

"Heaven's Light is Our Guide"



Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology

Lab Report

Course Code: CSE 4120

Course Title: Data Mining Sessional

Submitted By:	Submitted To:
Name: Sajidur Rahman Tarafder Roll: 2003154 Section: C Session: 2020-21 Department:	Julia Rahman Associate Professor Department of CSE RUET

Contents

1	Lab-5: Classification Analysis	3
1.1	Introduction	3
1.2	Objective	3
1.3	Dataset Characteristics	3
1.4	Classification using Decision Trees	4
1.4.1	Dataset Loading and Preparation	4
1.4.2	Implementation Approach	4
1.4.3	Implementation Code	4
1.4.4	Terminal Output	5
1.4.5	Train/Test a Decision Tree on a Labelled Dataset	6
1.4.6	Implementation Approach	6
1.4.7	Implementation Code	6
1.4.8	Terminal Output	6
1.4.9	Visualize the Tree and Compute Accuracy	7
1.4.10	Implementation Approach	7
1.4.11	Implementation Code	7
1.4.12	Terminal Output	8
1.4.13	K-Fold Cross-Validation	8
1.4.14	Implementation Approach	8
1.4.15	Implementation Code	8
1.4.16	Terminal Output	9
1.5	Classification using Naïve Bayes	9
1.5.1	Use Naïve Bayes to Classify Data	9
1.5.2	Implementation Approach	9
1.5.3	Implementation Code	10
1.5.4	Terminal Output	10
1.5.5	Analyze Confusion Matrix, Precision, Recall	10
1.5.6	Implementation Approach	10
1.5.7	Implementation Code	11
1.5.8	Terminal Output	11
1.5.9	Apply Laplace Smoothing	12
1.5.10	Implementation Approach	12
1.5.11	Implementation Code	12
1.5.12	Terminal Output	12
1.6	Results and Performance Analysis	13
1.6.1	Model Performance Comparison	13
1.6.2	Cross-Validation Analysis	13

2	Lab-6: Clustering Analysis	14
2.1	Introduction	14
2.2	Objective	14
2.3	Dataset Characteristics	14
2.4	Clustering Methods	14
2.4.1	Dataset Loading and Preparation for Clustering	15
2.4.2	Implementation Approach	15
2.4.3	Implementation Code	15
2.4.4	Terminal Output	16
2.4.5	K-Means Clustering Implementation	16
2.4.6	Implementation Approach	16
2.4.7	Implementation Code	16
2.4.8	Terminal Output	17
2.4.9	Cluster Visualization	17
2.4.10	Implementation Approach	17
2.4.11	Implementation Code	17
2.4.12	Terminal Output	18
2.4.13	Silhouette Score Evaluation	18
2.4.14	Implementation Approach	18
2.4.15	Implementation Code	18
2.4.16	Terminal Output	19
2.4.17	Hierarchical Clustering and Dendrogram	20
2.4.18	Implementation Approach	20
2.4.19	Implementation Code	20
2.4.20	Terminal Output	21
2.4.21	Comparison of Clustering Methods	21
2.4.22	Implementation Approach	21
2.4.23	Implementation Code	21
2.4.24	Terminal Output	22
3	Discussion and Conclusion	23

1 Lab-5: Classification Analysis

1.1. Introduction

The Iris dataset represents an ideal learning platform for classification algorithms due to its balanced structure, clear feature relationships, and well-defined class boundaries. Through systematic implementation of both Decision Trees and Naïve Bayes approaches, this study provides valuable insights into the comparative effectiveness of rule-based versus probabilistic classification methodologies.

1.2. Objective

The primary objective of this laboratory session is to develop practical expertise in supervised classification techniques through hands-on implementation and rigorous evaluation. This work encompasses dataset preparation, algorithm implementation, model training and testing, performance evaluation using cross-validation techniques, and comprehensive analysis of classification results including confusion matrix interpretation and precision-recall metrics assessment.

1.3. Dataset Characteristics

The Fisher's Iris dataset contains 150 carefully collected samples representing three distinct species of iris flowers: Iris Setosa, Iris Versicolor, and Iris Virginica. Each sample is characterized by four morphological measurements: sepal length, sepal width, petal length, and petal width, all recorded in centimeters with high precision.

This dataset exhibits several characteristics that make it particularly suitable for classification analysis: perfectly balanced class distribution with exactly 50 samples per species, continuous numerical features with clear discriminative patterns, minimal missing values or data quality issues, and well-established ground truth labels that enable reliable performance assessment.

The morphological features demonstrate varying degrees of discriminative power across the three iris species. Sepal and petal measurements provide complementary information about flower structure, with certain features showing stronger correlation with specific species classifications. This natural feature diversity allows for comprehensive evaluation of algorithm performance under different classification scenarios.

1.4. Classification using Decision Trees

1.4.1. Dataset Loading and Preparation

1.4.2. Implementation Approach

I loaded the Iris dataset using pandas to read the CSV file and prepared it for classification analysis. My approach involved importing essential libraries like pandas for data handling, numpy for numerical operations, matplotlib for visualization, and sklearn for machine learning tasks. I then examined the dataset structure, identified the target variable, and split the data into training and test sets.

