CALTech Lab Manual

Foundations of Data Science

SLOT - B21 + E14
FALL 2025-2026

Class No. 0410

Submitted By:
Mausam Kar
Reg. No: 24BAI10284

Devraj Vishnu
Assistant Professor
SCAI

*Collaborative and Active Learning through Technology

Contents

Chapter 1

Introduction

1.1 AbouttheLabManual

This laboratory manual provides comprehensive practical exercises for the Foundations of Data Science course. The manual covers essential data science techniques includ- ing regression analysis, dimensionality reduction, clustering, classification, and statistical testing.

1.2 Software Requirements

- Python 3.x
- Jupyter Notebook
- Required Python libraries: NumPy, Pandas, Matplotlib, Seaborn, Scikit-learn, SciPy

Chapter 2

Linear Regression

2.1 Objective

To implement linear regression using scikit-learn and evaluate the model performance.

2.2 Theory

Linear regression is a linear approach to modeling the relationship between a scalar response and one or more explanatory variables.

2.3 Code Implementation

```python
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Generating random data for demonstration
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Splitting the data 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)

# Creating a linear regression model
model = LinearRegression()

# Training the model
model.fit(X_train, y_train)

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

# Evaluating the model
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

2.4 Output

```
Mean Squared Error: 0.9190196881367937
```

2.5 Visualization

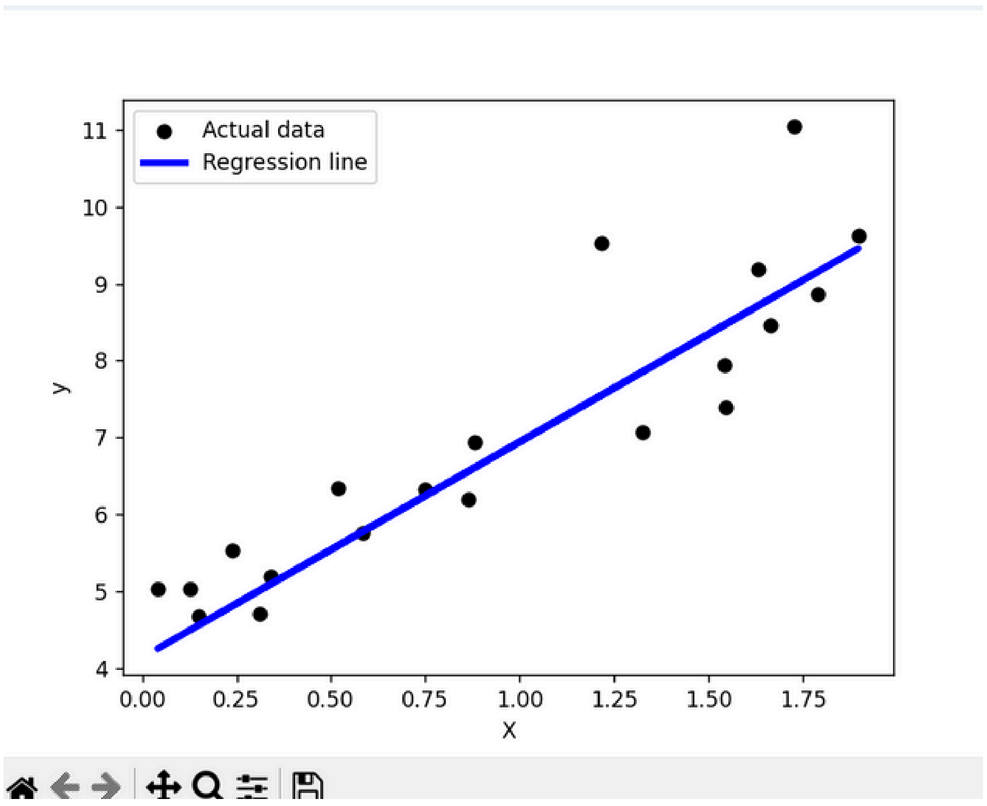


Figure 2.1: Linear Regression Visualization

Chapter 3

Singular Value Decomposition (SVD)

3.1 Objective

To perform Singular Value Decomposition on a matrix and reconstruct it.

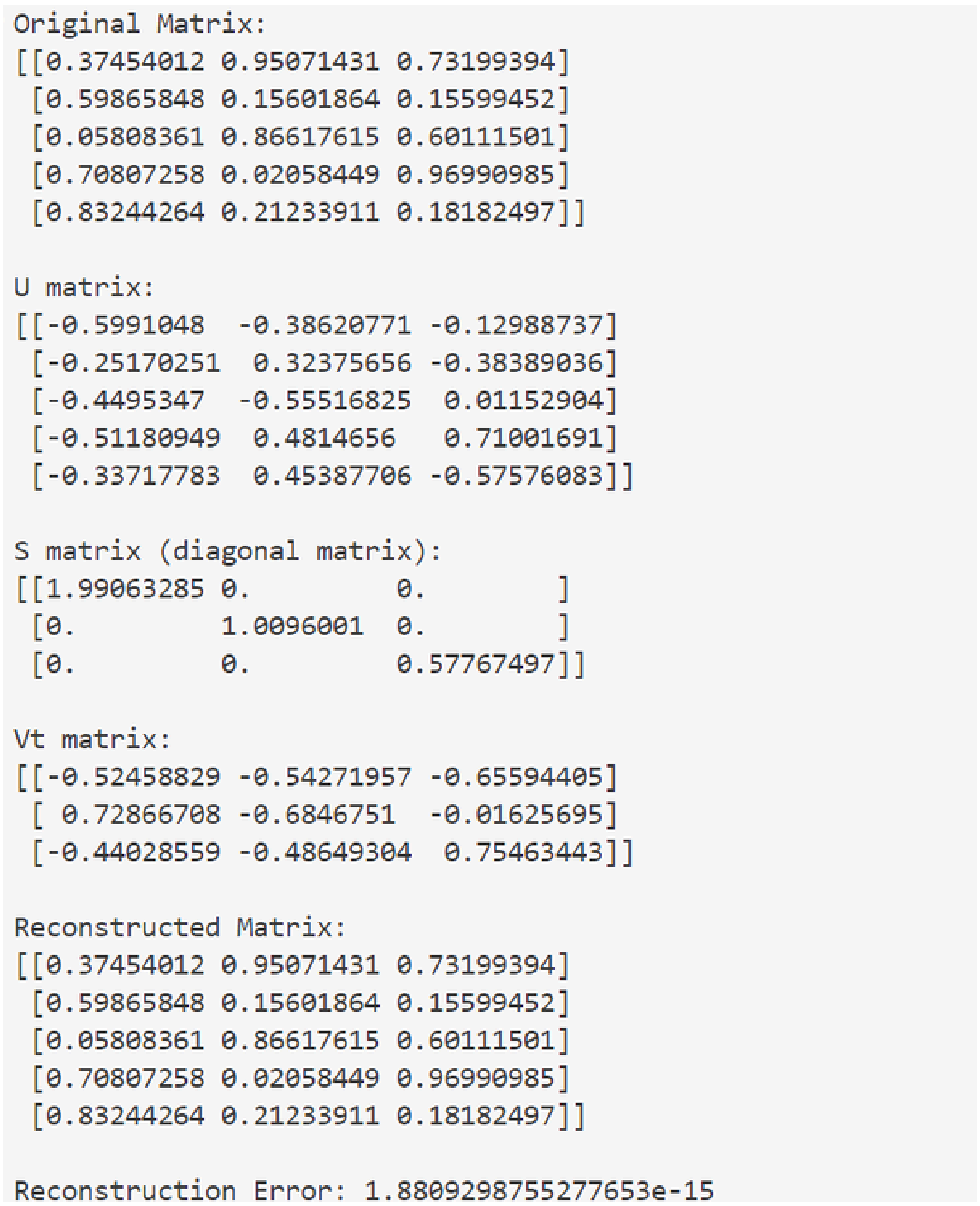
3.2 Theory

SVD is a matrix factorization method that generalizes the eigendecomposition of a square matrix to any m×n matrix.

3.3 Code Implementation

