UNIVERSITY OF LONDON

(CM3015)

Machine Learning and Neural Networks

A Comparative Analysis of kNN and Decision Tree for Multi-Class Classification

Ellen Faustine

220570273

Table of Contents

Abstract

This project aimed to compare the performance of the k-Nearest Neighbors (kNN) and Decision Tree algorithms for multi-class classification using the Wine dataset. The analysis focused on evaluating key metrics, including test accuracy, cross-validation accuracy, and precision, to assess the strengths and weaknesses of each algorithm. Results indicated that kNN outperformed the Decision Tree model across all evaluated metrics, achieving a test accuracy of 96.43%, cross-validation accuracy of 98.18%, and precision of 96.97%. Decision Trees, while less effective with a test accuracy of 89.29% and precision of 90.28%, demonstrated robustness to outliers and interpretability in their decision-making process. Both models exceeded the baseline accuracy of 39.89%, validating their suitability for the task. This study highlights the importance of data preprocessing and parameter tuning in optimizing model performance and emphasizes the superior predictive capability of kNN for this dataset. Future work could expand the analysis to larger, more complex datasets and additional evaluation metrics for broader insights.

Introduction

Understanding the Problem

Machine learning plays a crucial role in solving complex classification problems, particularly in domains such as healthcare, finance, and product recommendation systems. Among these, multi-class classification is a fundamental task where instances are categorized into one of several predefined classes. This project focuses on evaluating and comparing two widely used algorithms: k-Nearest Neighbors (kNN) and Decision Trees, to understand their strengths and limitations in multi-class classification.

Aim and Relevance

The aim of this project is to compare the performance of kNN and Decision Tree classifiers using the Wine dataset. Understanding the performance of these algorithms on multi-class problems is crucial for selecting the appropriate model based on dataset characteristics, computational efficiency, and accuracy requirements. By comparing these two algorithms, this study seeks to offer practical insights into their suitability for multi-class classification tasks.

Dataset Overview

The dataset utilized in this project is the Wine dataset from scikit-learn, which contains chemical analysis of wines produced by three different cultivators in the same region of Italy. It includes 178 samples with 13 numerical features representing various chemical constituents found in the wines, such as alcohol content, flavanoids, and proline levels. The target variable classifies the wines into three distinct categories: class_0, class_1, class_2. This dataset is widely used in machine learning research as a benchmark for multi-class

classification tasks, offering a structured and well-documented testbed for evaluating algorithm performance.

Challenges with the Dataset

Despite its small size, this dataset presents challenges such as highly correlated features and the presence of outliers. Features like 'proline' and 'magnesium' exhibit significant variance, which may influence the results of sensitive models like kNN. These challenges emphasize the importance of proper preprocessing to ensure reliable results.

Background

This project explores two machine learning algorithms: k-Nearest Neighbors (kNN) and Decision Trees, to perform multi-class classification on the Wine dataset. These algorithms represent distinct approaches to classification, providing an opportunity to evaluate their effectiveness under the same conditions.

k-Nearest Neighbors (kNN)

The k-Nearest Neighbors (kNN) algorithm is a supervised learning method used to classify or predict the outcome for a given data point based on its proximity to other points. While it can handle either classification or regression tasks, it is most commonly used for classification. For classification, the algorithm assigns a class label to the data point based on a majority vote among its nearest neighbors. Essentially, the core idea of kNN is that points that are closer together (or "neighbors") are more likely to share similar characteristics.

Two key parameters that influence the behavior of the kNN algorithm:

1. **k (the number of neighbors):** The number of closest points considered when determining the class of a given data point.
2. **Distance Metric:** The formula used to calculate the distance between data points. Common metrics include Euclidean, Manhattan, and Minkowski distances.

Strengths and Weaknesses

The kNN algorithm is straightforward to implement compared to other algorithms due to its simplicity and effectiveness. It requires only a few hyperparameters, such as the number of neighbours (k) and the distance metric. Unlike other algorithms that explicitly train a model, kNN relies on storing the training data and dynamically adjusts to new data without requiring retraining, making it a highly adaptable algorithm.

However, kNN comes with several notable limitations. It does not scale well with large datasets, as it requires storing all training data in memory and performing distance calculations for every prediction. This results in high computational costs in terms of time and memory, making it inefficient for large-scale applications. Additionally, kNN struggles with high-dimensional data due to the curse of dimensionality, where points in

high-dimensional spaces tend to become equidistant, diminishing the effectiveness of distance metrics. Furthermore, kNN is prone to overfitting when the value of k is too small, as the model becomes overly influenced by noise in the data. Contrarily, setting k too high can over smooth predictions, reducing the model's sensitivity to smaller patterns in the data.

Suitability for the Wine Dataset

The Wine dataset, comprising 13 numerical features and three target classes, is well-suited for kNN because the classes are distinct and the numerical nature of the features allow the algorithm to calculate the distances effectively. Additionally, with 178 samples, the dataset is small enough for kNN to handle efficiently without running into significant computational or memory issues. However, the dataset also contains some correlated features and outliers, which require careful preprocessing to ensure the effectiveness of kNN.

Decision Trees

The Decision Tree is a non-parametric supervised learning algorithm that recursively applies a "divide and conquer" approach to split the dataset into smaller subsets based on feature values. The primary objective is to build a model that predicts the value of a target variable by learning simple decision rules derived from the data's features. Decision Trees are versatile and can handle both classification and regression tasks. For classification tasks, the tree predicts a class label by traversing the branches based on feature values until it reaches a leaf node.

Structure of a Decision Tree

A Decision Tree follows a hierarchical structure, which consists of:

1. **Root node:** The topmost node of the tree that represents the initial split. It evaluates the most distinguishing feature to separate the classes
2. **Decision nodes:** Intermediate nodes that represent tests or conditions based on feature values, leading to further splits
3. **Leaf node:** Terminal nodes at the bottom of the tree, representing the final output or prediction

Types of Decision Tree Algorithms

1. **CART (Classification and Regression trees):** Used for both classification and regression tasks. It evaluates splits using Gini Impurity, which measures how often a randomly chosen attribute would be incorrectly classified
2. **ID3 (Iterative Dichotomiser 3):** Focuses on classification tasks, it uses entropy and information gain as metrics to evaluate candidate splits
3. **C4.5**: An extension of ID3 that handles both continuous and categorical data. It uses information gain ratio to determine the optimal split points

Strengths and Weaknesses

Decision Trees are easy to interpret due to their reliance on boolean logic and visual representation. They are based on a hierarchy of simple classification rules that are straightforward to visualize and understand. Decision Trees are also referred to as white box models because their decision making process is transparent and easy to explain. Additionally, they can handle both numerical and categorical data with little to no preprocessing required, making them versatile. This algorithm can be applied to both classification and regression tasks, offering greater flexibility compared to many other algorithms.

Despite their strengths, Decision Trees also have several limitations. They are prone to overfitting, especially when the tree grows too deep, capturing noise instead of general patterns in the data. This leads to poor generalization to new, unseen data. Moreover, Decision Trees are sensitive to small changes in the dataset, as slight variations can result in a significantly different tree structure, leading to high variance. Finally, they can also be more computationally expensive to train compared to other algorithms, as they utilize a greedy search approach, evaluating multiple splits at each node to determine the optimal one.

Suitability for the Wine Dataset

The Wine dataset, with its 13 numerical features and three target classes provide sufficient data for the algorithm to make effective splits and build a meaningful tree structure. Decision trees are also capable of handling correlated features, which are present in the Wine dataset, by selecting the most informative features at each split. Moreover, the algorithm's transparency and visual representation makes it ideal for understanding the relationships between the chemical attributes of the wines and their classifications.

Methodology

In this section, we describe the steps taken to explore the dataset, preprocess the data, and implement the algorithms. Additionally, I justify the decision behind key decisions such as using cross-validation for model evaluation and how our custom k-Nearest Neighbors (kNN) and Decision Tree implementations were developed.

Dataset Exploration

Before proceeding with model development, it is essential to thoroughly explore and understand the dataset's structure, characteristics, and potential challenges. This preliminary step ensures that appropriate preprocessing techniques and evaluation metrics are selected. By examining the dataset, we can identify issues such as outliers, missing data, or correlated features that may significantly influence the model performance.

