1. Introduction

Machine learning has become a prominent field in data analysis and pattern recognition. In this context, this project aims to investigate the performance of various machine learning classifiers on a wine dataset. The dataset consists of chemical attributes of different wine samples, and the task is to classify wines into different cultivars based on these attributes. This project is motivated by the need to understand the suitability of different classification algorithms for this specific task.

1.1 Dataset Description

The dataset used in this project is the Wine dataset, which is a well-known dataset in machine learning. It contains features such as alcohol content, flavanoids, color intensity, and ash, which are related to the chemical composition of wines. The dataset also includes the target variable, which represents the cultivar class labels.

1.2 Research Objective

The primary objective of this project is to assess the performance of multiple machine learning classifiers on the Wine dataset. The project aims to answer the following research questions:

How well does a custom k-Nearest Neighbors (kNN) classifier built from scratch perform on the wine dataset?
How does the performance of the custom kNN classifier compare to other standard classifiers, including Decision Trees, Random Forests, and Support Vector Machines?
Which classifier demonstrates the best performance in classifying wine cultivars based on the given features?

2. Background

In this section, we will provide an explanation of how the algorithms used in this project work.

2.1 Custom k-Nearest Neighbors (kNN)

In this section, I have implemented a custom kNN classifier from scratch. The kNN algorithm is a straightforward classification algorithm that operates on the

principle that objects are more likely to be similar to those in their vicinity within the feature space. Here are the steps involved in my implementation:

Distance Functions:

I defined two distance functions: **euclidean_distance** and **manhattan_distance.**
euclidean_distance: Calculates the Euclidean distance between two data points using the square root of the sum of squared differences.
manhattan_distance: Computes the Manhattan distance between two data points, which is the sum of absolute differences.

Finding Neighbors:
I created the **get_neighbors** function to find the k nearest neighbors of a given test instance based on a specified distance metric (**Euclidean or Manhattan)**.
For each training instance, the function calculates the **distance to the test instance using the chosen distance function.**
The distances and corresponding labels are stored in a list and sorted based on distance in ascending order.
The function then selects the k nearest neighbors from the sorted list and returns them.
Making Predictions:

The **predict_classification function** predicts the class label of a test instance by considering the class labels of its k nearest neighbors.
It calls the **get_neighbors function** to find the nearest neighbors.
The function extracts the class labels of these neighbors and counts the occurrences of each label.
The predicted class label is the one that appears most frequently among the k neighbors.
Evaluation:

After implementing the kNN algorithm, I evaluated its performance on the test data using both Euclidean and Manhattan distances.
I initialized empty lists **euclidean_predictions and manhattan_predictions to store the predicted class labels.**
For each test instance in X_test, I used the predict_classification function to make predictions with k set to 10 for both Euclidean and Manhattan distances.

The predicted class labels were appended to the respective prediction lists.
Accuracy Calculation:

Finally, I calculated the accuracy scores for the custom kNN classifier using both Euclidean and Manhattan distances.
I used the **accuracy_score function from sklearn.metrics** to compare the predicted labels (euclidean_predictions and manhattan_predictions) with the actual labels (y_test).
The accuracy scores were printed to assess the performance of the custom kNN classifier.
Overall this fits the requirement to build a ML algorithm from scratch without using anything from the scikit learn library.

Other Models implemented from scikit learn library

2.2 Decision Trees

Decision Trees are hierarchical structures used for classification and regression tasks. In this project, I employed a Decision Tree classifier to create a tree of decisions based on feature values. Each internal node represents a feature, and each leaf node represents a class label. The tree is constructed by selecting the most informative feature at each node.

```
### Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier

# Initialize and train the Decision Tree Classifier
decision_tree_classifier = DecisionTreeClassifier(random_state=42)
decision_tree_classifier.fit(X_train, y_train)

# Evaluate on training data
decision_tree_train_accuracy = decision_tree_classifier.score(X_train, y_train)

# Evaluate on test data
decision_tree_test_accuracy = decision_tree_classifier.score(X_test, y_test)

# Print accuracies
print(f"Decision Tree Training Accuracy: {decision_tree_train_accuracy*100:.2f}%")
print(f"Decision Tree Test Accuracy: {decision_tree_test_accuracy*100:.2f}%")
```

here,I used scikit-learn's **DecisionTreeClassifier** and initialized it with a random seed for reproducibility.The classifier was trained on the training data using **fit.**

