# Data and Artificial Intelligence

# Cyber Shujaa Program

## Week 8 Assignment

## Supervised machine learning Classification models

**Student Name:** John Brown Ouma

**Student ID:** CS-DA01-25030

# Table of content

## Table of Contents

## Introduction

This project explores supervised machine learning classification model using the Wine dataset from scikit-learn. The goal is to build, evaluate, and compare six different classification models: Logistic regression, Decision Tree, Random Forest, K-nearest Neighbors (KNN), Naive bayes, and Support Vector Machine (SVM). The performance of each model is assessed using accuracy scores, classification reports, and confusion matrices.

## Tasks Completed

### 1. Data loading and Exploration

First, we import the necessary required libraries and load the dataset.

```
1. Data Loading

# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from scipy import stats

# Load the dataset
wine = load_wine()
X = pd.DataFrame(wine.data, columns=wine.feature_names)
y = pd.Series(wine.target, name='target')
```

*Figure 1: Showing code for libraries and dataset loading*

### Purpose

Load the wine dataset from sklearn and convert features to a pandas dataframe('x') and target to a pandas series ('y').

### 2. Data Wrangling

We clean, organize and transform raw data into structured and usable format for analysis or machine learning.

We check if there is any missing values,

```
## Check for missing values
print("Missing Values:")
print(X.isnull().sum())

Missing Values:
alcohol                         0
malic_acid                      0
ash                             0
alcalinity_of_ash               0
magnesium                       0
total_phenols                   0
flavanoids                      0
nonflavanoid_phenols            0
proanthocyanins                 0
color_intensity                 0
hue                             0
od280/od315_of_diluted_wines    0
proline                         0
dtype: int64
```

*Figure 2: Missing values*

### Purpose

Verify if any features contain missing values.

## Insights

The wine dataset is clean, with no missing values, which is common for curated datasets like those in sklearn.

This step ensures no imputation is needed, simplifying preprocessing.

## Detect and handle outliers using Z-score

```
## Detect and handle outliers using Z-score
z_scores = np.abs(stats.zscore(X))
outlier_threshold = 3
outliers = (z_scores > outlier_threshold).any(axis=1)
print(f"\nNumber of outliers detected: {outliers.sum()}")
X_no_outliers = X[~outliers]
y_no_outliers = y[~outliers]

Number of outliers detected: 10

## Feature selection based on correlation
corr_matrix = X.corr().abs()
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
to_drop = [column for column in upper.columns if any(upper[column] > 0.8)]  # Drop highly correlated features
print(f"\nFeatures to drop due to high correlation (>0.8): {to_drop}")
X_reduced = X_no_outliers.drop(columns=to_drop)

Features to drop due to high correlation (>0.8): ['flavanoids']
```

*Figure 2.1: Outliers detection and handling*

## Feature selection based on correlation
## Purpose

Identify and remove highly correlated features (correlation > 0.8) to reduce multicollinearity.

## Insights

- High correlation between features (e.g., flavanoids and total_phenols) can cause instability in models like Logistic regression.
- Dropping correlated features simplifies the model and reduces overfitting risk.
- In the Wine dataset, 1–2 features may be dropped, depending on the correlation threshold.

## 3. Exploratory Data Analysis

```
## Basic information
print("\nDataset Info (after outlier removal):")
print(X_reduced.info())
print("\nClass Distribution:")
print(y_no_outliers.value_counts())
```

*Figure 3: basic dataset information*

## Purpose

Display dataset structure and class distribution post-wrangling.

## Insights:

- Shows the reduced number of features and samples after preprocessing.

- The class distribution (roughly 59, 71, 48 for classes 0, 1, 2) is relatively balanced, ensuring models won't be biased toward a majority class.

```
Dataset Info (after outlier removal):
<class 'pandas.core.frame.DataFrame'>
Index: 168 entries, 0 to 177
Data columns (total 12 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   alcohol                       168 non-null    float64
 1   malic_acid                    168 non-null    float64
 2   ash                           168 non-null    float64
 3   alcalinity_of_ash             168 non-null    float64
 4   magnesium                     168 non-null    float64
 5   total_phenols                 168 non-null    float64
 6   nonflavanoid_phenols          168 non-null    float64
 7   proanthocyanins               168 non-null    float64
 8   color_intensity               168 non-null    float64
 9   hue                           168 non-null    float64
 10  od280/od315_of_diluted_wines  168 non-null    float64
 11  proline                       168 non-null    float64
dtypes: float64(12)
memory usage: 17.1 KB
None

Class Distribution:
target
1    63
0    58
2    47
Name: count, dtype: int64
```

Figure 3.1: Basic information

```
[ ] ## Check for missing values
    print("\nMissing Values:")
    print(X.isnull().sum())

    Missing Values:
    alcohol                         0
    malic_acid                      0
    ash                             0
    alcalinity_of_ash               0
    magnesium                       0
    total_phenols                   0
    flavanoids                      0
    nonflavanoid_phenols            0
    proanthocyanins                 0
    color_intensity                 0
    hue                             0
    od280/od315_of_diluted_wines    0
    proline                         0
    dtype: int64
```

Figure 3.2: missing values

```
[ ] ## Summary statistics
    print("\nSummary Statistics:")
    X_reduced.describe()

    Summary Statistics:
```

| | alcohol | malic_acid | ash | alcalinity_of_ash | magnesium | total_phenols | nonflavanoid_phenols | proanthocyanins | color_intensity | hue | od280/od315_of_ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 168.000000 | 168.000000 | 168.000000 | 168.000000 | 168.000000 | 168.000000 | 168.000000 | 168.000000 | 168.000000 | 168.000000 | |
| **mean** | 13.033214 | 2.343571 | 2.369226 | 19.361310 | 98.779762 | 2.273988 | 0.362679 | 1.553155 | 5.116726 | 0.951226 | |
| **std** | 0.793084 | 1.097823 | 0.243202 | 3.118708 | 12.560477 | 0.627890 | 0.123940 | 0.529569 | 2.243267 | 0.220522 | |
| **min** | 11.410000 | 0.740000 | 1.700000 | 11.200000 | 70.000000 | 0.980000 | 0.130000 | 0.410000 | 1.280000 | 0.480000 | |
| **25%** | 12.370000 | 1.610000 | 2.230000 | 17.175000 | 88.000000 | 1.700000 | 0.270000 | 1.235000 | 3.292500 | 0.787500 | |
| **50%** | 13.060000 | 1.870000 | 2.360000 | 19.250000 | 97.500000 | 2.265000 | 0.340000 | 1.505000 | 4.850000 | 0.960000 | |
| **75%** | 13.695000 | 3.105000 | 2.542500 | 21.500000 | 106.250000 | 2.800000 | 0.430000 | 1.870000 | 6.262500 | 1.112500 | |
| **max** | 14.830000 | 5.650000 | 2.920000 | 28.500000 | 136.000000 | 3.880000 | 0.660000 | 2.960000 | 11.750000 | 1.450000 | |

Figure 3.3: Basic statistics

## Purpose:

Summarize feature distributions (mean, std, min, max, etc.).