```python
import numpy as np

# Create a random matrix for demonstration
np.random.seed(42)
matrix = np.random.random((3, 5))

# Perform Singular Value Decomposition
U, S, Vt = np.linalg.svd(matrix, full_matrices=True)

# Reconstruct the original matrix from the SVD components
reconstructed_matrix = np.dot(U, np.dot(np.diag(S), Vt))

# Print the original matrix
print("Original Matrix:")
print(matrix)

# Print the decomposed components
print("\nU matrix:")
print(U)
print("\nS matrix (diagonal matrix):")
print(np.diag(S))
print("\nVt matrix:")
print(Vt)

# Print the reconstructed matrix
print("\nReconstructed Matrix:")
print(reconstructed_matrix)
```

3.4 Output



```
Original Matrix:
[[0.37454012 0.95071431 0.73199394]
 [0.59865848 0.15601864 0.15599452]
 [0.05808361 0.86617615 0.60111501]
 [0.70807258 0.02058449 0.96990985]
 [0.83244264 0.21233911 0.18182497]]

U matrix:
[[-0.5991048  -0.38620771 -0.12988737]
 [-0.25170251  0.32375656 -0.38389036]
 [-0.4495347  -0.55516825  0.01152904]
 [-0.51180949  0.4814656   0.71001691]
 [-0.33717783  0.45387706 -0.57576083]]

S matrix (diagonal matrix):
[[1.99063285 0.         0.        ]
 [0.         1.0096001  0.        ]
 [0.         0.         0.57767497]]

Vt matrix:
[[-0.52458829 -0.54271957 -0.65594405]
 [ 0.72866708 -0.6846751  -0.01625695]
 [-0.44028559 -0.48649304  0.75463443]]

Reconstructed Matrix:
[[0.37454012 0.95071431 0.73199394]
 [0.59865848 0.15601864 0.15599452]
 [0.05808361 0.86617615 0.60111501]
 [0.70807258 0.02058449 0.96990985]
 [0.83244264 0.21233911 0.18182497]]

Reconstruction Error: 1.8809298755277653e-15
```

Chapter 4

Principal Component Analysis (PCA)

4.1 Objective

To implement Principal Component Analysis for dimensionality reduction.

4.2 Theory

PCA is a technique for reducing the dimensionality of datasets, increasing interpretability while minimizing information loss.

4.3 Code Implementation

```
# Predicting the training set result through scatter plot
from matplotlib.colors import ListedColormap

X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
    stop = X_set[:, 0].max() + 1, step = 0.01),
    np.arange(start = X_set[:, 1].min() - 1,
    stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
    X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,
    cmap = ListedColormap(('yellow', 'white', 'aquamarine')))

plt.xlim(X1.min(), X1.max()
plt.ylim(X2.min(), )
                    X2.max()
for i, j in enume)rate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
    c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression(Training set)')
plt.xlabel('PC1')    for Xlabel
plt.ylabel('PC2')  # for#Ylabel
plt.legend() # to show legend

# show scatter plot
plt.show()
```

4.4 Output

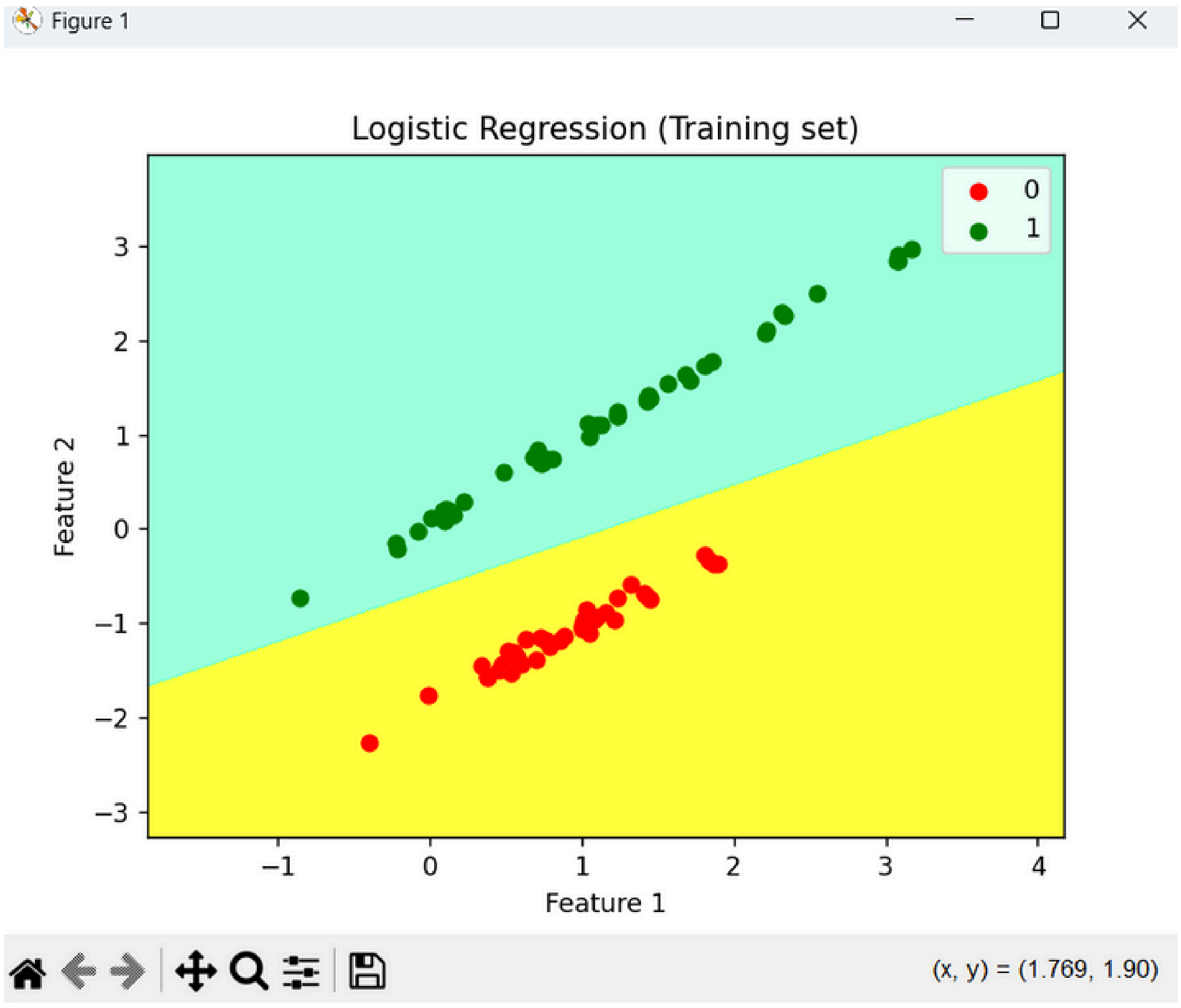


Figure 4.1: PCA Visualization

Chapter 5

Hypothesis Testing

5.1 Objective

To perform hypothesis testing using t-test to compare two samples.

5.2 Theory

Hypothesis testing is a statistical method that is used to determine whether there is enough evidence to reject a null hypothesis.

5.3 Code Implementation