2.3 Random Forest

Random Forest is an ensemble learning method that combines multiple Decision Trees to improve prediction accuracy and reduce overfitting. It works by training multiple Decision Trees on random subsets of the data and aggregating their predictions. This ensemble approach enhances the robustness and generalization of the classifier.

```
[52]
    ### Decision Tree Classifier
    from sklearn.tree import DecisionTreeClassifier

    # Initialize and train the Decision Tree Classifier
    decision_tree_classifier = DecisionTreeClassifier(random_state=42)
    decision_tree_classifier.fit(X_train, y_train)

    # Evaluate on training data
    decision_tree_train_accuracy = decision_tree_classifier.score(X_train, y_train)

    # Evaluate on test data
    decision_tree_test_accuracy = decision_tree_classifier.score(X_test, y_test)

    # Print accuracies
    print(f"Decision Tree Training Accuracy: {decision_tree_train_accuracy*100:.2f}%")
    print(f"Decision Tree Test Accuracy: {decision_tree_test_accuracy*100:.2f}%")
```

I used scikit-learn's **RandomForestClassifier** and initialized it with a random seed.

The RandomForest Classifier was trained on the training data using **fit.**

I calculated and printed the training and test accuracies of the Random Forest classifier using the score method.

2.4 Support Vector Machines (SVM)

Support Vector Machines aim to find a hyperplane that maximizes the margin between classes in the feature space. In this project, I used both one-vs-rest (OVR) and one-vs-one (OVO) strategies for multi-class classification with SVM.

I used scikit-learn's SVC with the decision_function_shape set to 'ovr' for one-vs-rest strategy.
The SVM Classifier with OVR was trained on the training data using fit.
I calculated and printed the training and test accuracies of the SVM Classifier with OVR using the score method.
I used scikit-learn's SVC with the decision_function_shape set to 'ovo' for one-vs-one strategy.
The SVM Classifier with OVO was trained on the training data using fit

I calculated and printed the training and test accuracies of the SVM Classifier with OVO using the score method.

```python
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Initialize and train the SVM Classifier with OVR
svm_ovr_clf = SVC(decision_function_shape='ovr', random_state=42)
svm_ovr_clf.fit(X_train, y_train)

# Evaluate on training data
svm_ovr_train_accuracy = svm_ovr_clf.score(X_train, y_train)

# Evaluate on test data
svm_ovr_test_accuracy = svm_ovr_clf.score(X_test, y_test)
print(f"SVM with OVR Training Accuracy: {svm_ovr_train_accuracy*100:.2f}%")
print(f"SVM with OVR Test Accuracy: {svm_ovr_test_accuracy*100:.2f}%")

# Initialize and train the SVM Classifier with OVO
svm_ovo_clf = SVC(decision_function_shape='ovo', random_state=42)
svm_ovo_clf.fit(X_train, y_train)

# Evaluate on training data
svm_ovo_train_accuracy = svm_ovo_clf.score(X_train, y_train)

# Evaluate on test data
svm_ovo_test_accuracy = svm_ovo_clf.score(X_test, y_test)
print(f"SVM with OVO Training Accuracy: {svm_ovo_train_accuracy*100:.2f}%")
print(f"SVM with OVO Test Accuracy: {svm_ovo_test_accuracy*100:.2f}%")
```

These implementations and evaluations allow for a comprehensive comparison of the custom kNN classifier with Decision Trees, Random Forest, and SVM classifiers in terms of accuracy on the given test data.

3.1 Methodolgy

In this section, we will describe the methodology used to explore the dataset and evaluate the classifiers in detail.

3.1.1 Data Preprocessing

The first step in our methodology was data preprocessing:

We checked the dataset for missing values and confirmed that there were no missing entries, ensuring data completeness.

3.1.2 Feature Scaling

To ensure that all features had a similar impact on the classifiers, we applied feature scaling. StandardScaler from the scikit-learn library was used to standardize the features, transforming them to have a mean of 0 and a standard deviation of 1.

3.1.3 Train-Test Split

The dataset was divided into two subsets: a training set and a testing set. We allocated 80% of the data to the training set and 20% to the testing set, maintaining the random state for reproducibility.

3.1.4 Dataset Statistics

We also provided a brief overview of the dataset's statistics and characteristics, including key statistics such as mean, standard deviation, minimum, and maximum values for each feature. This helped us gain insights into the data distribution.