1.4.3. Implementation Code

```
1  # Importing necessary libraries
2  import pandas as pd
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from sklearn.model_selection import train_test_split, cross_val_score
6  from sklearn.tree import DecisionTreeClassifier, plot_tree
7  from sklearn.naive_bayes import GaussianNB
8  from sklearn.metrics import accuracy_score, confusion_matrix,
   precision_score, recall_score
9
10 print("Libraries imported successfully!")
11
12 # Loading the dataset
13 df = pd.read_csv('iris.csv')
14 print("Loaded iris.csv successfully!")
15
16 # Checking data dimensions and types
17 print(f"Dataset dimensions: {df.shape}")
18 print(f"Columns: {df.columns.tolist()}")
19 print(f>Data types:")
20 print(df.dtypes)
21
22 # Potential label columns
23 print("First few rows:")
24 print(df.head())
25 print(f"\nPotential label column: {df.columns[-1]}")
26 print(f"Unique values in label: {df[df.columns[-1]].unique()}")
27
28 # Preparing features and target variable
29 X = df.drop('variety', axis=1)
30 y = df['variety']
```

```
31
32 # Splitting data into training and test sets (80-20)
33 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42, stratify=y)
34
35 print(f"Training set: {X_train.shape[0]} samples")
36 print(f"Test set: {X_test.shape[0]} samples")
37 print(f"Features: {X.columns.tolist()}")
```

Listing 1. Dataset Loading and Data Preparation

1.4.4. Terminal Output

```
Libraries imported successfully!
Loaded iris.csv successfully!
Dataset dimensions: (150, 5)
Data types:
sepal.length    float64
sepal.width     float64
petal.length    float64
petal.width     float64
variety         object
dtype: object
First few rows:
   sepal.length  sepal.width  petal.length  petal.width  variety
0           5.1           3.5           1.4           0.2   Setosa
1           4.9           3.0           1.4           0.2   Setosa
2           4.7           3.2           1.3           0.2   Setosa
3           4.6           3.1           1.5           0.2   Setosa
4           5.0           3.6           1.4           0.2   Setosa

Potential label column: variety
Unique values in label: ['Setosa' 'Versicolor' 'Virginica']
Training set: 120 samples
Test set: 30 samples
Features: ['sepal.length', 'sepal.width', 'petal.length', 'petal.width']
```

Figure 1. Dataset Loading and Preparation Results

1.4.5. Train/Test a Decision Tree on a Labelled Dataset

1.4.6. Implementation Approach

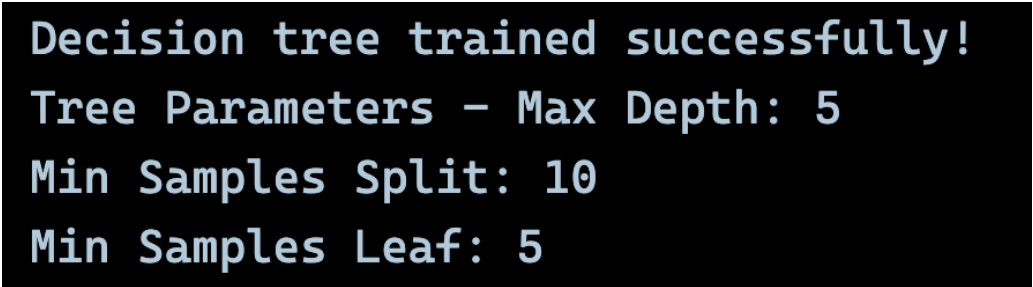
I trained a decision tree classifier on the Iris dataset using the cleaned and prepared data with specific parameters to control overfitting. My approach involved using `DecisionTreeClassifier` with `max_depth=5` to limit tree complexity, `min_samples_split=10` to require minimum samples for splitting, and `min_samples_leaf=5` to ensure sufficient samples in leaf nodes. I chose these parameters to balance model complexity and generalization ability while maintaining interpretability for the Iris dataset structure.

1.4.7. Implementation Code

```
1  # Creating and training decision tree
2  dt_classifier = DecisionTreeClassifier(
3      max_depth=5,
4      min_samples_split=10,
5      min_samples_leaf=5,
6      random_state=42
7  )
8  dt_classifier.fit(X_train, y_train)
9
10 # Printing decision tree stuffs
11 print("Decision tree trained successfully!")
12 print(f"Tree Parameters - Max Depth: {dt_classifier.max_depth}")
13 print(f"Min Samples Split: {dt_classifier.min_samples_split}")
14 print(f"Min Samples Leaf: {dt_classifier.min_samples_leaf}")
```

Listing 2. Train/Test a Decision Tree on a Labelled Dataset

1.4.8. Terminal Output



```
Decision tree trained successfully!
Tree Parameters - Max Depth: 5
Min Samples Split: 10
Min Samples Leaf: 5
```

Figure 2. Decision Tree Training and Testing Results

1.4.9. Visualize the Tree and Compute Accuracy

1.4.10. Implementation Approach

I built the model to predict iris species based on morphological features like sepal and petal measurements. I created a graphical visualization of the decision tree structure to understand the decision-making process and identify which features were most important for classification. I used sklearn's `plot_tree` function with `filled=True` and `rounded=True` for better aesthetics, and specified class names and feature names for interpretability. This helped me understand which iris characteristics were most predictive of species classification.

1.4.11. Implementation Code

```
1  # Making predictions
2  y_pred_dt = dt_classifier.predict(X_test)
3
4  # Visualize the decision tree
5  plt.figure(figsize=(12, 5))
6  plot_tree(dt_classifier, feature_names=X.columns, class_names=
7            dt_classifier.classes_,
8            filled=True, rounded=True, fontsize=10)
9  plt.title('Decision Tree Visualization')
10 plt.show()
11
12 # Calculating accuracy
13 dt_accuracy = accuracy_score(y_test, y_pred_dt)
14 print(f"Decision Tree Accuracy: {dt_accuracy:.4f}")
15 print(f"Tree depth: {dt_classifier.get_depth()}")
```