```python
import numpy as np
from scipy import stats

# Generate two independent samples for demonstration
np.random.seed(42)
sample1 = np.random.normal(loc=5, scale=2, size=30)
sample2 = np.random.normal(loc=7, scale=2, size=30)

# Perform a two-sample t-test
t_statistic, p_value = stats.ttest_ind(sample1, sample2)

# Set the significance level (alpha)
alpha = 0.05

# Print the results of the t-test
print(f'T-statistic: {t_statistic}')
print(f'P-value: {p_value}')

# Check if the null hypothesis can be rejected
if p_value < alpha:
    print(f'Reject the null hypothesis at alpha = {alpha}')
else:
    print(f'Fail to reject the null hypothesis at alpha = {alpha}')
```

5.4 Output

```
T-statistic: -6.564648057307771
P-value: 2.0402324667449437e-
08 Reject the null hypothesis at
alpha = 0.05
```

Chapter 6

Confusion Matrix

6.1 Objective

To create and visualize a confusion matrix for classification model evaluation.

6.2 Theory

A confusion matrix is a table that is used to evaluate the performance of a classification mo del.

6.3 Code Implementation

```python
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression

# Generate synthetic data for demonstration
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
    random_state=42)

# Split the data 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)

# Train a logistic regression model (example classifier)
model = LogisticRegression()
model.fit(X_train, y_train)

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

# Create a confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

6.4 Output

```
Confusion Matrix:
[[190  10]
 [  5  95]]
```

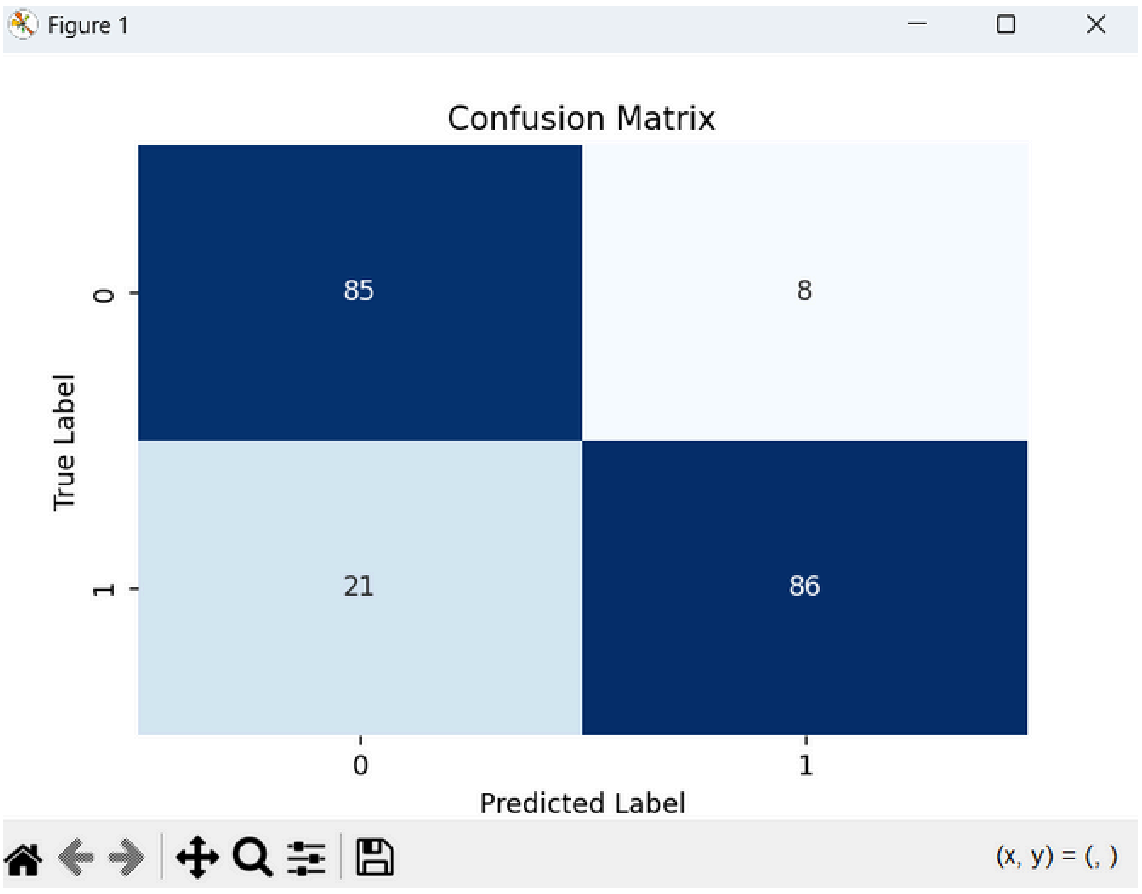


Figure 6.1: Confusion Matrix

Chapter 7

Decision Tree

7.1 Objective

To implement a Decision Tree classifier and evaluate its performance.

7.2 Theory

A Decision Tree is a flowchart-like tree structure where each internal node represents a feature, each branch represents a decision rule, and each leaf node represents an outcome.