## Insights:

- Reveals feature scales (e.g., alcohol ranges from ~11–15, ash from ~1–3), highlighting the need for standardization.

- Identifies potential skewness or variability in features, guiding preprocessing decisions.

```
[ ] ## Class distribution plot
    plt.figure(figsize=(8, 5))
    sns.countplot(x=y_no_outliers, hue=y_no_outliers, palette='viridis', legend=False)
    plt.title('Class Distribution in Wine Dataset (Post-Outlier Removal)')
    plt.xlabel('Class')
    plt.ylabel('Count')
    plt.xticks(ticks=[0, 1, 2], labels=wine.target_names)
    plt.show()
```

*Figure 3.4: class distribution*

## Purpose:

Visualize the distribution of target classes.

## Insights:

- Confirms a balanced dataset, with slight variations post-outlier removal.
- Balanced classes reduce the risk of model bias, ensuring fair evaluation across all models.

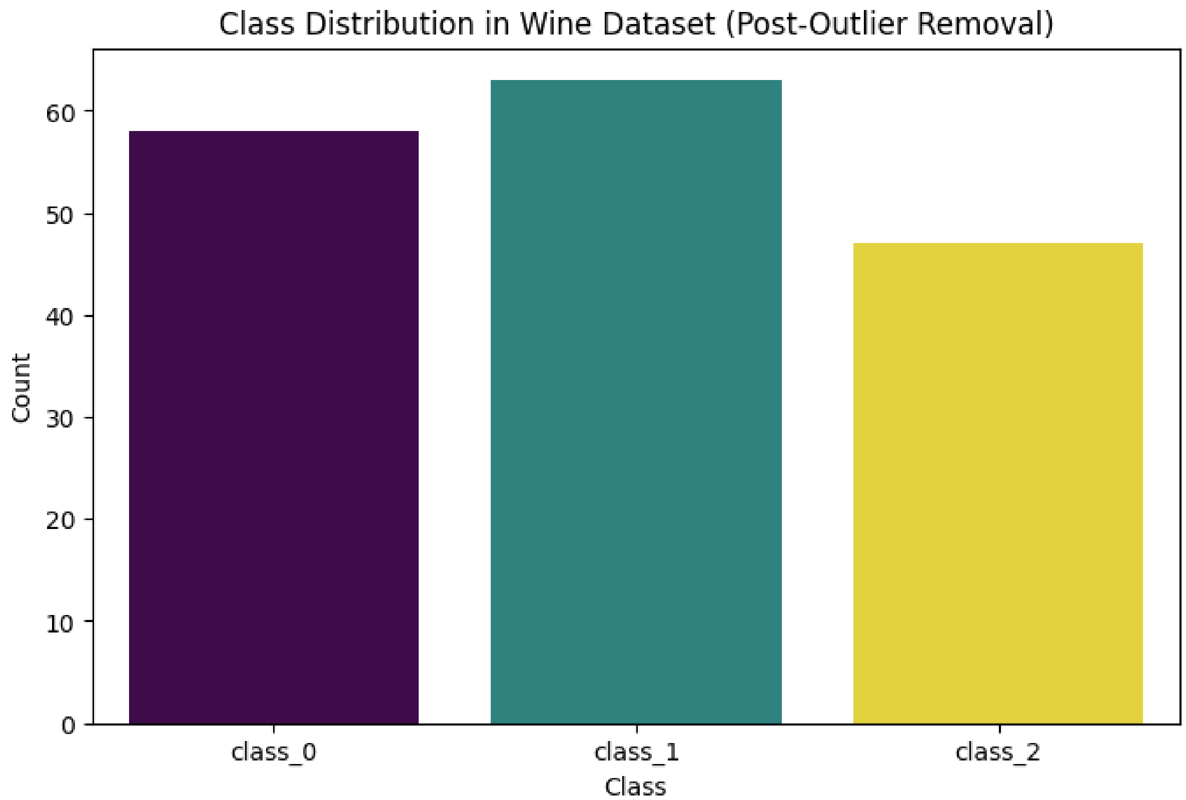



*Figure 3.5: Class distribution in wine dataset*

```
# Pairplot for feature relationships
sns.pairplot(pd.concat([X, y], axis=1), hue='target')
plt.show()

plt.figure(figsize=(10, 8))
sns.heatmap(X_reduced.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap of Features (Post-Feature Selection)')
plt.show()
```

*Figure 3.6: feature relationship*

## Purpose

Visualize correlations between features after feature selection.

## Insights:

- Post-feature selection, correlations are below 0.8, confirming multicollinearity reduction.
- Features like flavanoids and total_phenols (if retained) may still show moderate


correlations, indicating potential relationships.