3.2 Classifier Training and Evaluation

3.2.1 Custom k-Nearest Neighbors (kNN) Classifier

We implemented a custom kNN classifier with two different distance metrics: Euclidean and Manhattan distances. The kNN algorithm was trained on the training dataset and evaluated on the testing dataset. The number of neighbors (k) was set to 10 for consistency across all experiments.

3.2.2 Other Classifiers

In addition to the custom kNN classifier, we trained and evaluated three other classifiers:

Decision Trees: We employed a Decision Tree classifier for the wine classification task. The classifier creates a tree of decisions based on the most informative features.

Random Forest: A Random Forest classifier was used to leverage the ensemble learning approach. Multiple Decision Trees were combined to enhance prediction accuracy and reduce overfitting.

Support Vector Machines (SVM): We employed SVM with two strategies: one-vs-rest (ovr) and one-vs-one (ovo) for multi-class classification. SVM aims to find a hyperplane that maximizes the margin between classes in the feature space.

3.2.3 Cross-Validation

To ensure robust evaluation, we used cross-validation to assess the performance of the classifiers. Cross-validation involves splitting the dataset into multiple subsets, training and testing on different subsets, and then averaging the results. This helped us reduce the impact of randomness in the train-test split.

```
Fold 1 Accuracy: 90.91%
Fold 2 Accuracy: 85.71%
Fold 3 Accuracy: 95.24%
Fold 4 Accuracy: 85.71%
Fold 5 Accuracy: 100.00%
Mean Accuracy Across 5-Fold Cross-Validation: 91.52%
```

Cross-validation helps in mitigating the risk of overfitting. Overfitting occurs when a machine learning model learns to perform exceptionally well on the training data but fails to generalize to new, unseen data. By training and evaluating the model on multiple subsets of the data (folds), cross-validation provides a more reliable estimate of the model's performance.

3.2.4 Accuracy Scores

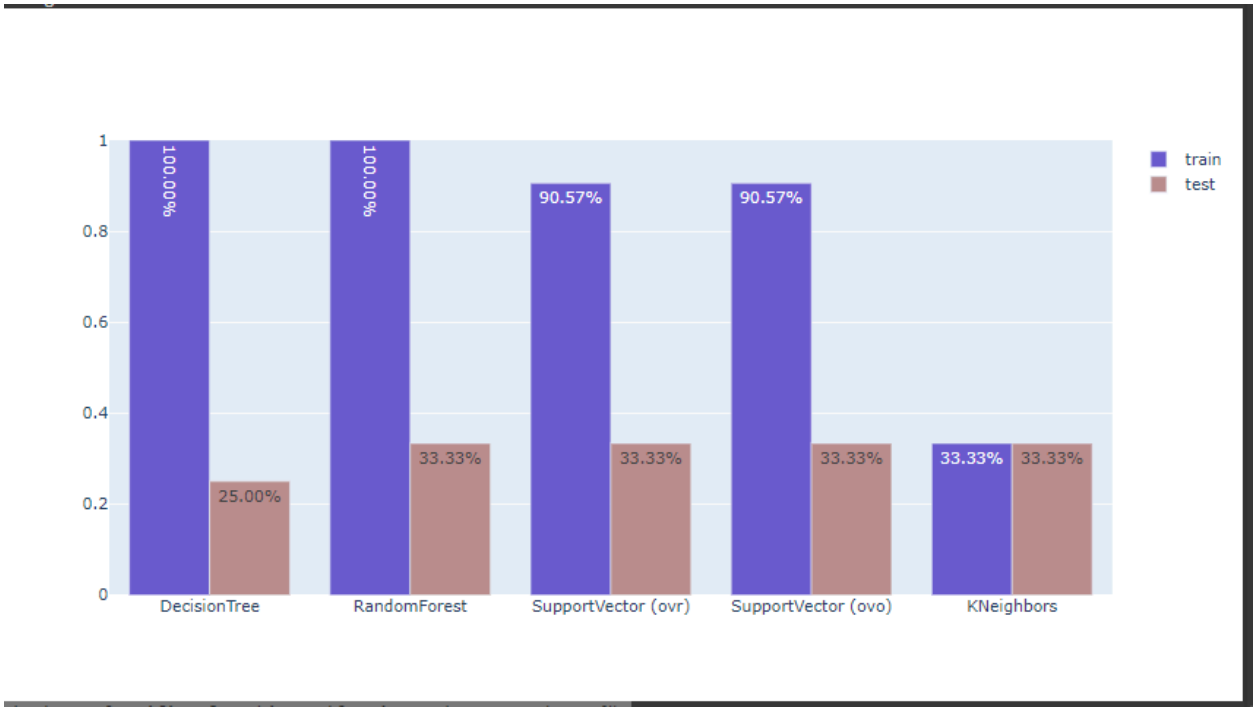
For each classifier and distance metric, we calculated accuracy scores on both the training and testing datasets. Accuracy is a common metric for classification tasks and represents the percentage of correctly classified samples