7.3 Code Implementation

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target

# Split the data 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 a decision tree classifier
clf = DecisionTreeClassifier(random_state=42)

# Train the classifier
clf.fit(X_train, y_train)

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

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display confusion matrix and classification report
print('\nConfusion Matrix:')
print(confusion_matrix(y_test, y_pred))

print('\nClassification Report:')
print(classification_report(y_test, y_pred))

# Visualize the decision tree
plt.figure(figsize=(12, 8))
plot_tree(clf, feature_names=iris.feature_names,
class_names=iris.target_names, filled=True, rounded=True)
plt.show()
```

7.4 Output

```
Accuracy: 1.00

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

Classification Report:
              precision    recall  f1-score   support

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

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

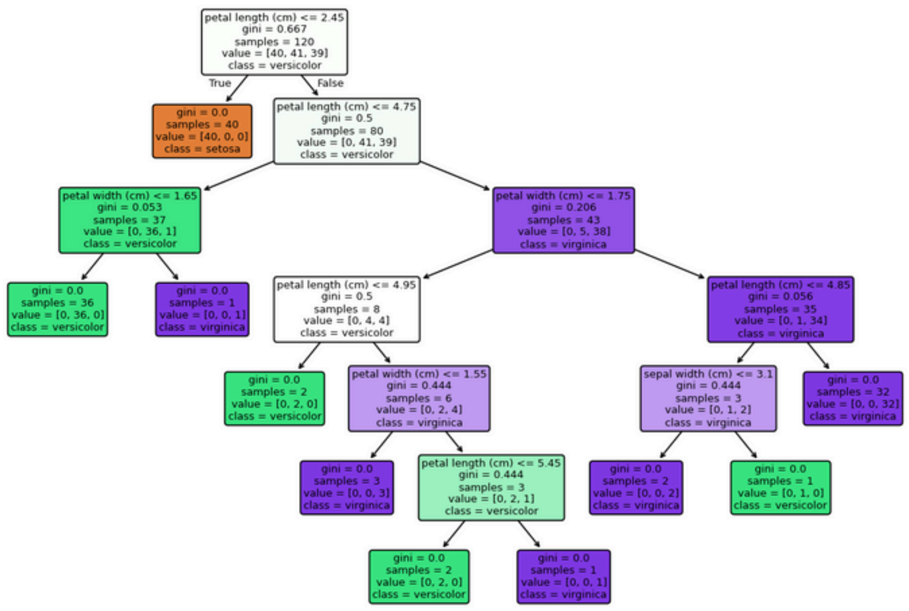


Figure 7.1: Decision Tree Structure

Chapter 8

Random Forest

8.1 Objective

To implement a Random Forest classifier and evaluate its performance.

8.2 Theory

Random Forest is an ensemble learning method that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes.

8.3 Code Implementation

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix

# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target

# Split the data 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 a Random Forest classifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the classifier
clf.fit(X_train, y_train)

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

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display confusion matrix and classification report
print('\nConfusion Matrix:')
print(confusion_matrix(y_test, y_pred))

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

8.4 Output

```
Accuracy: 1.00

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

Classification Report:
              precision    recall  f1-score   support

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

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

Chapter 9

K-means Clustering

9.1 Objective

To implement K-means clustering algorithm and determine the optimal number of clusters using the elbow method.

9.2 Theory

K-means clustering is a method of vector quantization that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean.

9.3 Code Implementation

```python
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

x = [4, 5, 10, 4, 3, 11, 14, 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x, y)
plt.show()

data = list(zip(x, y))
inertias = []

for i in range(1,11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)

plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()

kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

9.4 Output

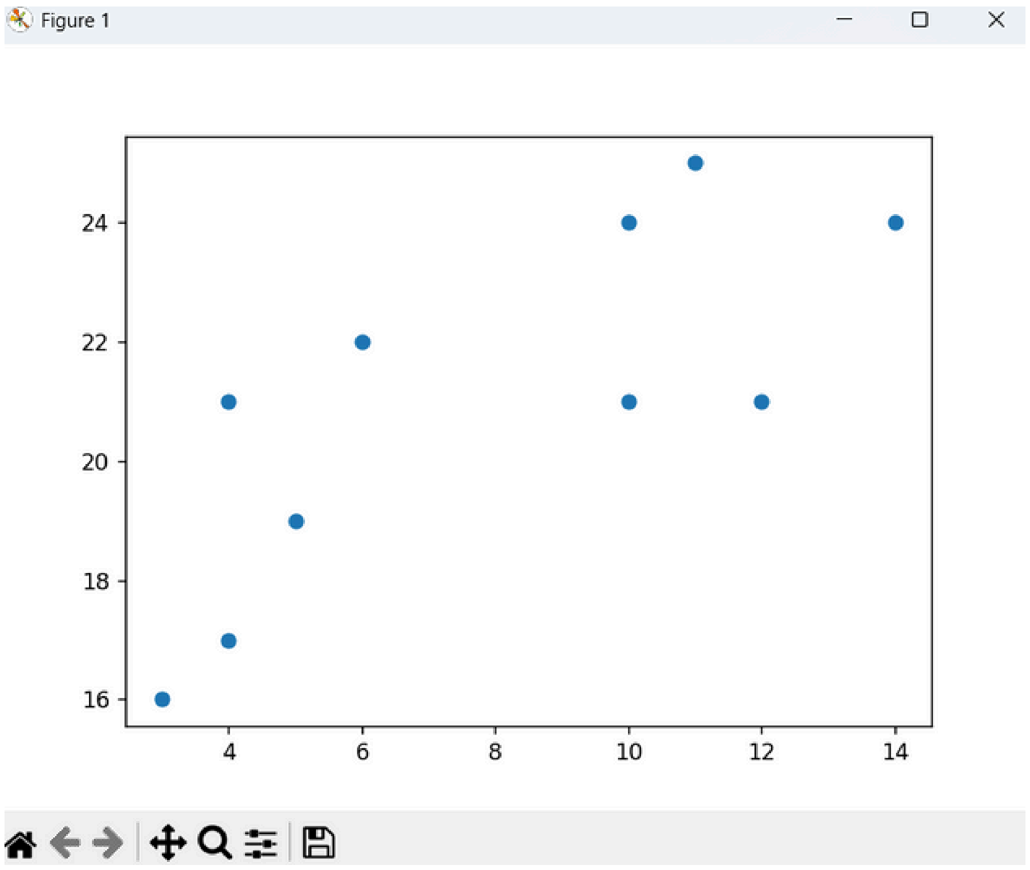


Figure 1
24
22
20
18
16
4
6
8
10
12
14

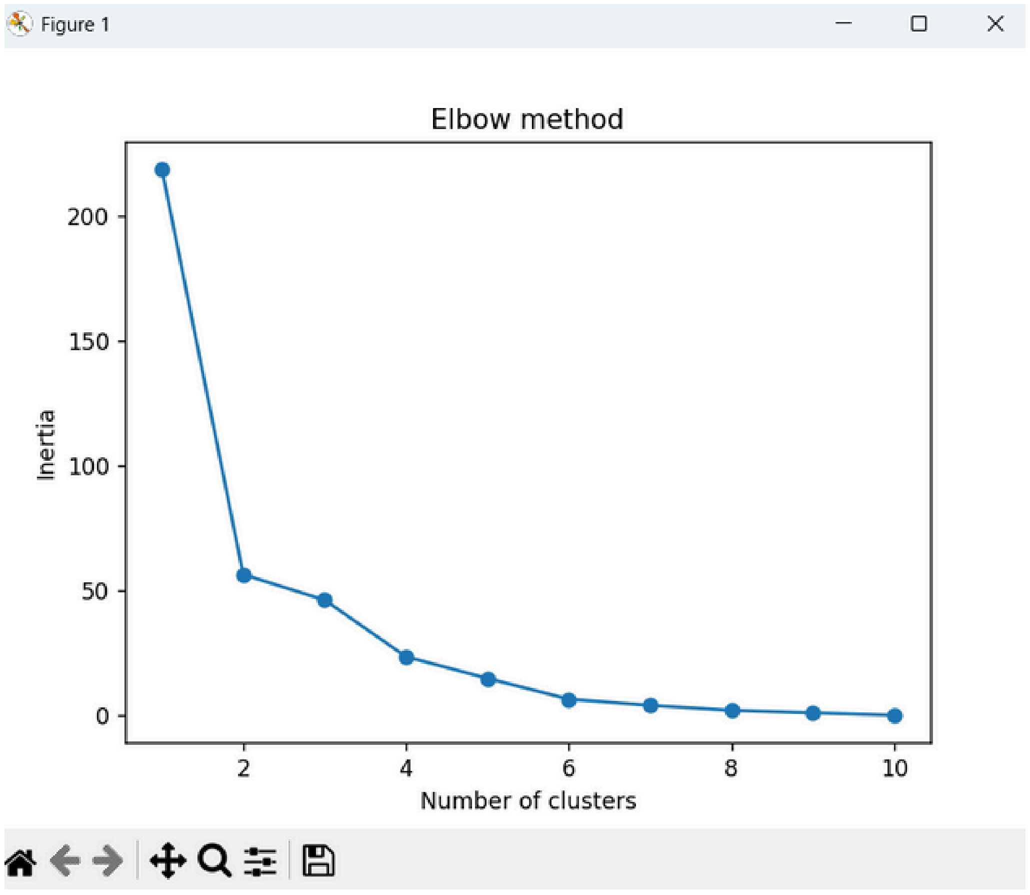


Figure 1
Elbow method
Inertia
200
150
100
50
0
2
4
6
8
10
Number of clusters

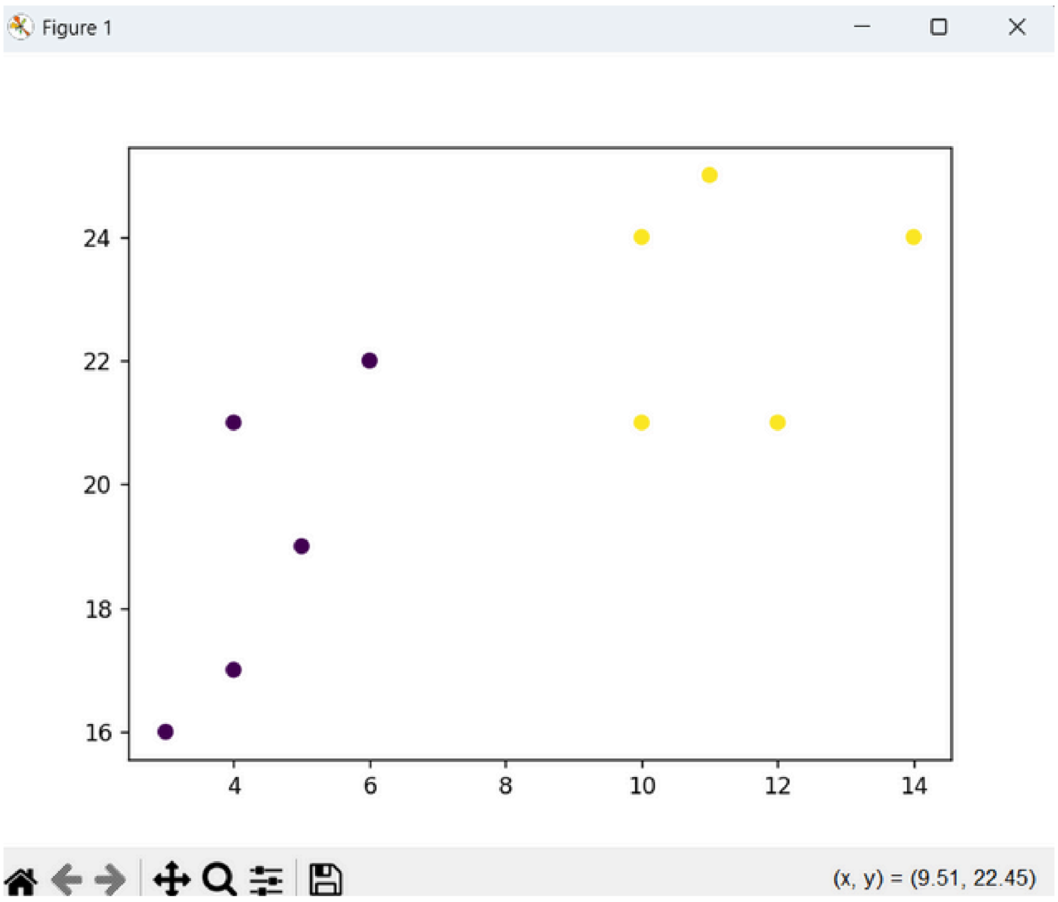


Figure 9.1: K-means Clustering Results

Chapter 10

DBSCAN Clustering

10.1 Objective

To implement DBSCAN clustering algorithm for customer segmentation.

10.2 Theory

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that can find arbitrarily shaped clusters and handle outliers.