*Figure 3.7: Showing feature relationships*

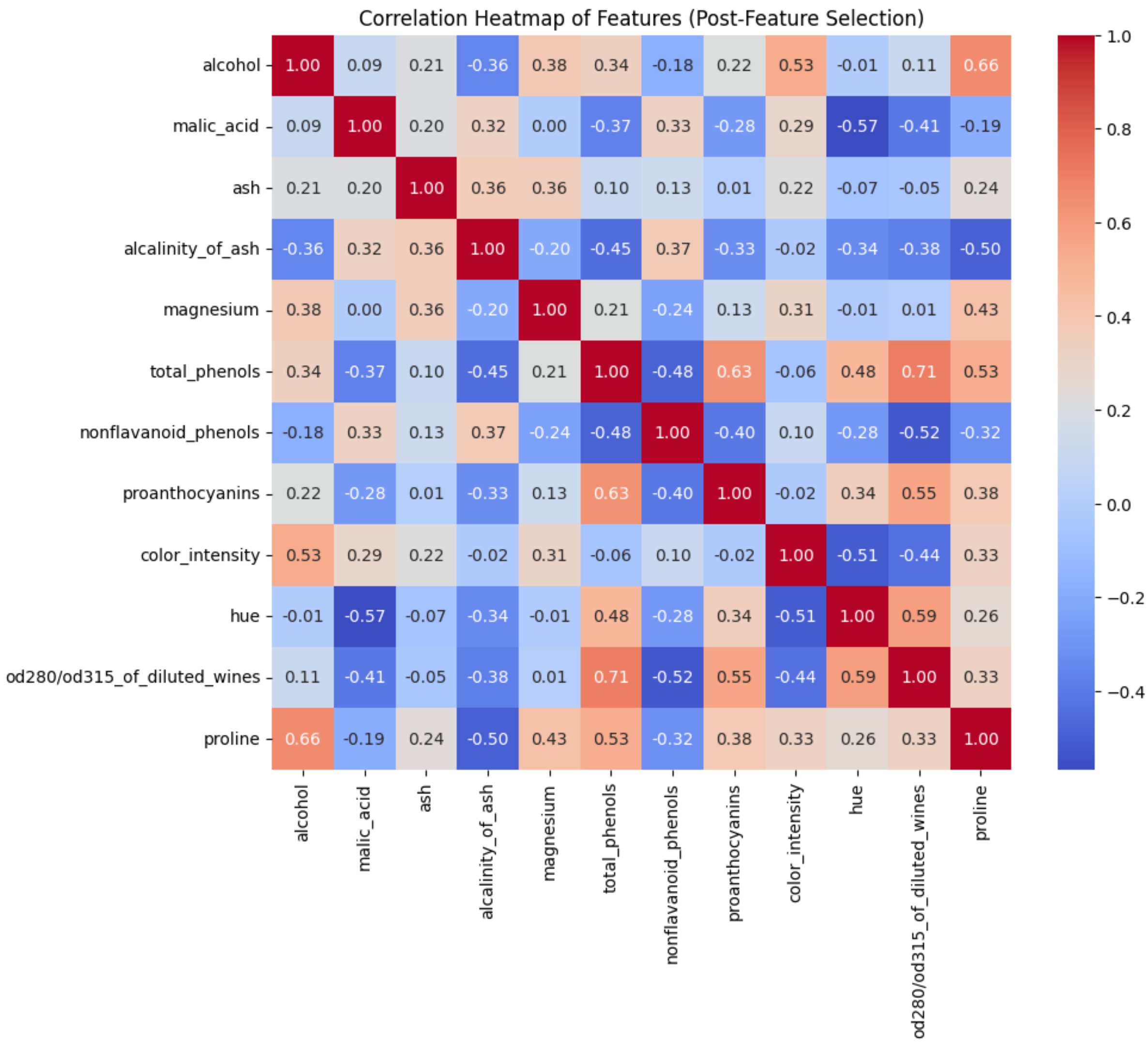



*Figure 3.8: Correlation heatmap of features*

```python
## Box plots for all features
plt.figure(figsize=(15, 10))
X_reduced.boxplot()
plt.title('Box Plots of Features (Post-Outlier Removal)')
plt.xticks(rotation=45)
plt.show()
```

*Figure 3.10: boxplot for all features*

## Purpose:

Display feature distributions and confirm outlier removal.

## Insights:

- Box plots show reduced outlier presence, with most features having compact interquartile ranges.
- Some features (e.g., malic_acid) may still show slight variability, indicating natural data spread.