We can do this by first analyzing the description of the dataset which provides valuable metadata such as attribute information, summary statistics and data characteristics

Wine recognition dataset

Data Set Characteristics:

:Number of Instances: 178
:Number of Attributes: 13 numeric, predictive attributes and the class
:Attribute Information:
- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

- class:
  - class_0
  - class_1
  - class_2

:Summary Statistics:

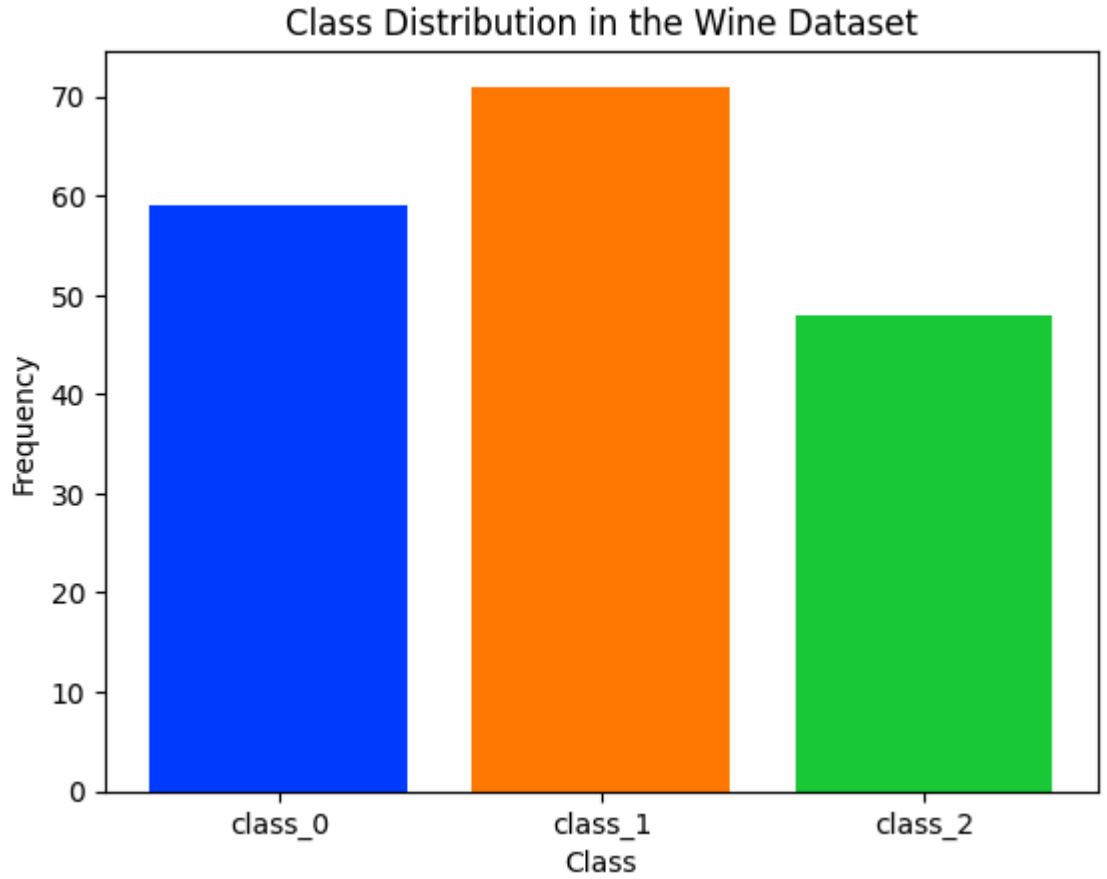
	Min	Max	Mean	SD
Alcohol:	11.0	14.8	13.0	0.8
Malic Acid:	0.74	5.80	2.34	1.12
Ash:	1.36	3.23	2.36	0.27
Alcalinity of Ash:	10.6	30.0	19.5	3.3
Magnesium:	70.0	162.0	99.7	14.3
Total Phenols:	0.98	3.88	2.29	0.63
Flavanoids:	0.34	5.08	2.03	1.00
Nonflavanoid Phenols:	0.13	0.66	0.36	0.12
Proanthocyanins:	0.41	3.58	1.59	0.57
Colour Intensity:	1.3	13.0	5.1	2.3
Hue:	0.48	1.71	0.96	0.23
OD280/OD315 of diluted wines:	1.27	4.00	2.61	0.71
Proline:	278	1680	746	315

:Missing Attribute Values: None
:Class Distribution: class_0 (59), class_1 (71), class_2 (48)
:Creator: R.A. Fisher
:Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
:Date: July, 1988

From this description snippet, we can see that this dataset is generally a small dataset with only 178 samples and 13 features and that there are no missing attribute values

Class Distribution Analysis

We begin by exploring the class distribution of the dataset, as understanding the class distribution is crucial for identifying any potential imbalance, which can inform our choice of appropriate evaluation metrics to assess classifier performance. We can observe from the description of data that there are three distinct classes followed by their class distribution. We can visualize this class distribution using bar charts

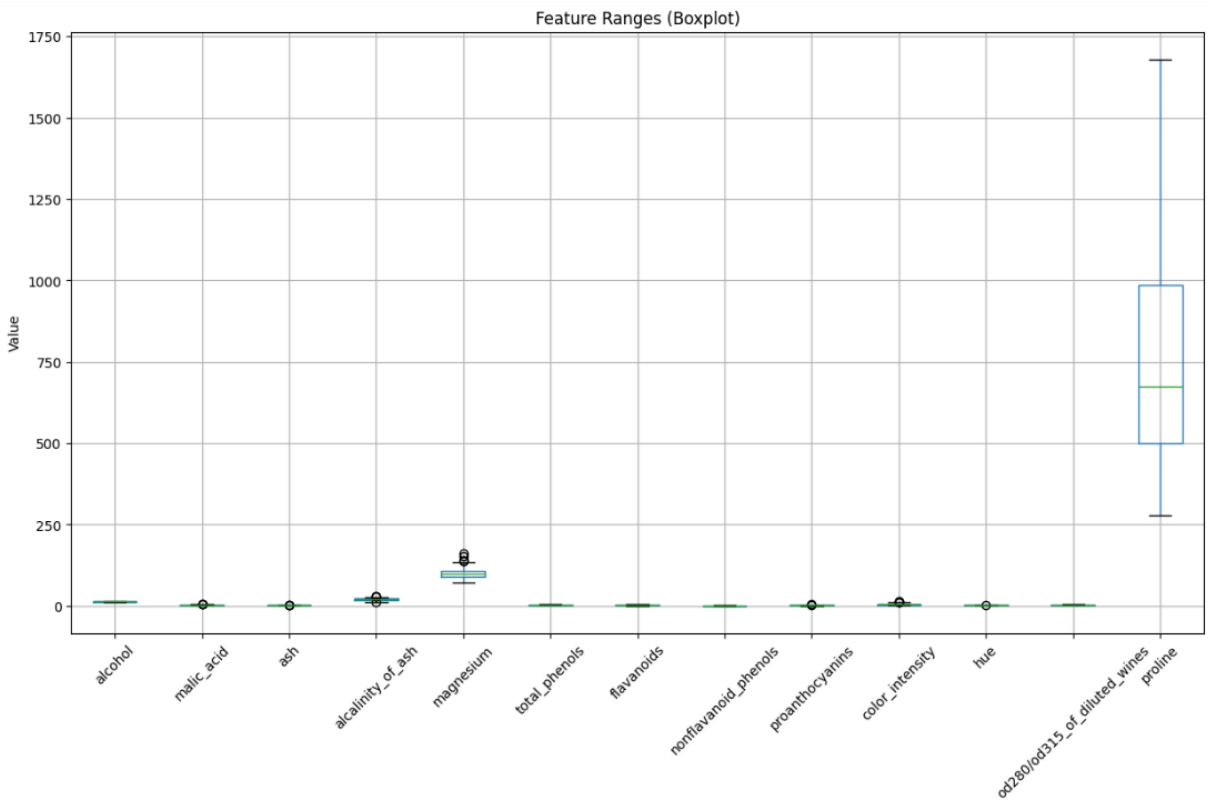


The class distribution plot above illustrates a relatively balanced dataset. Class_1 is the majority class, comprising 40% (71 samples), followed by class_0 with 33% (59 samples), and class_2 with the least representation at 27% (48 samples). While the class distribution is not perfectly equal, the imbalance is minor.

To determine whether a dataset is balanced or imbalanced, a commonly accepted threshold is a 65%-35% split for binary-class datasets. Any difference beyond this ratio is generally considered imbalanced (Hoffman, 2021). In this case, the difference between the majority and minority classes is only 13%, which is well below this threshold. Thus, we can confidently classify this dataset as balanced.

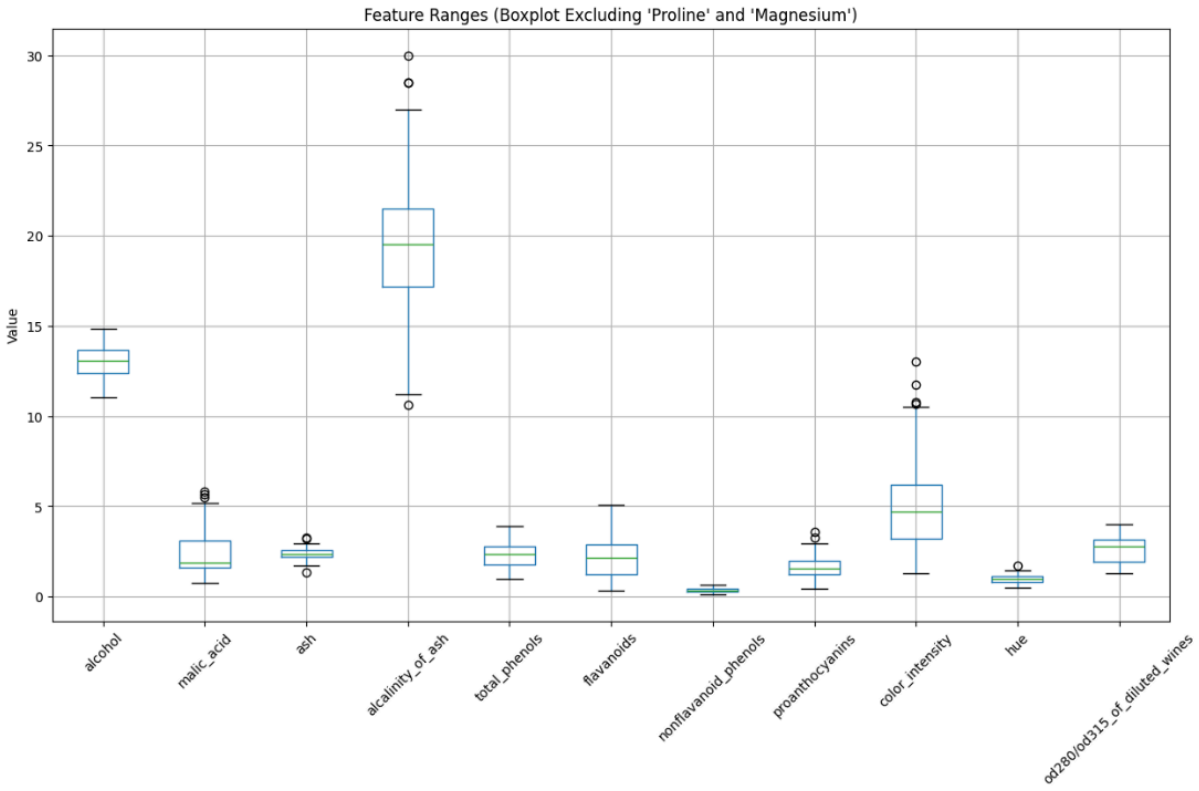
Feature Distribution Analysis

We will analyze the feature distributions to identify and evaluate relationships between features, detect potential issues such as correlated features, outliers, and high variance. This analysis will guide us in selecting appropriate preprocessing techniques and inform our decisions on how to address these challenges effectively. By understanding the feature characteristics, we can ensure that the dataset is well-prepared for model training and evaluation.



