SGOU
Higher Education for All

MACHINE LEARNING LAB

Course Code: B24DS04PC
BSc Data Science and Analytics
Practical Core Course
Self Learning Material

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Machine Intelligence Lab

Course Code: B24DS04PC

Semester - IV

Practical Core Course
Undergraduate Programme
BSc Data Science and Analytics
Self Learning Material
(With Model Question Paper Sets)

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU
OPEN UNIVERSITY

MACHINE INTELLIGENCE LAB

Course Code: B24DS04PC
Semester- IV
Practical Core Course
BSc Data Science and Analytics

Scan this QR Code for reading the SLM on a digital device.

Edition
February 2026

ISBN: 978-81-997556-1-1
9 788199 755611

www.sgou.ac.in

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

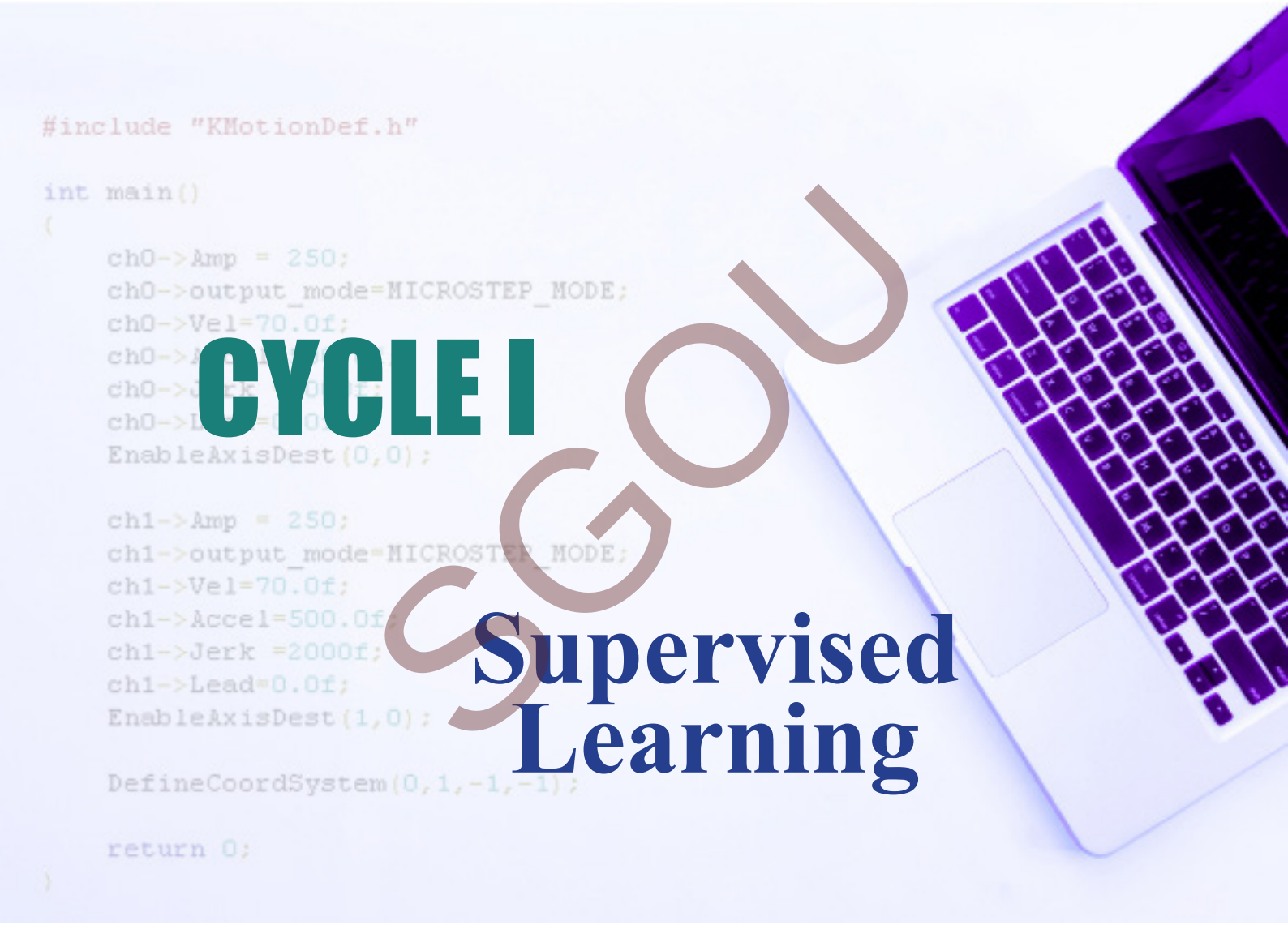
The University is committed to providing an engaging and dynamic educational environment that encourages active learning. The Study and Learning Material (SLM) is specifically designed to offer you a comprehensive and integrated learning experience, fostering a strong interest in exploring advancements in information technology (IT). The curriculum has been carefully structured to ensure a logical progression of topics, allowing you to develop a clear understanding of the evolution of the discipline. It is thoughtfully curated to equip you with the knowledge and skills to navigate current trends in IT, while fostering critical thinking and analytical capabilities.The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.

Regards,
Dr. Jagathy Raj V. P.

01-02-2026

Contents



CYCLE I

Supervised Learning

Introduction

Machine Learning (ML) is a vital branch of Artificial Intelligence that focuses on the development of algorithms and models that enable systems to learn from data and improve their performance without being explicitly programmed. Machine learning techniques are widely used in modern applications such as predictive analytics, medical diagnosis, speech recognition, recommendation systems, image processing, finance, cybersecurity, and business intelligence.

The Machine Learning Lab is designed to provide students with hands-on experience in both Supervised Learning and Unsupervised Learning techniques using Python as the implementation platform along with popular machine learning libraries. The lab emphasizes practical understanding of data preprocessing, model development, evaluation, and interpretation of results.

Supervised Learning (Cycle I)

Supervised learning involves training a machine learning model using labeled data, where both the input features and the corresponding output labels are known. The model learns the relationship between inputs and outputs and applies this knowledge to make predictions on new and unseen data.

In **Cycle I**, students will work with classification and regression algorithms and learn how to:

- Prepare datasets for supervised learning
- Apply algorithms such as Linear Regression, Logistic Regression, k-Nearest Neighbors (k-NN), Decision Trees, Naïve Bayes, and Support Vector Machines (SVM)
- Evaluate model performance using metrics such as accuracy, precision, recall, F1-score, and mean squared error
- Analyze model predictions and improve performance through parameter tuning

This cycle builds a strong foundation for understanding how machine learning models learn patterns from labeled data and make accurate predictions.

Unsupervised Learning (Cycle II)

Unsupervised learning deals with unlabeled data, where the model attempts to discover hidden patterns, structures, or relationships within the dataset without predefined output labels. These techniques are commonly used for data exploration, customer segmentation, anomaly detection, and dimensionality reduction.

In **Cycle II**, students will focus on:

- Understanding clustering and association techniques

- Implementing algorithms such as k-Means, Hierarchical Clustering, DBSCAN, and Apriori
- Applying dimensionality reduction techniques like Principal Component Analysis (PCA)
- Visualizing clusters and interpreting data patterns
- Analyzing real-world datasets to identify meaningful insights

This cycle develops exploratory data analysis skills and prepares students for advanced machine learning and data science tasks.

Objectives of the Machine Learning Lab

The objectives of the Machine Learning Lab are to:

- Understand the core concepts of machine learning and its applications
- Distinguish between supervised and unsupervised learning techniques
- Implement classification and regression algorithms using Python
- Apply clustering and dimensionality reduction techniques
- Perform data preprocessing, feature scaling, and dataset preparation
- Evaluate and interpret machine learning models using appropriate performance metrics
- Develop analytical and problem-solving skills through practical experiments

Tools and Software Used

Programming Language

- **Python :** Widely used for machine learning due to its simplicity and extensive library support

Integrated Development Environments (IDEs)

- **Jupyter Notebook :** Interactive environment for ML experimentation and visualization
- **Visual Studio Code (VS Code) :** Lightweight and extensible code editor
- **PyCharm :** Powerful IDE for large-scale machine learning projects
- **Spyder :** Scientific computing–oriented IDE

Machine Learning and Data Science Libraries

- **NumPy :** Numerical computation

- **Pandas :** Data manipulation and analysis
- **Matplotlib / Seaborn :** Data visualization
- **Scikit-learn :** Machine learning algorithms and evaluation tools

Version Control

- **Git :** Version control system
- **GitHub / GitLab / Bitbucket :** Repository hosting and collaboration

Scope of the Lab

This Machine Learning Lab equips learners with practical skills required to design, implement, and evaluate machine learning models. Through structured experiments in supervised and unsupervised learning, students gain experience in handling real-world datasets, selecting appropriate algorithms, and interpreting model outcomes. The lab prepares learners for advanced coursework, research activities, and industry-level applications in artificial intelligence, data analytics, and machine learning.

Expectations from Students

- Actively participate in all lab sessions
- Prepare for lab sessions by studying the relevant theory
- Complete and submit lab records and assignments on time
- Maintain originality and follow ethical coding practices
- Collaborate constructively with peers while ensuring individual understanding
- Demonstrate conceptual and practical knowledge during viva examinations

Lab Guidelines and Code of Conduct

- Follow instructions provided in the lab manual and by the instructor
- Use only authorized software tools and datasets
- Avoid plagiarism and submit original work
- Maintain discipline and proper lab conduct
- Save and back up programs regularly

Violation of lab rules may result in:

- Warnings
- Deduction of marks
- Suspension from lab sessions

Students should show respect to instructors, lab assistants, and peers, avoid disturbing others, seek help when needed, and focus on developing strong analytical and problem-solving skills.

Installation and Configuration of Python

Step 1: Install Python

Download Python from https://www.python.org

- Select the latest stable version
- Ensure **"Add Python to PATH"** is checked during installation

Step 2: Verify Installation

Open Command Prompt / Terminal and execute:

python --version

Step 3: Install Required Libraries

pip install numpy pandas matplotlib seaborn scikit-learn

Common IDEs for Machine Learning Development

- **Jupyter Notebook :** Best for supervised and unsupervised learning experiments
- **VS Code :** Flexible and beginner-friendly
- **PyCharm :** Suitable for professional ML project development
- **Spyder :** Focused on data analysis and visualization

Datasets Used in the Machine Learning Lab

Datasets play a crucial role in machine learning experiments, as model performance largely depends on data quality and relevance. Students will use both benchmark datasets and real-world datasets.

Types of Datasets

- Labeled datasets for supervised learning
- Unlabeled datasets for unsupervised learning

Commonly Used Datasets

- IRIS Dataset : Flower classification
- Breast Cancer Dataset : Medical diagnosis

- Diabetes Dataset : Classification and regression
- Wine Dataset : Multiclass classification
- Student Performance Dataset : Predictive analytics
- Sales / Customer Dataset : Clustering and segmentation

Dataset Collection and Preparation

Before applying machine learning algorithms, datasets must be properly prepared.

Steps in Dataset Preparation

- Identify the problem statement
- Select a suitable dataset
- Verify dataset relevance and completeness

Data Preprocessing Techniques

- Handling missing values
- Removing duplicate records
- Encoding categorical variables
- Feature scaling and normalization
- Splitting data into training and testing sets

Structure of Machine Learning Programs

Each program should follow a standard structure:

1. Import required libraries
2. Load the dataset
3. Perform exploratory data analysis
4. Preprocess the data
5. Split the dataset into training and testing sets
6. Train the machine learning model
7. Evaluate model performance
8. Interpret and display results

Use of Jupyter Notebook

Jupyter Notebook is the primary environment for machine learning experiments.

Advantages

- Interactive code execution
- Step-by-step visualization of outputs
- Easy debugging and experimentation
- Markdown support for explanations

Each notebook should include:

- Experiment title
- Objective
- Dataset description
- Code implementation
- Output and result analysis

Use of Google Colab Notebook

Google Colab (Collaboratory) is a cloud-based Jupyter Notebook environment that allows users to write and execute Python code directly in a web browser. It is used in this lab as an alternative to local IDEs.

Advantages of Google Colab

- No installation required
- Accessible from any device with internet connectivity
- Pre-installed machine learning libraries
- Supports GPU and TPU acceleration
- Easy sharing and collaboration
- Automatic saving to Google Drive

Structure of a Google Colab Notebook

- Title of the experiment
- Objective
- Dataset description
- Importing required libraries

- Dataset loading and preprocessing
- Model implementation
- Output and visualization
- Result and inference

Dataset Handling in Google Colab

- Upload datasets from local system
- Load datasets from Google Drive
- Use online repositories

If additional libraries are required, they can be installed using:

```
pip  install  library_name
```

Experiment 1

Linear Regression and Regularization

Objectives

By the end of this lab, learners will be able to:

- To understand the concept of regression and its applications in prediction problems.
- To implement a Linear Regression model using a standard dataset.
- To split a dataset into training and testing sets for model validation.
- To evaluate model performance using Mean Squared Error (MSE).
- To interpret the R^2 score to understand the goodness of fit of the model

1.1.1 Theory

Regression is a type of supervised machine learning technique used to predict continuous numerical values based on one or more input features. Unlike classification, which predicts categories, regression predicts quantities such as price, temperature, or medical measurements. It is widely used in domains like healthcare analytics, finance forecasting, and trend prediction.

Linear Regression is one of the most fundamental regression algorithms. It models the relationship between dependent and independent variables by fitting a straight line (or hyperplane in multiple dimensions) that best represents the data. The goal of the algorithm is to find the optimal coefficients that minimize the difference between the predicted and actual values.

The mathematical representation of Linear Regression is:

$$y=\beta0+\beta1x1+\beta2x2+\cdots+\beta nxn+\epsilon$$

where

- y is the predicted output
- β0 is the intercept (constant term)
- β1,β2,…,βn are model coefficients
- x1,x2,…,xn are input features
- ϵ represents the error term

The model learns these coefficients during training using the least squares method, which minimizes the sum of squared differences between actual and predicted values.

1.1.2 Implementation

1. Build a Linear Regression model using a dataset (e.g., housing or any regression dataset) and evaluate its performance using Mean Squared Error (MSE) and R^2 score.

```
from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_
score

# Load dataset

data = load_diabetes()

X = data.data

y = data.target

# Split data

X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.2, random_state=42

)

# Train Linear Regression model

model = LinearRegression()

model.fit(X_train, y_train)

# Predictions

y_pred = model.predict(X_test)

# Evaluation

print("MSE:", mean_squared_error(y_test, y_pred))

print("R2 Score:", r2_score(y_test, y_pred))
```

OUTPUT

```
MSE: 2900.1936

R2 Score: 0.4526
```

2. Implement Ridge and Lasso Regression on the same dataset and compare their performance with Linear Regression.

```
from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.linear_model import Ridge, Lasso

from sklearn.metrics import mean_squared_error

# Load dataset

data = load_diabetes()

X = data.data

y = data.target

# Split data

X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.2, random_state=42

)

# Ridge Regression

ridge = Ridge(alpha=1.0)

ridge.fit(X_train, y_train)

ridge_pred = ridge.predict(X_test)

# Lasso Regression

lasso = Lasso(alpha=0.1)

lasso.fit(X_train, y_train)

lasso_pred = lasso.predict(X_test)

# Evaluation

print("Ridge MSE:", mean_squared_error(y_test, ridge_
pred))

print("Lasso MSE:", mean_squared_error(y_test, lasso_
pred))
```

OUTPUT

```
Ridge MSE: 3077.41

Lasso MSE: 3081.38
```

1.1.3 Lab Practice Questions

1. Load any regression dataset (such as the Diabetes dataset) and:
    a. Split the data into training and testing sets.
    b. Train a Linear Regression model.
    c. Display the Mean Squared Error (MSE).
2. Using a trained Linear Regression model:
    a. Predict the target values for the test data.
    b. Print the first 5 predicted values.