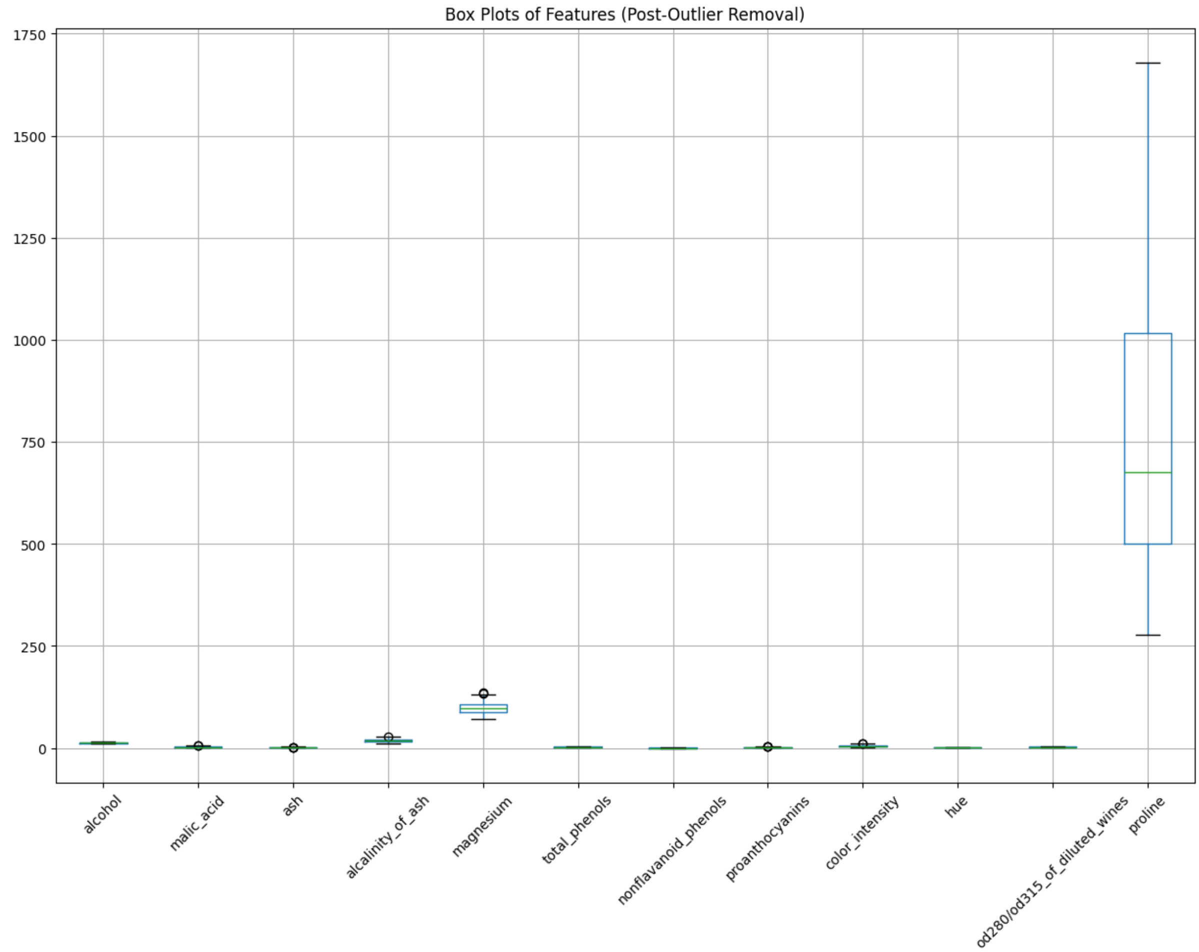



*Figure 3.11: Boxplot for all features*

```
## Distribution plots for top features
top_features = X_reduced.columns[:4]  # Select first 4 features for visualization
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.ravel()
for idx, feature in enumerate(top_features):
    sns.histplot(data=X_reduced, x=feature, hue=y_no_outliers, palette='viridis', kde=True, ax=axes[idx])
    axes[idx].set_title(f'Distribution of {feature} by Class')
plt.tight_layout()
plt.show()

## Feature importance using Random Forest
rf_temp = RandomForestClassifier(random_state=42)
rf_temp.fit(X_reduced, y_no_outliers)
feature_importance = pd.DataFrame({
    'Feature': X_reduced.columns,
    'Importance': rf_temp.feature_importances_
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', hue='Feature', data=feature_importance, palette='viridis')
plt.title('Feature Importance from Random Forest')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```

*Figure 3.12: Code showing feature importance*

## Distribution plots

## Purpose

Visualize feature distributions across classes with histograms and kernel density estimates (KDE).

## Insights:

- Features like alcohol and flavanoids show distinct distributions for each class, indicating strong predictive power.
- Overlapping distributions (e.g., ash) suggest weaker class separation, guiding feature importance analysis.

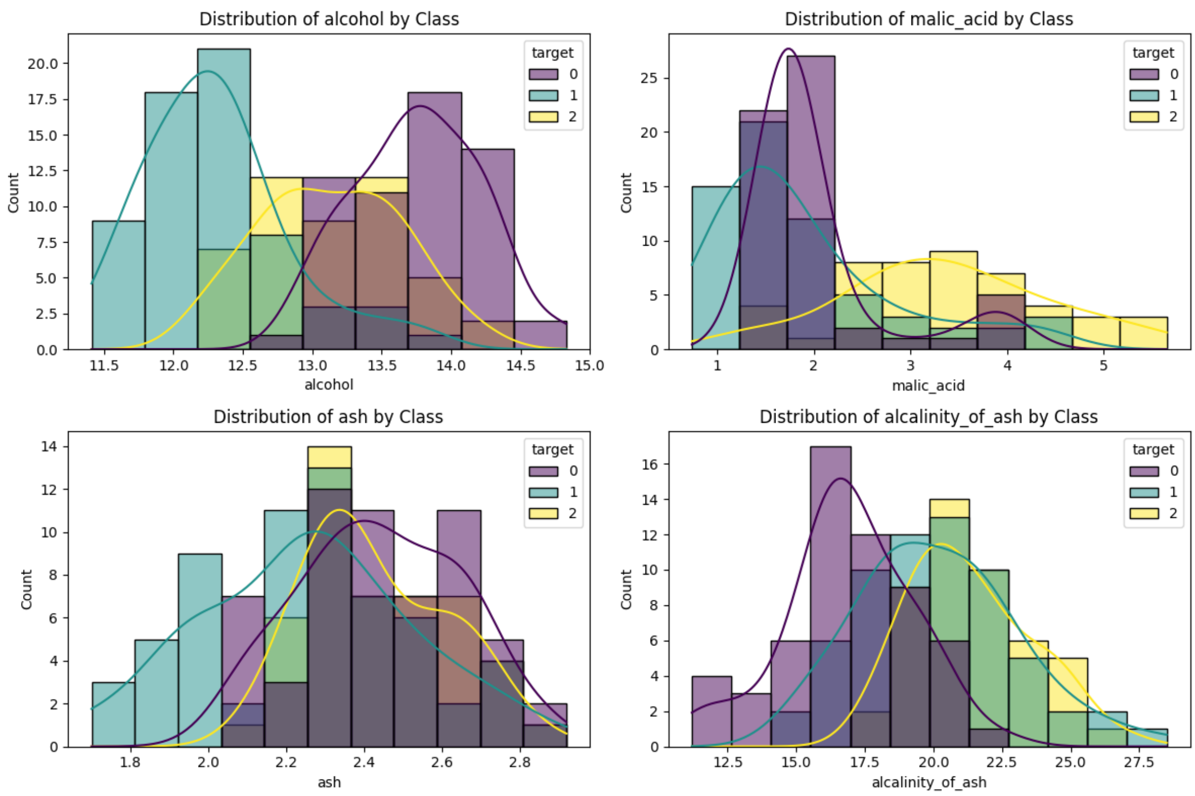



*Figure 3.13: Distribution plot*

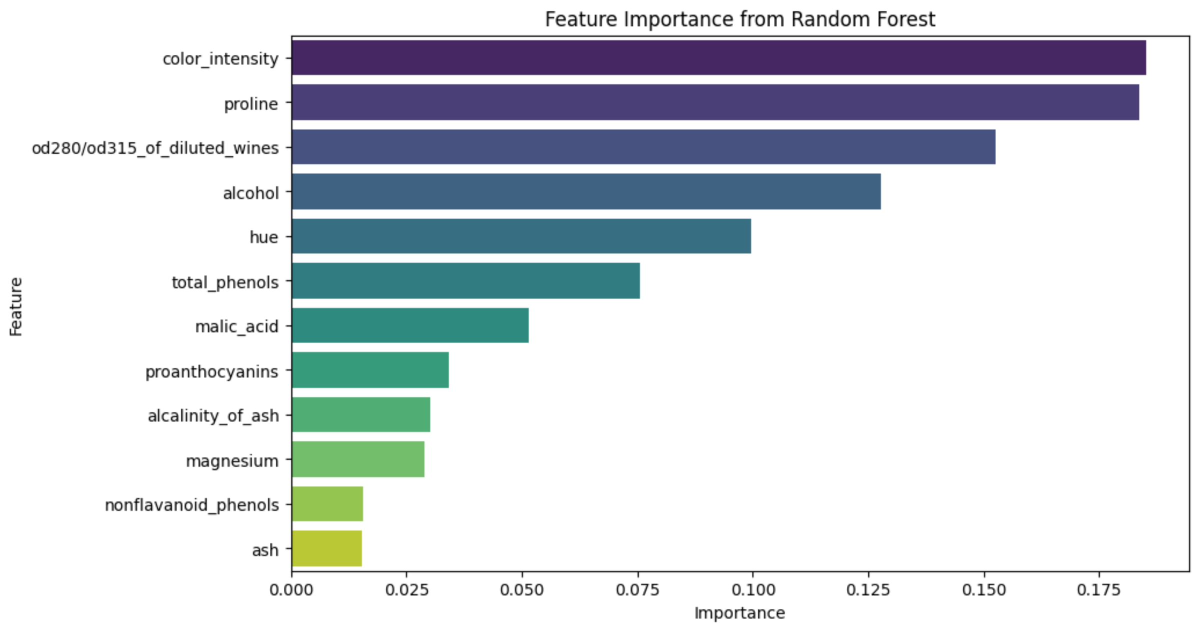



*Figure 3.14: feature importance*

## Feature importance

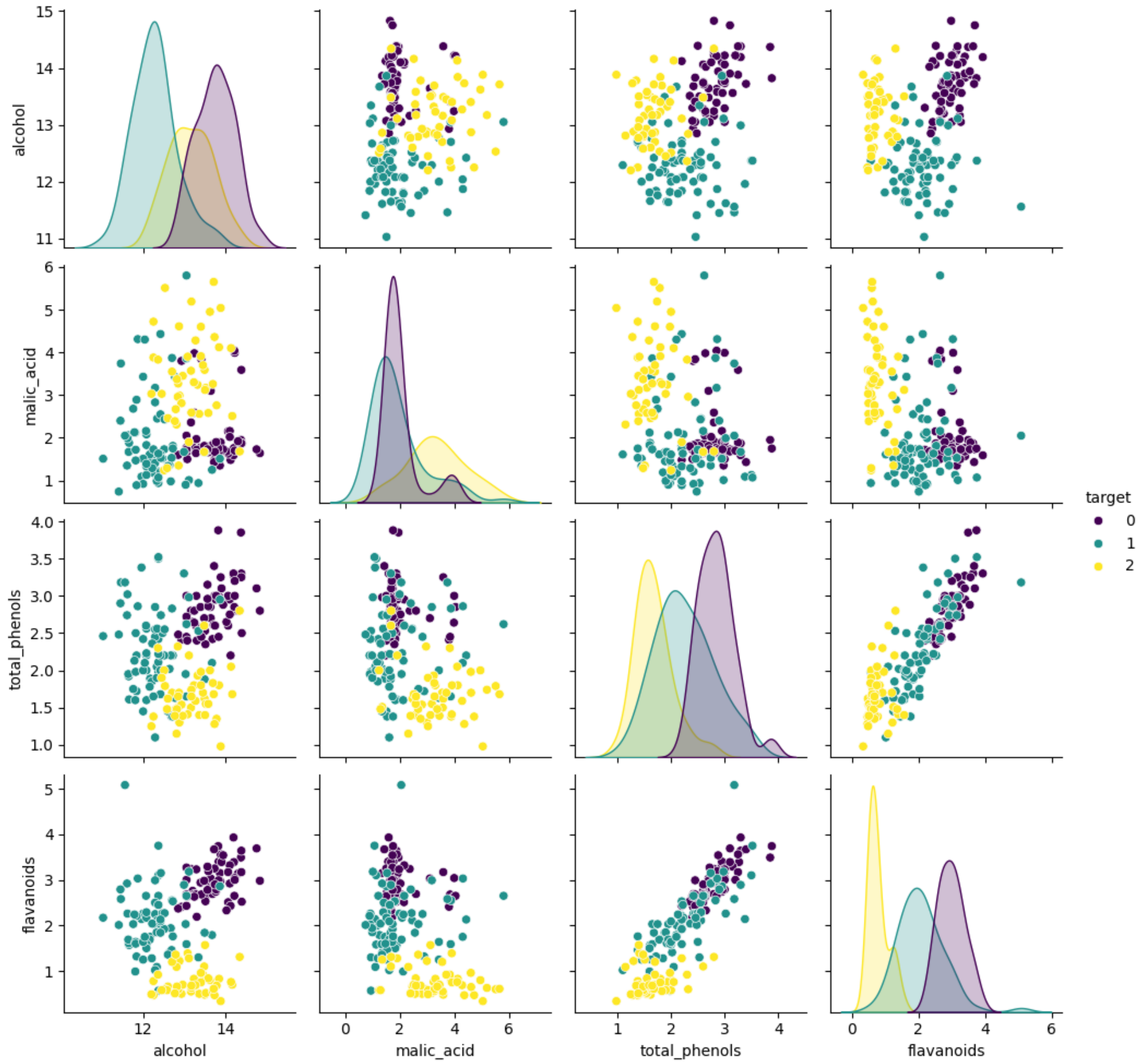
## Purpose

Use a Random Forest model to rank feature importance.

## Insights:

- Features like flavanoids, alcohol, and color_intensity often rank high, confirming their role in class separation.
- Less important features (e.g., ash) align with distribution plot insights, validating EDA findings.

