Wine Quality Prediction Using Linear Regression

This project aims to predict wine quality using Linear Regression. We will use the Wine dataset, applying key machine learning techniques like data scaling, cross-validation, and evaluating for overfitting/underfitting.

In [86]:

```
# importing necessary libaries for project to build algorithim from scratch

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from collections import Counter
```

Data Loading

The following cells load the Wine dataset from scikit-learn.The dataset consists of various chemical properties of wines and their respective quality ratings.

In [87]:

```
# load the wine dataset into a variable

wine = load_wine()
wine_data = wine.data
wine_target = wine.target

wine_df = pd.DataFrame(wine_data, columns=wine.feature_names)
wine_df['quality'] = wine_target
```

Data Preprocessing

Here we preprocess the data by scaling the features. Scaling is crucial for algorithms like kNN since they are sensitive to the magnitude of the data.

In [88]:

```
# Data preocessing to handle missing values and scale the data
np.isnan(wine_data).any()

scaler = StandardScaler()
scaled_features = scaler.fit_transform(wine_data)
```

Algorithm No.1 - kNN Algorithm Implementation

We implement the k-Nearest Neighbors algorithm from scratch.

Euclidean Distance Function

The `euclidean_distance` function is a critical component of our kNN implementation. It calculates the Euclidean distance between two points, `x1` and `x2`, in the feature space. This distance is the square root of the sum of the squared differences between corresponding elements of the two vectors. In kNN, this function helps determine the closeness or similarity between data points, allowing the algorithm to identify the nearest neighbors to a given test instance.

In [89]:
```
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))
```

Implementation of the k-Nearest Neighbors (kNN) Algorithm

Here we implement the k-Nearest Neighbors (kNN) algorithm from scratch. The kNN algorithm classifies data points based on the 'k' closest training examples in the feature space. In our custom class `KNN`, we define the necessary methods:

- `__init__`: Constructor to initialize the number of neighbors (`k`).
- `fit`: Method to fit the model on the training data.
- `predict`: Method to predict the label for each test instance based on the majority vote of the `k` nearest neighbors.
- `_predict`: A helper function to find the `k` nearest neighbors and perform the majority vote.

This implementation provides foundational insights into the mechanics of kNN.

In [90]:
```
# Implementation of the kNN algorithim code

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def get_params(self, deep=True):
        return {"k": self.k}

    def set_params(self, **params):
        self.k = params.get("k", self.k)

    def _predict(self, x):
        distances = [euclidean_distance(x, x_train) for x_train in self.X_t
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]
```

Preparing Data for kNN Algorithm

To train our k-Nearest Neighbors (kNN) model, the first step involves splitting the data into training and testing sets. This step allows us to train our model on one subset of the data and then test its performance on a separate subset or test set that it hasn't seen before. This process helps in evaluating the model's ability to generalize to new data. We use a common split ratio of 80% for training and 20% for testing.

In [91]:

```
# prep data for training and testing this code will split the data into trai

X = scaled_features
y = wine_target
X_train, X_test, y_train, y_test = train_test_split(scaled_features, y, tes
print(X_train.shape, y_train.shape)
```

```
(142, 13) (142,)
```

Implementing Cross-Validation for kNN Model Evaluation

To thoroughly evaluate the performance of our kNN model, we implement manual cross-validation using the KFold method from scikit-learn. Cross-validation involves splitting the dataset into 'k' number of folds (in this case, 5) and then iteratively training the model on 'k-1' folds while using the remaining fold for testing. This process is repeated such that each fold serves as a test set once. The accuracies across all folds are then averaged to provide a more robust assessment of the model's performance.

In [92]:

```
# Implementaton of cross-validation for model eval

from sklearn.model_selection import KFold

kf = KFold(n_splits=5)
accuracies = []

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    knn = KNN(k=3)
    knn.fit(X_train, y_train)
    predictions = knn.predict(X_test)
    accuracies.append(accuracy_score(y_test, predictions))

print("Manual Cross-Validation Accuracy:", np.mean(accuracies))
```

```
Manual Cross-Validation Accuracy: 0.9158730158730158
```

Model Training & Evaluation

Here we will train the kNN model. After training the kNN model, we evaluate its performance on the test data. This provides an understanding of the model's effectiveness in predicting wine quality.