From the box plot above, it is apparent that the feature proline exhibits a significantly larger range and higher variation compared to the other features. Similarly, the feature magnesium also displays relatively higher values, as its spread is more noticeable than most other features. These findings highlight the necessity of properly scaling the dataset to ensure all features contribute proportionally to the model's performance.

To better examine the remaining features, the proline and magnesium features were removed to avoid their high variance overshadowing the detail in the ranges of other features. This adjustment allows for a clearer focus on the characteristics of the remaining features.

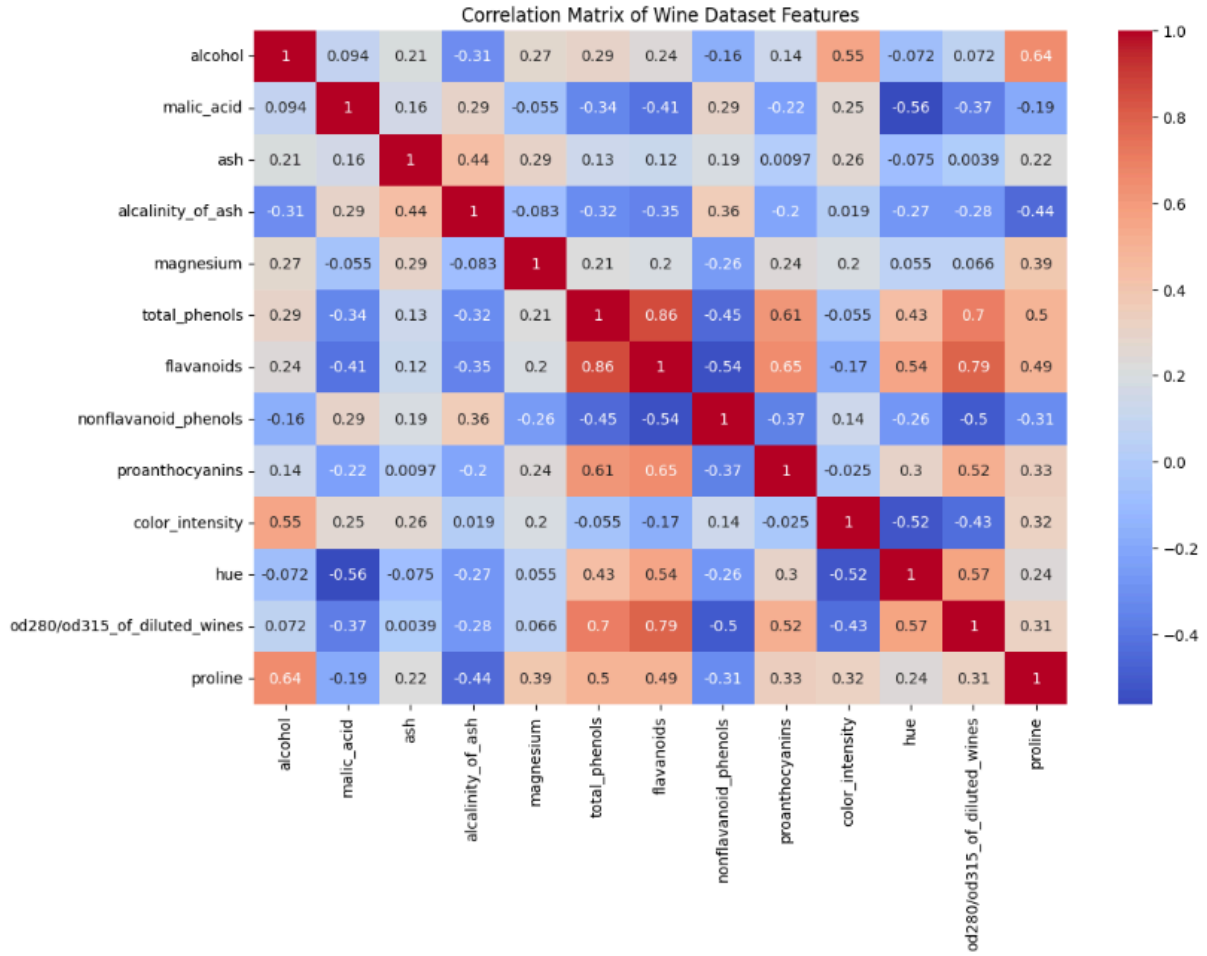


In the second plot, we observe that several features, such as malic_acid, ash, alcalinity_of_ash, proanthocyanins, color_intensity, and hue, contain outliers. These outliers can skew distance calculations and negatively impact the performance of algorithms like kNN. Therefore, it is crucial to address these outliers appropriately, either by removing them or applying appropriate scaling techniques, before proceeding with model development.

Correlation Analysis

Next, we will identify whether there are highly correlated features within the dataset, as strong correlations can negatively impact algorithm performance. For kNN, high correlation can skew distance calculations because the same information is effectively counted multiple times. This can lead to biased predictions and reduced model performance. Similarly, in decision trees, highly correlated features may result in overfitting as the tree might rely heavily on redundant splits.

To analyze these relationships, we will use Pearson's correlation coefficient, which quantifies the linear relationship between features. The coefficient ranges from +1 (perfect positive correlation) to -1 (perfect negative correlation), with a value of 0 indicating no correlation. Identifying and addressing these correlations will help improve model reliability and prevent redundant feature influence.



The heatmap above illustrates the correlation coefficients between features in the dataset. Features with correlation coefficients greater than 0.75 or less than -0.75 are considered highly correlated. For example, ‘flavanoids’ and ‘total phenols’ exhibit a strong positive correlation with a coefficient of 0.86, while ‘od280/od315_of_diluted_wines’ and ‘flavanoids’ show a correlation of 0.79. These strong correlations indicate potential redundancy, which may require feature selection or preprocessing to avoid negatively impacting model performance.

Summary of Findings

Through these analyses, we conclude that the wine dataset is a balanced dataset, as there is no major difference in the class distribution. However, the dataset exhibits some challenges, including high variance in certain features and the presence of outliers, which could disproportionately influence kNN predictions or result in suboptimal splits in decision trees. Additionally, there are a few highly correlated features that require appropriate handling.

Since the dataset is balanced, accuracy can be used as an evaluation metric to assess the performance of kNN and decision tree algorithms. To prepare the dataset effectively, the data can be scaled to normalize feature values and address high variance, ensuring that all

features contribute proportionally to the models. Outliers can be handled using the interquartile range (IQR) method to minimize their impact on model performance. Additionally, highly correlated features can be addressed by removing the ones least related to the target variable, thereby reducing redundancy and improving the reliability of the models. These preprocessing steps collectively aim to enhance the performance and interpretability of the classification algorithms.

Evaluation Metrics

Establishing a Baseline

Baseline models serve as a foundational reference for evaluating the performance of machine learning models. They fulfill a dual purpose: establishing a baseline performance metric against which improvements can be assessed, and they provide a benchmark for determining the efficiency of more complex models. At their core, baseline models set a minimum level of performance expectation, helping to contextualize the results of advanced models.

For multi-class classification tasks, baseline accuracy is calculated as the proportion of samples in the majority class to the total number of samples. Using the formula:

$$\text{Baseline Accuracy} = \frac{Number\ of\ samples\ in\ majority\ class}{Total\ number\ of\ samples}$$

For the Wine dataset, the majority class comprises 71 samples out of 178, resulting in a baseline accuracy of 39.89% as a percentage. This value provides a performance threshold for evaluating the effectiveness of kNN and Decision Tree classifiers on the dataset.

Accuracy

Accuracy is a fundamental metric that measures how often a machine learning model correctly predicts the outcome. It is a simple and effective metric for classification tasks when the dataset is balanced and all errors have equal importance. It is calculated using the formula:

$$\text{Accuracy} = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

In this project, accuracy is used to evaluate both the cross-validation and test set performances. Cross-validation accuracy represents the average accuracy achieved across multiple folds during the cross-validation process. Test set accuracy, on the other hand, measures the model's performance on unseen data, offering a direct evaluation of its predictive strength. To ensure consistency and interpretability, all accuracy values are expressed as percentages.

Precision

Precision is a metric that measures how often a machine learning model correctly predicts the positive class. It is calculated as the ratio of true positives to the total number of positive predictions, which includes both true positives and false positives. The formula is as follows:

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

In this project, precision is calculated for the test set using the confusion matrix to assess the model's ability to make accurate positive predictions. All precision values are expressed as percentages for clarity and consistency in presentation.

Data Preprocessing

In this section, I will perform all the necessary data preparation steps before using the dataset for training and testing our models. These steps are crucial for enhancing the dataset's quality and ensuring fair and accurate model comparisons.