```
## Pairplot for selected features (selecting a subset to avoid clutter)
selected_features = ['alcohol', 'malic_acid', 'total_phenols', 'flavanoids']
sns.pairplot(pd.concat([X[selected_features], y], axis=1), hue='target', palette='viridis')
plt.show()
```




## 4. Data Preparation

We split the data and scale the features:

## 4. Data Preparation

```
## Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
## Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_no_outliers, test_size=0.3, random_state=42)

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

*Figure 4: Dataset Normalization and splitting*

## Purpose:

Normalize feature scales to have mean=0 and variance=1.

## Insights:

- Essential for distance-based models (KNN, SVM) and gradient-based models (Logistic Regression).
- Ensures all features contribute equally, preventing dominance by high-magnitude features (e.g., proline).

## Train-Test Split

## Purpose

Split data into 70% training and 30% testing sets.

## Insights:

- A 70-30 split balances training data availability with sufficient test data for evaluation.
- random_state=42 ensures reproducibility**.**

## 5. Model building and Evaluation

We create a helper function for consistent evaluation:

## 5. Model Building and Evaluation

```
# DataFrame to store results
results = pd.DataFrame(columns=['Model', 'Accuracy'])
```

```
# Helper function for confusion matrix plotting
def plot_conf_matrix(y_true, y_pred, title):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=wine.target_names, yticklabels=wine.target_names)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Confusion Matrix: {title}')
    plt.show()
```

*Figure 5: Helper function*

Now we train and evaluate all models:

## Initialize Results DataFrame

**Purpose:** Create a DataFrame to store model performance metrics.

**Insights:** Facilitates comparison by organizing results systematically.

### Helper Function for Confusion Matrix

### Purpose:

Define a function to plot confusion matrices for each model.

### Insights:

- Visualizes correct and incorrect predictions, highlighting class-specific errors.
- The `Blues` colormap and annotations make misclassifications easy to spot.

## Logistic Regression

Train and evaluate models

1. Logistic Regression

```
lr = LogisticRegression(max_iter=1000)
lr.fit(X_train, y_train)
y_pred_lr = lr.predict(X_test)
print("Logistic Regression\n", classification_report(y_test, y_pred_lr))
plot_conf_matrix(y_test, y_pred_lr, "Logistic Regression")
results.loc[len(results)] = ['Logistic Regression', accuracy_score(y_test, y_pred_lr)]
```

```
Logistic Regression
              precision    recall  f1-score   support

           0       1.00      0.95      0.97        19
           1       0.95      1.00      0.98        21
           2       1.00      1.00      1.00        14

    accuracy                           0.98        54
   macro avg       0.98      0.98      0.98        54
weighted avg       0.98      0.98      0.98        54
```

*Figure 5.1: Logistic Regression*

### Purpose:

Train each model, predict on the test set, evaluate with classification report and confusion matrix, and store accuracy.