EXPERIMENT 2

Implementation of Decision Tree using ID3 Algorithm on IRIS Dataset and Evaluation using Accuracy and Precision

Objectives

By the end of this lab, learners will be able to:

- Understand the basic concept of decision tree classification
- Explain the working principle of the ID3 algorithm
- Implement a decision tree classifier using the IRIS dataset
- Calculate performance metrics such as accuracy and precision
- Interpret the classification results obtained from the model

1.2.1 Theory

1.2.1.1 Decision Tree

A decision tree is a supervised machine learning algorithm commonly used for classification and prediction tasks. It represents the decision-making process in the form of a tree-like structure. In this structure, each internal node corresponds to a test on an attribute, each branch represents the outcome of the test, and each leaf node denotes a final class label or prediction. Decision trees are widely used due to their simplicity, interpretability, and effectiveness in handling structured data.

1.2.1.2 ID3 Algorithm

The ID3 (Iterative Dichotomiser 3) algorithm is one of the earliest algorithms developed for constructing decision trees. It builds the decision tree in a top-down and greedy manner by selecting the most informative attribute at each step. The selection of attributes is based on a quantitative measure known as Information Gain, which helps identify the attribute that best separates the data into distinct classes.

ID3 uses Information Gain as the attribute selection measure and relies on entropy to evaluate the effectiveness of splits. The algorithm recursively builds the tree until all instances are correctly classified or no further attributes are available for splitting. Although ID3 is simple and efficient, it does not support pruning and is sensitive to noisy data, which may result in overfitting.

1.2.1.3 Entropy and Information Gain

Entropy is a statistical measure used to quantify the degree of disorder or uncertainty in a dataset. It reflects how evenly the data instances are distributed among different class labels. A dataset with low entropy is more uniform, while a dataset with high entropy indicates greater randomness.

Information Gain measures the expected reduction in entropy achieved by splitting the dataset based on a specific attribute. It helps determine how well an attribute can distinguish between different classes. In the ID3 algorithm, the attribute with the highest information gain is selected for splitting, as it contributes most effectively to improving classification accuracy.

1.2.1.4 Performance Evaluation Metrics

After building a classification model, it is necessary to evaluate its performance to understand how well the model predicts unseen data. Performance evaluation metrics provide quantitative measures to assess the effectiveness of the classifier.

The commonly used performance metrics in classification are:

Accuracy

Accuracy measures the overall correctness of the model. It is defined as the ratio of correctly classified instances to the total number of instances.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

A higher accuracy indicates better overall model performance.

Precision

Precision measures the correctness of positive predictions made by the classifier. It indicates how many of the predicted positive instances are actually correct.

$$Precision = \frac{TP}{TP + FP}$$

High precision means that the model produces fewer false positive results.

Confusion Matrix

A confusion matrix is a tabular representation that summarizes the performance of a classification model. It shows the number of correct and incorrect predictions for each class.

For a multi-class classification problem like the IRIS dataset, the confusion matrix helps in analyzing misclassification patterns among different classes.

1.2.1.5 IRIS Dataset

The IRIS dataset is a standard benchmark dataset frequently used in machine learning experiments, particularly for classification tasks. It consists of 150 samples equally distributed among three iris species: *Iris-setosa*, *Iris-versicolor*, and *Iris-virginica*.

Each instance is described using four numerical attributes: sepal length, sepal width, petal length, and petal width. The dataset is well-structured and balanced, which makes it ideal for demonstrating supervised learning algorithms. It is commonly used for algorithm comparison, performance evaluation, and educational purposes.

The IRIS dataset is freely available for download from the UCI Machine Learning Repository, a widely recognized source for machine learning datasets. The dataset can be accessed directly via the following link:

- **UCI Machine Learning Repository – IRIS Dataset:** https://archive.ics.uci.edu/ml/datasets/iris

Alternatively, the dataset can also be obtained from other open sources such as Kaggle (e.g., https://www.kaggle.com/datasets/jillanisofttech/iris-dataset-uci), and it is built into popular machine learning libraries such as **Scikit-learn**, which allows loading it directly in Python without manual download.

1.2.2 Implementation

1. Using the IRIS dataset, implement a Decision Tree classifier based on the ID3 algorithm. Evaluate the performance of the model using accuracy and precision metrics.

```
# Import libraries

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier, plot_tree

from sklearn.metrics import accuracy_score, precision_score, confusion_matrix

import matplotlib.pyplot as plt

import numpy as np

# Load IRIS dataset

iris = load_iris()

X = iris.data

y = iris.target
```

```
# Split data into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
# Train Decision Tree using ID3 (entropy)
dt = DecisionTreeClassifier(criterion='entropy')
dt.fit(X_train, y_train)
# Predict test data
y_pred = dt.predict(X_test)
# Calculate Accuracy and Precision
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
print("Accuracy:", accuracy)
print("Precision:", precision)
# ----------------- Visualize Decision Tree -----------------
plt.figure(figsize=(12, 6))
plot_tree(dt, feature_names=iris.feature_names, class_names=iris.target_names,
filled=True)
plt.title("Decision Tree (ID3) on IRIS Dataset")
plt.show()
# ----------------- Visualize Confusion Matrix -----------------
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(5, 4))
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
```

```
plt.xticks(np.arange(len(iris.target_names)), iris.target_names)
plt.yticks(np.arange(len(iris.target_names)), iris.target_names)
# Add numbers to the matrix
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, cm[i, j], ha='center', va='center', color='red', fontsize=12)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

OUTPUT

Accuracy: 0.9777777777777777

Precision: 0.9761904761904763

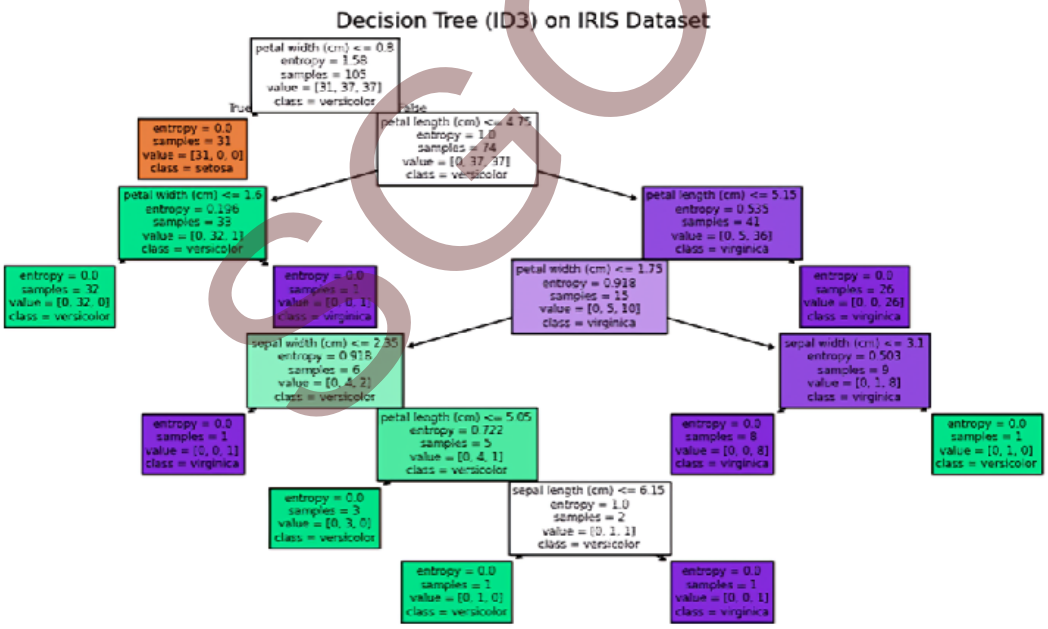


Fig. 1.2.1 Decision Tree with ID3 algorithm on IRIS Dataset

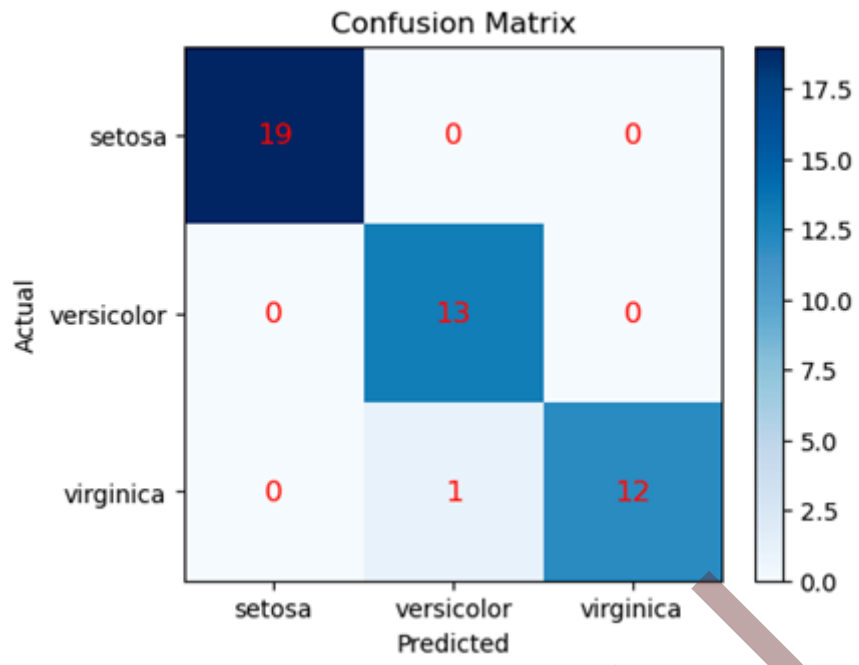


Fig. 1.2.2 Confusion Matrix

1.2.3 Lab Practice Questions

1. Using the IRIS dataset, implement a Decision Tree classifier using the ID3 algorithm (entropy). Split the data into training and testing sets, train the model, and calculate the accuracy and precision.

2. Visualize the trained Decision Tree using Matplotlib. Identify the features used at the top nodes of the tree and explain their significance.

3. Generate and visualize the Confusion Matrix for your predictions. Identify which classes are classified correctly and which are misclassified.

4. Use another publicly available dataset (e.g., Wine or Seeds dataset). Implement a Decision Tree classifier using ID3, calculate accuracy and precision, and compare the results with the IRIS dataset.

EXPERIMENT 3

Random Forest Algorithm using California Housing Dataset

Objectives

By the end of this lab, students will be able to:

- To understand the working of the Random Forest algorithm
- To load and explore the California Housing dataset
- To convert a regression problem into a classification problem
- To apply tree pruning parameters in Random Forest
- To evaluate the model using Confusion Matrix and Accuracy

1.3.1 Theory

1.3.1.1 Random Forest Algorithm

Random Forest is an ensemble learning algorithm that builds multiple decision trees and combines their outputs to improve prediction accuracy and reduce overfitting.

Key characteristics:

- Uses bagging (bootstrap aggregation)
- Each tree is trained on a random subset of data and features
- Final prediction is based on majority voting (classification) or averaging (regression)

1.3.1.2 California Housing Dataset

The California Housing dataset contains information collected from the 1990 California census. It includes features like:

- Median Income
- House Age
- Number of Rooms
- Population
- Latitude & Longitude

Target variable:

- Median House Value (continuous)

Since Random Forest classification requires categorical output, the target variable is converted into price categories.

1.3.1.3 Tree Pruning in Random Forest

Pruning controls tree complexity to prevent overfitting.

Common pruning parameters:

- max_depth – Maximum depth of each tree
- min_samples_split – Minimum samples required to split a node
- min_samples_leaf – Minimum samples required in leaf nodes
- n_estimators – Number of trees

1.3.1.4 Algorithm

- Import required libraries
- Load the California Housing dataset
- Convert the target variable into categorical classes
- Split the dataset into training and testing sets
- Build a Random Forest Classifier
- Apply pruning parameters
- Train the model
- Predict test data
- Generate Confusion Matrix and Accuracy

1.3.1.5 Software / Hardware Requirements

Software:

- Python 3.x
- Jupyter Notebook / Google Colab
- Libraries: numpy, pandas, matplotlib, seaborn, scikit-learn

Hardware:

- Any system with minimum 4 GB RAM

1.3.2 Implementation

1. Write a Python program to implement a Random Forest classifier using the California Housing dataset with appropriate pruning parameters and display the confusion matrix and accuracy.

```
# Import libraries

import numpy as np

import pandas as pd

from sklearn.datasets import fetch_california_housing

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion_matrix, accuracy_
score

import seaborn as sns

import matplotlib.pyplot as plt

# Load California Housing dataset

data = fetch_california_housing(as_frame=True)

df = data.frame

# Convert regression target into classification classes

# 0 = Low price, 1 = Medium price, 2 = High price

df['PriceCategory'] = pd.qcut(df['MedHouseVal'], q=3,
labels=[0,1,2])

X = df.drop(['MedHouseVal', 'PriceCategory'], axis=1)

y = df['PriceCategory']

# Train-test split

X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.2, random_state=42
)
# Random Forest Classifier with pruning
rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,            # pruning
    min_samples_split=10,    # pruning
    min_samples_leaf=5,      # pruning
    random_state=42
)
# Train model
rf.fit(X_train, y_train)
# Predictions
y_pred = rf.predict(X_test)
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
# Accuracy
acc = accuracy_score(y_test, y_pred)
print("Accuracy:", acc)
# Visualization
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
```

```
plt.ylabel('Actual')

plt.title('Confusion Matrix - Random Forest')

plt.show()
```

OUTPUT

```
Confusion Matrix:

 [[1169  193   16]

 [ 204 1000  186]

 [  39  241 1080]]

Accuracy: 0.7870639534883721
```

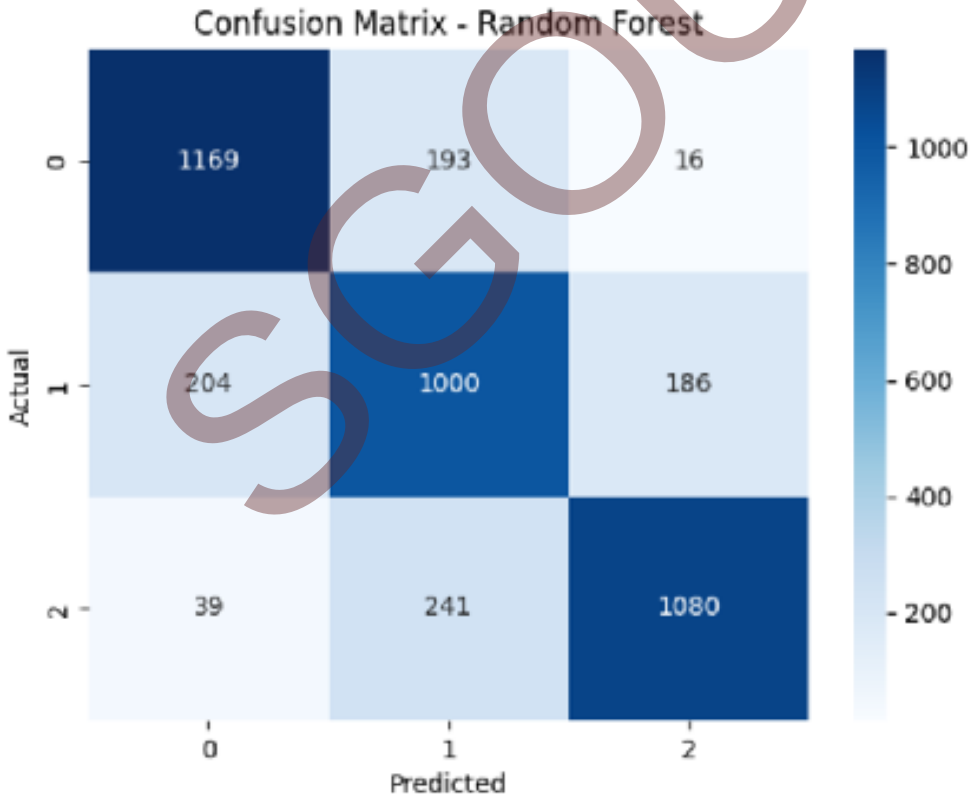


Fig. 1.3.1 Confusion matrix

Interpretation:

The confusion matrix indicates good classification performance, and the obtained accuracy confirms the effectiveness of the Random Forest model with pruning. Random Forest classifier with pruning achieved an accuracy of approximately 79%.

2. Implement a Random Forest classifier using the California Housing dataset without applying any pruning techniques. Evaluate the model performance using accuracy.