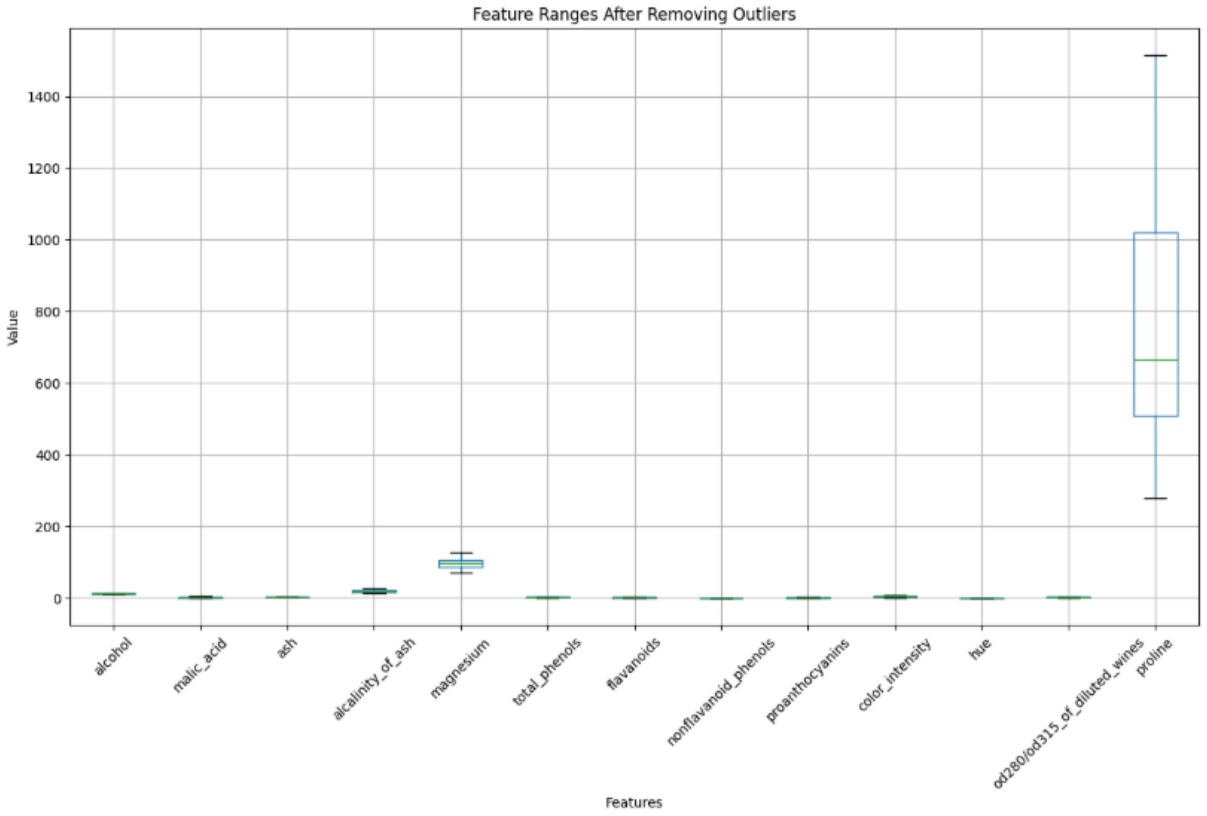
Outlier Removal

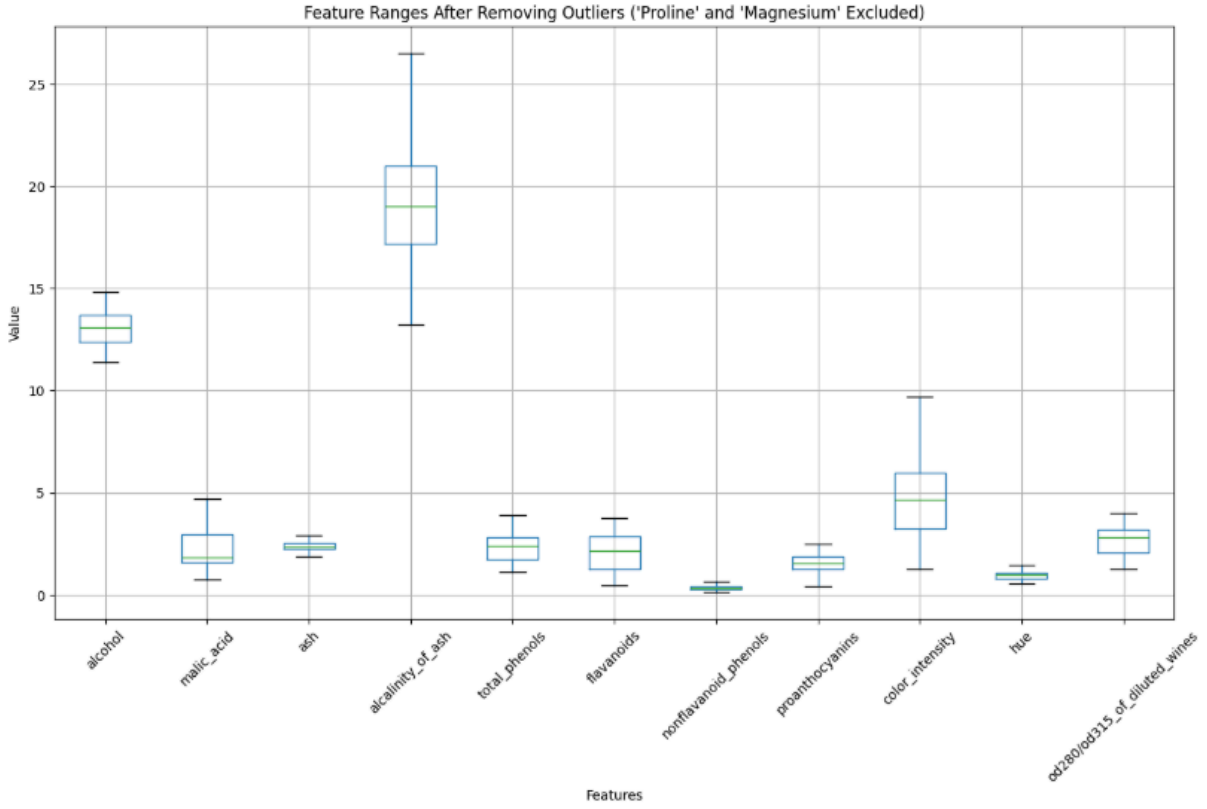
First, I will remove the outliers present in features such as magnesium, malic_acid, ash, alcalinity_of_ash, proanthocyanins, color_intensity, and hue. This step is essential to prevent distortion during scaling, as scaling methods are highly sensitive to extreme values. To achieve this, I will use the interquartile range (IQR) method, which calculates the bounds for outlier detection. Data points falling outside these bounds are identified as outliers and removed to ensure a more robust dataset.

```python
# Load the Wine dataset
wine = load_wine()
df = pd.DataFrame(wine.data, columns=wine.feature_names)
y = wine.target

# Remove outliers using the IQR method
# Calculate Q1 (25th percentile) and Q3 (75th percentile) for each column
Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1  # Interquartile Range

# Define lower and upper bounds
lower_bound = Q1 - 1.2 * IQR
upper_bound = Q3 + 1.2 * IQR

# Filter data within bounds
X_cleaned = df[(df >= lower_bound) & (df <= upper_bound)].dropna()

# Ensure target variable matches cleaned data
y_cleaned = y[X_cleaned.index]

# Print the resulting shapes of data
print("Original data shape:", df.shape)
print("Cleaned data shape:", X_cleaned.shape)
```

```
Original data shape: (178, 13)
Cleaned data shape: (138, 13)
```

I have adjusted the standard multiplier for the interquartile range (IQR) method from the typical value of 1.5 to 1.2. This stricter bound allows for the removal of a greater number of outliers, ensuring the dataset is better suited for kNN's distance-based calculations.





The boxplots above illustrate the feature ranges after applying outlier removal techniques

Train-Test Split

After removing the outliers, I will split the data into training and test sets. The training set will be used to train the machine learning models, enabling them to learn patterns, relationships, and decision boundaries from the data. The test set, on the other hand, is kept separate to evaluate the model's performance on unseen data, ensuring that the model generalizes well and does not overfit the training data. For this purpose, I utilized the train_test_split function from sklearn.model_selection, splitting the cleaned data into an 80%-20% ratio, with 80% of the data allocated for training and 20% for testing.