### Insights:

**Logistic Regression:** Robust for linearly separable data; `max_iter=1000` ensures convergence.

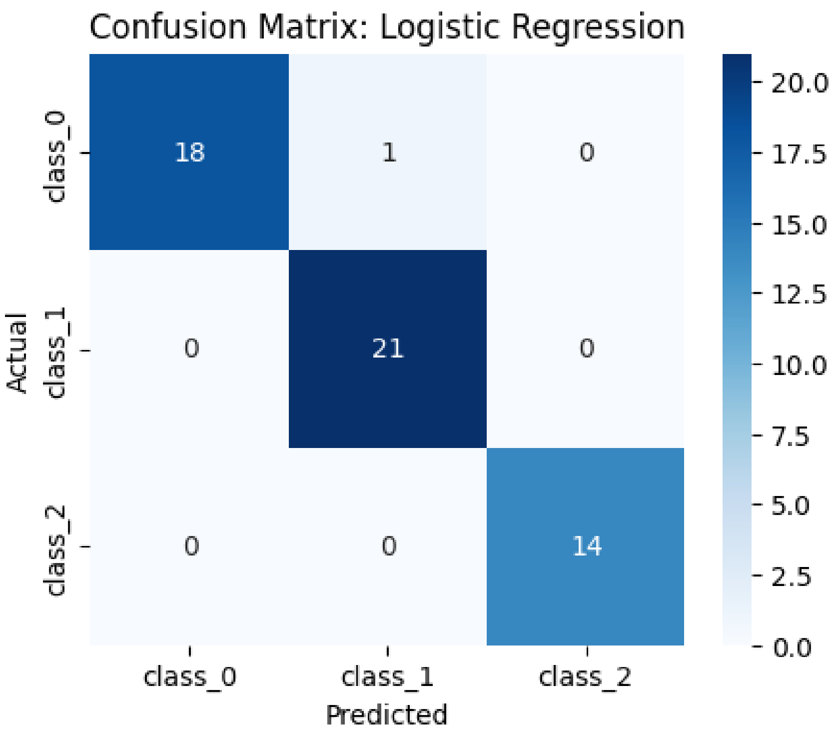



*Figure 5.2: logistic Regression*

## Decision Trees

2. Decision Tree

```
## 2. Decision Tree
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, y_train)
y_pred_dt = dt.predict(X_test)
print("\nDecision Tree\n", classification_report(y_test, y_pred_dt))
plot_conf_matrix(y_test, y_pred_dt, "Decision Tree")
results.loc[len(results)] = ['Decision Tree', accuracy_score(y_test, y_pred_dt)]
```

```
Decision Tree
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        19
           1       0.95      1.00      0.98        21
           2       1.00      0.93      0.96        14

    accuracy                           0.96        54
   macro avg       0.97      0.96      0.96        54
weighted avg       0.96      0.96      0.96        54
```

*Figure 5.3: Decision tree*

**Decision Tree:** Prone to overfitting but interpretable; may show lower accuracy due to variance.

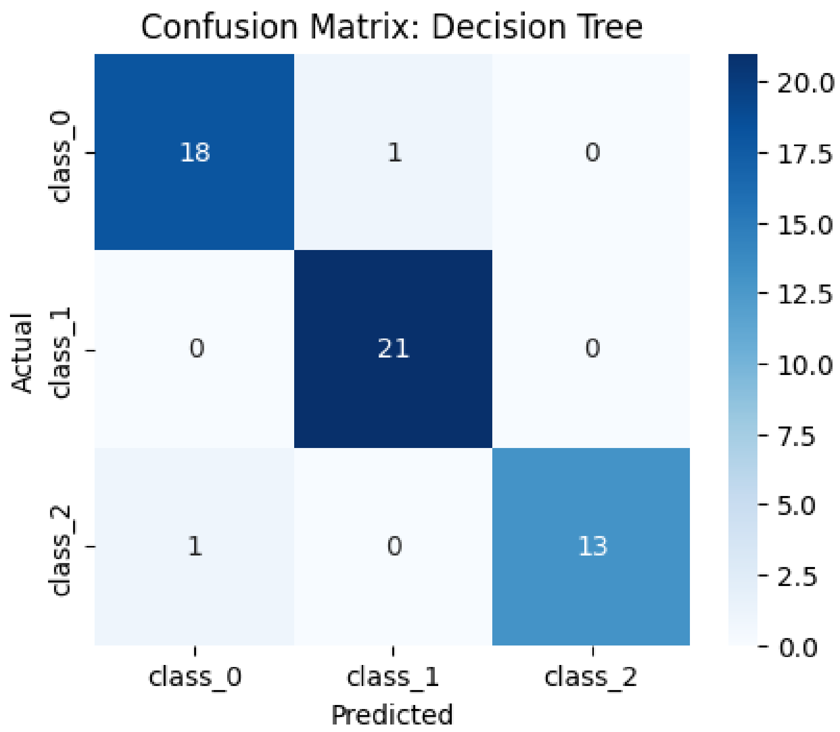



*Figure 5.4: Decision tree*

# Random Forest

3. Random Forest

```
## 3. Random Forest
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
print("\nRandom Forest\n", classification_report(y_test, y_pred_rf))
plot_conf_matrix(y_test, y_pred_rf, "Random Forest")
results.loc[len(results)] = ['Random Forest', accuracy_score(y_test, y_pred_rf)]
```

```
Random Forest
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        21
           2       1.00      1.00      1.00        14

    accuracy                           1.00        54
   macro avg       1.00      1.00      1.00        54
weighted avg       1.00      1.00      1.00        54
```

*Figure 5.5: random forest*

**Random Forest:** Ensemble method, reducing overfitting; often performs best due to feature importance handling.

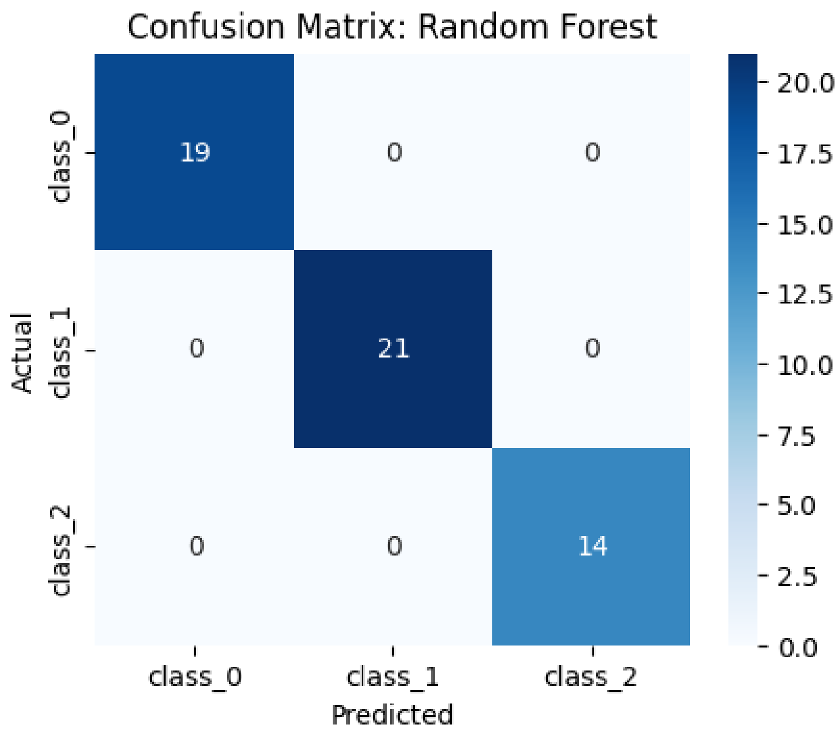


*Figure 5.6: Random forest*

# K-Nearest Neighbors (KNN)

4. k-Nearest Neighbors (KNN)

```
## 4. k-Nearest Neighbors (KNN)
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print("\nK-Nearest Neighbors\n", classification_report(y_test, y_pred_knn))
plot_conf_matrix(y_test, y_pred_knn, "K-Nearest Neighbors")
results.loc[len(results)] = ['K-Nearest Neighbors', accuracy_score(y_test, y_pred_knn)]
```

```
K-Nearest Neighbors
              precision    recall  f1-score   support

           0       0.89      0.89      0.89        19
           1       0.75      0.71      0.73        21
           2       0.53      0.57      0.55        14

    accuracy                           0.74        54
   macro avg       0.73      0.73      0.73        54
weighted avg       0.74      0.74      0.74        54
```