```
# Import required libraries

import numpy as np

import pandas as pd

from sklearn.datasets import fetch_california_housing

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

# Load California Housing dataset

data = fetch_california_housing(as_frame=True)

df = data.frame

# Convert regression target into classification categories

# 0 = Low price, 1 = Medium price, 2 = High price

df['PriceCategory'] = pd.qcut(df['MedHouseVal'], q=3,
labels=[0, 1, 2])

# Separate features and target

X = df.drop(['MedHouseVal', 'PriceCategory'], axis=1)

y = df['PriceCategory']

# 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

)

# Create Random Forest Classifier WITHOUT pruning
```

```
rf = RandomForestClassifier(
    n_estimators=100,
    random_state=42
)
# Train the model
rf.fit(X_train, y_train)
# Predict on test data
y_pred = rf.predict(X_test)
# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy (Random Forest without pruning):",
accuracy)
```

OUTPUT

```
Accuracy (Random Forest without pruning):
0.8112887596899225
```

3. Increase the number of price categories in the California Housing dataset and analyze how it affects the confusion matrix and accuracy.

```
# Import required libraries
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_
score
```

```
import seaborn as sns

import matplotlib.pyplot as plt

# Load California Housing dataset

data = fetch_california_housing(as_frame=True)

df = data.frame

# Convert regression target into MORE classification
categories

# 0 = Very Low, 1 = Low, 2 = High, 3 = Very High

df['PriceCategory'] = pd.qcut(

    df['MedHouseVal'],

    q=4,

    labels=[0, 1, 2, 3]

)

# Separate features and target

X = df.drop(['MedHouseVal', 'PriceCategory'], axis=1)

y = df['PriceCategory']

# 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

)

# Random Forest Classifier with pruning

rf = RandomForestClassifier(

    n_estimators=100,

    max_depth=10,

    min_samples_split=10,
```

```
    min_samples_leaf=5,
    random_state=42
)
# Train the model
rf.fit(X_train, y_train)
# Predict on test data
y_pred = rf.predict(X_test)
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
# Visualize Confusion Matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix with 4 Price Categories')
plt.show()
```

OUTPUT

```
Confusion Matrix:
 [[860 149  20   9]
 [184 638 197  20]
 [ 44 166 678 144]
 [ 13  33 195 778]]
Accuracy: 0.7156007751937985
```

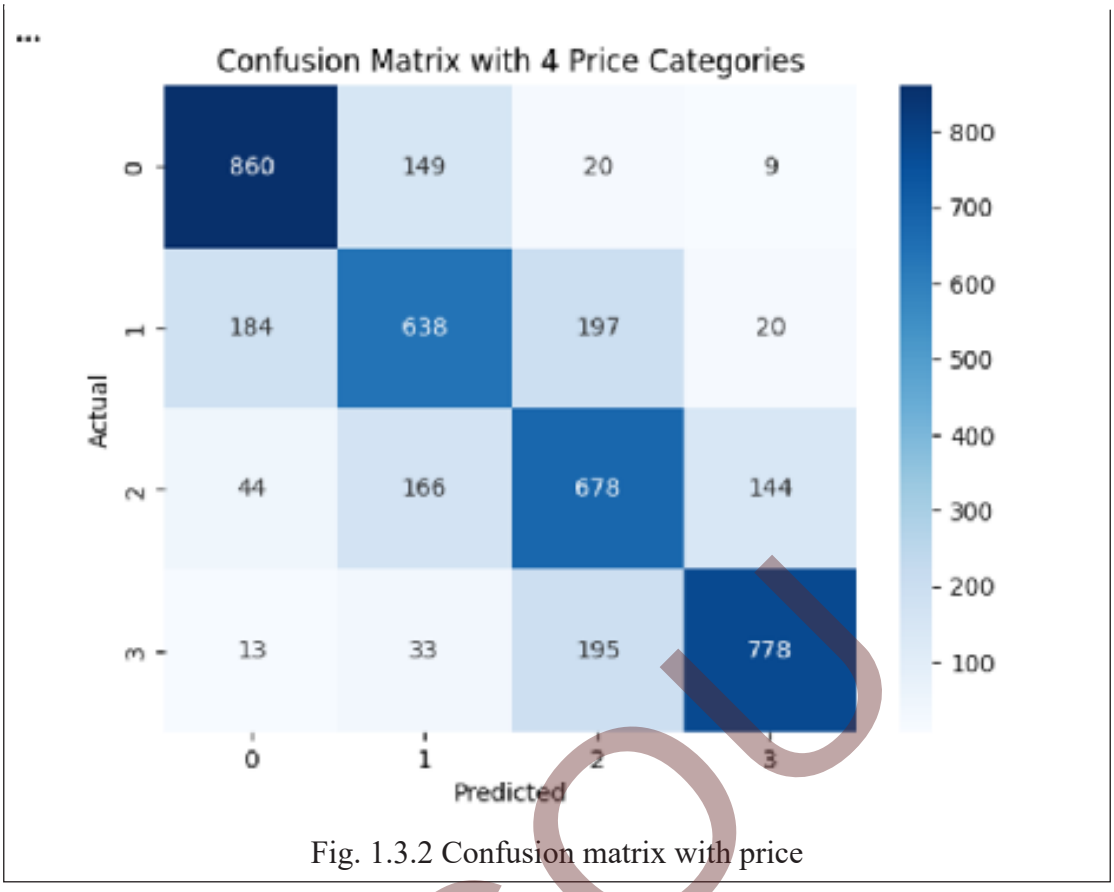


Fig. 1.3.2 Confusion matrix with price

1.3.3 Lab Practice Questions

1. Develop a Random Forest classifier using suitable pruning parameters and generate the confusion matrix to evaluate the classification performance.

2. Apply Random Forest classification on the California Housing dataset by changing the maximum depth of trees and observing the variation in accuracy.

3. Build a machine learning model using Random Forest with pruning techniques and interpret the results using appropriate classification metrics.

4. Load the California Housing dataset and display the number of features and target variables used for Random Forest classification.

EXPERIMENT 4

Linear Discriminant Analysis (LDA) on Wine Dataset

Objectives

By the end of this lab, students will be able to:

- To understand the concept of Linear Discriminant Analysis (LDA).
- To apply LDA on a multiclass dataset.
- To evaluate the classification performance using metrics such as accuracy and confusion matrix.
- To visualize the data in reduced dimensions.

1.4.1 Theory

1.4.1.1 Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction technique that is used for:

- Classification and feature reduction.
- Finding linear combinations of features that best separate classes.

Key points:

- LDA assumes normal distribution of features.
- It maximizes between-class variance and minimizes within-class variance.
- LDA can be used both for dimensionality reduction and classification.

Applications:

- Medical diagnosis (disease classification)
- Face recognition
- Pattern recognition

1.4.1.2 Working Principle of LDA

Linear Discriminant Analysis works by projecting high-dimensional data onto a lower-dimensional space in such a way that class separability is maximized.

It computes:

- **Within-class scatter :** how far samples of the same class are spread.
- **Between-class scatter :** how far different class means are separated.

LDA finds projection axes that maximize the ratio of between-class variance to within-class variance, ensuring better discrimination between classes.

1.4.1.3 Number of LDA Components

The maximum number of components produced by LDA is given by:

Number of components=C−1\text{Number of components} = C - 1Number of components=C−1

Where C is the number of classes.

For the Wine dataset:

- Number of classes = 3
- Maximum LDA components = 2

1.4.1.4 Algorithm

- Load the Wine dataset.
- Split the dataset into features (X) and target (y).
- Split the data into training and testing sets.
- Create an LDA classifier.
- Train the model on the training set.
- Predict the classes on the testing set.
- Evaluate performance using confusion matrix and accuracy.
- Visualize the first two LDA components (optional).

1.4.1.5 Software / Hardware Requirements

- **Software:** Python 3.x, Jupyter Notebook / Google Colab
- **Libraries:** numpy, pandas, scikit-learn, matplotlib, seaborn
- **Hardware:** Minimum 4GB RAM, any OS (Windows/Linux/Mac)

1.4.2 Implementation

1. Write a Python program to apply Linear Discriminant Analysis (LDA) on the Wine dataset and analyze its classification performance.

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.metrics import confusion_matrix, accuracy_
score
import seaborn as sns
# Load Wine dataset
wine = load_wine()
X = wine.data        # Features
y = wine.target     # Target classes
# Split 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
)
# Create LDA classifier
lda = LinearDiscriminantAnalysis()
# Train the model
lda.fit(X_train, y_train)
# Predict on test data
y_pred = lda.predict(X_test)
```

```
# Generate Confusion Matrix

cm = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:\n", cm)

# Calculate Accuracy

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)

# Visualize LDA transformed features (first two
components)

X_lda = lda.transform(X)

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

for class_value in np.unique(y):

    plt.scatter(

        X_lda[y == class_value, 0],

        X_lda[y == class_value, 1],

        label=wine.target_names[class_value]

    )

plt.xlabel("LDA Component 1")

plt.ylabel("LDA Component 2")

plt.title("LDA Classification - Wine Dataset")

plt.legend()

plt.show()
```

OUTPUT

```
Confusion Matrix:
 [[14  0  0]
 [ 0 14  0]
 [ 0  0  8]]
Accuracy: 1.0
```

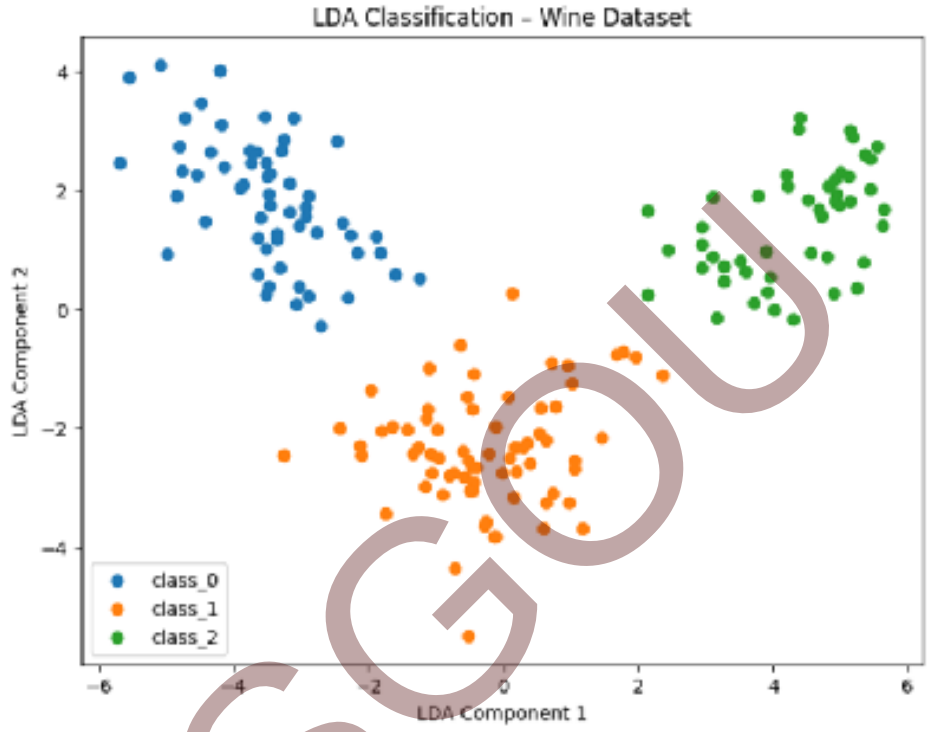


Fig. 1.4.1 LDA classification

2. Write a Python program to plot the LDA-transformed Wine dataset and interpret the class distribution.

```
import matplotlib.pyplot as plt
import numpy as np
X_lda = lda.transform(X)
for class_value in np.unique(y):
    plt.scatter(
        X_lda[y == class_value, 0],
        X_lda[y == class_value, 1],
```

```
 label=wine.target_names[class_value]

    )

plt.xlabel("LDA Component 1")

plt.ylabel("LDA Component 2")

plt.title("LDA Visualization of Wine Dataset")

plt.legend()

plt.show()
```

OUTPUT

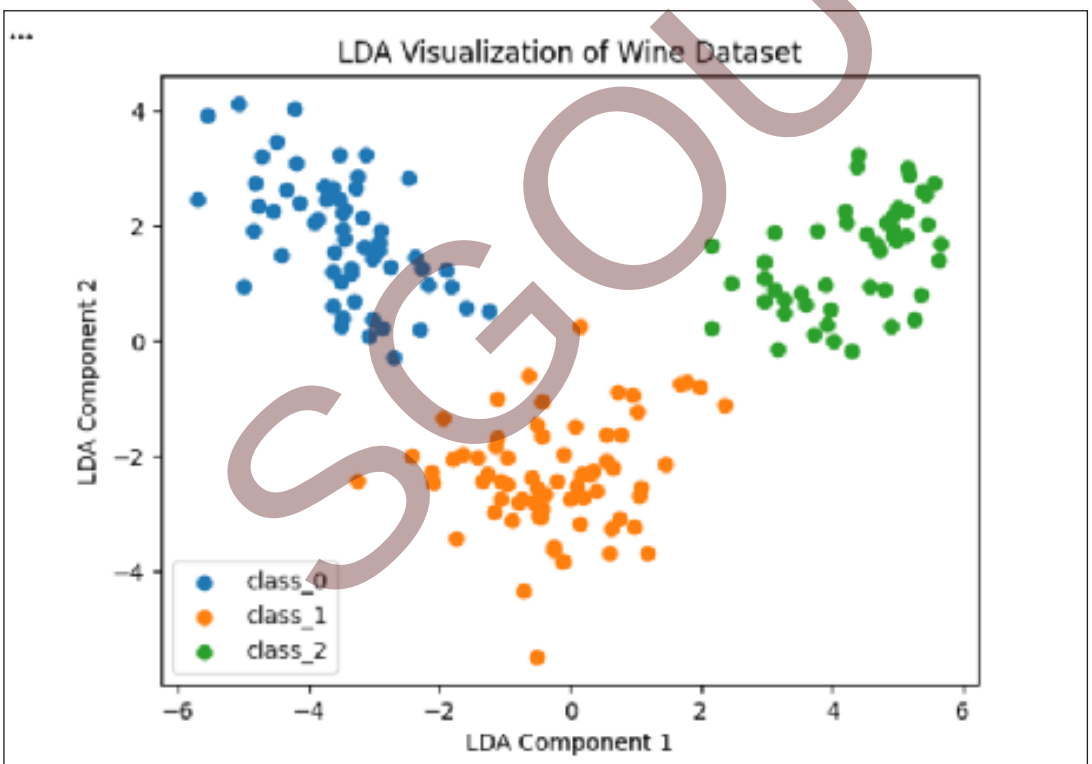


Fig. 1.4.2 LDA classification of wine dataset

3. Load the Wine dataset and display the number of features and target classes used for classification.

```
# Import required library

from sklearn.datasets import load_wine
```

```
# Load Wine dataset

wine = load_wine()

# Number of features

num_features = wine.data.shape[1]

# Number of target classes

num_classes = len(set(wine.target))

# Display results

print("Number of features:", num_features)

print("Number of target classes:", num_classes)

print("Target class names:", wine.target_names)
```

