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 [ ]:
```
# 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 [59]:
```
# 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 [61]:
```
# 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 [62]:
```
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 [73]:
```
# 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 [74]:
```
# 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, test
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 [75]:
```
# 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 [76]:
```
# 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 [67]:
```
# 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 [69]:
```
 # 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 [70]:
```
# Training the Linear Regression model code
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

Out[70]:
```
▾ 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 [71]:
```
# Cross-validation code
cv_scores = cross_val_score(lin_reg, X, y, cv=5, scoring='neg_mean_squared_e
```

```
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 R^2 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 R^2 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 [72]:
```
# 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 R^2 score, which indicates the proportion of variance in the dependent variable predictable from the independent variables.