Feature Scaling

Next, I will scale the features to address the high variance observed in the dataset, particularly in features like proline and magnesium. Since this project involves a distance-based algorithm, I opted to use the Min-Max Scaler, which is particularly effective for such scenarios. The Min-Max Scaler normalizes feature values to a fixed range (0, 1), ensuring fair and consistent distance calculations. On the other hand, decision trees are unaffected by feature scaling, as they split data based on thresholds derived directly from feature values. Therefore, this scaling does not impact the decision tree's performance. To implement this, I used the MinMaxScaler function from sklearn.preprocessing to fit and transform the training data and then applied the same scaler to transform the test data.

```
# Split the cleaned data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_cleaned, y_cleaned, test_size=0.2, stratify=y_cleaned, random_state=36)

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the training data
X_train_scaled = scaler.fit_transform(X_train)

# Transform the test data using the same scaler
X_test_scaled = scaler.transform(X_test)

# Print the resulting shapes of data
print("Training set shape:", X_train_scaled.shape)
print("Test set shape:", X_test_scaled.shape)
```

```
Training set shape: (110, 13)
Test set shape: (28, 13)
```

Handling Correlated Features

The next step is to address the correlated features observed in the dataset, such as 'flavanoids' and 'total_phenols', as well as 'flavanoids' and 'od280/od315_of_diluted_wines'. To handle this, I have opted for Principal Component Analysis (PCA). This decision is based on an analysis of the correlation of each feature with the target variable, which revealed that all three features exhibit strong correlations with the target: 'flavanoids' (-0.847498), 'od280/od315_of_diluted_wines' (-0.788230), and 'total_phenols' (-0.719163). Dropping two of these features could risk losing valuable information that contributes to the model's performance so by employing PCA, I can retain essential information while effectively addressing the issue of correlated features, ensuring a more balanced and robust dataset for model training.

```python
df = pd.DataFrame(wine.data, columns=wine.feature_names)
df['target'] = wine.target

# Compute correlations
corr_with_target = df.corr()['target'].sort_values(ascending=False)
print("Correlation with Target:\n", corr_with_target)
```

```
Correlation with Target:
 target                          1.000000
alcalinity_of_ash                0.517859
nonflavanoid_phenols             0.489109
malic_acid                       0.437776
color_intensity                  0.265668
ash                             -0.049643
magnesium                       -0.209179
alcohol                         -0.328222
proanthocyanins                 -0.499130
hue                             -0.617369
proline                         -0.633717
total_phenols                   -0.719163
od280/od315_of_diluted_wines    -0.788230
flavanoids                      -0.847498
Name: target, dtype: float64
```

```python
# Initialize PCA to retain components explaining 95% of variance
pca = PCA(n_components=0.95)

# Fit PCA on the training set and transform both training and test sets
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

# Print the resulting shapes of PCA-transformed data
print("Training set after PCA shape:", X_train_pca.shape)
print("Test set after PCA shape:", X_test_pca.shape)
```

```
Training set after PCA shape: (110, 10)
Test set after PCA shape: (28, 10)
```

Algorithm Implementation

In this project, I implemented the k-Nearest Neighbors (kNN) from scratch and Decision Tree algorithms with the help of **DecisionTreeClassifier** from the scikit-learn library. These algorithms were chosen for their contrasting approaches to classification: kNN as a distance-based algorithm and Decision Trees as a rule-based algorithm. This section details the step-by-step implementation and highlights key considerations.

k-Nearest Neighbors (kNN)

Cross-Validation

For the kNN algorithm, k-fold cross-validation was conducted to determine the optimal value of k, the number of nearest neighbors considered for predictions. Since the kNN algorithm is

highly sensitive to parameter choices, particularly k, cross-validation helps identify the value that generalizes best to unseen data by evaluating performance across multiple folds of the dataset.

Cross-validation also mitigates the risks of bias and overfitting by averaging performance across multiple folds and testing the model on diverse subsets of data. This is important for kNN as its predictions are heavily influenced by data distribution and the presence of outliers. Additionally, cross-validation allows for a fair comparison of different models by evaluating their performance under consistent conditions

To achieve this, I implemented a custom function with the help of NumPy library which operates as follows:

1. The dataset is shuffled using **np.random.shuffle** to randomize the data points. A random seed is applied using **np.random.seed** to ensure reproducibility, so the results remain consistent across different runs.
2. The dataset is divided into k equally sized subsets (folds), where the size of each fold is calculated by dividing the total number of samples by the number of folds (k_folds).
3. During each iteration of the cross-validation process:
    - The current fold is designated as the validation set.
    - The remaining folds are combined into the training set using **np.concatenate**
4. The training and validation sets are passed to my custom kNN function, which uses the training data to predict the class labels for the validation set.
5. The predictions are evaluated using a custom **calculate_accuracy** function, which computes the accuracy as the ratio of correctly predicted labels to the total number of predictions.
6. The accuracy for each fold is stored in an array, and at the end of all iterations, the mean accuracy is computed using **np.mean** to provide a reliable performance metric for the kNN algorithm.

```python
def cross_validate_knn(X, y, k_neighbors=3, k_folds=5, random_seed=36):

    # Shuffle the data
    indices = np.arange(len(X))
    np.random.seed(random_seed)
    np.random.shuffle(indices)
    X = X[indices]
    y = y[indices]

    # Split data into folds
    fold_size = len(X) // k_folds
    accuracies = []

    for fold in range(k_folds):
        # Create training and validation splits
        start = fold * fold_size
        end = (fold + 1) * fold_size

        X_val = X[start:end] # Validation set features
        y_val = y[start:end] # Validation set labels

        # Combine the remaining folds into the training set
        X_train = np.concatenate([X[:start], X[end:]], axis=0)
        y_train = np.concatenate([y[:start], y[end:]], axis=0)

        # Predict using the custom kNN function
        y_pred = knn_algorithm(X_train, y_train, X_val, k=k_neighbors)

        # Evaluate accuracy
        accuracy = calculate_accuracy(y_val, y_pred)
        accuracies.append(accuracy)

    # Return the average accuracy across folds
    return np.mean(accuracies)
```

Implementation

For this project, I also implemented my own version of the kNN algorithm using the NumPy library. The algorithm operates as follows:

1. For each test point, I used the **np.sqrt** and **np.sum** functions to calculate the Euclidean distance between the test point and all training points.
2. The **np.argsort** function was then used to sort the distances in ascending order and retrieve the indices of the nearest neighbors. These indices were then used to fetch the corresponding labels of the training points.
3. Next, the **np.unique** function was applied to identify all unique labels among the nearest neighbors and count their occurrences.
4. The label with the highest count is determined using **np.argmax**, which retrieves the index of the maximum value. This label is considered the predicted class for the test point.

5. Finally, the predicted label is added to an array that stores predictions for all the test points.

```python
def knn_algorithm(X_train, y_train, X_test, k=3):
    predictions = []
    for test_point in X_test:
        # Calculate Euclidean distances
        distances = np.sqrt(np.sum((X_train - test_point) ** 2, axis=1))

        # Find the k-nearest neighbors
        k_indices = np.argsort(distances)[:k]
        k_labels = y_train[k_indices]

        # Get unique labels and their counts
        unique_labels, counts = np.unique(k_labels, return_counts=True)

        # Find the label with the maximum count
        most_frequent = unique_labels[np.argmax(counts)]

        # Append the most frequent label to predictions
        predictions.append(most_frequent)

    return np.array(predictions)
```

Model Training

Since kNN does not create an explicit model during the training phase, cross-validation was applied before implementation to determine the optimal value of k, which represents the number of nearest neighbors. Given the small size of the dataset, k-values ranging from 1 to 5 were tested. Using the custom cross_validate_knn function, the average accuracy for each k-value was calculated and the results alongside their corresponding cross-validation accuracies were displayed.

```python
# Test different values of k
k_values = range(1, 6)
results = []

# Iterate through each k value and find the average accuracy
for k in k_values:
    avg_accuracy = cross_validate_knn(X_train_scaled, y_train, k_neighbors=k, k_folds=5)
    results.append((k, avg_accuracy))
    print(f"k={k}, Cross-Validation Accuracy: {avg_accuracy * 100:.2f}%")

# Find the best k
best_k = max(results, key=lambda x: x[1])[0]
print(f"Best k: {best_k}")
```

```
k=1, Cross-Validation Accuracy: 96.36%
k=2, Cross-Validation Accuracy: 97.27%
k=3, Cross-Validation Accuracy: 98.18%
k=4, Cross-Validation Accuracy: 98.18%
k=5, Cross-Validation Accuracy: 97.27%
Best k: 3
```

After evaluating the cross-validation results, we observed that the k-value of 3 achieved the highest accuracy, with a score of 98.18%. Based on this, we can conclude that the optimal k-value for this dataset is 3.

Decision Tree

Cross-Validation

K-fold cross-validation was conducted for decision trees to address their tendency to overfit, especially when the depth of the tree is unconstrained. By evaluating the model across multiple folds of the training data, cross-validation helps prevent overfitting while ensuring the tree remains generalizable to unseen data. It also aids in selecting the optimal value for hyperparameters, such as max_depth, by testing different values and identifying the one that achieves the best average performance across folds.

To achieve this, I utilized the cross_val_score function from sklearn.model_selection to compute the cross-validation accuracy for each value of max_depth. The accuracies for all folds were averaged to determine the mean performance for each parameter value.

```python
# Range of max_depth values to test
max_depth_values = range(1, 6)
mean_accuracies = []

# Perform cross-validation for each max_depth value
for max_depth in max_depth_values:
    dt_model = DecisionTreeClassifier(max_depth=max_depth, random_state=36)
    cv_scores = cross_val_score(dt_model, X_train_pca, y_train, cv=5, scoring='accuracy')
    mean_accuracy = np.mean(cv_scores)
    mean_accuracies.append(mean_accuracy)
    print(f"max_depth={max_depth}, Cross-Validation Accuracy: {mean_accuracy * 100:.2f}%")

# Find the best max_depth
best_max_depth = max_depth_values[np.argmax(mean_accuracies)]
print(f"Best max_depth: {best_max_depth}")
```

```
max_depth=1, Cross-Validation Accuracy: 68.18%
max_depth=2, Cross-Validation Accuracy: 93.64%
max_depth=3, Cross-Validation Accuracy: 92.73%
max_depth=4, Cross-Validation Accuracy: 92.73%
max_depth=5, Cross-Validation Accuracy: 92.73%
Best max_depth: 2
```

From the results, we observed that the decision tree achieved the highest cross-validation accuracy of 93.64% when max_depth was set to 2. Thus, we can conclude that the optimal value for max_depth is 2.