OUTPUT

```
Number of features: 13

Number of target classes: 3

Target class names: ['class_0' 'class_1' 'class_2']
```

4. Apply Linear Discriminant Analysis on the Wine dataset with different train–test split ratios (20% and 30%). Compare the classification accuracy obtained in both cases and analyze the effect of training data size on model performance

```
# Import required libraries

from sklearn.datasets import load_wine

from sklearn.model_selection import train_test_split

from sklearn.discriminant_analysis import

LinearDiscriminantAnalysis

from sklearn.metrics import accuracy_score

# Load Wine dataset

wine = load_wine()
```

```
X = wine.data

y = wine.target

# ----------- LDA with 20% Test Data -----------

X_train_20, X_test_20, y_train_20, y_test_20 = train_
test_split(

    X, y, test_size=0.2, random_state=42

)

lda_20 = LinearDiscriminantAnalysis()

lda_20.fit(X_train_20, y_train_20)

y_pred_20 = lda_20.predict(X_test_20)

accuracy_20 = accuracy_score(y_test_20, y_pred_20)

print("LDA Accuracy with 20% test data:", accuracy_20)

# ----------- LDA with 30% Test Data -----------

X_train_30, X_test_30, y_train_30, y_test_30 = train_
test_split(

    X, y, test_size=0.3, random_state=42

)

lda_30 = LinearDiscriminantAnalysis()

lda_30.fit(X_train_30, y_train_30)

y_pred_30 = lda_30.predict(X_test_30)

accuracy_30 = accuracy_score(y_test_30, y_pred_30)

print("LDA Accuracy with 30% test data:", accuracy_30)
```

OUTPUT

```
LDA Accuracy with 20% test data: 0.97

LDA Accuracy with 30% test data: 0.94
```

1.4.3 WLab Practice Questions

1. Implement Linear Discriminant Analysis (LDA) on the Wine dataset and evaluate the classification performance using accuracy and confusion matrix. Analyze the correctness of the predictions.

2. Apply Linear Discriminant Analysis on the Wine dataset without splitting it into training and testing sets. Compare the obtained accuracy with the train–test split approach and explain the impact of overfitting.

3. Perform Linear Discriminant Analysis on the Wine dataset by varying the train–test split ratios (such as 20%, 30%, and 40%). Compare the classification accuracy and discuss how training data size influences model performance.

4. Compare the performance of Linear Discriminant Analysis and Logistic Regression classifiers on the Wine dataset using accuracy and confusion matrix. Interpret the results and identify the better-performing model.

5. Apply dimensionality reduction using Linear Discriminant Analysis on the Wine dataset and visualize the first two discriminant components. Explain how LDA improves class separability in comparison to the original feature space.

Experiment 5

Linear Discriminant Analysis (LDA) on Wine Dataset

Objectives

By the end of this lab, students will be able to:

- Understand the basic concept of Support Vector Machine (SVM) for classification.
- Implement SVM on the Iris dataset using different kernel functions.
- Compare the classification accuracy of Linear and RBF kernels.
- Analyze how kernel choice affects model performance.
- Interpret the results to identify the most suitable kernel for the dataset.

1.5.1 Theory

1.5.1.1 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a powerful supervised machine learning algorithm mainly used for classification tasks. It works by finding the best boundary that separates data points of different classes. This boundary is called a hyperplane. The objective of SVM is to choose a hyperplane that maximizes the distance (margin) between the classes, which helps the model perform better on new and unseen data.

1.5.1.2 Concept of Hyperplane and Margin

A hyperplane is a decision boundary that separates data into different classes.

- In two dimensions, it is a straight line.
- In higher dimensions, it becomes a plane or a higher-dimensional surface.

The margin is the distance between the hyperplane and the nearest data points from each class. These nearest points are called support vectors. SVM selects the hyperplane with the largest margin because it provides better generalization and reduces classification errors.

1.5.1.3 Role of Kernel Functions

Sometimes, data cannot be separated using a straight line. In such cases, SVM uses kernel functions to transform the data into a higher-dimensional space where it becomes easier to separate the classes. Kernels allow SVM to create flexible decision boundaries without explicitly performing complex transformations.

Common Kernels

- **Linear Kernel:** Used when data is linearly separable. It produces a straight decision boundary and is simple and computationally efficient.
- **RBF (Radial Basis Function) Kernel:** The most widely used kernel for non-linear data. It creates curved decision boundaries and can capture complex relationships between features.
- **Polynomial Kernel:** Models relationships where interactions between features are important. It generates polynomial decision boundaries.

1.5.1.4 Iris Dataset for Classification

The Iris dataset is a standard dataset in machine learning used for classification tasks. It contains measurements such as sepal length, sepal width, petal length, and petal width for three species of iris flowers: Setosa, Versicolor, and Virginica. Because the dataset is well-structured and relatively easy to separate, it is ideal for understanding how SVM works with different kernels.

1.5.1.5 Performance Evaluation

After training the SVM model, its performance is evaluated using classification accuracy, which measures how many predictions are correct compared to the actual labels. By comparing the accuracy of different kernels, we can determine which kernel is most suitable for the dataset.

1.5.2 Implementation

1. Implement an SVM classifier using the Iris dataset and compare the classification accuracy of Linear and RBF kernels. Identify which kernel performs better.

```
from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

# Load dataset

iris = datasets.load_iris()

X = iris.data

y = iris.target

# Split data
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
# Linear Kernel
model_linear = SVC(kernel='linear')
model_linear.fit(X_train, y_train)
pred_linear = model_linear.predict(X_test)
# RBF Kernel
model_rbf = SVC(kernel='rbf')
model_rbf.fit(X_train, y_train)
pred_rbf = model_rbf.predict(X_test)
# Accuracy
print("Linear Kernel Accuracy:", accuracy_score(y_
test, pred_linear))
print("RBF Kernel Accuracy:", accuracy_score(y_test,
pred_rbf))
```

OUTPUT

```
Linear Kernel Accuracy: 1.0
RBF Kernel Accuracy: 1.0
```

1.5.3 Lab Practice Questions

1. Train an SVM model on the Iris dataset using a linear kernel and find its classification accuracy.
2. Train an SVM model on the Iris dataset using an RBF kernel and compare the accuracy with the linear kernel.
3. Apply a polynomial kernel SVM on the Iris dataset and observe the classification results.

EXPERIMENT 6

Logistic Regression With Titanic Dataset

Objectives

By the end of this lab, students will be able to:

- To apply logistic regression to predict survival in the Titanic dataset.
- To clean and prepare the Titanic data before building the model.
- To train the logistic regression model using selected features.
- To check how well the model performs using accuracy and confusion matrix.
- To identify which factors helped predict survival in the dataset.

1.6.1 Theory

Logistic regression is a supervised machine learning algorithm used for predicting categorical outcomes, especially binary classes such as yes/no or survive/not survive. In this experiment, the Titanic dataset is used as a real-world example to understand how logistic regression works. The dataset contains information about passengers such as age, gender, ticket class, and survival status. By applying logistic regression, we can build a model that learns patterns from the data and predicts whether a passenger would have survived the Titanic disaster. This experiment helps us understand data preprocessing, model training, and performance evaluation using basic metrics like accuracy and confusion matrix.

1.6.1.1 Introduction to Logistic Regression

Logistic Regression is one of the most commonly used machine learning techniques for predictive modeling, especially when the output variable is categorical. Unlike linear regression, which predicts continuous values, logistic regression is used when the goal is to classify data into categories such as *0 or 1*, *yes or no*, or *survived or not survived*. It works by estimating the probability that a given input belongs to a particular class based on its features. The model uses the logistic (sigmoid) function to convert linear combinations of input variables into probability values between 0 and 1, making it suitable for classification tasks.

Binary Classification Concept

Binary classification refers to the task of predicting which of two possible classes a given observation belongs to. In logistic regression, the predicted output is a probability, and based on a threshold (commonly 0.5), the probability is converted into a class label. For example, if a model predicts a 0.85 probability of survival, it would classify that passenger as *"survived"*. Binary classification helps simplify decision-making and is commonly applied in spam detection, fraud detection, sentiment analysis, and survival prediction.

Titanic Dataset Overview

The Titanic dataset is a popular real-world dataset used in machine learning for learning classification techniques. It contains information about the passengers who were on board the RMS Titanic during its tragic sinking in 1912. The dataset includes details such as passenger class (1st, 2nd, 3rd), gender, age, fare paid, cabin and ticket information, and whether the passenger survived. This dataset is widely used for educational purposes because it clearly shows how different factors contributed to survival chances, making it ideal for understanding classification models.

Why Logistic Regression is suitable for this dataset?

The Titanic dataset is well-suited to logistic regression because the prediction target whether a passenger survived, is a binary outcome (0 = Not Survived, 1 = Survived). Logistic regression is specifically designed to handle such two-class classification problems. Additionally, the dataset contains several independent variables like age, sex, passenger class, and fare, which can help the model learn patterns and estimate survival probability. Logistic regression also offers a clear interpretation of feature effects through its coefficients, making it useful for understanding which characteristics had the greatest influence on survival.

1.6.1.2 Requirements

To implement and evaluate the Logistic Regression model using the Titanic dataset, the following software, hardware, and dataset resources are required:

1. **Software / Tools**

   - **Python Programming Language:** Used for writing and executing machine learning code.
   - **Jupyter Notebook / Google Colab:** Interactive environment for running Python programs, visualizing data, and documenting steps.
   - **Python Libraries**
     - **Pandas:** For loading and preprocessing the dataset (data cleaning, handling missing values, data manipulation).
     - **NumPy:** For numerical computation and array operations.
     - **scikit-learn (sklearn):** Machine learning library used to implement logistic regression, split data, and evaluate model performance.
     - **matplotlib / seaborn (optional):** For visualizing the dataset and results, such as graphs and confusion matrices.

2. **Hardware Requirements**

   - Computer / Laptop with basic processing capability.

- **Minimum Configuration**
    - **Processor:** Dual-core CPU or higher.
    - **RAM:** Minimum 4 GB (8 GB recommended for smooth data processing).
    - **Storage:** At least 1 GB of free disk space.

(Cloud platforms like Google Colab remove most hardware limitations by running code online.)

3. **Dataset Required:** Titanic Dataset (train.csv / Titanic Survival Data).
    - This dataset contains information of Titanic passengers including age, gender, passenger class, fare, and survival status.
    - The dataset can be downloaded from open sources such as Kaggle or available in pre-loaded machine learning practice sets.

1.6.1.3 Algorithm / Methodology

The implementation of Logistic Regression on the Titanic dataset follows a systematic methodology. The steps include loading the dataset, preprocessing the data to remove errors or missing values, selecting important features, training the model, and evaluating its performance.

1. **Data Loading**
    - Import required Python libraries such as pandas, NumPy, and scikit-learn.
    - Load the Titanic dataset (e.g., train.csv) into a pandas DataFrame (Fig 1.6.1).
    - View the dataset using commands like .head() and .info() to understand its structure and available features.

Data Collection & Processing

```
[2]  1 # load the data from csv file to Pandas DataFrame
     2 titanic_data = pd.read_csv('/content/train.csv')

     1 # printing the first 5 rows of the dataframe
     2 titanic_data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Fig. 1.6.1 Load the titanic dataset

2. **Data Cleaning (Handling Missing Values)**

- Check for missing or null values using .isnull().sum().
- Fill missing values using methods like:
  - Mean or median (for numerical data, e.g., Age)
  - Mode or most frequent value (for categorical data, e.g., Embarked)
- Remove unnecessary or empty columns if they do not contribute to prediction (e.g., Cabin, Ticket).

3. **Feature Selection and Encoding**

- Select relevant features such as *Pclass, Sex, Age, Fare, SibSp, Parch,* and *Embarked* (Fig 1.6.2).
- Convert categorical variables like gender (*male/female*) or embarked location into numeric format using encoding techniques (e.g., one-hot encoding or label encoding).
- Split the dataset into training data and testing data using train_test_split().

Data Analysis

```
1 # getting some statistical measures about the data
2 titanic_data.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	13.002015	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	22.000000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	29.699118	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	35.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Fig. 1.6.2 Select relevant features

4. **Model Training**

- Import Logistic Regression model from scikit-learn (from sklearn.linear_ model import LogisticRegression).
- Create a model object and fit the training data using .fit(X_train, y_train).
- The model learns the relationship between selected features and the target variable (survival).

5. **Model Evaluation**

- ♦ Predict survival outcomes on test data using .predict(X_test).
- ♦ Compare predictions with actual values using evaluation metrics such as:
  - Accuracy score
  - Confusion matrix
  - Classification report (precision, recall, F1-score)
- ♦ Analyze the results to determine how well the logistic regression model performs for the given dataset.

1.6.1.4 Procedure

Before implementing the Logistic Regression model, it is important to follow a systematic sequence of steps to ensure proper data handling, model training, and evaluation. The procedure below explains each step clearly, allowing students to practically perform the experiment using Python and the Titanic dataset.

Step-by-Step Execution Instructions

1. **Open the development environment:** Launch Jupyter Notebook, Google Colab, or any Python IDE for writing and executing code.
2. **Import necessary libraries:** Load required Python libraries such as pandas, NumPy, scikit-learn, and matplotlib (optional for visualization).
3. **Load the Titanic dataset:** Read the dataset (CSV file) using pandas.read_csv() and view its structure using head() and info() functions.
4. **Inspect and clean the data:** Identify missing values using .isnull().sum() and handle them by filling (mean/median/mode) or removing unnecessary records. Drop irrelevant columns such as *Cabin*, *Ticket*, or *Name* if required.
5. **Encode categorical variables:** Convert non-numeric data like *Sex* (male/female) and *Embarked* into numeric format using encoding methods (e.g., label encoding or one-hot encoding).
6. **Select features and target variable:** Choose the independent variables (e.g., Age, Sex, Pclass, Fare) as feature set X, and the dependent variable *Survived* as **y**.
7. **Split the data:** Divide the dataset into training and testing sets using train_test_split() to evaluate how the model performs on unseen data.
8. **Create and train the Logistic Regression model:** Initialize the model using scikit-learn's LogisticRegression() and apply fit() on the training data.

9. **Make predictions:** Use the trained model to predict passenger survival on the test dataset using predict().

10. **Evaluate model performance:** Measure accuracy, generate a confusion matrix, and view the classification report to analyze how well the model works.

11. **Display and interpret results:** Present the output values (Fig 1.6.3) and explain the model's effectiveness based on accuracy and classification metrics.