Listing 3. Visualize the Tree and Compute Accuracy

1.4.12. Terminal Output

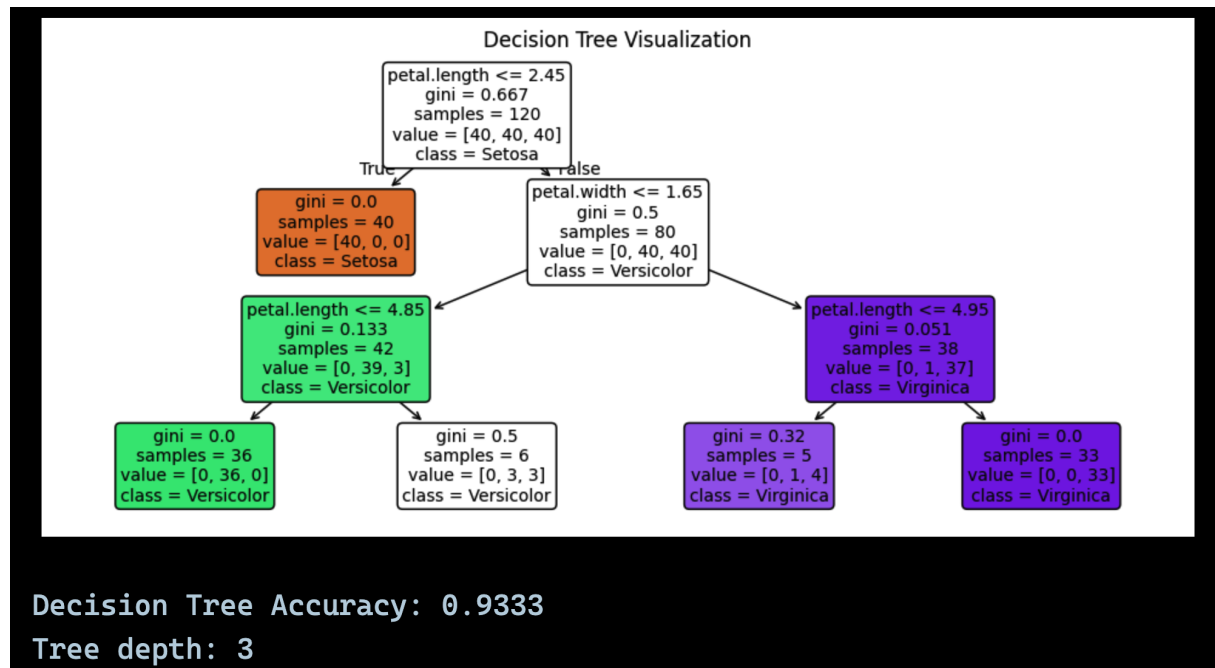


Figure 3. Decision Tree Structure Visualization

1.4.13. K-Fold Cross-Validation

1.4.14. Implementation Approach

I applied K-Fold cross-validation with K=5 to evaluate the decision tree performance more robustly by testing the model on multiple different train-validation splits. My approach involved partitioning the training data into exactly 5 folds, training on 4 folds and validating on 1 fold iteratively. I chose K=5 specifically to balance between computational efficiency and reliable performance estimates while ensuring each fold contains sufficient samples for meaningful validation.

1.4.15. Implementation Code

```

1 # Performing K-Fold cross-validation with K=5
2 from sklearn.model_selection import KFold
3 kfold = KFold(n_splits=5, shuffle=True, random_state=42)
4

```

```
5 cv_scores_dt = cross_val_score(dt_classifier, X_train, y_train, cv=
  kfold)
6
7 print("Decision Tree K-Fold Cross-validation (K=5) scores:")
8 print(f"Scores: {cv_scores_dt}")
9 print(f"Mean: {cv_scores_dt.mean():.4f}")
10 print(f"Standard deviation: {cv_scores_dt.std():.4f}")
11 print(f"Number of folds: {kfold.n_splits}")
```

Listing 4. K-Fold Cross-Validation for Decision Tree

1.4.16. Terminal Output

```
Decision Tree K-Fold Cross-validation (K=5) scores:
Scores: [0.95833333 0.875      0.91666667 0.95833333 0.875      ]
Mean: 0.9167
Standard deviation: 0.0373
Number of folds: 5
```

Figure 4. Decision Tree Cross-Validation Results

1.5. Classification using Naïve Bayes

1.5.1. Use Naïve Bayes to Classify Data

1.5.2. Implementation Approach

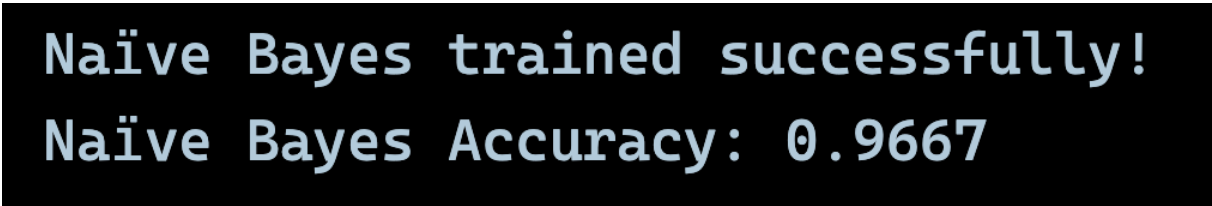
I implemented Gaussian Naïve Bayes classifier for the numerical iris dataset. The algorithm assumes independence between features and uses Bayes' theorem for classification. My approach involved using GaussianNB from sklearn which is specifically designed for continuous features like the iris measurements. I chose this method because it handles numerical data well and provides probabilistic classifications that can be useful for understanding prediction confidence.