Implementation

In this project, I implemented the Decision Tree algorithm using the “DecisionTreeClassifier” from the scikit-learn library, which leverages the CART (Classification and Regression Trees) approach. The classifier was initialized with the optimal value of max_depth, determined through cross-validation as 2, and Gini impurity was selected as the evaluation metric to measure the quality of splits at each node. Additionally, a random seed was set to ensure reproducibility of the model's results during the training and evaluation process.

```python
# Initialize the Decision Tree Classifier with the best max_depth
dt_model = DecisionTreeClassifier(max_depth=best_max_depth, criterion='gini', random_state=36)
```

Model Training

The model is then trained using the **fit()** method on the training data. During training, the algorithm begins at the root node, which contains all the training data. At each step, It evaluates all possible splits for the features and selects the split that minimizes Gini impurity. This process is recursively repeated at each decision node, progressively splitting the dataset into smaller subsets until a node becomes pure (contains data points of only one class), no further splits are possible, or the max_depth has been reached. The final result is a set of leaf nodes, where each leaf represents the predicted class for a subset of the data.

```
# Train the model on the training data
dt_model.fit(X_train_pca, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=2, random_state=36)
```

Results

In this section, I will continue making actual predictions after training both models and evaluate their performance by plotting confusion matrices, as well as calculating and comparing the accuracy and precision of their predictions.

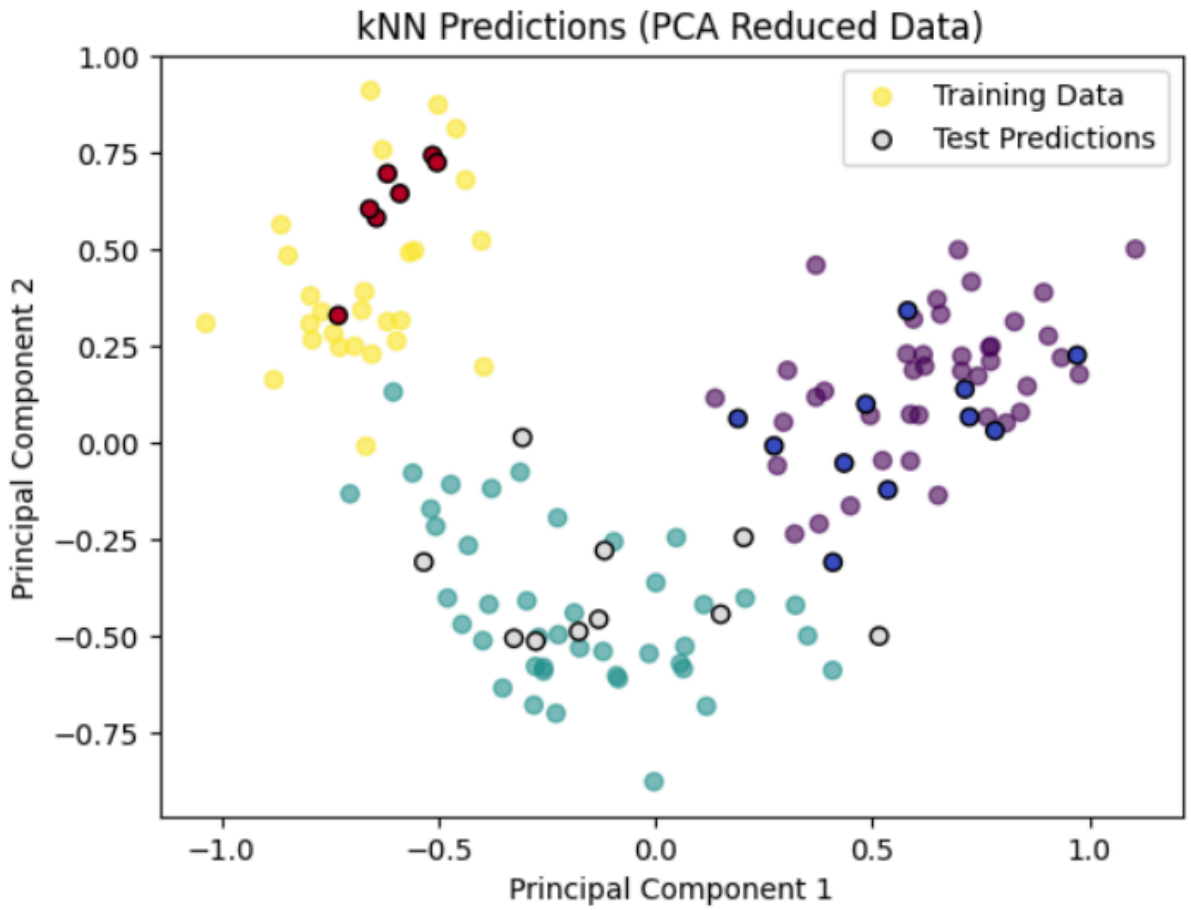
k-Nearest Neighbors (kNN)

After determining the optimal value of k, which is 3, through k-fold cross-validation, the next step is to evaluate the kNN model's performance on the test set. Using the best k value, I applied the custom **knn_algorithm** function to predict the class labels for the test data. The training data was used to identify the nearest neighbors for each test instance, and the corresponding labels were assigned based on majority voting.

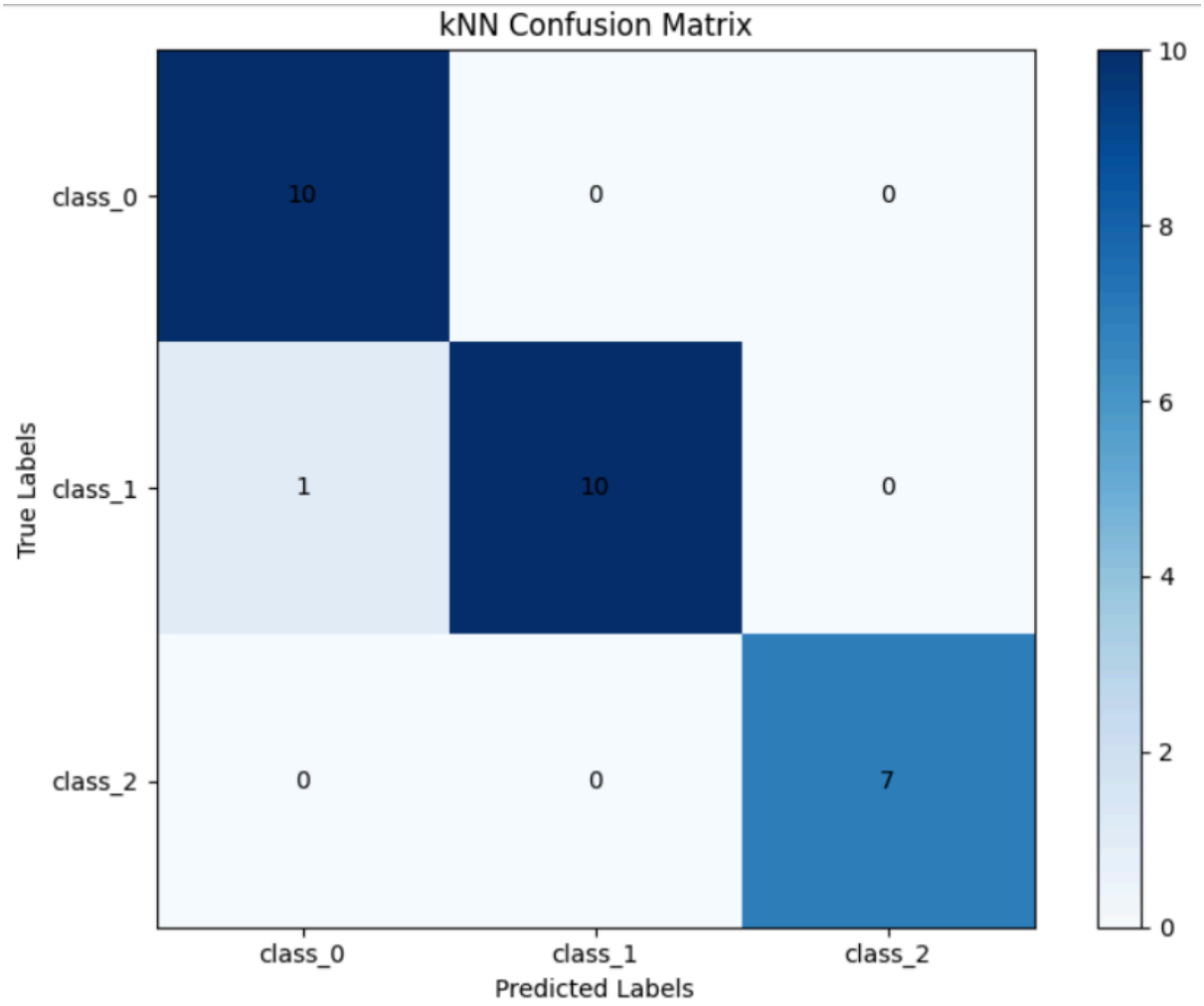
To measure performance, I utilized my custom calculate_accuracy function, which computes accuracy by dividing the number of correct predictions by the total number of predictions. The kNN algorithm achieved an accuracy of 96.43% on the test data.

```
# Train the kNN algorithm with the best k
final_predictions = knn_algorithm(X_train_pca, y_train, X_test_pca, k=best_k)

# Calculate accuracy for kNN predictions
accuracy = calculate_accuracy(y_test, final_predictions)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Accuracy: 96.43%
```



Additionally, I calculated the macro average precision using the confusion matrix. Precision for each class was computed as the ratio of true positives to the total predicted values for that class. By averaging the precision values across all classes, the overall macro-average precision was determined to be 96.97%, demonstrating strong predictive performance across all classes.




```
# True positives (diagonal elements)
tp_kkn = np.diagonal(kkn_cm)

# Total predicted values for all classes (TP + FP for all classes)
predicted_totals = np.sum(kkn_cm, axis=0)

# Overall Precision (Macro Average)
macro_average_precision = np.mean(tp_kkn / predicted_totals) * 100  # Convert to percentage

# Print overall precision
print(f"Overall Macro Average Precision: {macro_average_precision:.2f}%")
```

```
Overall Macro Average Precision: 96.97%
```

The high accuracy and precision values indicate that the kNN algorithm generalizes well to unseen data, confirming the effectiveness of the optimal k value determined during cross-validation to make accurate and reliable predictions.