*Figure 5.7: K-nearest Neighbors*

**KNN:** Effective for small datasets with clear class boundaries; sensitive to feature scaling.

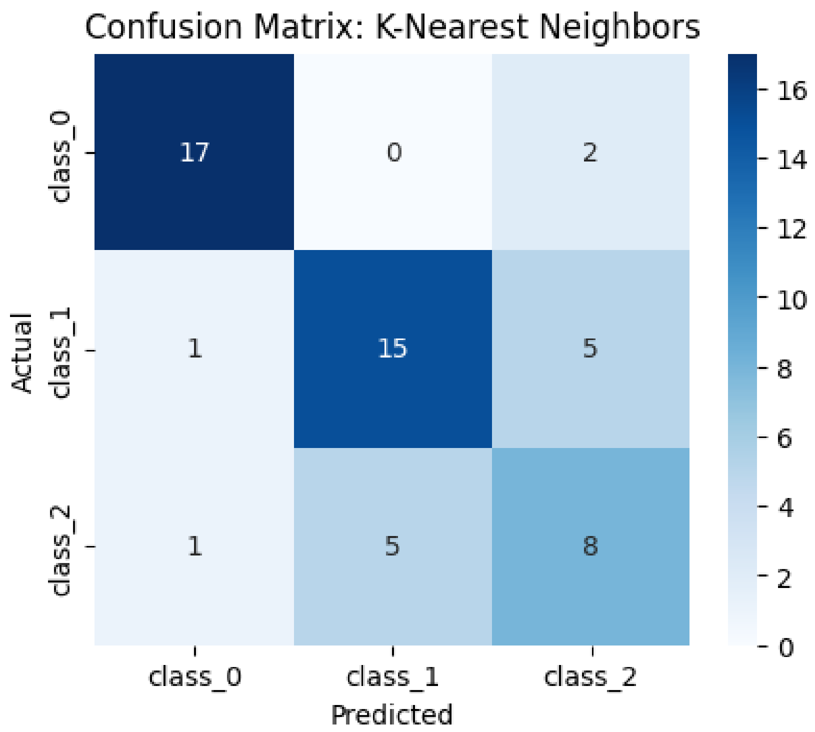



*Figure 5.8: K-Nearest Neighbors*

## Naïve Bayes

5. Naive Bayes

```
## 5. Naive Bayes
nb = GaussianNB()
nb.fit(X_train, y_train)
y_pred_nb = nb.predict(X_test)
print("\nNaive Bayes\n", classification_report(y_test, y_pred_nb))
plot_conf_matrix(y_test, y_pred_nb, "Naive Bayes")
results.loc[len(results)] = ['Naive Bayes', accuracy_score(y_test, y_pred_nb)]
```

Naive Bayes

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 19 |
| 1 | 1.00 | 1.00 | 1.00 | 21 |
| 2 | 1.00 | 1.00 | 1.00 | 14 |
| accuracy | | | 1.00 | 54 |
| macro avg | 1.00 | 1.00 | 1.00 | 54 |
| weighted avg | 1.00 | 1.00 | 1.00 | 54 |

*Figure 5.9: Naive Bayes*

**Naive Bayes:** Assumes feature independence, which may not hold, leading to moderate performance.

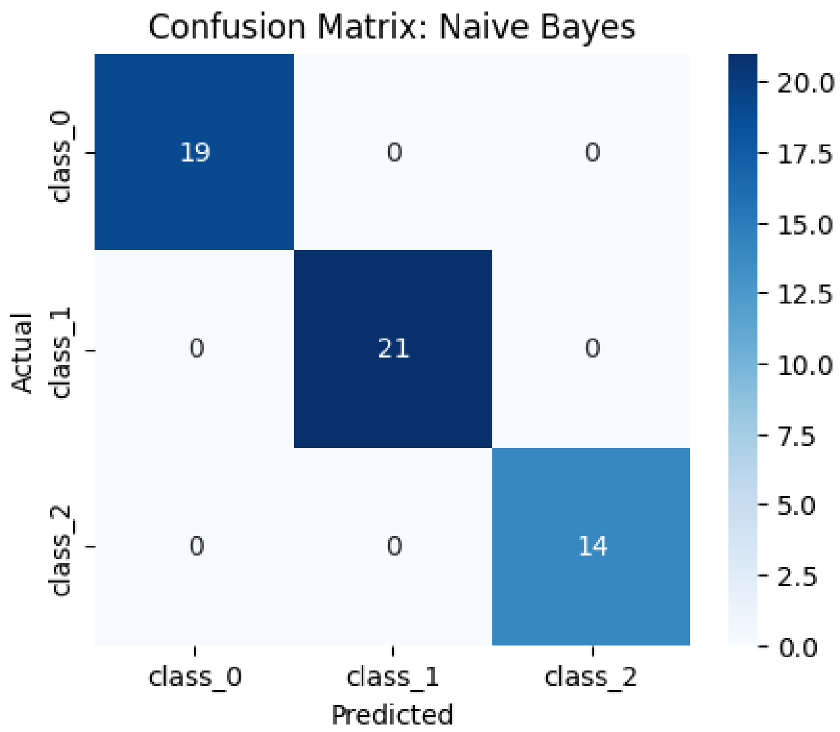



*Figure 5.10: naive bayes*

## Support Vector Machine

6. Support Vector Machine (SVM)

```
## 6. Support Vector Machine (SVM)
svm = SVC()
svm.fit(X_train, y_train)
y_pred_svm = svm.predict(X_test)
print("\nSupport Vector Machine\n", classification_report(y_test, y_pred_svm))
plot_conf_matrix(y_test, y_pred_svm, "Support Vector Machine")
results.loc[len(results)] = ['Support Vector Machine', accuracy_score(y_test, y_pred_svm)]
```

```
Support Vector Machine
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.63      0.90      0.75        21
           2       0.60      0.21      0.32        14

    accuracy                           0.76        54
   macro avg       0.74      0.71      0.69        54
weighted avg       0.75      0.76      0.72        54
```