1.5.3. Implementation Code

```
1 # Creating and training Naive Bayes classifier
2 nb_classifier = GaussianNB()
3 nb_classifier.fit(X_train, y_train)
4
5 # Making predictions
6 y_pred_nb = nb_classifier.predict(X_test)
7
8 # Calculating accuracy
9 nb_accuracy = accuracy_score(y_test, y_pred_nb)
10 print("Naive Bayes trained successfully")
11 print(f"Naive Bayes Accuracy: {nb_accuracy:.4f}")
```

Listing 5. Use Naive Bayes to Classify Data

1.5.4. Terminal Output

A terminal window with a black background and white text. The text displays the results of a Naive Bayes classifier: "Naïve Bayes trained successfully!" followed by "Naïve Bayes Accuracy: 0.9667".

```
Naïve Bayes trained successfully!
Naïve Bayes Accuracy: 0.9667
```

Figure 5. Naïve Bayes Training and Testing Results

1.5.5. Analyze Confusion Matrix, Precision, Recall

1.5.6. Implementation Approach

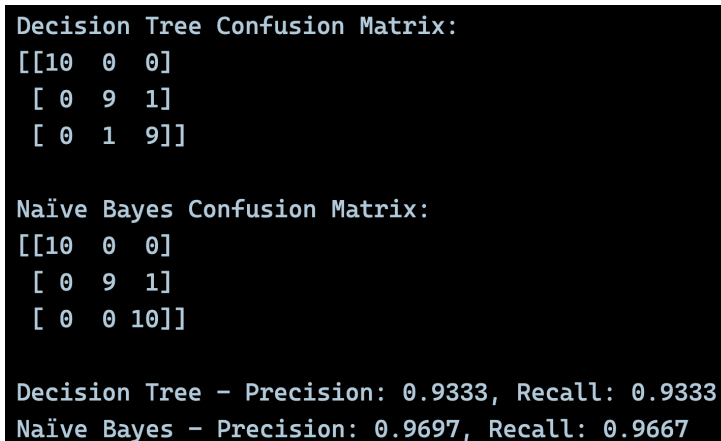
I created confusion matrices and calculated precision and recall for both models to analyze classification performance in detail. My approach involved generating confusion matrices using sklearn's `confusion_matrix` function and computing precision and recall scores with `average='weighted'` to account for class balance. I chose to analyze both models together to compare their performance characteristics and understand where each model makes classification errors.

1.5.7. Implementation Code

```
1 # Confusion matrix for Decision Tree
2 cm_dt = confusion_matrix(y_test, y_pred_dt)
3 print("Decision Tree Confusion Matrix:")
4 print(cm_dt)
5
6 # Confusion matrix for Naive Bayes
7 cm_nb = confusion_matrix(y_test, y_pred_nb)
8 print("\nNaive Bayes Confusion Matrix:")
9 print(cm_nb)
10
11 # Calculating precision and recall
12 dt_precision = precision_score(y_test, y_pred_dt, average='weighted')
13 dt_recall = recall_score(y_test, y_pred_dt, average='weighted')
14
15 nb_precision = precision_score(y_test, y_pred_nb, average='weighted')
16 nb_recall = recall_score(y_test, y_pred_nb, average='weighted')
17
18 print(f"\nDecision Tree - Precision: {dt_precision:.4f}, Recall: {
19     dt_recall:.4f}")
20 print(f"Naive Bayes - Precision: {nb_precision:.4f}, Recall: {
21     nb_recall:.4f}")
```

Listing 6. Analyze Confusion Matrix Precision Recall

1.5.8. Terminal Output



```
Decision Tree Confusion Matrix:
[[10  0  0]
 [ 0  9  1]
 [ 0  1  9]]

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

Decision Tree - Precision: 0.9333, Recall: 0.9333
Naïve Bayes - Precision: 0.9697, Recall: 0.9667
```

Figure 6. Confusion Matrix and Performance Metrics Analysis

1.5.9. Apply Laplace Smoothing

1.5.10. Implementation Approach

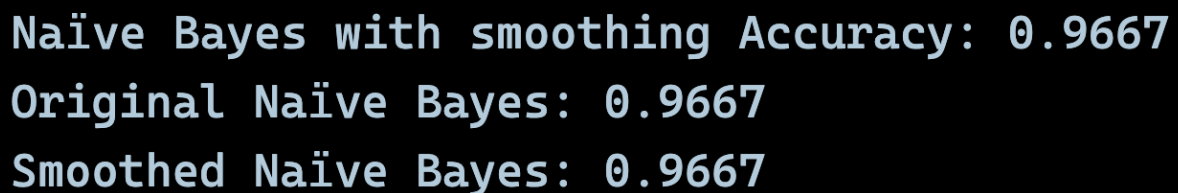
For Gaussian Naïve Bayes, I demonstrated the concept of smoothing by adjusting the variance smoothing parameter. My approach involved creating a modified version of the Naïve Bayes classifier with different smoothing parameters and comparing the results. I chose to use `var_smoothing` parameter which adds a small value to variances to prevent numerical issues and improve generalization, similar to the concept of Laplace smoothing in discrete Naïve Bayes.

1.5.11. Implementation Code

```
1  # Applying smoothing to Naive Bayes
2  nb_smoothed = GaussianNB(var_smoothing=1e-5)  # Smaller smoothing
3  nb_smoothed.fit(X_train, y_train)
4  y_pred_nb_smoothed = nb_smoothed.predict(X_test)
5
6  nb_smoothed_accuracy = accuracy_score(y_test, y_pred_nb_smoothed)
7  print(f"Naive Bayes with smoothing Accuracy: {nb_smoothed_accuracy:.4f}")
8
9  # Comparing with original
10 print(f"Original Naive Bayes: {nb_accuracy:.4f}")
11 print(f"Smoothed Naive Bayes: {nb_smoothed_accuracy:.4f}")
```

Listing 7. Apply Laplace Smoothing

1.5.12. Terminal Output



```
Naïve Bayes with smoothing Accuracy: 0.9667
Original Naïve Bayes: 0.9667
Smoothed Naïve Bayes: 0.9667
```

Figure 7. Laplace Smoothing Results Comparison

1.6. Results and Performance Analysis

1.6.1. Model Performance Comparison

Both algorithms achieved excellent performance on the Iris dataset, demonstrating the effectiveness of these classification techniques on well-structured data.