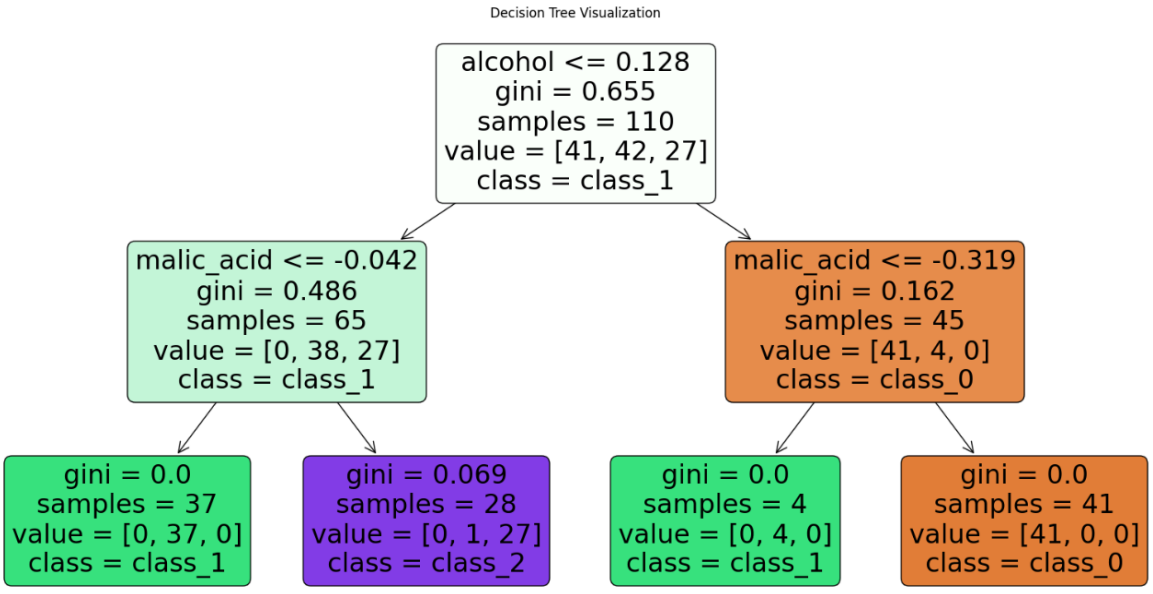
Decision Tree

After training, the decision tree model had learned a series of decision rules derived from the training data. These decision rules were then applied to classify new test points using the **predict()** method.

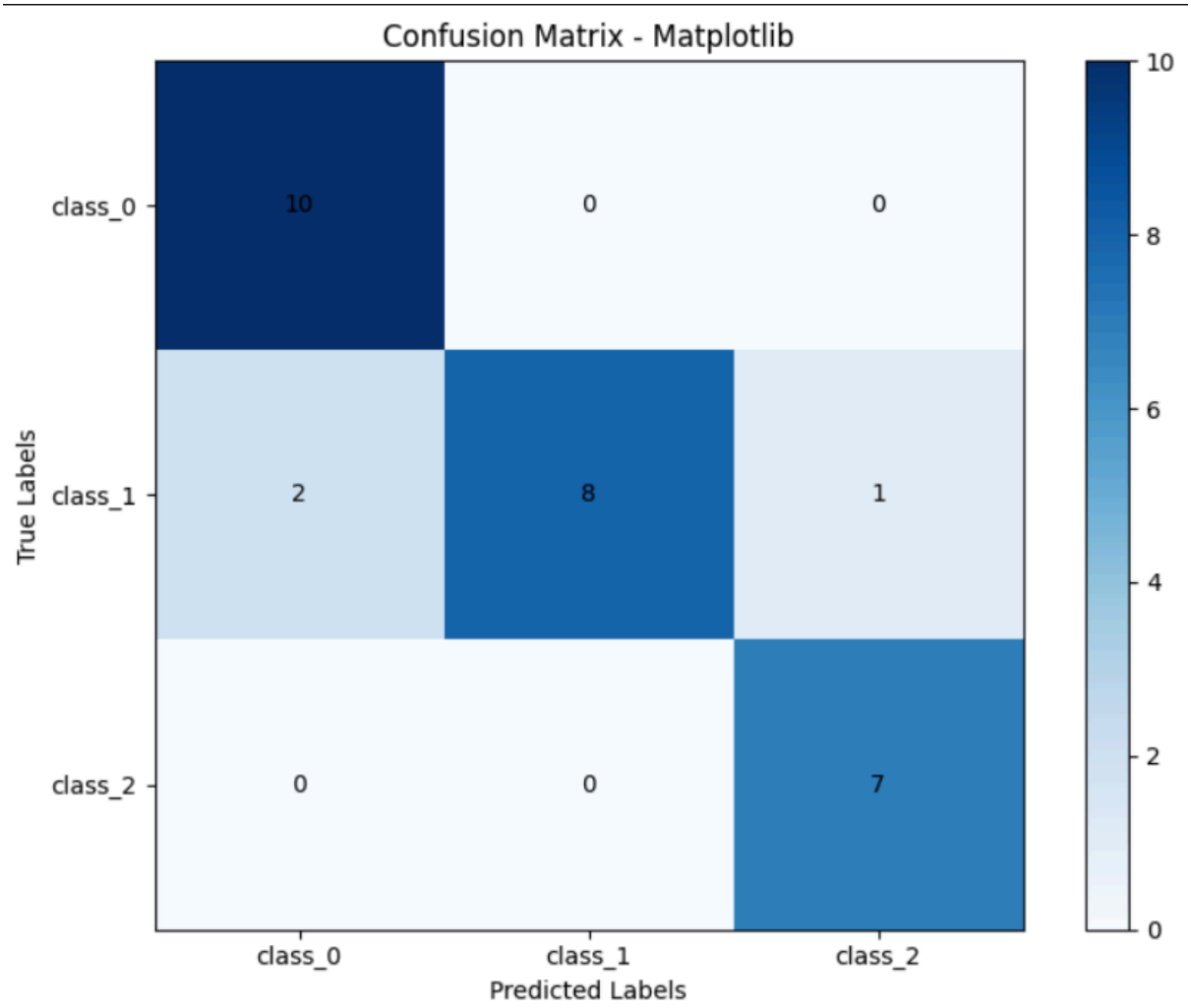
To evaluate its performance, I used my calculate_accuracy function, which determined the accuracy of the decision tree on the test set to be 89.29%. This result indicates the effectiveness of the decision tree model with the optimal value of max_depth.

```
# Predict the labels for the test set
y_pred = dt_model.predict(X_test_pca)

# Calculate accuracy for kNN predictions
accuracy = calculate_accuracy(y_test, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Accuracy: 89.29%
```



Additionally, using the confusion matrix, I calculated the macro average precision for the model. Precision for each class was computed as the ratio of true positives to predicted totals, and the overall macro-average precision was determined to be 90.28%.




```python
# True positives (diagonal elements)
tp_dt = np.diagonal(dt_cm)

# Total predicted values for all classes (TP + FP for all classes)
predicted_totals = np.sum(dt_cm, axis=0)

# Overall Precision (Macro Average)
macro_average_precision = np.mean(tp_dt / predicted_totals) * 100  # Convert to percentage

# Print overall precision
print(f"Overall Macro Average Precision: {macro_average_precision:.2f}%")
```

```
Overall Macro Average Precision: 90.28%
```

These metrics demonstrate that the decision tree model performs reliably, with solid precision across all classes, although it slightly trails the kNN model in overall accuracy and precision.

Summary

Algorithm	Test Accuracy (%)	Precision (%)	Cross-Validation Accuracy (%)
kNN	96.43	96.97	98.18
Decision Tree	89.29	90.28	93.64

From the table above, we observe that the kNN model outperforms the Decision Tree model in terms of test accuracy, cross-validation accuracy, and precision. This suggests that the kNN model demonstrates stronger predictive capability and accuracy when applied to unseen data. Additionally, both models significantly surpass the previously calculated baseline accuracy of 39.89%, emphasizing their successful implementation and effectiveness in classifying the dataset.

Evaluation

The evaluation section assesses the performance of the k-Nearest Neighbors (kNN) and Decision Tree algorithms based on the results obtained from the test data. Key metrics such as accuracy and precision are analyzed, while the strengths and weaknesses of each algorithm are critically examined. This evaluation also considers the impact of dataset characteristics and the preprocessing techniques employed, providing a comprehensive understanding of the algorithms' effectiveness.

Model Performance Comparison

From the summary above, it is evident that the kNN model outperforms the Decision Tree model for this dataset. The test accuracy, cross-validation accuracy, and precision all indicate superior performance by kNN compared to the Decision Tree algorithm. The cross-validation accuracy for kNN is 98.18%, compared to 93.64% of the Decision Tree, indicates that kNN generalizes better across different folds of the dataset. Similarly, the test accuracy for kNN is 96.43%, while the Decision Tree achieves a lower value of 89.29%, The close alignment between kNN's cross-validation and test accuracy demonstrates the model's consistent and robust performance across both validation and unseen data.