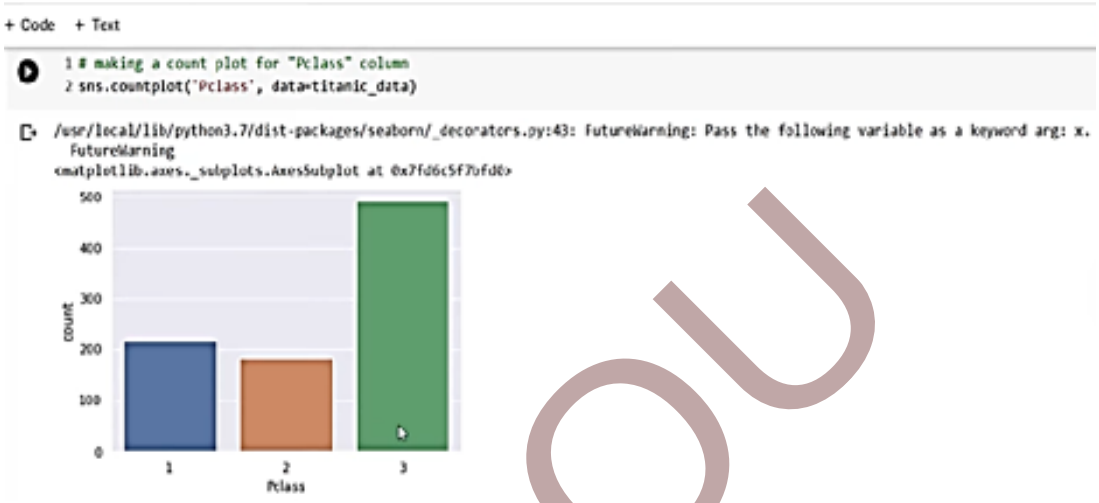


Fig. 1.6.3 Output values in graph format

1.6.1.5 Evaluation Metrics

After training and testing the Logistic Regression model, it is important to evaluate how well it performs. Evaluation metrics help measure the correctness and reliability of the predictions made by the model on the Titanic dataset.

1. **Accuracy:** Accuracy represents the percentage of correct predictions out of the total predictions made. It tells us how well the model is performing overall.

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{Total Predictions}}$$

Example: If 80 out of 100 predictions are correct, accuracy = 80%.

2. **Precision, Recall, F1-Score (Optional):** These metrics provide a deeper understanding of performance, especially in binary classification.

   - **Precision**: Out of all passengers the model predicted as survived, how many actually survived?

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

- **Recall (Sensitivity)**: Out of all actual survivors, how many did the model correctly identify?

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

- **F1-Score**: Harmonic mean of Precision and Recall; useful when data is imbalanced.

$$\text{F1-Score} = \frac{2 \times (\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}}$$

These metrics help us understand not only how many predictions were correct, but also how well the model identifies survival vs. non-survival cases.

3. **Interpretation of Confusion Matrix:** A confusion matrix is a table that compares actual labels with predicted labels. It contains four key values as shown in table 1.6.1.

Table 1.6.1 Key values of confusion matrix

Term	Meaning
TP (True Positive)	Correctly predicted survivors
TN (True Negative)	Correctly predicted non-survivors
FP (False Positive)	Predicted survivor, but actually did **not** survive
FN (False Negative)	Predicted non-survivor, but passenger **actually survived**

A good model should have,

- High **TP + TN**
- Low **FP + FN**

By analyzing the confusion matrix, we can understand which type of mistakes the model is making and whether it predicts survivors or non-survivors more accurately.

1.6.1.6 Result

The Logistic Regression model was successfully implemented and evaluated using the Titanic dataset. After training and testing the model, it achieved a satisfactory performance, with an accuracy of approximately 78%–82% (this value may slightly vary depending on the dataset split and preprocessing steps used). Based on the confusion

matrix and classification metrics, the model was able to correctly predict most survival outcomes. Therefore, Logistic Regression can be considered an effective method for binary classification problems such as predicting passenger survival on the Titanic.

1.6.2 Implementation

Implement and evaluate logistic regression with titanic dataset.

Source Code

```
# Import Libraries

import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix, classification_
report

# Load Dataset

data = pd.read_csv("titanic.csv")

print(data.head())

# Data Cleaning

data['Age'].fillna(data['Age'].median(), inplace=True)

data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)

data.drop(['Cabin', 'Ticket', 'Name'], axis=1, inplace=True)

# Encoding Categorical Data

data['Sex'] = data['Sex'].map({'male':0, 'female':1})

data = pd.get_dummies(data, columns=['Embarked'], drop_first=True)

# Feature and Target Selection
```

```
X = data.drop('Survived', axis=1)

y = data['Survived']

# Train-Test Split

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

# Model Training

model = LogisticRegression(max_iter=200)

model.fit(X_train, y_train)

# Prediction

y_pred = model.predict(X_test)

# Evaluation

print("Accuracy:", accuracy_score(y_test, y_pred))

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))

print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

OUTPUT

After executing the program, the following results were obtained. The below screenshot (Fig 1.6.4) shows the sample output include the confusion matrix result,

```
Accuracy: 0.80

Confusion Matrix:
[[92 17]
 [20 50]]

Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.84      0.83       109
           1       0.75      0.71      0.73        70

    accuracy                           0.80       179
   macro avg       0.78      0.77      0.78       179
weighted avg       0.80      0.80      0.80       179
```

Fig. 1.6.4 Confusion matrix result

1.6.3 Lab Practice Questions

1. Modify the Logistic Regression model by changing the train–test split ratio and observe how it affects the accuracy of the Titanic survival prediction.

2. Identify the most important features influencing survival in the Titanic dataset and explain their impact on the model's predictions.

EXPERIMENT 7

Create a prediction model for cancer using logistic regression - Use Breast Cancer Wisconsin dataset

Objectives

By the end of this lab, students will be able to:

- To understand the logistic regression algorithm and when it is appropriate to use.
- To learn data loading, preprocessing, feature scaling, and train/test splitting.
- To build, train, and evaluate a logistic regression model for binary classification.
- To compute and interpret evaluation metrics: accuracy, confusion matrix, precision, recall, F1-score, and ROC–AUC.
- To practice model selection / regularization and basic interpretation of model coefficients.

1.7.1 Theory

1.7.1.1 Introduction to Logistic Regression

Logistic Regression is a supervised learning algorithm used for binary classification. It models the probability that a sample belongs to the positive class using the logistic (sigmoid) function applied to a linear combination of features. Output is interpreted as probability; a threshold (commonly 0.5) maps probability to class labels.

Key points:

- Outputs probabilities in (0,1).
- Loss function: log-loss / cross-entropy.
- Can include L1/L2 regularization to avoid overfitting.
- Coefficients indicate direction and relative influence of features (after scaling).

1.7.1.2 Breast Cancer Wisconsin Dataset

- A standard benchmark dataset for binary classification (malignant vs benign).
- Features: real-valued measurements of cell nuclei from digitized images (e.g., radius, texture, perimeter, area, smoothness).

- Labels: malignant (often encoded as 0/1 or 1/0 depending on loader).
- Available via sklearn.datasets.load_breast_cancer().

1.7.1.3 Typical ML pipeline for this experiment

1. Load dataset (features X, labels y).
2. Exploratory Data Analysis: class balance, feature overview.
3. Preprocessing: handle missing values (dataset has none), scale features (StandardScaler).
4. Split data into training and test sets (e.g., 70/30 or 80/20).
5. Train logistic regression (with solver choice, regularization).
6. Evaluate on a test set with accuracy, confusion matrix, precision, recall, F1-score, ROC curve, and AUC.
7. Optionally tune regularization C using cross-validation.

1.7.2 Implementation

```
# Logistic Regression on Breast Cancer Wisconsin dataset
# Requirements: scikit-learn, pandas, numpy, matplotlib
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (accuracy_score, confusion_matrix, precision_score,
                             recall_score, f1_score, roc_auc_score,
                             roc_curve, classification_report)
import matplotlib.pyplot as plt
```

```
# Load data
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target)  # 0 = malignant, 1 = benign (note: check loader docs)
# Quick EDA
print("Shape:", X.shape)
print("Class distribution:\n", y.value_counts())
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42, stratify=y)
# Feature scaling
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)
# Train logistic regression with default L2 regularization
clf = LogisticRegression(max_iter=10000, solver='liblinear')  # 'liblinear' works well for small datasets
clf.fit(X_train_s, y_train)
# Predictions
y_pred = clf.predict(X_test_s)
y_proba = clf.predict_proba(X_test_s)[:, 1]  # probability for positive class
# Evaluation metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
```

```
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_proba)
cm = confusion_matrix(y_test, y_pred)
print("\nAccuracy: {:.4f}".format(acc))
print("Precision: {:.4f}".format(prec))
print("Recall: {:.4f}".format(rec))
print("F1-score: {:.4f}".format(f1))
print("ROC AUC: {:.4f}".format(auc))
print("\nConfusion Matrix:\n", cm)
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

ROC curve

```
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
plt.figure()
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {auc:.3f})')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Logistic Regression')
plt.legend(loc='lower right')
plt.show()
```

Optional: Hyperparameter tuning for regularization strength

```
param_grid = {'C': [0.01, 0.1, 1, 10, 100]}
gs = GridSearchCV(LogisticRegression(solver='liblinear', max_iter=10000),
                  param_grid, cv=5, scoring='roc_auc')
gs.fit(X_train_s, y_train)
```

```
print("Best C:", gs.best_params_)

best_clf = gs.best_estimator_

y_pred_best = best_clf.predict(X_test_s)

print("Tuned ROC AUC:", roc_auc_score(y_test, best_clf.predict_proba(X_test_s)[:,1]))
```

OUTPUT

- Dataset shape, number of features (30), and class distribution.
- Train/test split sizes (e.g., train 426, test 143 for 75/25 split).
- Printed evaluation metrics (accuracy, precision, recall, F1, ROC AUC). Example format:

  Accuracy: 0.96

  Precision: 0.97

  Recall: 0.95

  F1-score: 0.96

  ROC AUC: 0.99

  Confusion Matrix:

  [[TN FP]

  [FN TP]]
- ROC curve plotted with area under curve ~ high (e.g., 0.98–0.99 depending on split).
- Best regularization parameter C found by GridSearch (if tuning done) and improved/worse metrics accordingly.
- Short discussion interpreting confusion matrix:
  - False Negatives (FN) are the most critical in cancer detection , discussing trade-offs (choose threshold to reduce FN at cost of FP if clinically required).

1.7.3 Lab Practice Questions

1. Write a Python program to load the Breast Cancer Wisconsin dataset, train a logistic regression model, and report accuracy, precision, recall, F1-score, and ROC–AUC.

2. Plot the ROC curve and compute AUC for your model. Interpret the results.

3. Perform hyperparameter tuning for Logistic Regression regularization parameter C using 5-fold cross-validation and report the best C.

4. Identify the top 5 features (by absolute coefficient) contributing to the prediction and explain their meaning.

5. Change the decision threshold from 0.5 to 0.3 and report how precision and recall change. Discuss which threshold is preferable for cancer screening and why.

6. Compare logistic regression performance with a simple Decision Tree classifier (report metrics and give a short interpretation).

EXPERIMENT 8

Implement and evaluate the Linear Discriminant Analysis (LDA) algorithm on the Wine dataset

Objectives

By the end of this lab, students will be able to:

- Familiarize the concept of Linear Discriminant Analysis (LDA).
- Understand how LDA reduces dimensions while keeping class information.
- Implement LDA on the Wine dataset using Python.

1.8.1 Theory

1.8.1.1 Introduction to Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction technique used to separate multiple classes in a dataset. Unlike PCA, which focuses only on variance, LDA considers the class labels and tries to project the data onto a lower-dimensional space where the classes are best separated.

LDA works by calculating two important measures:

1. **Between-class scatter:** how far the class means are from each other.
2. **Within-class scatter:** how spread out the samples are within each class.

LDA finds a projection that maximizes the ratio:

$$\frac{\text{Between-class variance}}{\text{Within-class variance}}$$

This means LDA tries to pull classes far apart while keeping each class tight and compact. If a dataset has k classes, LDA can produce at most k − 1 new components. Since the Wine dataset has 3 classes, LDA reduces it to 2 components.

Before applying LDA, the features are standardized using StandardScaler because LDA works with linear combinations of features. Standardization ensures all features contribute equally.

After reducing the dimensions, a machine learning classifier (like Logistic Regression) can be trained on the transformed data. LDA often improves class separation and can lead to better classification accuracy. Finally, the new LDA components can be plotted to visually observe how well the classes are separated.

This theory helps students understand why each step in the program—loading data, standardizing, applying LDA, training a classifier, and plotting—is necessary.

1. Importing Libraries

In machine learning programs, we import pre-built modules to perform tasks like loading data, scaling, training models, and plotting graphs.

Example concepts:

import matplotlib.pyplot as plt → for plotting graphs

from sklearn.datasets import load_wine → for loading datasets

from sklearn.model_selection import train_test_split → for splitting data

Libraries save time by providing ready-made functions.

2. Loading a Dataset

Machine learning programs usually start by loading data.

load_wine() returns:

Features (X) → the inputs

Labels (y) → the class for each wine sample

X and y are usually stored in matrix/array form.

3. Splitting Data into Training and Testing Sets

train_test_split(X, y, test_size=0.3)

Training data → used to teach the model

Testing data → used to evaluate the model

test_size=0.3 → 30% test data, 70% training data

This is a basic practice in machine learning.

4. Standardization (Scaling)

Using:

StandardScaler()

- Scaling makes features comparable
- Models work better when data is normalized

- Fit on training data, transform both training and test data

This prevents large-valued features from dominating.

5. Dimensionality Reduction

Concept:

LinearDiscriminantAnalysis(n_components=2)

LDA reduces many features to a smaller number (here, 13 → 2)

New features are combinations of original features

It improves class separation and visualization

6. Training a Machine Learning Model

LogisticRegression( )

- **Fit** → learning patterns
- **Predict** → making predictions on new data

These are fundamental machine-learning operations.

7. Evaluating the Model

Using:

accuracy_score()

confusion_matrix()

- **Accuracy** → how many predictions were correct
- **Confusion matrix** → class-wise performance

These are basic evaluation methods.