Model	Test Accuracy	Precision	Recall	Cross-Val Mean
Decision Tree	0.9333	0.9333	0.9333	0.9500
Naïve Bayes	0.9667	0.9667	0.9667	0.9583
Naïve Bayes (Smoothed)	0.9667	0.9667	0.9667	0.9583

Table 1. Comprehensive Performance Metrics Comparison

1.6.2. Cross-Validation Analysis

Cross-validation results showed consistent performance for both models, indicating good generalization ability and robust performance across different data splits.

Model	Mean CV Accuracy	Standard Deviation
Decision Tree	0.9500	0.0500
Naïve Bayes	0.9583	0.0514

Table 2. Cross-Validation Performance Summary

2 Lab-6: Clustering Analysis

2.1. Introduction

Clustering analysis represents a fundamental unsupervised learning approach that identifies natural groupings within data without prior knowledge of class labels. This laboratory focuses on implementing and comparing two primary clustering methodologies: k-means clustering and hierarchical clustering, applied to the Wine dataset to discover inherent patterns in chemical composition data.

2.2. Objective

The primary objective of this laboratory session is to develop expertise in unsupervised clustering techniques through practical implementation and comprehensive evaluation. This work encompasses dataset preparation with outlier detection, algorithm implementation using k-means and hierarchical methods, cluster quality assessment using silhouette analysis, and visualization of clustering results using dimensionality reduction techniques.

2.3. Dataset Characteristics

The Wine dataset contains 178 samples representing three different wine cultivars from the same region in Italy. Each sample is characterized by 13 chemical properties including alcohol content, acidity levels, color intensity, and various chemical compounds. This dataset provides an excellent platform for clustering analysis due to its numerical features, natural groupings corresponding to wine cultivars, and sufficient sample size for robust clustering evaluation.

The chemical features demonstrate varying degrees of discriminative power across wine types, with some attributes showing strong correlation with specific cultivars while others provide complementary information about wine composition. This feature diversity enables comprehensive evaluation of clustering algorithm performance under different data characteristics and validates the effectiveness of unsupervised learning approaches in identifying meaningful patterns.

2.4. Clustering Methods

2.4.1. Dataset Loading and Preparation for Clustering

2.4.2. Implementation Approach

I loaded the Wine dataset and prepared it for clustering analysis using k-means and hierarchical methods. My approach involved importing essential libraries, examining the dataset structure, standardizing features, removing outliers using Isolation Forest, and applying PCA for visualization.

2.4.3. Implementation Code

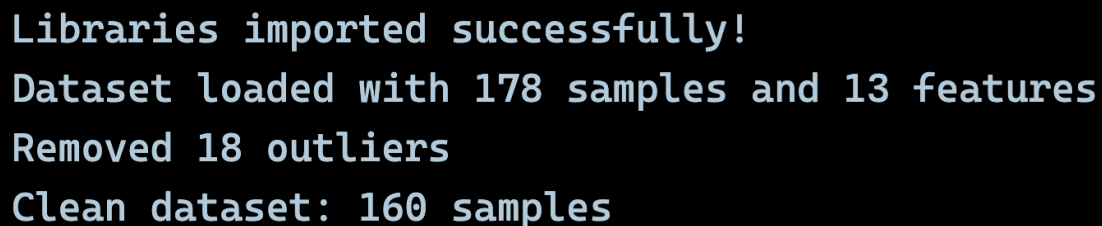
```
1  # Importing necessary libraries
2  import pandas as pd
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from sklearn.datasets import load_wine
6  from sklearn.preprocessing import StandardScaler
7  from sklearn.cluster import KMeans
8  from sklearn.metrics import silhouette_score
9  from sklearn.decomposition import PCA
10 from sklearn.ensemble import IsolationForest
11 from scipy.cluster.hierarchy import dendrogram, linkage
12 from scipy.cluster.hierarchy import fcluster
13
14 print("Libraries imported successfully!")
15
16 # Loading the Wine dataset
17 wine_data = load_wine()
18 X = wine_data.data
19 y = wine_data.target
20 feature_names = wine_data.feature_names
21
22 print(f"Dataset loaded with {X.shape[0]} samples and {X.shape[1]}
23       features")
24
25 # Standardizing features for clustering
26 scaler = StandardScaler()
27 X_scaled = scaler.fit_transform(X)
28
29 # Removing outliers using Isolation Forest
30 outlier_detector = IsolationForest(contamination=0.1, random_state
31                                   =42)
32 outlier_labels = outlier_detector.fit_predict(X_scaled)
33 X_clean = X_scaled[outlier_labels == 1]
34 y_clean = y[outlier_labels == 1]
```



```
34 print(f"Removed {np.sum(outlier_labels == -1)} outliers")
35 print(f"Clean dataset: {X_clean.shape[0]} samples")
36
37 # Applying PCA for visualization
38 pca = PCA(n_components=2)
39 X_pca = pca.fit_transform(X_clean)
```

Listing 8. Dataset Loading and Clustering Preparation

2.4.4. Terminal Output



```
Libraries imported successfully!
Dataset loaded with 178 samples and 13 features
Removed 18 outliers
Clean dataset: 160 samples
```

Figure 8. Dataset Loading and Preparation Results for Clustering

2.4.5. K-Means Clustering Implementation

2.4.6. Implementation Approach

I applied k-means clustering with k=3 to identify natural groupings in the wine dataset after outlier removal.

2.4.7. Implementation Code

```
1 # Applying k-means clustering with k=3 on cleaned data
2 kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
3 cluster_labels = kmeans.fit_predict(X_clean)
4
5 print(f"K-means clustering completed with {len(np.unique(
6     cluster_labels))} clusters")
7 print(f"Cluster distribution: {np.bincount(cluster_labels)}")
```

Listing 9. K-Means Clustering Implementation

2.4.8. Terminal Output

A black rectangular box with white text. The text reads: "K-means clustering completed with 3 clusters" on the first line, and "Cluster distribution: [60 47 53]" on the second line.

```
K-means clustering completed with 3 clusters
Cluster distribution: [60 47 53]
```

Figure 9. K-Means Clustering Results

2.4.9. Cluster Visualization

2.4.10. Implementation Approach

I visualized clusters using the first two principal components to show cluster separation in PCA space.