Additionally, the precision of the kNN algorithm is 96.97%, compared to 90.27% for the Decision Tree, highlighting kNN's stronger predictive accuracy for positive classifications. Analyzing precision per class reveals that kNN achieves 90.91% precision for class_0, 100.00% for class_1, and 100.00% for class_2. In contrast, the Decision Tree model achieves 83.33% precision for class_0, 100.00% for class_1, and 87.50% for class_2. These

results indicate that while both models perform well for class_1, kNN provides superior precision for the other classes, reinforcing its overall precision advantage.

Lastly, the small difference present between cross-validation accuracy and test accuracy for both models, with cross-validation accuracy being slightly higher, suggests that both models generalize well to unseen data without significant overfitting.

Analysis of Strengths and Weaknesses

The kNN algorithm, in addition to achieving high accuracy and precision, has proven its ability to generalize effectively to unseen data. It is also relatively simple to implement as it does not require complex mathematical models or a formal training phase. Instead, predictions are made at runtime by calculating distances between data points. However, this simplicity also introduces challenges. The kNN algorithm is highly sensitive to outliers, making thorough data preprocessing essential. In this project, removing outliers was critical to maintain its reliability. Additionally, since kNN relies on distance calculations, features with larger ranges, such as proline, could dominate the metric and lead to biased predictions. To address this, the MinMaxScaler was used to normalize feature ranges, ensuring that all features contributed proportionally. Another limitation of kNN is its computational expense, particularly for larger datasets, as it requires calculating distances to all training samples during prediction. While the Wine dataset is relatively small, this characteristic highlights potential scalability challenges.

In contrast, the Decision Tree algorithm is less sensitive to outliers and variations in features compared to the kNN algorithm. This is because its splits are determined by thresholds rather than distance calculations which allows the Decision Tree model to adapt more flexibly to the dataset’s characteristics. However, Decision Trees are prone to overfitting, especially when the tree depth is not constrained. In this project, this risk was mitigated by using k-fold cross-validation to determine the optimal max_depth, ensuring a balance between complexity and generalizability. Despite this, Decision Trees may show bias toward features with larger ranges, though this issue was minimized by scaling the data. Furthermore, correlated features can lead to redundant splits, reducing the model's efficiency and interpretability. To address this, PCA was applied to combine correlated features into principal components, resulting in a more robust dataset for training.

Impact of Dataset Characteristics

Since the Wine dataset is balanced, with all classes having a similar number of samples, it ensures that no single class dominates during the prediction process. This balance allows kNN to evaluate majority and minority classes fairly and simplifies the splitting process in Decision Trees, reducing the risk of predictions being biased toward more frequent classes. Additionally, the dataset's dimensionality is relatively low, which benefits kNN, as high-dimensional datasets can reduce the effectiveness of distance metrics.

With 178 samples and 13 features, the dataset's small size also favors kNN by minimizing the computational effort required for distance calculations. However, small datasets can

pose challenges for Decision Trees, as they may overfit by creating overly specific splits to fit the limited training data, demanding careful tuning. This dataset also contains outliers and correlated features, which can heavily impact kNN by distorting distance calculations and lead to redundant splits in Decision Trees. These challenges were addressed through outlier removal using the IQR method and PCA for correlated features, ensuring more reliable model performance.

Suitability for Multi-Class Classification

Both kNN and Decision Tree algorithms are inherently capable of handling multi-class classification tasks. kNN classifies test instances by evaluating the majority class among the nearest neighbors, while Decision Trees handle multi-class classification by hierarchically splitting data and assigning class labels to each leaf node. Decision Trees also offer the advantage of interpretability, making it easier to understand the classification rules, and are more robust against issues like feature scaling and outliers.

In this project, kNN demonstrated superior performance for multi-class classification compared to Decision Trees, as evidenced by higher test accuracy, precision, and cross-validation accuracy scores across all evaluation metrics. This indicates that kNN is slightly more effective for multi-class classification in this specific context. However, Decision Trees performed adequately, offering high interpretability and robustness to specific dataset characteristics, such as variations in feature scaling.

Both algorithms proved to be suitable for multi-class classification when paired with appropriate preprocessing techniques, such as outlier removal, scaling, and addressing correlated features. These preprocessing steps were essential to optimize the performance of both models on the Wine dataset.

Critical Reflection

Although this project provided valuable insights into the performance and suitability of two widely used algorithms for multi-class classification tasks, several aspects warrant further consideration. The balanced nature and relatively small size of the Wine dataset offered advantages, such as simplicity and reduced computational effort, but also limited the scope for testing the algorithms on more challenging datasets like those with significant class imbalances, higher dimensionality, or larger sample sizes.

Moreover, this project's scope was confined to a single dataset and a limited set of evaluation metrics. Expanding the study in the future to include larger, more complex, and diverse datasets, additional algorithms, and a broader range of evaluation metrics, such as recall, F1-score, or AUC. These enhancements could provide deeper insights and a more comprehensive understanding of algorithm performance and behavior across varied scenarios.

The preprocessing choices in this project were effective and ensured fair evaluations for both models. However, the use of PCA to handle correlated features, while suitable for kNN,

may not be ideal for Decision Trees, where preserving the interpretability of original features is crucial. Exploring alternative methods for addressing feature correlation, such as feature selection or domain-specific techniques, could further enhance model performance and applicability.

Conclusions

This project aimed to compare the performance of the k-Nearest Neighbors (kNN) and Decision Tree algorithms for multi-class classification tasks. The results show that kNN outperformed the Decision Tree model across all evaluated metrics, achieving 96.43% test accuracy, 98.18% cross-validation accuracy, and 96.97% precision. These metrics highlight kNN's strong predictive capability and its ability to generalize effectively to unseen data.

Based on these findings, we conclude that the kNN algorithm is particularly well-suited for multi-class classification tasks when supported by appropriate data preprocessing and careful parameter tuning. Its effective use of distance metrics, combined with its simplicity, makes it a reliable and robust choice for such tasks.

The Decision Tree model, while achieving slightly lower performance compared to kNN, attained 89.29% test accuracy, 90.28% cross-validation accuracy, and 93.64% precision. Despite these slightly lower metrics, it demonstrated notable strengths in interpretability and robustness to variations in feature scaling. Additionally, both models significantly surpassed the baseline accuracy of 39.89%, confirming their effectiveness in classifying the dataset.

The balanced nature and relatively small size of the Wine dataset contributed to the algorithms' success but also limited the scope of this analysis. Future studies could extend this work by exploring larger, more complex datasets and additional evaluation metrics, such as recall and F1-score, to provide a more comprehensive understanding of these algorithms.

Overall, this project has highlighted the suitability of kNN and Decision Tree algorithms for multi-class classification tasks, emphasizing the importance of data preprocessing and careful parameter tuning in achieving optimal performance.

References

1. GeeksforGeeks, 2023. *Wine dataset*. Available at: https://www.geeksforgeeks.org/wine-dataset/#characteristics-of-wine-dataset (Accessed 20 December 2024).
2. Elastic, 2023. *KNN distance metrics*. Available at: https://www.elastic.co/what-is/knn#4-types-of-computing-knn-distance-metrics (Accessed 21 December 2024).
3. IBM, 2023. *What is the KNN algorithm?*. Available at: https://www.ibm.com/think/topics/knn#:~:text=the%20KNN%20algorithm-,What%20is%20the%20KNN%20algorithm%3F,of%20an%20individual%20data%20point. (Accessed 23 December 2024).
4. Scikit-learn, 2023. *Tree module documentation*. Available at: https://scikit-learn.org/1.5/modules/tree.html (Accessed 21 December 2024).
5. IBM, 2023. *Decision trees*. Available at: https://www.ibm.com/think/topics/decision-trees (Accessed 21 December 2024).
6. Hoffman, K., 2021. *Machine Learning: How to handle class imbalance*. Available at: https://medium.com/analytics-vidhya/machine-learning-how-to-handle-class-imbalance (Accessed: 21 December 2024).
7. Iguazio, 2024. *What is a Baseline Model?* [online] Available at: https://www.iguazio.com/glossary/baseline-models/ (Accessed 21 December 2024).
8. Towards Data Science, 2023. *Calculating a baseline accuracy for a classification model*. Available at: https://towardsdatascience.com/calculating-a-baseline-accuracy-for-a-classification-model-a4b342ceb88f (Accessed 23 December 2024).
9. ProCogia, 2023. *Interquartile range method for reliable data analysis*. Available at: https://procogia.com/interquartile-range-method-for-reliable-data-analysis/ (Accessed 22 December 2024).
10. Towards Data Science, 2023. *Metrics to evaluate your machine learning algorithm*. Available at: https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234 [Accessed 22 December 2024].
11. Evidently AI, 2023. *Accuracy, precision, and recall*. Available at: https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall#:~:text=Accuracy%20is%20a%20metric%20that,the%20total%20number%20of%20predictions. (Accessed 23 December 2024).
12. Neptune.ai, 2023. *Balanced accuracy*. Available at: https://neptune.ai/blog/balanced-accuracy (Accessed 21 December 2024).
13. Machine Learning Mastery, 2023. *K-fold cross-validation*. Available at: https://machinelearningmastery.com/k-fold-cross-validation/ (Accessed 21 December 2024).
14. Towards Data Science, 2023. *Understanding confusion matrix*. Available at: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62 (Accessed 22 December 2024).
15. Abdallahashraf90x, 2023. *All you need to know about correlation for machine learning*. Medium. Available at:

https://medium.com/@abdallahashraf90x/all-you-need-to-know-about-correlation-for-machine-learning-e249fec292e9 (Accessed 23 December 2024)