10.3 Code Implementation

```python
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN

df = pd.read_csv('Wall_Customers.csv')
X_train = df[['Age', 'Annual Income (k$)', 'Spending Score (1-100)']]
clustering = DBSCAN(eps=12.5, min_samples=4).fit(X_train)
DBSCAN_dataset = X_train.copy()
DBSCAN_dataset.loc[:,'Cluster'] = clustering.labels_
DBSCAN_dataset.Cluster.value_counts().to_frame()

outliers = DBSCAN_dataset[DBSCAN_dataset['Cluster']==-1]

fig2, axes = plt.subplots(1,2,figsize=(12,5))

sns.scatterplot('Annual Income (k$)', 'Spending Score (1-100)',
data=DBSCAN_dataset[DBSCAN_dataset['Cluster']!=-1],
hue='Cluster', ax=axes[0], palette='Set2',
,              s=200)
legend='full',
sns.scatterplot('Age', 'Spending Score (1-100)',
data=DBSCAN_dataset[DBSCAN_dataset['Cluster']!=-1],
hue='Cluster', palette='Set2', ax=axes[1],
,              s=200)
legend='full',
axes[0].scatter(outliers['Annual Income (k$)'], outliers['Spending
    Score (1-100)'], s=10, label='outliers', c="k")
axes[1].scatter(outliers['Age'], outliers['Spending Score (1-100)'], s
    =10, label='outliers', c="k")
axes[0].legend()
axes[1].legend()
```

```

plt.setp(axes[0].get_legend().get_texts(),  fontsize='12')
plt.setp(axes[1].get_legend().get_texts(), fontsize='12')

plt.show()
```

10.4 Output

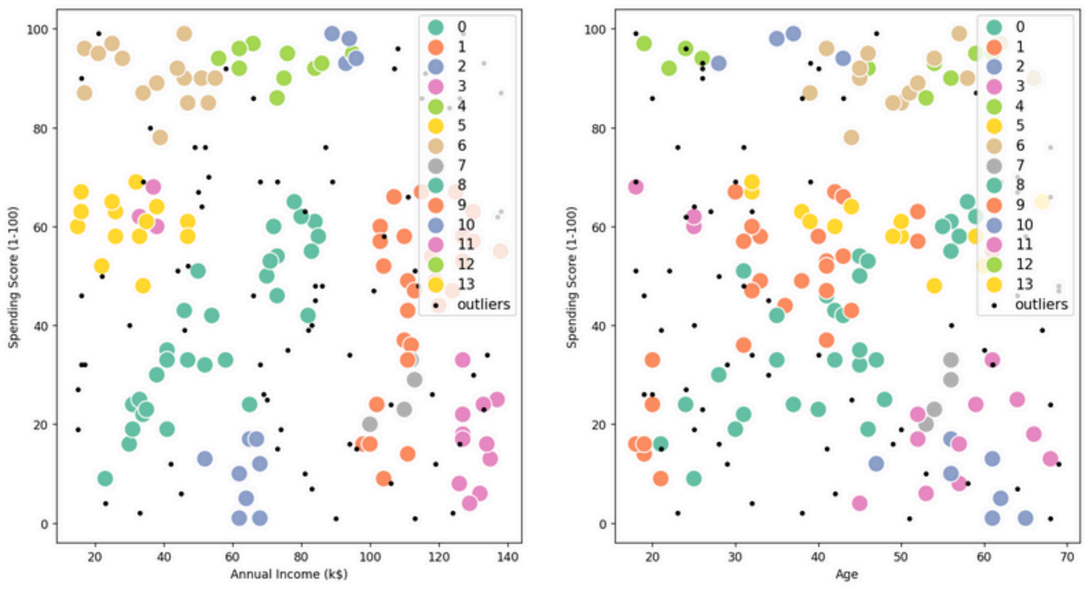


Figure 10.1: DBSCAN Clustering Results

Chapter 11

Boosting Techniques

11.1 Objective

To implement Gradient Boosting classifier and evaluate its performance.

11.2 Theory

Boosting is an ensemble technique that combines multiple weak learners to create a strong learner. Gradient Boosting builds the model in a stage-wise fashion.

11.3 Code Implementation

load_iris

```
from sklearn.datasets import
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target

# Split the data 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 a Gradient Boosting classifier
gb_clf = GradientBoostingClassifier(n_estimators=100, learning_rate
    =0.1, max_depth=3, random_state=42)

# Train the Gradient Boosting classifier
gb_clf.fit(X_train, y_train)

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

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Gradient Boosting Accuracy: {accuracy:.2f}')
```

11.4 Output

```
Gradient Boosting Accuracy: 1.00
```

Chapter 12

Ensemble Learning

12.1 Objective

To implement an ensemble classifier using Voting Classifier.

12.2 Theory

Ensemble learning combines multiple machine learning models to improve predictive performance and robustness.

12.3 Code Implementation

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target

# Split the data 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 individual classifiers
logistic_clf = LogisticRegression(random_state=42)
svm_clf = SVC(probability=True, random_state=42)
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Create an ensemble using a VotingClassifier
ensemble_clf = VotingClassifier(estimators=[('lr', logistic_clf), ('svm
    ', svm_clf), ('rf', rf_clf)], voting='soft')

# Train the ensemble classifier
ensemble_clf.fit(X_train, y_train)

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

# Evaluate the ensemble classifier
```

```
accuracy= accuracy_score(y_test,y_pred)
print(f'EnsembleAccuracy:{accuracy:.2f}')
```

12.4 Output

Ensemble Accuracy: 0.97

Chapter 13

Naive Bayes

13.1 Objective

To implement a Gaussian Naive Bayes classifier and evaluate its performance.

13.2 Theory

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong independence assumptions between the features.

13.3 Code Implementation

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix

# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target

# Split the data 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 a Gaussian Naive Bayes classifier
nb_classifier = GaussianNB()

# Train the classifier
nb_classifier.fit(X_train, y_train)

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

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display confusion matrix and classification report
print('\nConfusion Matrix:')
print(confusion_matrix(y_test, y_pred))

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

13.4 Output

```
Accuracy: 0.97

Confusion Matrix:
 [[10  0  0]
 [ 0  9  1]
 [ 0   0 10]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      0.9       0.95        10
           2       0.91      0         0.95        10
                             1.00
    accuracy                           0.97        30
   macro avg       0.97      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30
```

Chapter 14

Linear Discriminant Analysis (LDA)

14.1 Objective

To implement Linear Discriminant Analysis for dimensionality reduction and classification.

14.2 Theory

LDA is a method used in statistics, pattern recognition, and machine learning to find a linear combination of features that characterizes or separates two or more classes of objects or events.

14.3 Code Implementation

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix

# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target

# Split the data 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 a Linear Discriminant Analysis (LDA) model
lda = LinearDiscriminantAnalysis()

# Fit the model to the training data
lda.fit(X_train, y_train)

# Transform the data to the reduced-dimensional space
X_train_lda = lda.transform(X_train)
X_test_lda = lda.transform(X_test)

# Train a classifier (e.g., Logistic Regression) on the reduced-
    dimensional data
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(X_train_lda, y_train)

```

```python
# Make predictions on the test set
y_pred = classifier.predict(X_test_lda)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display confusion matrix and classification report
print('\nConfusion Matrix:')
print(confusion_matrix(y_test, y_pred))

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

14.4 Output

```
Accuracy: 0.97

Confusion Matrix:
 [[10  0  0]
 [ 0  9  1]
 [ 0  0 10]]

Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      0.9       0.95        10
           2       0.91      0         0.95        10
                             1.00
    accuracy                           0.97        30
   macro avg       0.97      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30
```

Chapter 15

Gradient Descent

15.1 Objective

To implement Gradient Descent algorithm for linear regression.

15.2 Theory

Gradient Descent is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient.

15.3 Code Implementation

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data for demonstration
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Add a bias term to X
X_b = np.c_[np.ones((100, 1)), X]

# Set the learning rate and number of iterations
learning_rate = 0.01
n_iterations = 1000

# Initialize random values for the parameters
theta = np.random.randn(2, 1)

# Gradient Descent
for iteration in range(n_iterations):
    gradients = 2 / 100 * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - learning_rate * gradients

# Print the final parameters
print("Final Parameters (theta):", theta)

# Plot the data and the linear regression line
plt.scatter(X, y)
plt.plot(X, X_b.dot(theta), color='red', label='Linear Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

15.4 Output

Final Parameters (theta): [[4.21509616] [2.77011339]]

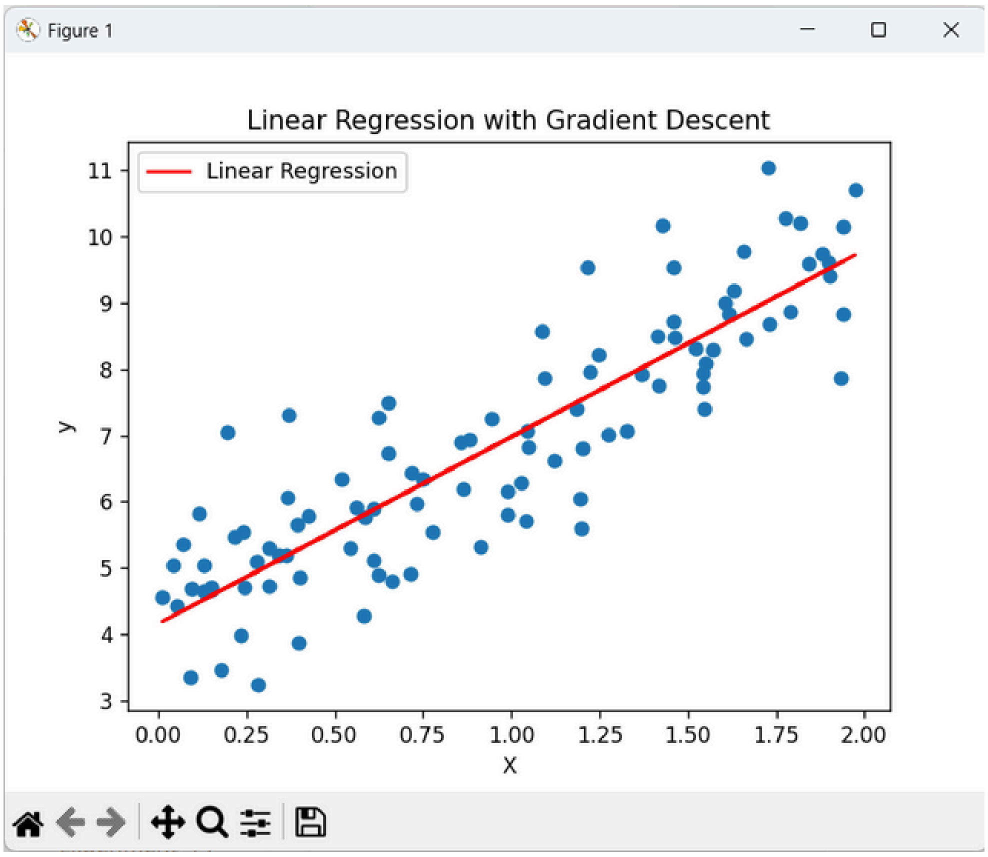


Figure 15.1: Gradient Descent Results

Chapter 16

Logistic Regression

16.1 Objective

To implement Logistic Regression for binary classification and visualize the decision boundary.

16.2 Theory

Logistic Regression is a statistical model that uses a logistic function to model a binary dependent variable.

16.3 Code Implementation

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix

# Generate synthetic data for binary classification
X, y = make_classification(n_samples=100, n_features=2, n_informative
    =2, n_redundant=0, n_clusters_per_class=1, random_state=42)

# Split the data 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 a Logistic Regression model
logreg = LogisticRegression()

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

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

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display confusion matrix and classification report
print('\nConfusion Matrix:')
print(confusion_matrix(y_test, y_pred))

print('\nClassification Report:')
print(classification_report(y_test, y_pred))

# Plot decision boundary
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolors='k', s
    =50)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Plot decision boundary
h = .02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max
    , h))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contour(xx, yy, Z, cmap=plt.cm.Paired)
plt.title('Logistic Regression Decision Boundary')
plt.show()
```

16.4 Output

```
Accuracy: 0.95

Confusion Matrix:
 [[10   1]
 [ 0    9]]

Classification Report:
             precision   recall   f1-score   support

           0     1.00     0.91      0.95        11
           1     0.9      1.00      0.95         9
                 0
    accuracy                        0.95        20
   macro avg     0.95     0.95      0.95        20
weighted avg     0.96     0.95      0.95        20
```