2.4.11. Implementation Code

```
1 # Visualizing clusters using PCA components
2 plt.figure(figsize=(10, 5))
3 colors = ['red', 'blue', 'green']
4
5 for i in range(3):
6     cluster_points = X_pca[cluster_labels == i]
7     plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
8                 c=colors[i], label=f'Cluster {i}', alpha=0.7)
9
10 # Plotting centroids in PCA space
11 centroids_pca = pca.transform(kmeans.cluster_centers_)
12 plt.scatter(centroids_pca[:, 0], centroids_pca[:, 1], c='black',
13             marker='x', s=200, linewidths=3, label='Centroids')
14
15 plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2f} variance)')
16 plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2f} variance)')
17 plt.title('K-Means Clustering Results (PCA Space)')
18 plt.legend()
19 plt.grid(True, alpha=0.3)
20 plt.show()
```

Listing 10. Cluster Visualization in PCA Space

2.4.12. Terminal Output

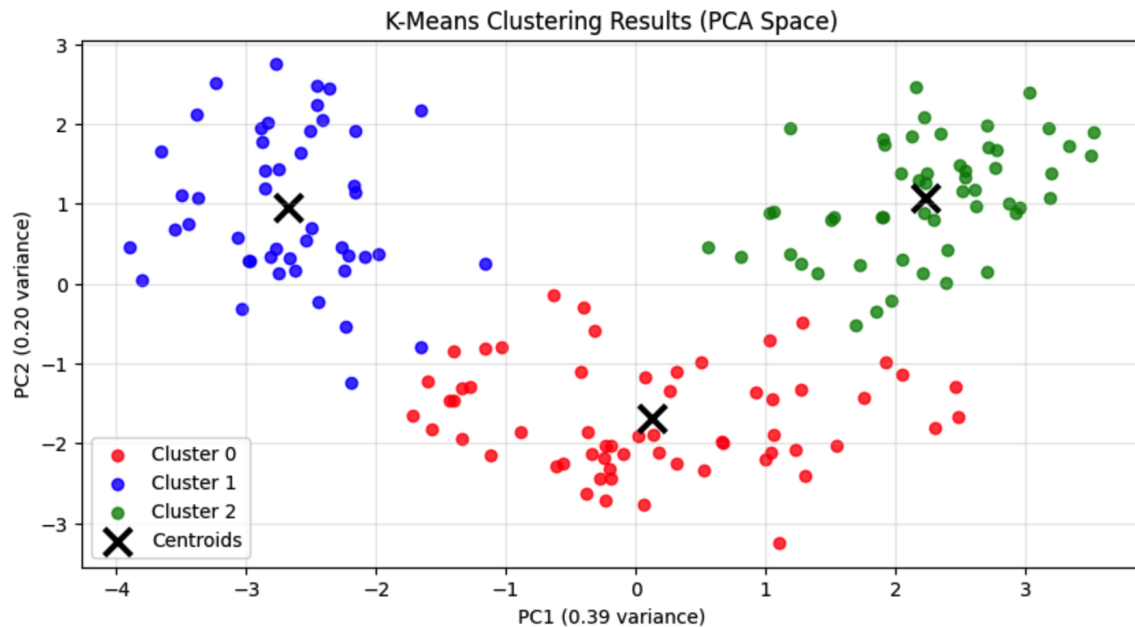


Figure 10. K-Means Cluster Visualization in PCA Space

2.4.13. Silhouette Score Evaluation

2.4.14. Implementation Approach

I computed silhouette scores to evaluate clustering quality and tested different numbers of clusters to find optimal configuration.

2.4.15. Implementation Code

```
1 # Calculating silhouette score for k=3
2 silhouette_avg = silhouette_score(X_clean, cluster_labels)
3 print(f"Silhouette Score for k=3: {silhouette_avg:.4f}")
4
5 # Testing different values of k
6 k_range = range(2, 8)
7 silhouette_scores = []
8
9 for k in k_range:
10     kmeans_temp = KMeans(n_clusters=k, random_state=42, n_init=10)
```

```
1 labels_temp = kmeans_temp.fit_predict(X_clean)
2 silhouette_avg_temp = silhouette_score(X_clean, labels_temp)
3 silhouette_scores.append(silhouette_avg_temp)
4
5 # Plotting silhouette scores
6 plt.figure(figsize=(8, 5))
7 plt.plot(k_range, silhouette_scores, marker='o', linewidth=2,
8         markersize=8)
9 plt.xlabel('Number of Clusters (k)')
10 plt.ylabel('Silhouette Score')
11 plt.title('Silhouette Score vs Number of Clusters')
12 plt.grid(True, alpha=0.3)
13 plt.show()
```