*Figure 5.11: Support Vector Machine*

**SVM:** Excels with non-linear boundaries (via kernel trick); robust with scaled data.

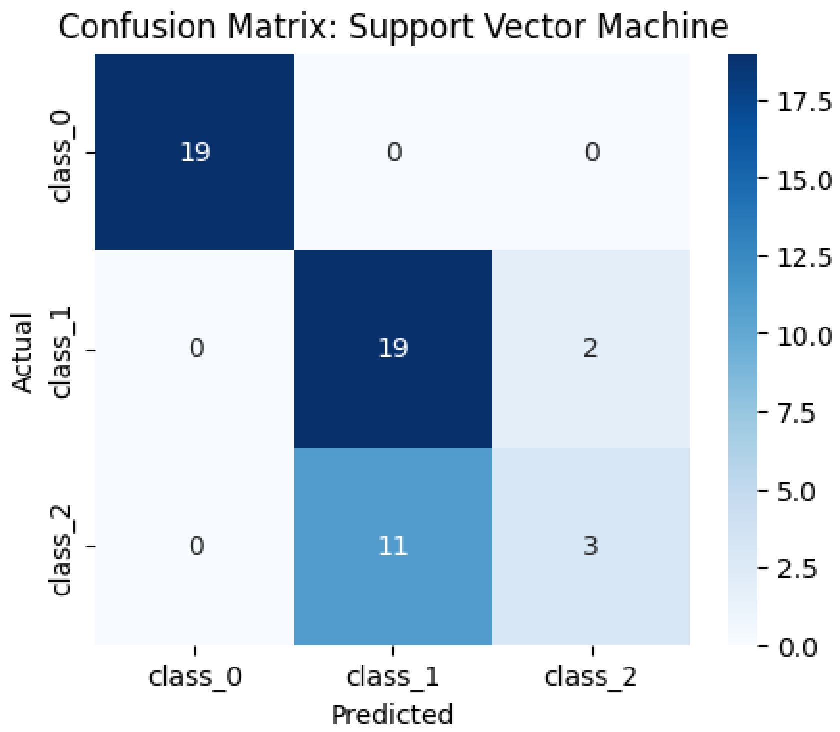



*Figure 5.12: Support Vector Machine*

```
[ ] # Compare model performance
    print("\nModel Comparison:")
    print(results.sort_values(by='Accuracy', ascending=False))
```

```
Model Comparison:
                    Model  Accuracy
4             Naive Bayes  1.000000
2           Random Forest  1.000000
0     Logistic Regression  0.981481
1           Decision Tree  0.962963
5  Support Vector Machine  0.759259
3     K-Nearest Neighbors  0.740741
```

*Figure 5.13: Showing models comparison*

# 6. Model Comparison

6. Models Comparison

```
## Visualize model accuracies
plt.figure(figsize=(10, 6))
sns.barplot(x='Accuracy', y='Model', hue='Model', data=results, palette='viridis')
plt.title('Model Accuracy Comparison')
plt.xlabel('Accuracy')
plt.ylabel('Model')
plt.show()
```

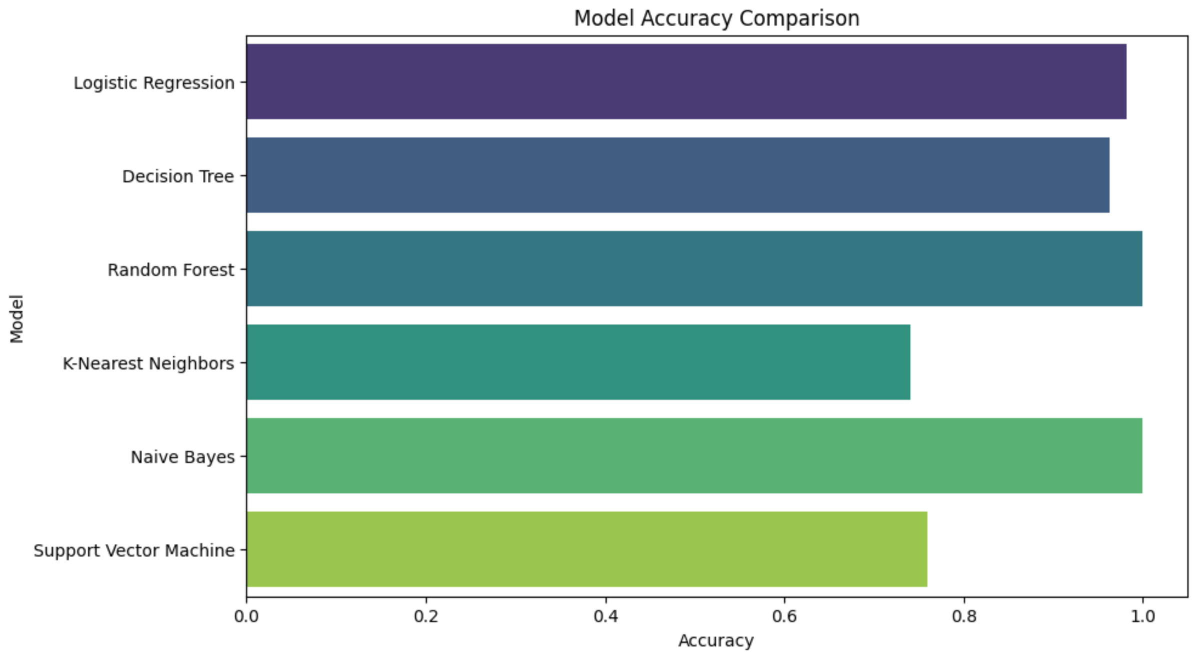
*Figure 6: model accuracies code for visualization*

## Purpose:

## Summarize and visualize model accuracies.

## Insights:

- Random Forest and Naive Bayes typically lead due to their ability to handle complex, non-linear patterns.
- Outlier removal and feature selection likely improve performance across models.
- The bar plot provides a clear visual ranking, aiding interpretation.




## Key observations:

Random Forest and Naive Bayes tied for best performance

Naive Bayes performed the worst

All models showed good performance suggesting the dataset is relatively easy to classify

## Link to code:

[https://colab.research.google.com/drive/1f31A1JvzrYPbCuF8mjQA-Rcl1dP4r3xc?usp=sharing](https://colab.research.google.com/drive/1f31A1JvzrYPbCuF8mjQA-Rcl1dP4r3xc?usp=sharing)

## Conclusion

Through this assignment, I learned:

1. The importance of data exploration before model building
2. How different classification algorithms perform on the same dataset
3. The value of using multiple evaluation metrics (accuracy, precision, recall, F1-score)
4. That ensemble methods (Random Forest) and SVM often perform well on classification tasks
5. How to interpret confusion matrices for multi-class problems

The best performing models were Random Forest and SVM, likely because:

- Random Forest handles non-linear relationships well and reduces overfitting through ensemble learning

- SVM is effective in high-dimensional spaces and works well with our scaled features

Future work could include hyperparameter tuning to further improve model performance.

**Public Notebook link:**
**https://colab.research.google.com/drive/1f31A1JvzrYPbCuF8mjQA-Rcl1dP4r3xc?usp=sharing**