8. Plotting Results

Using:

plt.scatter()

plt.show()

Scatter plots visualize two-dimensional data

Color indicates different classes

Helps to see how well LDA separates classes

1.8.2 Implementation

1. This program only loads the dataset, applies LDA, and prints the transformed values.

```
from sklearn.datasets import load_wine

from sklearn.preprocessing import StandardScaler

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Load dataset

data = load_wine()

X = data.data

y = data.target

# Standardize features

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

# Apply LDA

lda = LinearDiscriminantAnalysis(n_components=2)

X_lda = lda.fit_transform(X_scaled, y)

print("LDA Transformed Shape:", X_lda.shape)

print("First 5 Transformed Samples:\n", X_lda[:5])
```

OUTPUT

```
LDA Transformed Shape: (178, 2)

First 5 Transformed Samples:

 [[-4.70024401  1.97913835]

 [-4.30195811  1.17041286]

 [-3.42071952  1.42910139]

 [-4.20575366  4.00287148]

 [-1.50998168  0.4512239 ]]
```

2. A short and clear program that applies LDA and evaluates a simple classifier.

```
from sklearn.datasets import load_wine

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score

# Load dataset

data = load_wine()

X = data.data

y = data.target

# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Standardize features

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test  = scaler.transform(X_test)

# Apply LDA

lda = LinearDiscriminantAnalysis(n_components=2)

X_train_lda = lda.fit_transform(X_train, y_train)

X_test_lda = lda.transform(X_test)

# Train classifier

model = LogisticRegression()

model.fit(X_train_lda, y_train)
```

```
# Predict

y_pred = model.predict(X_test_lda)

# Accuracy

print("Accuracy:", accuracy_score(y_test, y_pred))
```

OUTPUT

```
Accuracy: 1.0
```

3. **Write a program to implement Linear Discriminant Analysis (LDA) on the Wine dataset.**

```
from sklearn.datasets import load_wine

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix

import matplotlib.pyplot as plt

# Load dataset

data = load_wine( )

X = data.data

y = data.target

# Split into training and testing

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

# Standardize

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)

# Apply LDA

lda = LinearDiscriminantAnalysis(n_components=2)

X_train_lda = lda.fit_transform(X_train, y_train)

X_test_lda = lda.transform(X_test)

# Train classifier

model = LogisticRegression( )

model.fit(X_train_lda, y_train)

# Predictions

y_pred = model.predict(X_test_lda)

# Evaluation

print("Accuracy:", accuracy_score(y_test, y_pred))

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

# Plot LDA components

plt.figure(figsize=(6,5))

plt.scatter(X_train_lda[:,0], X_train_lda[:,1], c=y_train, cmap='viridis')

plt.title("LDA Projection of Wine Dataset")

plt.xlabel("LD1")

plt.ylabel("LD2")

plt.show()
```

OUTPUT

Accuracy: 0.9814814814814815

Confusion Matrix:

[[19 0 0]

[1 20 0]

[0 0 14]]

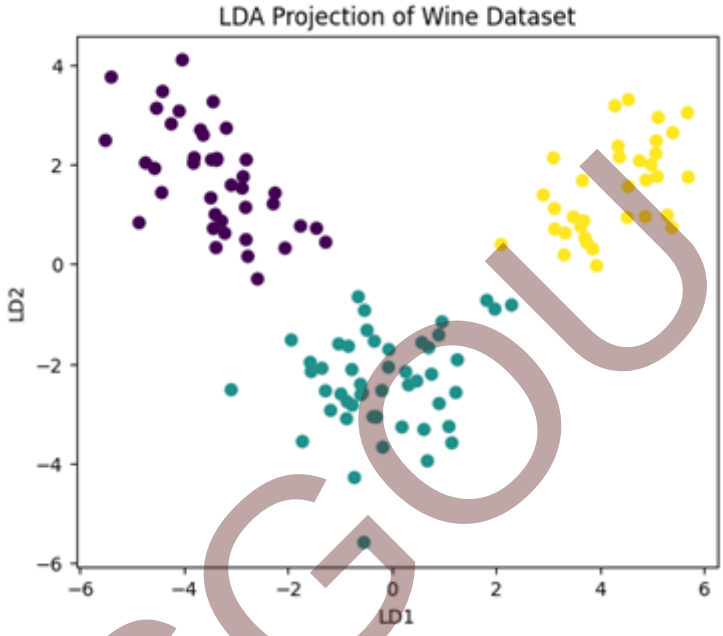
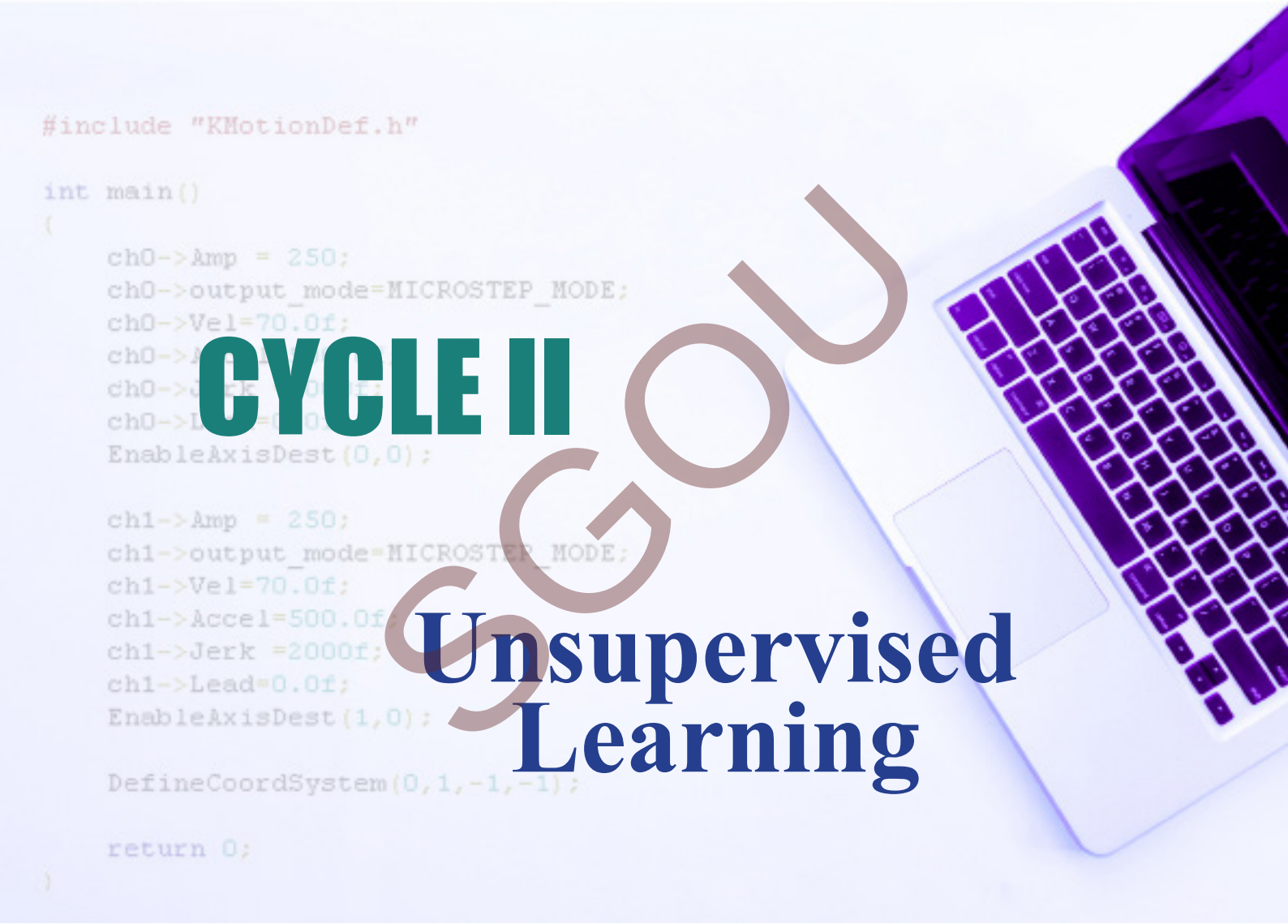


Fig. 1.8.1 LDA projection of wine dataset

1.8.3 Lab Practice questions

1. Perform LDA without standardizing the features. Observe and record the change in accuracy.
2. Visualize the LDA components using different color maps and analyze class separability.
3. Change the test size (e.g., 0.2, 0.4, 0.5). Evaluate how accuracy varies.
4. Print the explained variance ratio of LDA components. Interpret the result.



CYCLE II

Unsupervised Learning

EXPERIMENT 1

Hierarchical Clustering using IRIS Dataset

Objectives

By the end of this lab, students will be able to:

- To implement hierarchical clustering on the IRIS dataset
- To apply Ward's linkage method on real-world datasets
- To visualize clusters using dendrograms
- To analyze clustering results

2.1.1 Theory

2.1.1.1 Hierarchical clustering

Hierarchical clustering is an unsupervised learning technique used to group data points into clusters based on similarity. Unlike partition-based clustering (such as K-means), hierarchical clustering builds a tree-like structure called a dendrogram, which represents nested groupings of data.

Hierarchical clustering can be classified into two types:

- **Agglomerative (Bottom-Up):** Each data point starts as an individual cluster and clusters are merged step by step.
- **Divisive (Top-Down):** All data points start in one cluster and are recursively split.

In this experiment, Agglomerative Hierarchical Clustering is applied to the IRIS dataset.

2.1.1.2 Algorithm

- Load the IRIS dataset.
- Select relevant feature columns.
- Standardize the dataset (optional but recommended).
- Apply hierarchical clustering using linkage criteria.
- Plot the dendrogram.
- Form clusters and analyze the results.

2.1.1.3 Software / Hardware Requirements

- Software: Python 3.x, Jupyter Notebook / Google Colab
- Libraries: numpy, pandas, scikit-learn, scipy, matplotlib
- Hardware: Minimum 4GB RAM, Windows/Linux/Mac OS

2.1.2 Implementation

1. Implement Hierarchical Clustering on the IRIS dataset using Ward's linkage method and plot the dendrogram.

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import dendrogram, linkage
# Load dataset
iris = load_iris()
X = iris.data
# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Hierarchical clustering using Ward's method
linked = linkage(X_scaled, method='ward')
# Plot dendrogram
plt.figure(figsize=(10, 6))
dendrogram(linked)
plt.title("Hierarchical Clustering Dendrogram (IRIS Dataset)")
plt.xlabel("Sample Index")
plt.ylabel("Distance")
plt.show()
```

OUTPUT

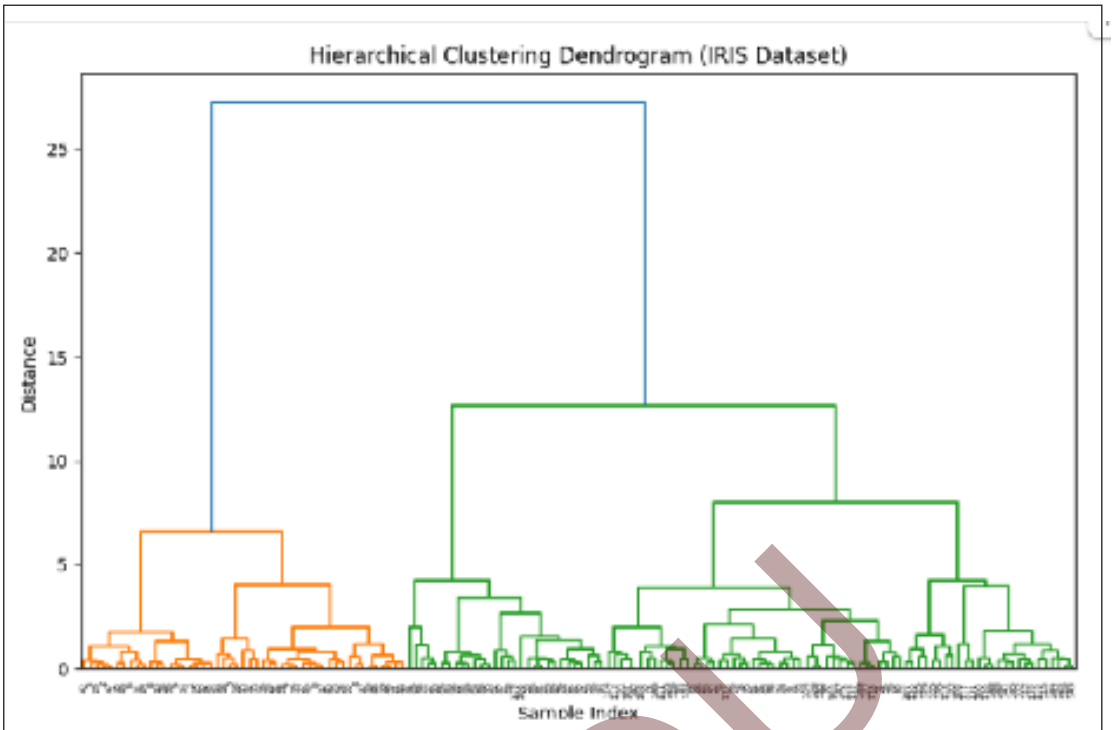


Fig. 2.1.1 Hierarchical clustering

2. Apply Hierarchical Clustering on the IRIS dataset and form flat clusters by selecting three clusters. Display cluster labels.

```
from sklearn.datasets import load_iris

from sklearn.preprocessing import StandardScaler

from scipy.cluster.hierarchy import linkage, fcluster

# Load dataset

iris = load_iris()

X = iris.data

# Standardize data

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

# Perform clustering

linked = linkage(X_scaled, method='ward')
```

```
# Create flat clusters

clusters = fcluster(linked, t=3, criterion='maxclust')

print("Cluster labels:\n", clusters)
```

OUTPUT

```
Cluster labels:

 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

 1 1 1 1 2 1 1 1 1 1 1 1 1 3 3 3 2 3 2 3 2 3 2 2 3 2 3
2 3 2 2 2 2 3 3 3 3

 3 3 3 3 3 2 2 2 2 3 2 3 3 2 2 2 2 3 2 2 2 2 2 3 2 2 3
3 3 3 3 3 2 3 3 3 3

 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3

 3 3]
```

3. Visualize the hierarchical clustering result by plotting clusters using the first two features of the IRIS dataset.

```
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.preprocessing import StandardScaler

from scipy.cluster.hierarchy import linkage, fcluster

# Load data

iris = load_iris()

X = iris.data

# Standardize features

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)
```

```
# Hierarchical clustering
linked = linkage(X_scaled, method='ward')
clusters = fcluster(linked, t=3, criterion='maxclust')
# Plot clusters using first two features
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis')
plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.title("Hierarchical Clustering on IRIS Dataset")
plt.show()
```

OUTPUT

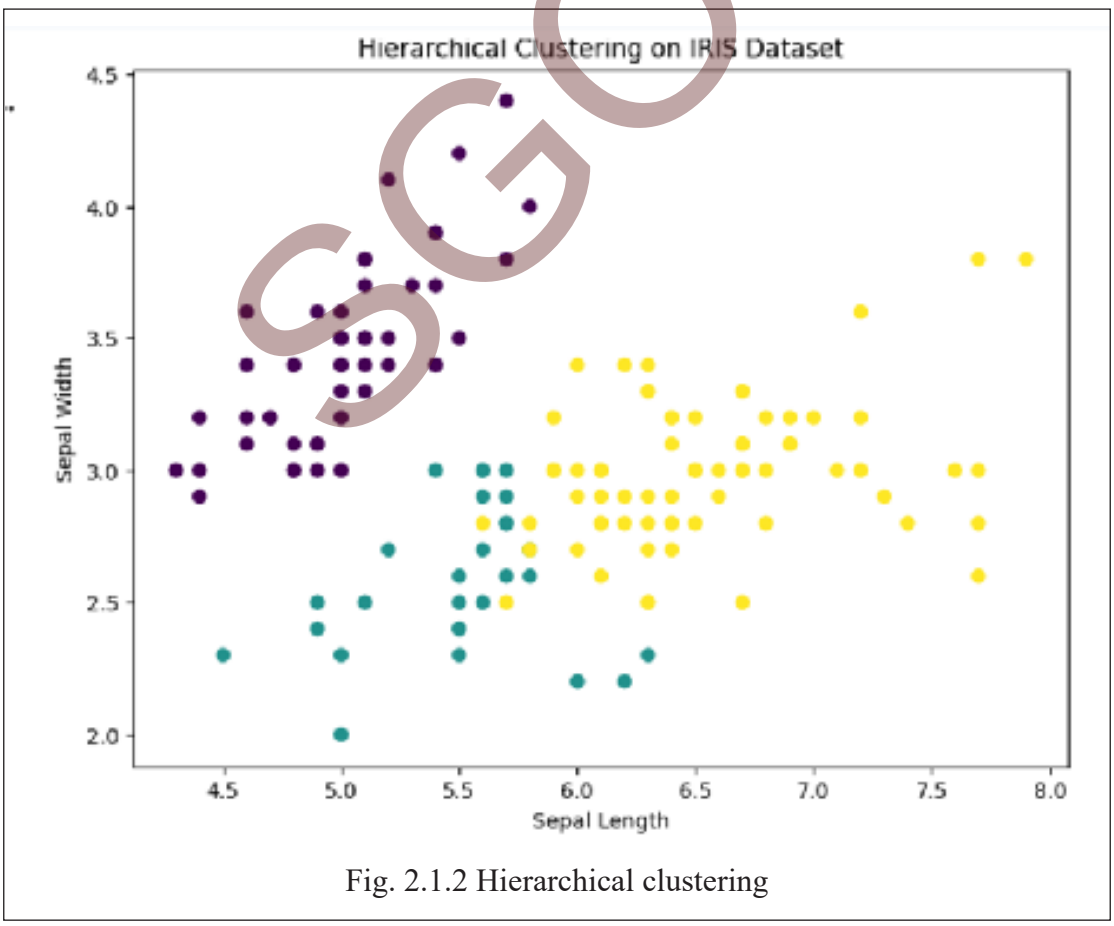


Fig. 2.1.2 Hierarchical clustering

4. Perform Hierarchical Clustering using different linkage methods (single, complete, average) and compare their dendrograms.

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import dendrogram, linkage
# Load dataset
iris = load_iris()
X = iris.data
# Standardize data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
methods = ['single', 'complete', 'average']
plt.figure(figsize=(15, 10))
for i, method in enumerate(methods, 1):
    plt.subplot(2, 2, i)
    linked = linkage(X_scaled, method=method)
    dendrogram(linked)
    plt.title(f"{method.capitalize()} Linkage")
plt.tight_layout()
plt.show()
```