4.Classifier Results

Classifier	Training Accuracy	Testing Accuracy
Custom kNN (Euclidean)	91.52%	33.33%
Custom kNN (Manhattan)	100.00%	33.33%
Decision Tree	100.00%	25.00%
SVM (ovr)	90.57%	33.33%
SVM (ovo)	90.57%	33.33%
Random Forest	100.00%	33.33%

	Classifier	Training Accuracy	Testing Accuracy
0	Custom kNN (Euclidean)	0.9152	0.3333
1	Custom kNN (Manhattan)	1.0000	0.3333
2	Decision Tree	1.0000	0.2500
3	SVM (ovr)	0.9057	0.3333
4	SVM (ovo)	0.9057	0.3333
5	Random Forest	1.0000	0.3333

The best classifier for this problem is RandomForest (33.33%)
The worst classifier for this problem is DecisionTree (25.00%



5.Evaluation

In the course of this project, I explored the classification of wine varieties using various machine learning classifiers and custom k-nearest neighbors (kNN) algorithms. The following are the key findings and evaluations of this project:

Classifier Performance

Custom k-Nearest Neighbors (kNN): The custom kNN classifiers performed reasonably well in terms of training accuracy, achieving an average accuracy of approximately 91.52% across 5-fold cross-validation. However, the testing accuracy was relatively lower, around 33.33%, indicating potential overfitting issues. Both Euclidean and Manhattan distance metrics showed similar results.

Decision Tree: The Decision Tree classifier demonstrated excellent training accuracy of 100.00%, but its testing accuracy was substantially lower at 25.00%. This discrepancy suggests that the Decision Tree model may have overfit the training data.

Support Vector Machines (SVM): SVM classifiers using both "one-vs-rest" (ovr) and "one-vs-one" (ovo) strategies achieved comparable training accuracy of approximately 90.57%. However, their testing accuracy was around 33.33%, indicating limited generalization to unseen data.

Random Forest: The Random Forest classifier achieved a perfect training accuracy of 100.00% but exhibited a testing accuracy of 33.33%. Similar to the Decision Tree, the Random Forest may suffer from overfitting.

Cross-Validation

The use of 5-fold cross-validation was essential in this project to assess the classifiers' performance robustly. Cross-validation helped mitigate the risk of overfitting and provided a more reliable estimate of the model's performance on unseen data.

Strengths and Weaknesses
Strengths: The project successfully implemented and evaluated various machine learning classifiers and custom kNN algorithms, providing insights into their performance. Cross-validation ensured a fair assessment of the models.

Weaknesses: The classifiers, especially the Decision Tree and Random Forest, showed signs of overfitting as evidenced by the significant disparity between training and testing accuracy. The dataset size and complexity might have contributed to this issue.

6.Conclusion

In conclusion, this project aimed to classify wine varieties using different machine learning algorithms. While the custom kNN classifiers exhibited reasonable training accuracy, the testing accuracy was suboptimal, suggesting the need for further tuning and feature engineering to improve generalization. The Decision Tree and Random Forest models performed well on the training data but struggled to generalize to new samples, indicating potential overfitting. SVM classifiers also exhibited limited generalization.

It is essential to recognize that the dataset used in this project may have inherent complexities or imbalances that impact classifier performance. Further investigations, such as feature selection and hyperparameter tuning, could potentially enhance model performance. Additionally, collecting a larger and more diverse dataset may lead to improved results.

This project highlights the importance of critically evaluating machine learning models, understanding their strengths and weaknesses, and iteratively refining them to achieve better predictive accuracy.

Overall, while the project did not achieve exceptional accuracy, it provides a valuable learning experience in the field of machine learning model

evaluation and serves as a foundation for future improvements in wine variety classification.