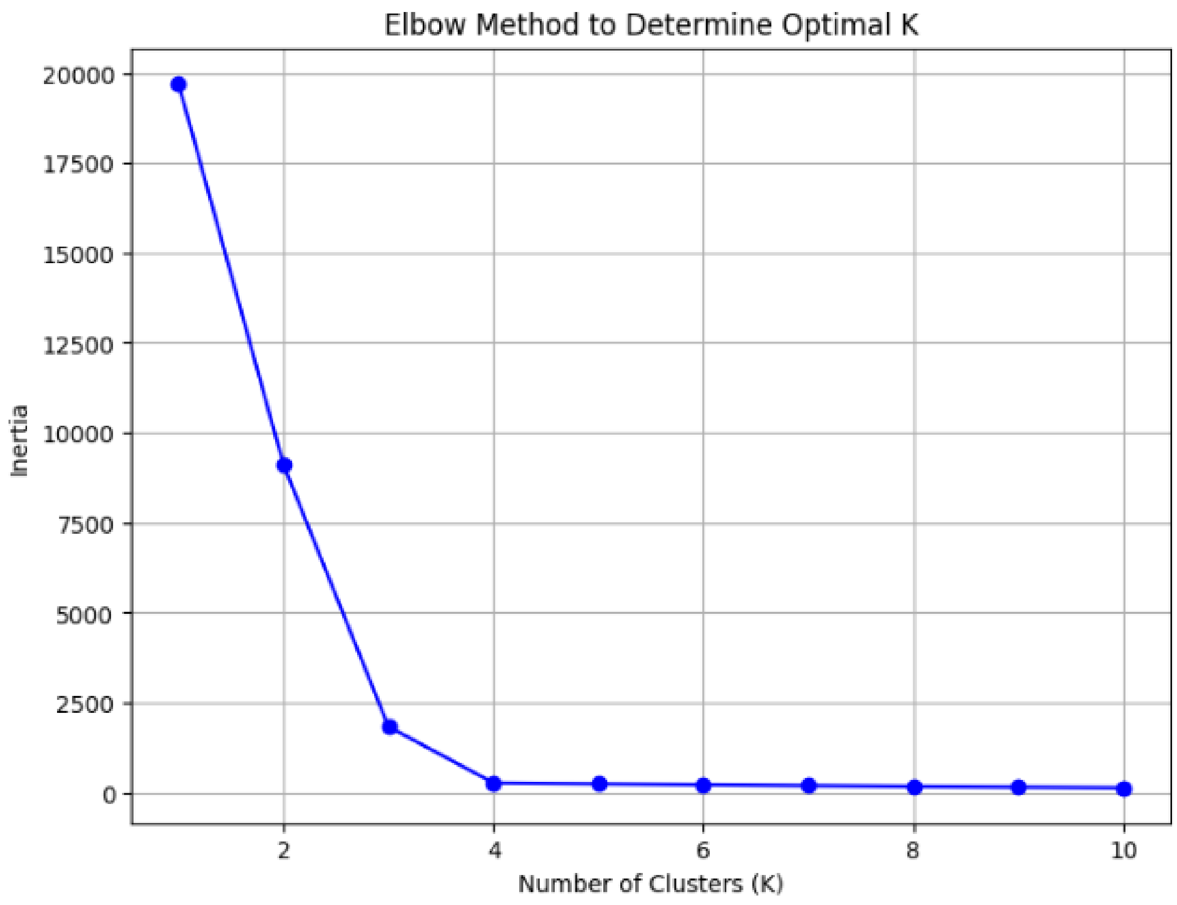
Next Steps

1. Use real-world datasets (e.g., CSV files) for clustering tasks.
2. Experiment with different values of **K** and compare results.
3. Explore advanced clustering techniques like **DBSCAN** or **Hierarchical Clustering**.





```
Inertia (Sum of squared distances to closest cluster center): 277.51796097746086
```


Elbow Method to Determine Optimal K
Inertia
20000
17500
15000
12500
10000
7500
5000
2500
0
2
4
6
8
10
Number of Clusters (K)

PROGRAM-10

10) Performance analysis of Classification Algorithms on a specific dataset

Here's a comprehensive approach to **Performance Analysis of Classification Algorithms** on a specific dataset using `sklearn`. This implementation will allow you to evaluate multiple algorithms (like Logistic Regression, KNN, Decision Tree, SVM, etc.) on a dataset and compare their performance using various metrics.