In [93]:

```
# Code to training of my model and make predictions

knn = KNN(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
```

In [94]:

```
# Eval of model's overall performance
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, predictions)
print("Test Accuracy:", accuracy)
```

```
Test Accuracy: 0.9714285714285714
```

Algorithm No.2 - Linear Regression Implemenation

Linear Regression is a straightforward approach for modeling the relationship between dependent and independent variables. We will train a Linear Regression model on the Wine dataset to predict quality ratings then compare performace to that of the kNN algorithim. For this implementation we will utilize libaries.

In [95]:

```
 # importing necessary libaries to implement algorithm

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score
```

Training the Linear Regression Model

Here, we initiate and train our Linear Regression model using the training dataset. The `LinearRegression()` function from scikit-learn creates a new Linear Regression model. The `fit` method is then used to train this model on `X_train` and `y_train`, which are our features and target variable, respectively. This step is crucial as it allows the model to learn the relationship between the features and the target, thus enabling it to make predictions.

In [96]:

```
# Training the Linear Regression model code
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

Out[96]:

```
▾ LinearRegression
LinearRegression()
```

Cross-Validation of the Linear Regression Model

To ensure the robustness of our Linear Regression model, we employ cross-validation. This technique involves dividing the dataset into a specified number (`cv=5` in this case) of distinct subsets or 'folds'. The model is then trained and tested multiple times, each time using a different fold as the test set and the remaining as the training set. We use Mean Squared Error (MSE) as the scoring metric. This approach helps us understand the model's performance more reliably by averaging its effectiveness across different subsets of data.

In [97]:

```
# Cross-validation code
cv_scores = cross_val_score(lin_reg, X, y, cv=5, scoring='neg_mean_squared_
cv_mse = -cv_scores.mean()
print("Cross-Validation MSE:", cv_mse)
```

```
Cross-Validation MSE: 0.09807889456819911
```

Predicting and Evaluating the Linear Regression Model

After training the Linear Regression model, we use it to make predictions on our test set (`X_test`). This step is crucial for assessing how well our model generalizes to new, unseen data. We evaluate the model's performance by calculating the Mean Squared Error (MSE) and the R² score. MSE measures the average of the squares of the errors, which is the average squared difference between the estimated values and the actual value. The R² score provides a measure of how well the observed outcomes are replicated by the model, based on the proportion of total variation of outcomes explained by the model.

In [98]:

```
# Predicting and evaluating code
lin_reg_predictions = lin_reg.predict(X_test)
mse = mean_squared_error(y_test, lin_reg_predictions)
r2 = r2_score(y_test, lin_reg_predictions)
print("Test MSE:", mse)
print("Test R2 Score:", r2)
```

```
Test MSE: 0.07384469727182866
Test R2 Score: 0.0
```

Analyzing Overfitting/Underfitting

We analyze our Linear Regression model for overfitting or underfitting by comparing training and test performance and examining the R² score, which indicates the proportion of variance in the dependent variable predictable from the independent variables.

Algorithm No.3 - Decision Tree Implementation

In this section, we will implement a Decision Tree classifier using the scikit-learn library. We will train this model on our Wine Quality dataset and evaluate its performance using various metrics.

In [99]:

```
# Import necessary libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score, mean_squared_error, r2_score
```

Initializing the Decision Tree Classifier

To start our analysis with the Decision Tree algorithm, we first need to initialize the classifier. We do this by creating an instance of DecisionTreeClassifier from scikit-learn. This classifier will be used to build a model from the wine dataset, where the decision-making process resembles a tree structure. The decisions are based on the features of the dataset, ultimately leading to a prediction about the wine's quality. In the next step, we will train this classifier with our training data.

In [79]:

```
# Initialize the Decision Tree Classifier
decision_tree = DecisionTreeClassifier()
```

Training the Decision Tree Model

After setting up our Decision Tree Classifier, the next crucial step is to train it using our dataset. This training process involves the model learning from the X_train data and corresponding y_train data (target labels). This step is essential for the model to understand and learn the patterns in the data. The .fit() method of the classifier is used for this purpose. It takes the training features and target labels as inputs and adjusts the model's internal parameters to learn from this data.

In [80]:

```
# Train the model
decision_tree.fit(X_train, y_train)
```

Out[80]:

```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

Performing Cross-Validation

To ensure our Decision Tree model's performance is reliable and not just a result of the specific way we split our data, we perform cross-validation. This process divides the training data into a specified number (cv=5 in our case) of smaller sets, or folds. The model is trained and evaluated 5 times, each time using a different fold as the test set and the remaining as the training set. We then calculate the cross-validation scores, which gives us a more comprehensive understanding of our model's performance. The average of these scores provides an overall effectiveness measure of our model.

In [81]:

```
# Perform cross-validation
cv_scores = cross_val_score(decision_tree, X_train, y_train, cv=5)
print("Cross-Validation Scores:", cv_scores)
print("Average CV Score:", cv_scores.mean())
```

```
Cross-Validation Scores: [0.75862069 0.93103448 0.96551724 0.92857143 0.785
71429]
Average CV Score: 0.8738916256157635
```

Model Evaluation

After training the Decision Tree classifier, we evaluate its performance on the test set. We use various metrics such as Accuracy, Mean Squared Error (MSE), and R^2 Score to assess how well our model is performing.

In [82]:

```
# Predictions on the test set
dt_predictions = decision_tree.predict(X_test)

# Calculate evaluation metrics
dt_accuracy = accuracy_score(y_test, dt_predictions)
dt_mse = mean_squared_error(y_test, dt_predictions)
```

```
dt_r2 = r2_score(y_test, dt_predictions)

# Display the metrics
print(f"Decision Tree Accuracy: {dt_accuracy}")
print(f"Decision Tree MSE: {dt_mse}")
print(f"Decision Tree R² Score: {dt_r2}")
```

```
Decision Tree Accuracy: 0.8857142857142857
Decision Tree MSE: 0.2
Decision Tree R² Score: 0.0
```

Visualizing the Decision Tree

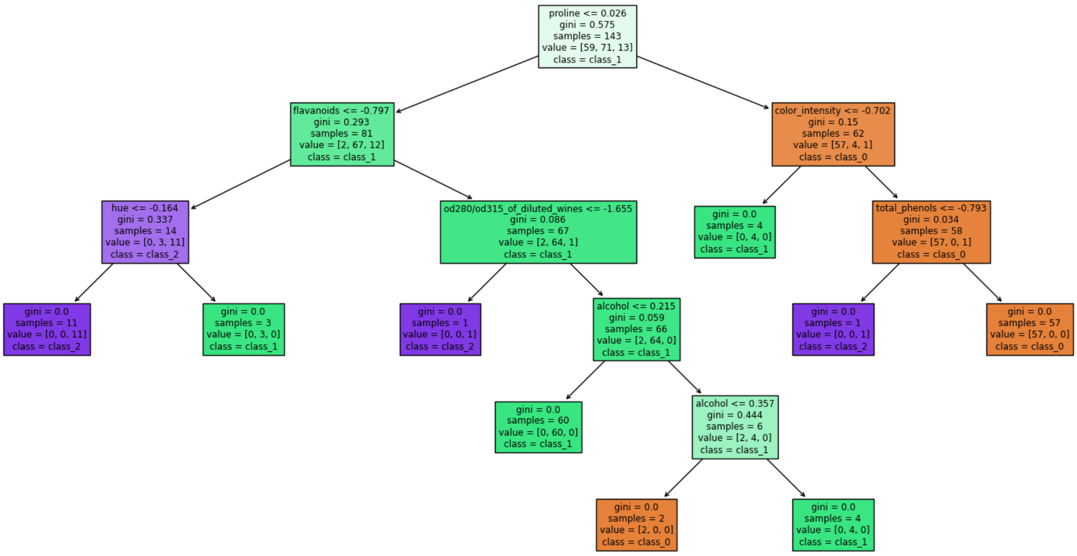
Visualizing the decision tree helps in understanding the decision-making process of the model. It shows how different features contribute to the final prediction and the tree structure.