OUTPUT

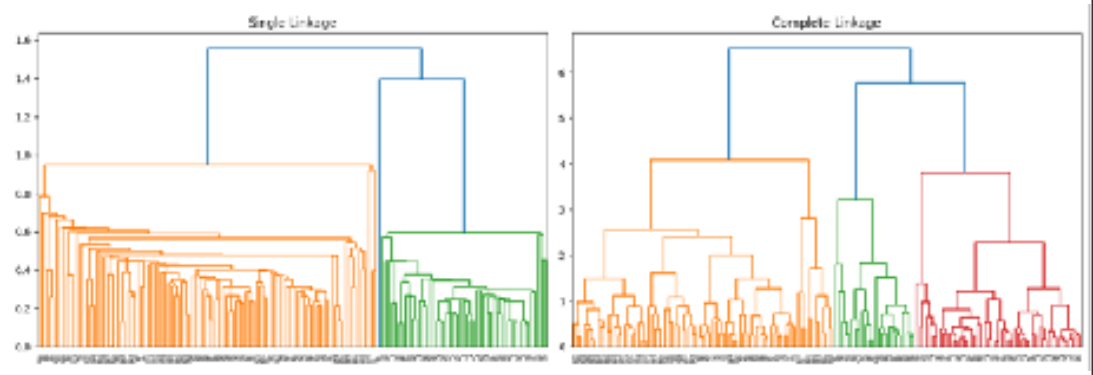


Fig. 2.1.3 Hierarchical clustering with linkage

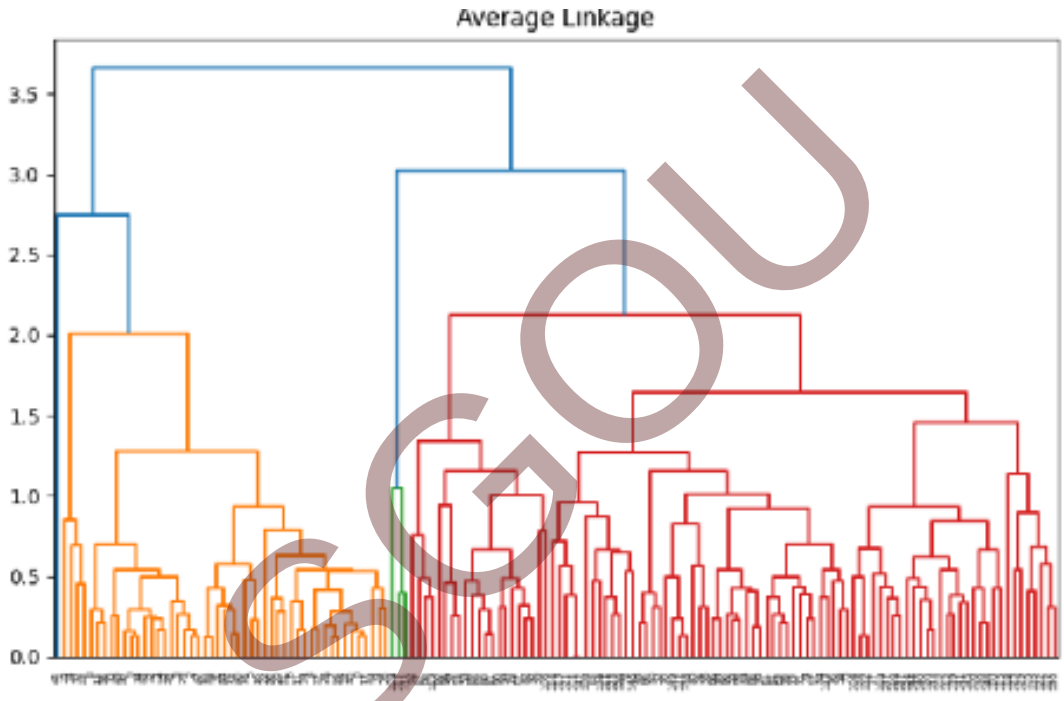


Fig. 2.1.4 Average linkage

2.1.2 Lab Practice Questions

1. Perform hierarchical clustering on the IRIS dataset without standardization and analyze the effect on clustering.
2. Compare the results of Hierarchical Clustering and K-means Clustering on the IRIS dataset.
3. Analyze how changing the number of clusters affects the structure of flat clusters.
4. Apply hierarchical clustering on a different real-world dataset and interpret the results.
5. Identify situations where Hierarchical Clustering is preferred over K-means.

EXPERIMENT 2

Implementation of BIRCH and DBSCAN Clustering Algorithms Using a Publicly Available Dataset

Objectives

By the end of this lab, learners will be able to:

- Understand the concept of clustering in unsupervised learning
- Explain the working principles of BIRCH and DBSCAN algorithms
- Implement BIRCH and DBSCAN clustering using a publicly available dataset
- Observe and interpret the clusters formed by each algorithm

2.2.1 Theory

In this experiment, clustering algorithms are implemented using a publicly available dataset, namely the IRIS dataset, with the help of Python and machine learning libraries. Since clustering is an unsupervised learning technique, the class labels present in the dataset are not used during the implementation. Only the numerical feature values are considered for grouping the data.

The experiment involves loading the dataset, preprocessing the data, applying clustering algorithms, and observing the clusters formed.

2.2.1.1 Dataset Used – IRIS Dataset

The IRIS dataset is used in this experiment for implementing clustering algorithms. It consists of 150 data instances, each described by four numerical attributes:

- Sepal length
- Sepal width
- Petal length
- Petal width

Although the dataset contains class labels for three iris species, these labels are ignored during clustering, as clustering does not require predefined categories. The dataset is suitable for laboratory experiments because it is clean, balanced, and easy to understand.

In the implementation, the dataset is loaded directly using the Scikit-learn library, which avoids the need for manual data downloading and simplifies laboratory execution.

2.2.1.2 BIRCH Clustering

BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) is implemented in this experiment using the Birch class available in the Scikit-learn library.

Initially, the dataset is loaded into the program and only the feature values are considered for clustering. The data is then preprocessed using standardization, which scales the features so that all attributes contribute equally to distance calculations.

The BIRCH algorithm works by incrementally building a Clustering Feature (CF) Tree from the input data. Instead of storing all data points, the CF tree stores summarized information such as the number of points and their statistical properties. This makes BIRCH memory-efficient and suitable for large datasets.

In the implementation, the number of clusters is specified using the n_clusters parameter. After fitting the model to the dataset, each data point is assigned a cluster label. These labels are printed and observed to understand how the data is grouped.

2.2.1.3 DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is implemented using the DBSCAN class from Scikit-learn.

After loading and preprocessing the dataset, DBSCAN groups data points based on density rather than distance to cluster centers. Two important parameters are defined during implementation:

- eps – the radius of the neighborhood around a data point
- min_samples – the minimum number of points required to form a dense region

The algorithm identifies core points, border points, and noise points based on these parameters. Data points that do not belong to any dense region are marked as noise and assigned a label -1.

During implementation, the fit_predict() method is used to assign cluster labels to each data point. The output labels are examined to identify clusters and noise points formed by DBSCAN.

Observation of Clustering Results

The cluster labels obtained from BIRCH and DBSCAN are analyzed to observe how data points are grouped. In BIRCH, each data point is assigned to one of the predefined clusters. In DBSCAN, some data points may not belong to any cluster and are identified as noise.

By changing the parameter values such as the number of clusters in BIRCH and the eps and min_samples values in DBSCAN, different clustering patterns can be observed. This helps in understanding the working behavior of both clustering algorithms in a practical environment.