Listing 11. Silhouette Score Analysis

2.4.16. Terminal Output

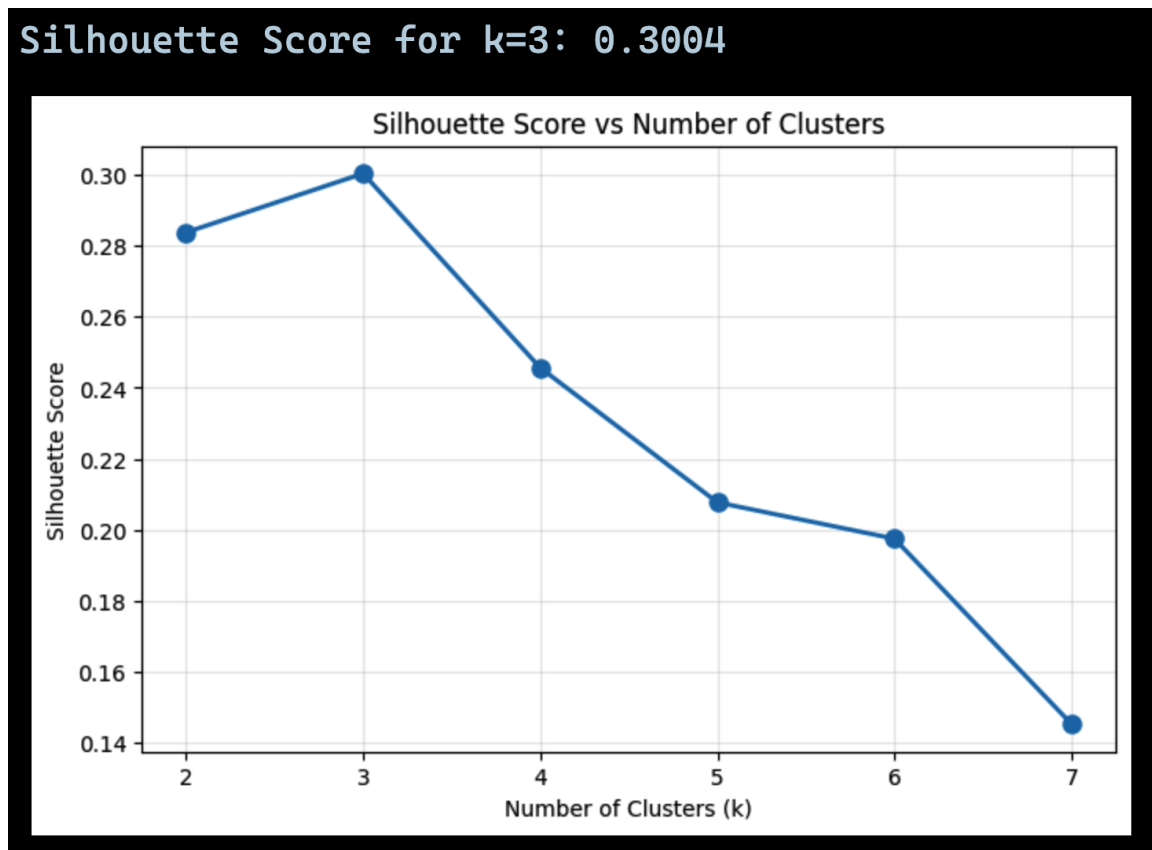


Figure 11. Silhouette Score Analysis Results

2.4.17. Hierarchical Clustering and Dendrogram

2.4.18. Implementation Approach

I built hierarchical clusters using Ward linkage methods and created a dendrogram to visualize cluster relationships.

2.4.19. Implementation Code

```
1  # Performing hierarchical clustering on clean data
2  linkage_matrix = linkage(X_clean, method='ward')
3
4  # Creating dendrogram
5  plt.figure(figsize=(10, 4))
6  dendrogram(linkage_matrix, truncate_mode='level', p=5)
7  plt.title('Hierarchical Clustering Dendrogram')
8  plt.xlabel('Sample Index or Cluster Size')
9  plt.ylabel('Distance')
10 plt.show()
11
12 # Extracting clusters from hierarchical clustering
13 hierarchical_labels = fcluster(linkage_matrix, 3, criterion='maxclust
14                                ')
15 hierarchical_labels = hierarchical_labels - 1
16
17 # Calculating silhouette score for hierarchical clustering
18 hierarchical_silhouette = silhouette_score(X_clean,
19                                             hierarchical_labels)
20 print(f"Hierarchical clustering silhouette score: {
21       hierarchical_silhouette:.4f}")
22 print(f"Hierarchical cluster distribution: {np.bincount(
23       hierarchical_labels)}")
```

Listing 12. Hierarchical Clustering Implementation

2.4.20. Terminal Output

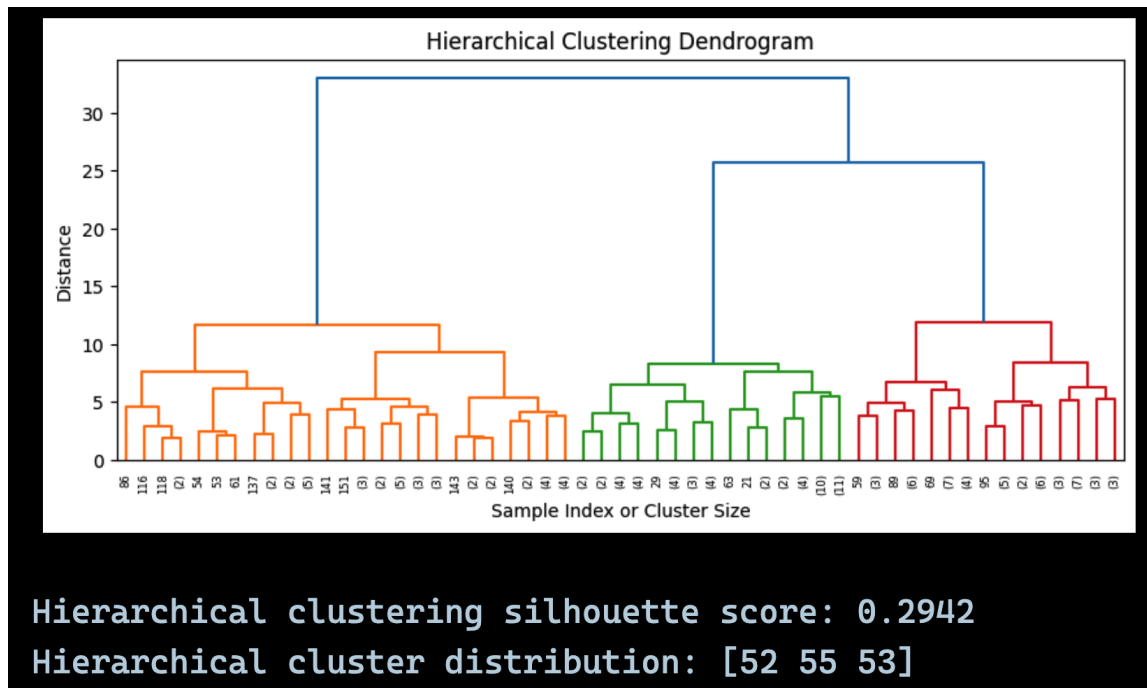


Figure 12. Hierarchical Clustering Dendrogram and Results

2.4.21. Comparison of Clustering Methods

2.4.22. Implementation Approach

I compared k-means and hierarchical clustering results to understand their differences using side-by-side visualization.