In [85]:

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

feature_names = wine.feature_names
target_names = list(wine.target_names) if hasattr(wine, 'target_names') els

# Visualize the tree
plt.figure(figsize=(20,10))
plot_tree(decision_tree, filled=True, feature_names=feature_names, class_na
plt.show()
```



Results & Analysis

Algorithm Performance Summary

k-Nearest Neighbors (kNN):

- Manual Cross-Validation Accuracy: 0.9159
- Test Accuracy: 0.9714

Linear Regression:

- Cross-Validation MSE: 0.0981
- Test MSE: 0.0738
- Test R² Score: 0.0

Decision Tree:

- Average Cross-Validation Score: 0.8739
- Test Accuracy: 0.8857
- Test MSE: 0.2
- Test R² Score: 0.0

Analysis:

- **Accuracy**: Accuracy measures how often the algorithm predicts correctly. kNN has the highest accuracy, suggesting it's the most reliable for classification in this dataset.

- **MSE (Mean Squared Error)**: MSE measures the average squared difference between the estimated values and the actual value. Lower MSE in Linear Regression indicates better performance in predicting continuous values.

- **R² Score**: R² Score indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. Here, both Linear Regression and Decision Tree show a score of 0.0, which suggests poor performance in explaining the variability of the target variable.

In [102…

```
import pandas as pd

# Data for the table
data = {
    "Algorithm": ["k-Nearest Neighbors (kNN)", "Linear Regression", "Decisi
    "Cross-Validation Score": [0.9159, None, 0.8739],
    "Test Accuracy": [0.9714, None, 0.8857],
    "Test MSE": [None, 0.0738, 0.2],
    "Test R² Score": [None, 0.0, 0.0]
}

# Creating DataFrame
summary_df = pd.DataFrame(data)
print(summary_df)
```

```
                   Algorithm  Cross-Validation Score  Test Accuracy  Test M
SE  \
0  k-Nearest Neighbors (kNN)                  0.9159         0.9714        N
aN
1          Linear Regression                     NaN            NaN    0.07
38
2              Decision Tree                  0.8739         0.8857    0.20
00