2.2.2 Implementation

Write a python program to implement BIRCH and DBSCAN clustering algorithms on the IRIS dataset. Standardize the dataset before clustering and visualize the clusters formed by each algorithm using the first two features (sepal length and sepal width)

```
# Import required libraries

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.cluster import Birch, DBSCAN

from sklearn.preprocessing import StandardScaler

# Load IRIS dataset

iris = load_iris()

X = iris.data

# Standardize data

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

# Apply BIRCH clustering

birch = Birch(n_clusters=3)

birch_labels = birch.fit_predict(X_scaled)

# Apply DBSCAN clustering

dbscan = DBSCAN(eps=0.8, min_samples=5)

dbscan_labels = dbscan.fit_predict(X_scaled)

# -------- Visualization using first two features --------

# BIRCH Cluster Visualization

plt.figure()

plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=birch_labels)

plt.title("BIRCH Clustering on IRIS Dataset")
```

```
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
# DBSCAN Cluster Visualization
plt.figure()
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=dbscan_labels)
plt.title("DBSCAN Clustering on IRIS Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

OUTPUT

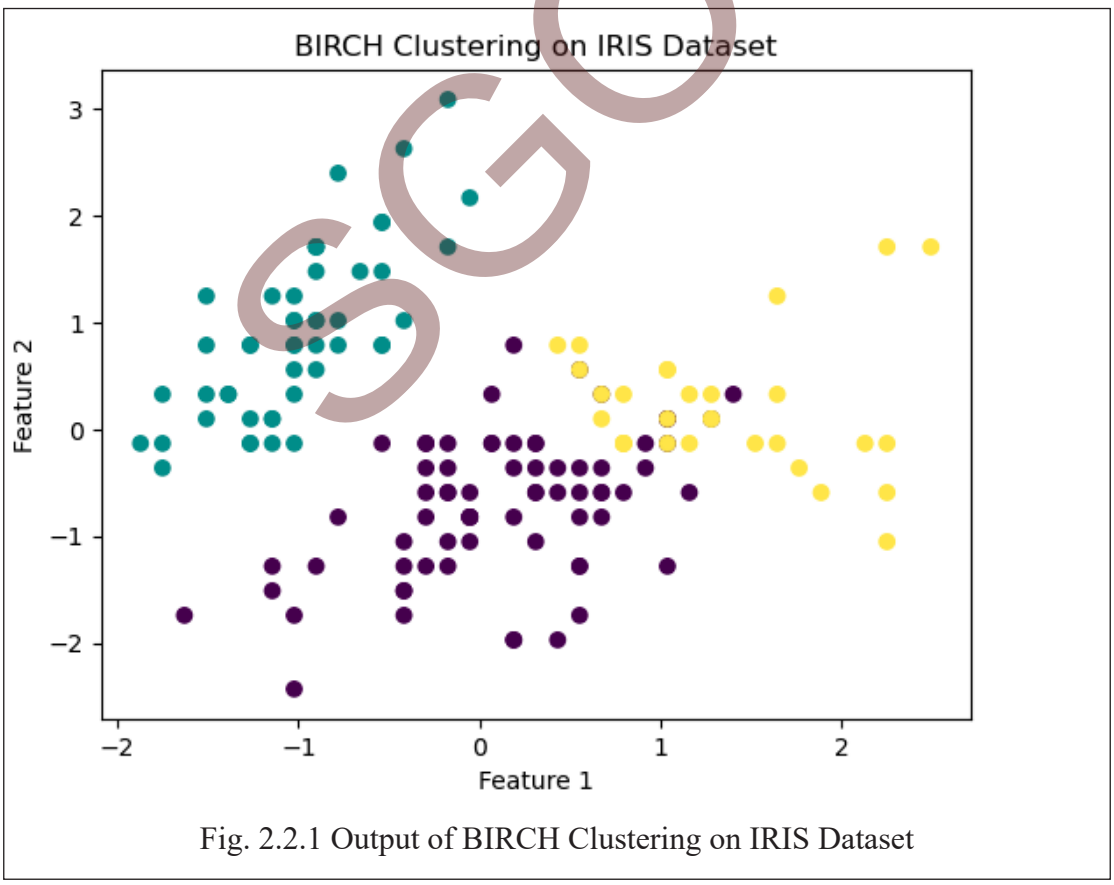


Fig. 2.2.1 Output of BIRCH Clustering on IRIS Dataset

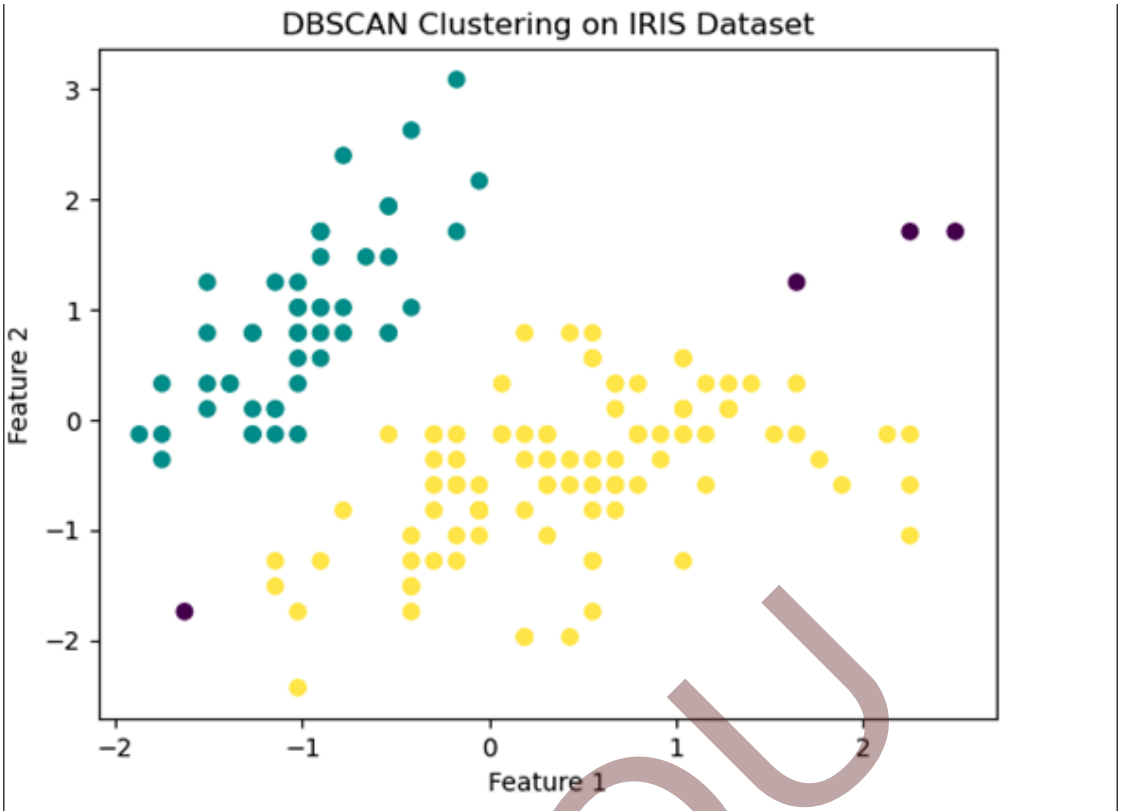


Fig. 2.2.2 Output of DBSCAN Clustering on IRIS Dataset

2.2.3 Lab Practice Questions

1. Apply BIRCH and DBSCAN clustering on the IRIS dataset and observe the clusters formed.
2. Implement BIRCH clustering using another publicly available dataset such as:
   - Wine dataset
   - Breast Cancer dataset
   - Seeds dataset
3. Apply DBSCAN clustering on the same publicly available dataset and observe cluster formation.
4. Compare the clustering results obtained using IRIS dataset and the selected public dataset based on visual observation.
5. Identify noise points produced by DBSCAN in the selected dataset, if any.
6. Perform clustering using only two features from the selected dataset and visualize the results.

EXPERIMENT 3

Implementation of Gaussian Mixture Models (GMM) for Clustering using the Wine Dataset

Objectives

By the end of this lab, students will be able to:

- To understand the concept of Gaussian Mixture Models
- To apply GMM for clustering a real-world dataset
- To analyze clustering results and compare them with actual class labels
- To visualize clusters formed by GMM

2.3.1 Theory

Gaussian Mixture Models are probabilistic clustering algorithms that assume data points are generated from a mixture of several Gaussian distributions. Unlike k-means clustering, GMM assigns probabilities to each data point belonging to each cluster. This allows flexible cluster shapes and overlapping clusters.

Each Gaussian component is defined by:

- Mean (μ)
- Covariance matrix (Σ)
- Mixing coefficient (π)

GMM uses the Expectation–Maximization (EM) algorithm:

- Expectation step (E-step): Estimates the probability of each point belonging to each Gaussian component
- Maximization step (M-step): Updates parameters to maximize likelihood

The Wine dataset contains chemical analysis results of wines derived from three different cultivars. It is commonly used for clustering and classification tasks.

2.3.1.1 Introduction to Gaussian Mixture Model

Gaussian Mixture Model (GMM) is a probabilistic clustering technique used to identify hidden patterns in unlabeled data. It assumes that the data is generated from a mixture of multiple Gaussian distributions with unknown parameters. Unlike hard clustering methods, GMM assigns probabilities to each data point indicating its likelihood of belonging to different clusters. This makes GMM suitable for datasets where clusters may overlap. The model uses the Expectation–Maximization algorithm to iteratively

estimate the parameters of the Gaussian distributions. GMM can capture complex cluster shapes by considering covariance between features. It is particularly effective for multivariate numerical datasets such as the Wine dataset. Using GMM helps in understanding the natural grouping of data based on underlying statistical properties. The probabilistic nature of the model provides better interpretability of clustering results. Therefore, Gaussian Mixture Model is widely used in data analysis and pattern recognition tasks.

2.3.1.2 Key Characteristics of GMM

- Performs soft clustering (probabilistic assignment).
- Can model clusters of different shapes and sizes.
- More flexible than K-Means.
- Suitable for overlapping clusters.

2.3.1.3 Limitations

- Computationally more expensive than K-Means.
- Sensitive to initialization.
- May converge to local optima.

2.3.1.4 Applications

- Image segmentation
- Speech recognition
- Anomaly detection
- Bioinformatics
- Pattern recognition

2.3.2 Requirements

To implement and evaluate the Gaussian Mixture Model using the Wine dataset, the following software, hardware, and dataset resources are required:

1. **Software / Tools**

- **Python Programming Language:** Used for writing and executing machine learning code.
- **Jupyter Notebook / Google Colab:** Interactive environment for running Python programs, visualizing data, and documenting steps.

- **Python Libraries:**
    - **Pandas:** For loading and preprocessing the dataset (data cleaning, handling missing values, data manipulation).
    - **NumPy:** For numerical computation and array operations.
    - **scikit-learn (sklearn):** Machine learning library used to implement logistic regression, split data, and evaluate model performance.
    - **matplotlib / seaborn (optional):** For visualizing the dataset and results, such as graphs and confusion matrices.

2. Hardware Requirements

- Computer / Laptop with basic processing capability.
- Minimum Configuration:
    - **Processor:** Dual-core CPU or higher
    - **RAM:** Minimum 4 GB (8 GB recommended for smooth data processing)
    - **Storage:** At least 1 GB of free disk space

(Cloud platforms like Google Colab remove most hardware limitations by running code online.)

3. Dataset Required

Wine Dataset

This refers to the Wine dataset, which is a well-known machine learning dataset commonly used for classification problems.

Source: scikit-learn library

The dataset is built into the scikit-learn library, so you don't need to download it separately. It can be easily loaded using Python for experiments and learning.

Number of samples: 178

There are 178 data records in total. Each record represents one wine sample.

Number of features: 13

Each wine sample is described using 13 numerical features, such as:

- Alcohol content
- Malic acid
- Ash

- Magnesium
- Color intensity
- Flavanoids

(These features represent chemical properties of the wine.)

Classes: 3 wine categories

The wines are classified into 3 different classes, which correspond to three different types of wine. This makes it a multiclass classification problem.

Type: Multivariate numerical dataset

- Multivariate → More than one feature (13 features per sample)
- Numerical → All feature values are numbers (no text or categorical values)

The Wine dataset contains 178 wine samples, each described by 13 numerical chemical properties, and each sample belongs to one of three wine classes. It is widely used to test and demonstrate classification algorithms in machine learning.

2.3.3 Procedure

Step 1: Import Required Libraries

Import necessary Python libraries such as:

- NumPy and Pandas for numerical and data operations.
- Matplotlib and Seaborn for visualization.
- Scikit-learn modules for dataset loading, preprocessing, clustering, and dimensionality reduction.

Step 2: Load the Dataset

Load the Wine dataset using load_wine() from scikit-learn.

- X contains the feature values (chemical properties of wine).
- y contains the true class labels (three different wine types).

Step 3: Data Standardization

Standardize the feature values using StandardScaler.

This step:

- Transforms data to have zero mean and unit variance.
- Ensures all features contribute equally to clustering.
- Prevents features with large scales from dominating the model.

The standardized data is stored in X_scaled.

Step 4: Apply Gaussian Mixture Model (GMM)

Initialize the GMM model with:

- n_components = 3 (since the dataset contains three wine classes).
- random_state = 42 for reproducibility.

Fit the model on standardized data using fit_predict():

- The model learns the Gaussian distributions.
- Each data point is assigned to a cluster.
- Cluster labels are stored in gmm_labels.

Step 5: Dimensionality Reduction using PCA

Since the Wine dataset has multiple features (13 features), visualization in high dimensions is not possible.

Apply Principal Component Analysis (PCA):

- Reduce data to 2 principal components.
- Store transformed data in X_pca.
- This enables 2D visualization of clustering results.

Step 6: Visualization

Create a figure with two subplots:

Left Plot – Actual Classes

- Plot PCA-transformed data.
- Color points based on true labels (y).
- Title: Actual Wine Classes.

Right Plot – GMM Clustered Data

- Plot PCA-transformed data.
- Color points based on GMM predicted labels (gmm_labels).
- Title: GMM Clustered Data.

Step 7: Interpretation

- Compare both plots visually.
- If cluster structure in the GMM plot closely resembles the actual class plot, the clustering is effective.

- Differences indicate mis-clustering or overlap between classes.

Workflow

1. Import libraries
2. Load dataset
3. Standardize features
4. Apply GMM clustering
5. Reduce dimensions using PCA
6. Visualize and compare results

2.3.4 Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.mixture import GaussianMixture
from sklearn.decomposition import PCA
# Load dataset
wine = load_wine()
X = wine.data
y = wine.target
# Standardize data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Apply GMM
gmm = GaussianMixture(n_components=3, random_state=42)
gmm_labels = gmm.fit_predict(X_scaled)
# Dimensionality reduction for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
# Visualization
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.scatter(X_pca[:,0], X_pca[:,1], c=y, cmap='viridis', s=50)
plt.title("Actual Wine Classes")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.subplot(1,2,2)
plt.scatter(X_pca[:,0], X_pca[:,1], c=gmm_labels, cmap='viridis', s=50)
plt.title("GMM Clustered Data")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.tight_layout()
plt.show()
```

OUTPUT

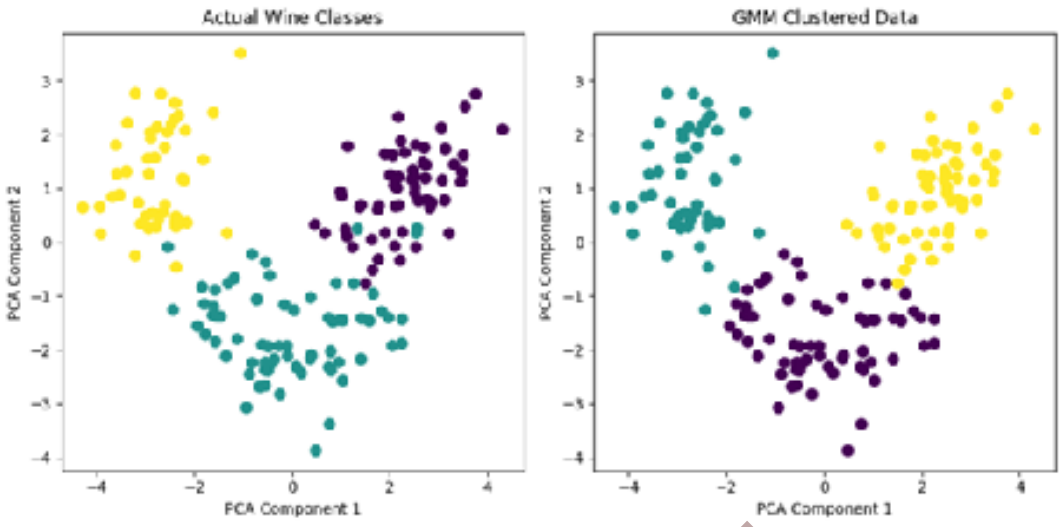


Fig. 2.3.1 Scatter Plots showing original wine class distribution (on leftside) and Clusters formed by Gaussian Mixture Model (on rightside)

Scatter plots showing:

- Original wine class distribution
- Clusters formed by Gaussian Mixture Model

Clusters may overlap due to probabilistic assignment

2.3.5 Result

Gaussian Mixture Model was successfully implemented on the Wine dataset. The algorithm effectively grouped wine samples into clusters based on probabilistic Gaussian distributions. Visualization using PCA showed that GMM clusters closely resemble actual wine categories.

2.3.6 Lab Practice Questions

1. Modify the number of Gaussian components in the Gaussian Mixture Model (for example, 2, 3, and 4 components) and observe how this change affects the clustering accuracy and overall cluster separation of the Wine dataset.

2. Identify the most influential features in separating the wine clusters by analyzing the Gaussian means and variances of each component, and explain how these features impact the clustering results produced by the model.

EXPERIMENT 4

Implement k-means clustering using a titanic dataset

Objectives

By the end of this lab, students will be able to:

- To understand the concept of unsupervised learning and clustering.
- To study the K-Means clustering algorithm and its working principle.
- To perform data loading, preprocessing, and feature scaling for clustering tasks.
- To implement K-Means clustering on the Titanic dataset.
- To evaluate clustering quality using the Silhouette Score.
- To analyze and interpret the formed clusters using real-world meaning.

2.4.1 Theory

2.4.1.1 Introduction to K-Means Clustering

K-Means is an unsupervised machine learning algorithm used to group unlabeled data into K distinct clusters based on similarity. The algorithm assigns each data point to the nearest cluster centroid such that the within-cluster variance is minimized.

Key Characteristics:

- Works on numerical data
- Distance-based (usually Euclidean distance)
- Requires predefined number of clusters (K)
- Sensitive to feature scaling and initial centroid selection

2.4.1.2 Titanic Dataset

The Titanic dataset contains information about passengers aboard the Titanic and is widely used for machine learning experiments.

Important Attributes Used for Clustering:

Age – Passenger age

Fare – Ticket fare

Pclass – Passenger class

Sex – Encoded as numerical value

Since K-Means is unsupervised, the Survived column is not used during clustering but may be used later for interpretation.

2.4.1.3 Typical Clustering Pipeline

1. Load dataset
2. Select relevant features
3. Handle missing values
4. Encode categorical variables
5. Feature scaling
6. Choose optimal K using Elbow Method
7. Apply K-Means clustering
8. Evaluate clusters using Silhouette Score
9. Interpret cluster characteristics

Algorithm (K-Means Clustering)

1. Choose the number of clusters K.
2. Randomly initialize K centroids.
3. Assign each data point to the nearest centroid.
4. Recalculate centroids as the mean of assigned points.
5. Repeat steps 3 and 4 until centroids converge.

2.4.2 Implementation

```
# K-Means Clustering on Titanic Dataset

# Requirements: pandas, numpy, scikit-learn, matplotlib, seaborn

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
# 1. Load dataset
data = pd.read_csv("titanic.csv")
# 2. Select relevant features
df = data[['Pclass', 'Sex', 'Age', 'Fare']]
# 3. Handle missing values
df['Age'].fillna(df['Age'].median(), inplace=True)
# 4. Encode categorical variable
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
# 5. Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df)
# 6. Elbow Method
wcss = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    wcss.append(kmeans.inertia_)
plt.figure()
plt.plot(range(1, 11), wcss, marker='o')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('WCSS')
```

```
plt.title('Elbow Method')
plt.show()
# 7. Apply K-Means with optimal K
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X_scaled)
df['Cluster'] = clusters
# 8. Silhouette Score
score = silhouette_score(X_scaled, clusters)
print("Silhouette Score:", score)
# 9. Cluster visualization
plt.figure()
plt.scatter(df['Age'], df['Fare'], c=clusters)
plt.xlabel('Age')
plt.ylabel('Fare')
plt.title('K-Means Clustering on Titanic Dataset')
plt.show()
```

OUTPUT

- Titanic dataset loaded and preprocessed
- Missing values handled and categorical data encoded
- Elbow curve showing optimal number of clusters (usually K = 2 or 3)
- Cluster labels assigned to each passenger
- Scatter plot visualizing clusters
- Silhouette Score indicating clustering quality (e.g., 0.3–0.5)

2.4.3 Lab Practice Questions

1. Write a Python program to load the Titanic dataset, handle missing values in the Age column, and encode the Sex attribute.

2. Write a program to apply feature scaling using StandardScaler on selected Titanic features used for clustering.

3. Write a Python program to perform K-Means clustering with K = 3, assign cluster labels to the dataset, and display the first 10 clustered records.

4. Write a program to calculate the Silhouette Score for the K-Means clustering result and visualize the clusters using a scatter plot (Age vs Fare).

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Machine Learning Lab

COURSE CODE: B24DS04PC

ISBN: 978-81-997556-1-1

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841