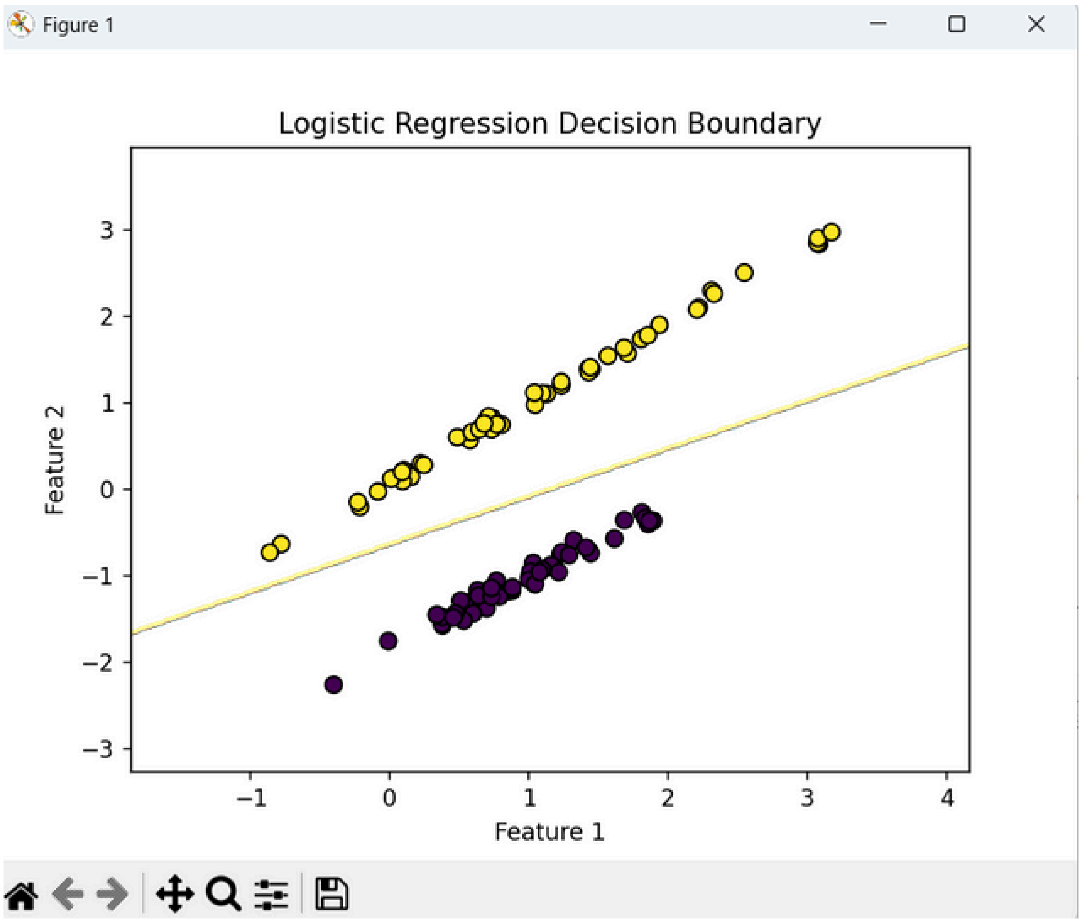


Figure 16.1: Logistic Regression Decision Boundary

Chapter 17

Hierarchical Agglomerative Clustering

17.1 Objective

To implement Hierarchical Agglomerative Clustering and visualize the dendrogram.

17.2 Theory

Hierarchical clustering is a method of cluster analysis which seeks to build a hierarchy of clusters.

17.3 Code Implementation

```python
import numpy as np
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# randomly chosen dataset
X = np.array([[1, 2], [1, 4], [1, 0],
    [4, 2], [4, 4], [4, 0]])

# Perform hierarchical clustering
Z = linkage(X, 'ward')

# Plot dendrogram
dendrogram(Z)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data point')
plt.ylabel('Distance')
plt.show()
```

17.4 Output

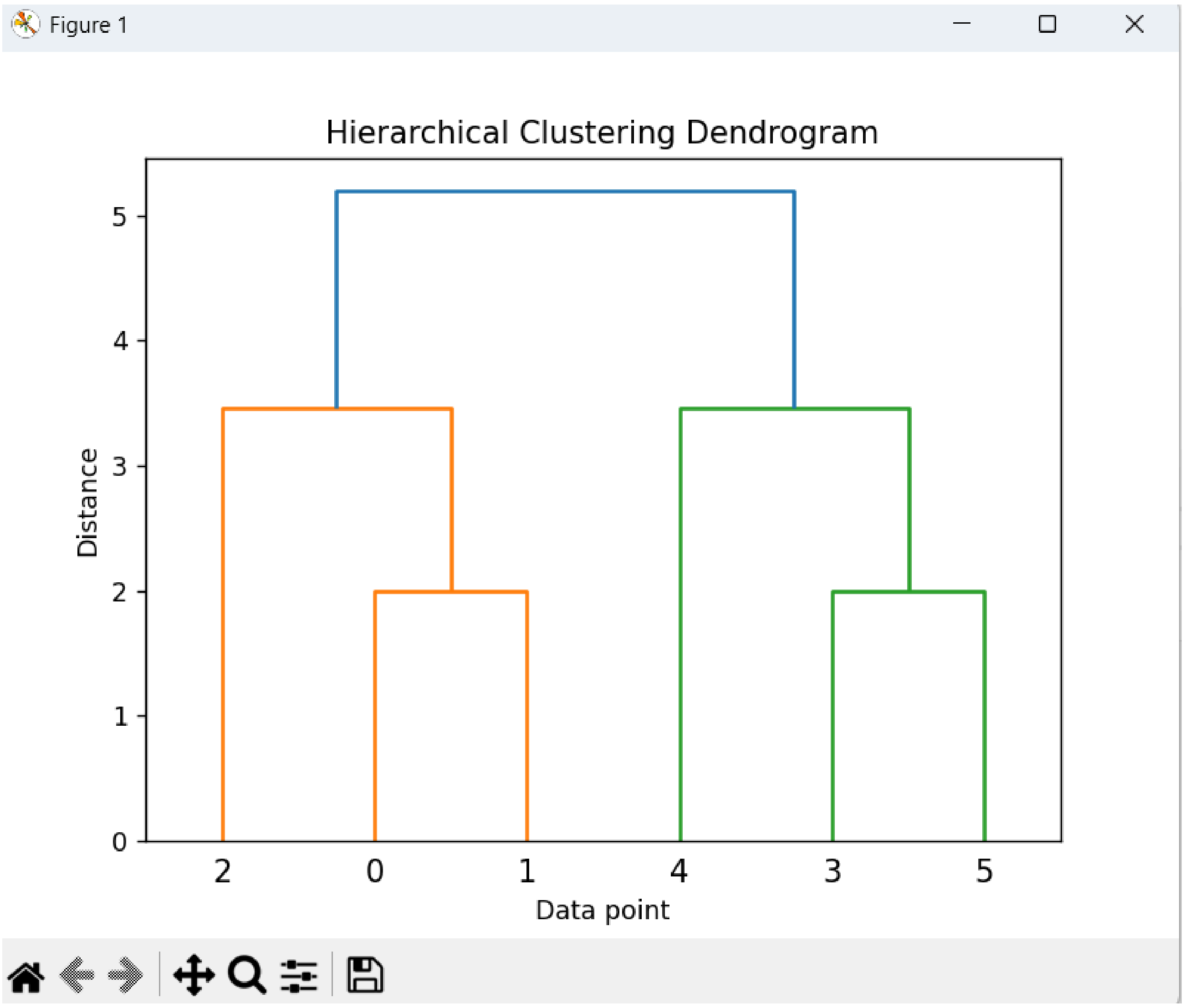


Figure 17.1: Hierarchical Clustering Dendrogram

Chapter 18

Appendix

18.1 AdditionalResources

- Scikit-learn documentation: https://scikit-learn.org
- NumPy documentation: https://numpy.org/doc/
- Matplotlib documentation: https://matplotlib.org/stable/contents.html

18.2 References

1. Introduction to Machine Learning with Python by Andreas C. Mü¨ller and Sarah Guido
2. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow by Auré´lien Gé´ron
3. Python Data Science Handbook by Jake VanderPlas

18.3 Github Repo Link: https://github.com/Mausam5055/Data-Science