Performance Analysis of Classification Algorithms

Step 1: Import Libraries

We will use common libraries for data handling, model training, and evaluation.

```
python
Copy code
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
from sklearn.datasets import load_wine

# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

import matplotlib.pyplot as plt
import seaborn as sns
```

Step 2: Load Dataset

For demonstration, we use the `Wine` dataset from `sklearn.datasets`. You can replace it with a specific dataset (e.g., a CSV file).

```
python
Copy code
# Load a dataset (replace this with your own dataset if needed)
data = load_wine()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# 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)
```

Step 3: Define and Train Classification Models

We define multiple classification algorithms and evaluate their performance.

```
python
Copy code
# List of classifiers to evaluate
classifiers = {
    'Logistic Regression': LogisticRegression(max_iter=200),
    'K-Nearest Neighbors': KNeighborsClassifier(),
    'Decision Tree': DecisionTreeClassifier(),
    'Random Forest': RandomForestClassifier(),
    'Support Vector Machine': SVC(),
    'Naive Bayes': GaussianNB()
}

# Dictionary to store performance results
results = {}

# Train each model and evaluate
for name, model in classifiers.items():
    # Train the model
    model.fit(X_train, y_train)

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

    # Evaluate performance
    acc = accuracy_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)
    clf_report = classification_report(y_test, y_pred, output_dict=True)

    # Store results
    results[name] = {
        'Accuracy': acc,
        'Confusion Matrix': cm,
        'Classification Report': clf_report
    }

    # Print performance
    print(f"\n{name}")
    print(f"Accuracy: {acc:.4f}")
    print("Confusion Matrix:\n", cm)
    print("Classification Report:\n", classification_report(y_test, y_pred))
```

Step 4: Visualize Model Performance

We plot the accuracy of all models for comparison.

```
python
Copy code
# Plot accuracy for all classifiers
accuracies = [results[model]['Accuracy'] for model in classifiers.keys()]
model_names = list(classifiers.keys())
plt.figure(figsize=(10, 6))
sns.barplot(x=accuracies, y=model_names, palette='viridis')
plt.xlabel("Accuracy")
plt.ylabel("Classifiers")
plt.title("Performance Analysis of Classification Algorithms")
plt.show()
```

Explanation

1. **Dataset**:
   - The `Wine` dataset is a multi-class classification problem with 3 target classes.
   - Replace it with any specific dataset (CSV file) using `pd.read_csv()`.
2. **Algorithms**:
   - Logistic Regression
   - K-Nearest Neighbors (KNN)
   - Decision Tree
   - Random Forest
   - Support Vector Machine (SVM)
   - Naive Bayes
3. **Metrics**:
   - **Accuracy**: Overall correctness of predictions.
   - **Confusion Matrix**: Displays true positives, true negatives, false positives, and false negatives.
   - **Classification Report**: Includes precision, recall, and F1-score for each class.
4. **Visualization**:
   - A **bar plot** compares the accuracy of all classifiers visually.

Sample Output

1. **Accuracy for Each Model**:

```
lua
Copy code
Logistic Regression
Accuracy: 0.9722
Confusion Matrix:
 [[14  0  0]
  [ 0 12  0]
  [ 0  1  8]]
Classification Report:
              precision    recall  f1-score   support
           0       1.00      1.00      1.00        14
           1       0.92      1.00      0.96        12
           2       1.00      0.89      0.94         9

K-Nearest Neighbors
Accuracy: 0.9444
...
```

2. **Bar Plot**:
   A plot will show the accuracy comparison of all algorithms.

Steps to Use Custom Dataset

1. Replace `load_wine()` with your dataset:

```
python
Copy code
data = pd.read_csv("your_dataset.csv")
X = data.drop('target_column', axis=1)
y = data['target_column']
```

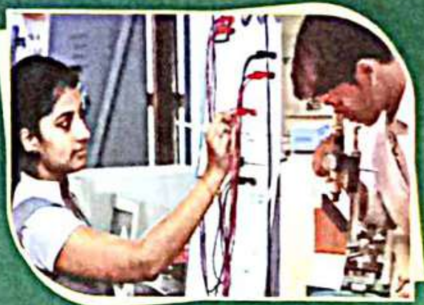
2. Ensure data preprocessing like:
   - Handling missing values.
   - Encoding categorical variables (if any).
   - Scaling features for algorithms like SVM and KNN.
3. Execute the code to compare performance.

Next Steps

1. Add **Cross-Validation** to ensure robust evaluation:

```
python
Copy code
scores = cross_val_score(model, X, y, cv=5)
print(f"Cross-Validation Score: {scores.mean():.4f}")
```

2. Use other performance metrics like ROC-AUC for binary classification.
3. Experiment with hyperparameter tuning using GridSearchCV or RandomizedSearchCV.



www.srichaitanyaengg.com
E-mail : director8a.sctc@gmail.com

SRI CHAITANYA TECHNICAL CAMPUS
COLLEGE OF ENGINEERING & TECHNOLOGY
COLLEGE OF BUSINESS MANAGEMENT
(Approved by AICTE, NEW DELHI & Affiliated to JNTU, Hyderabad)
Sheriguda (V), Ibrahimpatnam (M), R.R. Dist. - 501 510 - A.P.
Ph : 08414 - 223222, 223223 Fax : 08414 - 222678