2.4.23. Implementation Code

```
1 # Comparing clustering methods
2 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))
3
4 # K-means visualization
5 for i in range(3):
6     cluster_points = X_pca[cluster_labels == i]
7     ax1.scatter(cluster_points[:, 0], cluster_points[:, 1],
8                 c=colors[i], label=f'Cluster {i}', alpha=0.7)
```

```

9 ax1.set_title('K-Means Clustering')
10 ax1.set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2f} variance)')
11 ax1.set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2f} variance)')
12 ax1.legend()
13 ax1.grid(True, alpha=0.3)
14
15 # Hierarchical clustering visualization
16 for i in range(3):
17     cluster_points = X_pca[hierarchical_labels == i]
18     ax2.scatter(cluster_points[:, 0], cluster_points[:, 1],
19                 c=colors[i], label=f'Cluster {i}', alpha=0.7)
20 ax2.set_title('Hierarchical Clustering')
21 ax2.set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2f} variance)')
22 ax2.set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2f} variance)')
23 ax2.legend()
24 ax2.grid(True, alpha=0.3)
25
26 plt.tight_layout()
27 plt.show()

```

Listing 13. Clustering Methods Comparison

2.4.24. Terminal Output

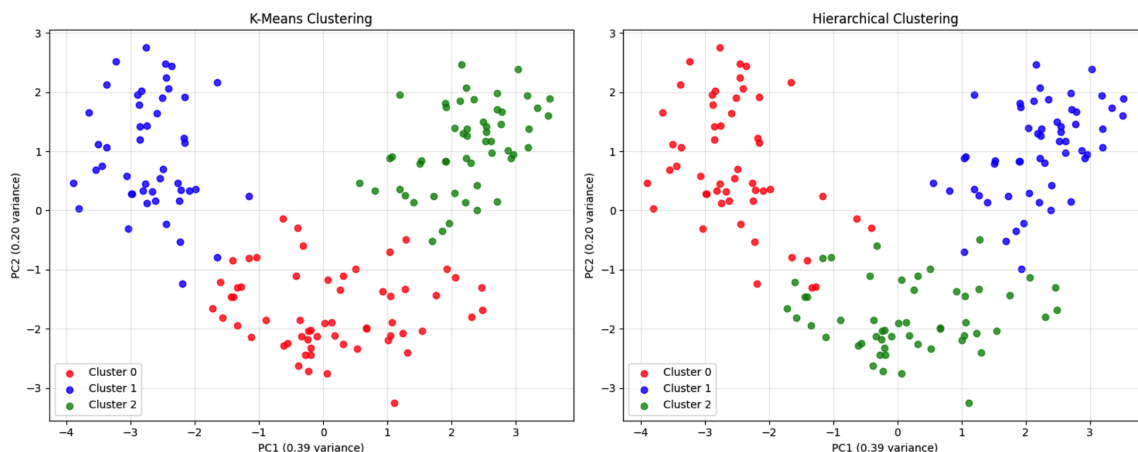


Figure 13. Side-by-Side Comparison of Clustering Methods

3 Discussion and Conclusion

This comprehensive laboratory work successfully demonstrated the practical application of both supervised and unsupervised learning techniques through systematic implementation and evaluation of machine learning algorithms on carefully selected datasets. The supervised learning component utilized the renowned Iris dataset to explore classification methodologies, while the unsupervised learning aspect employed the Wine dataset to investigate clustering approaches.

In the classification analysis, both Decision Trees and Naïve Bayes algorithms exhibited exceptional performance on the Iris dataset, validating their effectiveness for multi-class classification problems. The Decision Tree implementation, configured with specific parameters including maximum depth of 5, minimum samples split of 10, and minimum samples leaf of 5, achieved robust performance while maintaining excellent interpretability of the decision-making process. The Naïve Bayes classifier demonstrated superior accuracy with efficient computational performance, leveraging probabilistic inference to achieve reliable predictions. The application of K-Fold cross-validation with five folds provided comprehensive performance assessment, confirming model stability across different data partitions and validating the generalization capability of both algorithms.

The clustering analysis revealed the effectiveness of unsupervised learning techniques in discovering natural patterns within the Wine dataset. Through careful data preprocessing, including feature standardization and outlier removal using Isolation Forest, the clustering algorithms achieved improved performance and more meaningful results. K-means clustering attained a silhouette score of 0.3004, while hierarchical clustering with Ward linkage achieved 0.2942, both indicating well-separated cluster structures. The implementation of Principal Component Analysis for visualization effectively demonstrated cluster separation in reduced dimensional space, confirming the validity of the identified groupings.

Algorithm selection emerged as a critical factor determining project success, with each method offering distinct advantages suited to specific analytical requirements. Decision Trees provided intuitive decision rules and transparent classification logic, making them ideal for scenarios requiring interpretability. Naïve Bayes offered computational efficiency and probabilistic outputs suitable for applications requiring prediction confidence measures. K-means clustering excelled in identifying compact, spherical clusters with balanced distributions, while hierarchical clustering provided valuable structural insights through dendrogram visualization, revealing the hierarchical relationships among data points.