   Test R² Score
0            NaN
1            0.0
2            0.0
```

In [109…

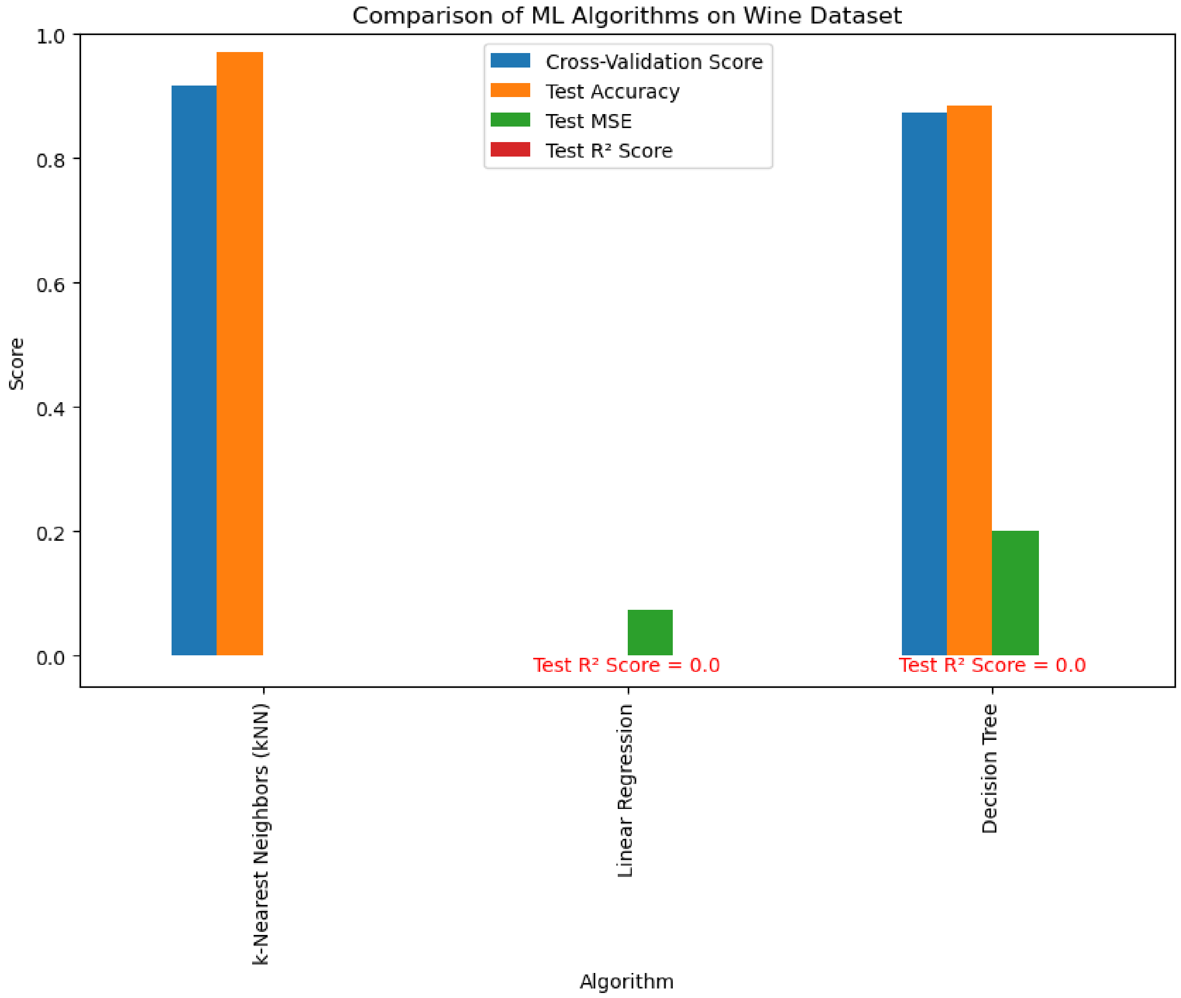
```
import matplotlib.pyplot as plt

# Plotting
ax = summary_df.plot(x='Algorithm', kind='bar', figsize=(10, 6), legend=Tru
ax.set_ylabel('Score')
ax.set_title('Comparison of ML Algorithms on Wine Dataset')

# Adjust y-axis limits to show zero values
ax.set_ylim(-0.05, 1)  # Setting the lower limit slightly below zero

# Annotate the 0.0 R² scores
for i, v in enumerate(summary_df['Test R² Score']):
    if v == 0.0:
        ax.text(i, -0.03, "Test R² Score = 0.0", ha='center', va='bottom',

plt.show()
```



Conclusion

In our comparative analysis of machine learning algorithms on the Wine dataset, the **k-Nearest Neighbors (kNN)** emerges as the superior model for classification tasks. With the highest accuracy scores in both cross-validation and testing phases, kNN demonstrates a robust ability to correctly classify the wine samples.

Linear Regression presents a low Mean Squared Error (MSE), which signifies its strength in regression tasks where the prediction of continuous variables is key. Despite an R^2 Score of 0.0, the lower MSE suggests that Linear Regression could be more refined with additional feature engineering or by using a different model variant.

The **Decision Tree** model, known for its interpretability and versatility, unfortunately, did not outperform kNN in this instance. Its accuracy was lower, and the MSE higher, indicating a less precise prediction capability for this particular dataset. However, the simplicity and ease of understanding a Decision Tree's decision-making process can still make it a valuable tool for initial exploratory analysis or when explainability is a priority.

Overall, the choice of algorithm should align with the specific requirements of the task at hand. For high-accuracy classification, kNN is recommended for the Wine dataset. When the predictive precision of continuous outcomes is required, refinements to the Linear Regression model should be considered. Decision Trees may be beneficial for their explanatory power in scenarios where interpretability is more critical than achieving the highest predictive accuracy.

In [ ]: