UNIVERSITY OF LONDON

Singapore Institute of Management

CM3015 Machine Learning and Neural Networks Midterm Coursework Report

Abstract

This project aims to classify wine quality using the UCI Wine dataset [1] by evaluating the performance of various machine learning models, including k-Nearest Neighbors (k-NN), Naive Bayes, Logistic Regression, and Decision Trees. The study involved both manual implementations of k-NN, Naive Bayes, and Logistic Regression with Softmax to gain deeper algorithmic insights, as well as Scikit-learn implementations for optimized performance. Additionally, Principal Component Analysis (PCA) was applied for dimensionality reduction through custom and Scikit-learn approaches to assess its impact on model generalization and computational efficiency. Models were evaluated using accuracy, precision, recall, and F1-score metrics. The findings highlight that Scikit-learn implementations consistently outperformed manual models in terms of efficiency and accuracy, while PCA improved generalization for Decision Trees without compromising overall performance across the evaluated models.

Introduction

Machine learning plays a pivotal role in solving real-world classification challenges, where accurately distinguishing between classes enables predictive analytics and informed decision-making. This project focuses on classifying wine quality using the Wine dataset from the Scikit-learn Machine Learning Repository. The dataset comprises 178 samples with 13 chemical attributes [1], including **alcohol content**, **malic acid**, and **flavonoids**, which serve as predictors for categorizing wines into three quality classes: **Class 0**, **Class 1**, and **Class 2**.

The aim of this project is to evaluate and compare the performance of machine learning models implemented manually with their Scikit-learn equivalents. Specifically, the project explores **k-Nearest Neighbors (k-NN)**, **Naive Bayes**, and **Logistic Regression with Softmax** manually and using sk-learn, and **Decision Trees** just with sk-learn. The manual implementations provide a deeper understanding of the mathematical operations and algorithmic mechanics behind these methods, while Scikit-learn's optimized models serve as robust tools designed for real-world applications. Furthermore, the project investigates the impact of **Principal Component Analysis (PCA)**, a widely used dimensionality reduction technique, on model performance. PCA reduces the number of features in a dataset while retaining the variance, simplifying models, improving computational efficiency, and addressing potential overfitting issues. Both a custom PCA implementation, developed through **eigen decomposition of the covariance matrix**, and Scikit-learn's PCA were applied to provide a comparative analysis.

The k-Nearest Neighbors algorithm was implemented manually to compute distances between data points, identify the nearest neighbors, and classify the target variable using **majority voting**. Naive Bayes was developed to model feature distributions using the **Gaussian likelihood function**, calculating class probabilities and making predictions accordingly. For Logistic Regression, a manual implementation employed the **softmax activation function** for multi-class classification, with gradient descent as the optimization technique. In addition to these custom methods, Scikit-learn models for k-NN, Naive Bayes, Logistic Regression, and Decision Trees were applied to the dataset, providing a benchmark for performance comparison.

To ensure a robust evaluation, key performance metrics including **accuracy, precision, recall, and F1-scores** were used. These metrics offer a comprehensive understanding of the effectiveness of the models, particularly in handling class imbalance challenges present in the dataset, where certain wine classes dominate. Class imbalance often introduces biases in predictions, requiring careful evaluation to validate model performance across all categories.

By integrating manual and Scikit-learn implementations, this project highlights the trade-offs between understanding algorithmic fundamentals and leveraging optimized tools for efficiency. Furthermore, the application of PCA demonstrates its significance in improving computational performance without considerable loss in accuracy. Through a structured analysis of multiple machine learning models, this study provides a thorough comparison of their performance while emphasizing the importance of dimensionality reduction techniques for complex and feature-rich datasets.

Background

k-Nearest Neighbors (k-NN)

The **k-Nearest Neighbors (k-NN)** algorithm is a non-parametric and instance-based machine learning method widely used for classification tasks. [2] Unlike many algorithms that build an explicit predictive model, k-NN works by memorizing the training data and predicting new samples based on similarity with existing data points. For this project, a custom implementation of k-NN was created alongside Scikit-learn's KNeighborsClassifier to provide a deeper understanding of its mechanics.

The manual implementation involves key steps such as **distance calculation, neighbor selection, and majority voting**. For each test sample, the **Euclidean distance** to all training points is computed. These distances are then sorted, and the labels of the k closest neighbors are retrieved. The most frequently occurring label among these neighbors determines the predicted class. By explicitly defining these steps, including sorting and label voting, the custom implementation provides insight into the inner workings of k-NN. In contrast, Scikit-learn automates these operations efficiently while delivering identical results.

Naive Bayes Classifier

The **Naive Bayes classifier** is a probabilistic machine learning algorithm based on **Bayes' theorem**, which calculates the probability of a class given a set of features. It is called "naive" because it assumes that all features are conditionally independent of each other given the class label. [3] For this project, Gaussian Naive Bayes was implemented, as wine dataset are continuous data , making it particularly suitable.

The performance of both the custom and Scikit-learn implementations was evaluated using key metrics such as accuracy, precision, recall, and F1-score. The evaluation process ensured that both approaches were comparable in their results, with minimal deviations observed. The consistency between the manual and library-based models highlights the correctness of the custom implementation and demonstrates its alignment with Scikit-learn's optimized version. The Naive Bayes classifier proved to be both computationally efficient and effective in handling the Wine dataset, making it a suitable model for multi-class classification problems.

Logistic Regression

Logistic Regression is a fundamental statistical and machine learning algorithm used for classification tasks. While it originates from regression models, it applies a transformation to predict discrete class labels rather than continuous outcomes. Logistic Regression is particularly suitable for binary and multi-class classification problems, where it estimates the probability of an input belonging to a specific class. In the case of multi-class classification, the algorithm is extended using techniques such as the softmax function, which normalizes output probabilities across all classes. [4]

Logistic Regression was applied to the Wine dataset because of its simplicity, interpretability, and suitability for multi-class classification problems. The dataset consists continuous chemical attributes, such as alcohol content, flavonoids, and malic acid, which can effectively serve as predictors for classifying wines into three quality classes. Logistic Regression works well with continuous data and does not require feature transformations beyond standardization, which makes it a strong candidate for this dataset. Moreover, its ability to assign probabilistic scores to class predictions allows for better evaluation and decision-making when handling classification tasks.

Decision Trees

Decision Trees are versatile and widely used machine learning algorithms for both classification and regression tasks. They are based on a tree-like structure where the data is split recursively according to specific conditions on feature values, resulting in branches and terminal nodes that represent class labels or outcomes. The intuitive nature of Decision Trees makes them highly interpretable, as the classification process resembles a series of logical "if-then" decisions. [5]

Decision Trees were chosen for the Wine dataset due to their ability to handle continuous numerical data, their flexibility in capturing non-linear relationships, and their inherent interpretability. By efficiently splitting the data at optimal thresholds, Decision Trees provide a powerful and transparent approach to solving multi-class classification problems, making

them a valuable tool for predicting wine quality and understanding the influence of chemical attributes on classification outcomes.

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a widely used dimensionality reduction technique that simplifies high-dimensional datasets while retaining as much information (variance) as possible. PCA works by transforming the original features into a new set of orthogonal variables called principal components, which are linear combinations of the original features. These components are ranked based on the amount of variance they explain in the dataset, with the first principal component accounting for the largest variance, the second for the next largest, and so on. By selecting a reduced number of principal components, PCA projects the data into a lower-dimensional space, reducing complexity while maintaining most of the data's informative structure. [6]

PCA was employed to reduce the Wine dataset's dimensionality and test its effect on the performance of k-NN and Decision Trees. By simplifying the feature space, PCA aims to improve model efficiency without significantly compromising accuracy. This analysis underscores the importance of dimensionality reduction for distance-based algorithms like k-NN and demonstrates how Decision Trees can benefit from a cleaner, more concise feature set.

Methodology

For my project, the wine dataset from the UCI Machine Learning Repository was used, and before building and training the machine learning models, **Exploratory Data Analysis (EDA)**, **Train-Test Split**.

Exploratory Data Analysis (EDA)

To thoroughly analyze and prepare the Wine dataset for machine learning tasks, an extensive Exploratory Data Analysis (EDA) was conducted. The EDA aimed to identify patterns, relationships among features, and potential issues like feature correlations or class imbalances. Furthermore, preprocessing steps such as feature scaling and data splitting were carried out to ensure the dataset was ready for model training and evaluation.

To gain deeper insights into the dataset, multiple visualizations were employed. Violin plots, box plots, pair plots, radar charts, and heatmaps were used to analyze class distributions and feature relationships, uncovering significant patterns and trends within the data. These visualizations highlighted critical features that contributed to differentiating between wine classes, while also exposing correlations between certain features.

After completing the EDA, the dataset was divided into training and testing subsets using the train_test_split function, with a 70/30 split ratio. This ensured that 70% of the data was used for training the models, while the remaining 30% was reserved for testing to assess the

models' ability to generalize to unseen data. A random state of 34 was specified to maintain consistency and reproducibility across multiple runs.

Train-Test Split

```
# Split the Dataset
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.3, random_state=34)

print(f"Training Features Shape: {X_train.shape}")
print(f"Testing Features Shape: {X_test.shape}")
```

A fundamental step in machine learning is splitting the dataset into training and testing subsets to assess a model's generalization performance on unseen data. The train-test split method was used for this purpose, dividing the data into 70% for training and 30% for testing. The training subset was used to fit and optimize the machine learning models, while the test set served to evaluate their predictive performance.

Splitting the Dataset

The goal of splitting the wine dataset is to evaluate the performance of the machine learning model on unseen data, ensuring its generalizability.

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.3, random_state=34)
```

The `train_test_split` function is used to divide the dataset into training and testing subsets, ensuring proper evaluation of the model's performance. The `X_scaled` variable represents the scaled feature matrix containing the input variables, while `y` is the target vector with class labels. The parameter `test_size=0.3` specifies that 30% of the dataset will be allocated to the test set, leaving 70% for the training set, providing sufficient data for both training and evaluation. The parameter `random_state=34` ensures reproducibility by fixing the randomization of data splitting, allowing consistent results with each execution. After splitting, the `X_train` and `y_train` subsets are used to train the machine learning model, while the `X_test` and `y_test` subsets are reserved to assess the model's performance on unseen data, validating its generalizability.

k-Nearest Neighbors (k-NN)

To evaluate the performance of the k-Nearest Neighbors (k-NN) classifier, I implemented the algorithm from scratch and validated it using cross-validation. Cross-validation was necessary to ensure that the custom model generalizes well to unseen data, as it systematically splits the dataset into multiple training and validation folds, providing a robust assessment of the model's predictive capabilities. [7]

Manual k-NN Classifier

```
# Custom k-NN Classifier
class KNNClassifier:
    def __init__(self, k=5):
        self.k = k
```

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

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

    def _predict(self, x):
        distances = np.linalg.norm(self.X_train - x, axis=1)
        k_indices = np.argsort(distances)[:self.k]
        k_labels = [self.y_train[i] for i in k_indices]
        return np.bincount(k_labels).argmax()

# Cross-validation for custom k-NN
def cross_validate_knn(X, y, k_values, folds=5):
    np.random.seed(10)
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    fold_size = len(indices) // folds

    results = {}

    for k in k_values:
        accuracies = []
        model = KNNClassifier(k=k)

        for fold in range(folds):
            val_start = fold * fold_size
            val_end = val_start + fold_size if fold != folds - 1 else
len(indices)
            val_indices = indices[val_start:val_end]
            train_indices = np.concatenate([indices[:val_start],
indices[val_end:]])

            X_train, X_val = X[train_indices], X[val_indices]
            y_train, y_val = y[train_indices], y[val_indices]

            model.fit(X_train, y_train)
            y_pred = model.predict(X_val)
            accuracy = np.mean(y_val == y_pred)
            accuracies.append(accuracy)

        results[k] = {
            'accuracy': np.mean(accuracies),
        }

    return results
```

Initialization

Firstly, I initializes the k value, which determines the number of neighbors to consider

```
def __init__(self, k=5):
    self.k = k
```

This parameter directly influences the model's sensitivity. Smaller `k` values make the model highly responsive to local variations, potentially leading to overfitting. Larger values smooth

out predictions, reducing sensitivity to noise but potentially missing finer details.

Training the Model (fit method):

Next, the training data (X_train and y_train) is stored for reference during prediction

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

Unlike algorithms that build a predictive model during training, k-NN will retain the entire dataset for future comparisons.

Predicting Labels (predict and _predict methods):

For each input sample in the test set, the distances to all training samples are calculated using Euclidean distance

```
distances = np.linalg.norm(self.X_train - x, axis=1)
```

The `np.linalg.norm` function calculates the norm (length) of each row vector in the difference matrix. Setting axis=1 ensures that the norm is computed for each row (corresponding to a training data point).

The `_predict` method sorts these distances and retrieves the k smallest ones

```
k_indices = np.argsort(distances)[:self.k]
k_labels = [self.y_train[i] for i in k_indices]
```

Finally, the label with the highest frequency (majority vote) among the k neighbors is chosen

```
return np.bincount(k_labels).argmax()
```

By explicitly defining the distance calculation and majority voting, the implementation showcase a hands-on understanding of the algorithm. The modular design (separate `fit` and `predict` methods) makes the code reusable and adaptable.

Cross-Validation for Custom K-NN

To evaluate the performance of my custom k-NN implementation across different k values, I used cross-validation. This ensures that the model's performance generalizes well across unseen data.

Splitting the Dataset

The dataset is shuffled and divided into `folds`

```
indices = np.arange(X.shape[0])
np.random.shuffle(indices)
fold_size = len(indices) // folds
```

This ensures that all samples are used for both training and validation across iterations.

Training and Validation

For each fold, the validation set is defined, and the remaining folds are used for training

```
val_indices = indices[val_start:val_end]
train_indices = np.concatenate([indices[:val_start], indices[val_end:]])
X_train, X_val = X[train_indices], X[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
```

The `fit` and `predict` methods of the k-NN implementation are called to train and evaluate the model

```
model.fit(X_train, y_train)
y_pred = model.predict(X_val)
```

Evaluating Accuracy:

The accuracy for each fold is calculated and stored

```
accuracy = np.mean(y_val == y_pred)
accuracies.append(accuracy)
```

Aggregating Results:

After evaluating all folds, the average accuracy for each value of k is stored

```
results[k] = {
    'accuracy': np.mean(accuracies),
}
```

Manual calculations of evaluation metrics

I implemented functions manual calculations of evaluation metrics for multi-class classification problems. Each metric serves a distinct purpose in measuring the performance of a classification model.

manual_accuracy_score

The manual_accuracy_score function calculates the accuracy of a classification model, which is the ratio of correctly predicted instances to the total number of instances. The function iterates through the true labels (`y_true`) and predicted labels (`y_pred`) using the `zip` function, comparing each pair. For every match (correct prediction), the counter increments. The sum of these correct predictions is then divided by the total number of samples (`len(y_true)`) to compute the accuracy. This metric provides a general measure of how often the model correctly classifies, but it does not account for imbalanced datasets, where accuracy alone may not provide a full picture of the model's performance.

```
correct_predictions = sum(y_t == y_p for y_t, y_p in zip(y_true,
y_pred))
return correct_predictions / len(y_true)
```

manual_precision_score

I use it to calculate precision for multi-class classification. It finds the fraction of correctly predicted instances among all instances predicted as a given class.

The manual_precision_score function calculates the precision, which measures the proportion of correctly predicted positive instances among all instances predicted as positive. The function first identifies the unique class labels from the true labels (`y_true`).

For each class, it computes the number of true positives (TP)—instances where the model correctly predicts the class—and the total number of predicted positives, which includes both correct and incorrect predictions for that class.

The function supports two averaging methods: weighted average, where precision values are weighted by the number of true instances (support) for each class, and macro average, which calculates the unweighted mean of precision values across all classes.

```
precision = true_positive / predicted_positive if predicted_positive > 0
else 0
if average == 'weighted':
    precisions.append(precision * support)
else:
    precisions.append(precision)
```

manual_recall_score

The manual_recall_score function calculates the recall, which quantifies the proportion of actual positive instances correctly identified by the model. Similar to the precision calculation, the function iterates through all class labels. For each class, it determines the number of true positives (TP) and the total number of actual positives, which includes both correctly and incorrectly classified instances of that class.

As with precision, the function supports weighted and macro averaging. Weighted averaging takes into account the class distribution, whereas macro averaging treats all classes equally. Recall is critical in applications where it is important to minimize false negatives, such as disease detection or identifying defective products in manufacturing.

```
recall = true_positive / actual_positive if actual_positive > 0 else 0
if average == 'weighted':
    recalls.append(recall * support)
else:
    recalls.append(recall)
```

manual_f1_score

The `manual_f1_score` function calculates the F1 score, which is the harmonic mean of precision and recall. The F1 score provides a balance between these two metrics, making it especially useful in scenarios where precision and recall have an inverse relationship. For each class, precision and recall are calculated using the logic described in the `manual_precision_score` and `manual_recall_score` functions.

```
f1 = 2 * (precision * recall) / (precision + recall) if (precision +
recall) > 0 else 0
if average == 'weighted':
    f1_scores.append(f1 * support)
else:
    f1_scores.append(f1)
```

The function handles cases where both precision and recall are zero to avoid division by zero errors. Like the other metrics, it supports weighted and macro averaging methods. The F1 score is particularly valuable in imbalanced datasets where focusing solely on precision or recall might be misleading.

Evaluation of manual (build from scratch) and sklearn k-Nearest Neighbors (k-NN) classifier

Manual k-Nearest Neighbors

Choosing range of k to calculate

```
k_values = range(1, 16)
manual_test_manually = {}  # Initialize the dictionary to store results

for k in k_values:
    manual_knn = KNNClassifier(k=k)
    manual_knn.fit(X_train, y_train)
    y_test_pred_manual = manual_knn.predict(X_test)
```

Here, I use k_values to define the range of k values to evaluate. A KNNClassifier instance is created for each k, and the training data (`X_train` , `y_train`) is stored using the fit method. The model predicts labels for the test data (`X_test`) using the predict method, which calculates distances between the test sample and training data, identifies the k nearest neighbors, and selects the majority class among these neighbors.

Storing and Printing Metrics

```
manual_test_manually[k] = {
    'accuracy': manual_accuracy_score(y_test, y_test_pred_manual),
    'precision': manual_precision_score(y_test, y_test_pred_manual,
average='weighted'),
    'recall': manual_recall_score(y_test, y_test_pred_manual,
average='weighted'),
    'f1': manual_f1_score(y_test, y_test_pred_manual,
average='weighted')
}
```

The metrics are stored in a dictionary `manual_test_manually` , with k as the key.

Output

k	Accuracy	Precision	Recall	F1 Score
1	0.94	0.94	0.94	0.94
2	0.93	0.93	0.93	0.92
3	0.96	0.97	0.96	0.96
4	0.94	0.94	0.94	0.94
5	0.96	0.97	0.96	0.96
6	0.94	0.95	0.94	0.94
7	0.94	0.95	0.94	0.94
8	0.93	0.93	0.93	0.92
9	0.93	0.94	0.93	0.92
10	0.96	0.97	0.96	0.96
11	0.94	0.95	0.94	0.94
12	0.94	0.95	0.94	0.94

k	Accuracy	Precision	Recall	F1 Score
13	0.94	0.95	0.95	0.94
14	0.94	0.95	0.95	0.94
15	0.94	0.95	0.94	0.94

SK-Learn k-Nearest Neighbors

Initialization and Training

For each k, a `KNeighborsClassifier` is initialized with `n_neighbors=k` and trained on the training set

```
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
```

Prediction and Metric Calculation

The model predicts the labels for the test set (`X_test`), and the predictions are compared to the ground truth (`y_test`) using Scikit-learn metrics

```
y_test_pred_sklearn = knn.predict(X_test)
sklearn_results[k] = {
    'Accuracy': accuracy_score(y_test, y_test_pred_sklearn),
    'Precision': precision_score(y_test, y_test_pred_sklearn,
average='weighted', zero_division=0),
    'Recall': recall_score(y_test, y_test_pred_sklearn,
average='weighted', zero_division=0),
    'F1 Score': f1_score(y_test, y_test_pred_sklearn,
average='weighted', zero_division=0)
}
```

Output

k	Accuracy	Precision	Recall	F1 Score
1	0.94	0.94	0.94	0.94
2	0.93	0.93	0.93	0.92
3	0.96	0.97	0.96	0.96
4	0.94	0.94	0.94	0.94
5	0.96	0.97	0.96	0.96
6	0.94	0.95	0.94	0.94
7	0.94	0.95	0.94	0.94
8	0.93	0.93	0.93	0.92
9	0.93	0.94	0.93	0.92
10	0.96	0.97	0.96	0.96
11	0.94	0.95	0.94	0.94
12	0.94	0.95	0.94	0.94
13	0.94	0.95	0.95	0.94

k	Accuracy	Precision	Recall	F1 Score
14	0.94	0.95	0.95	0.94
15	0.94	0.95	0.94	0.94

Comparison between Manual and Sk-learn Implementation

Aspect	Manual k-NN Implementation	Scikit-learn k-NN Implementation
Distance Calculation	Explicitly implemented using `np.linalg.norm`.	Automatically handled within Scikit-learn.
Majority Voting	Implemented manually using `np.bincount`.	Built-in functionality in Scikit-learn.
Metric Calculation	Custom functions for accuracy, precision, recall, and F1 score.	Uses Scikit-learn's prebuilt metric functions.
Code Reusability	Modular but less flexible.	Highly modular and optimized for reuse.

Output results

By referring to the two output above. We can tell that the manual k-NN and Scikit-learn k-NN both yield the same results across accuracy, precision, recall, and F1 score when rounded to 2 decimal places. This demonstrates the correctness of my manual implementation in reproducing the expected results from Scikit-learn's optimized version. While Scikit-learn abstracts the implementation details, it provides the same performance metrics, validating the consistency between both approaches.

Summary

From a performance perspective, both implementations exhibited excellent generalization ability, achieving consistent high accuracy (up to 96% for optimal `k-values`) and balanced precision, recall, and F1 scores across the dataset. These results highlight the robustness of the k-NN algorithm in handling this particular dataset. Ultimately, while the manual implementation is beneficial for educational purposes and algorithmic understanding, Scikit-learn's version stands out for its efficiency and reusability in real-world applications.

PCA on Wine dataset

Using PCA for the Wine dataset improves computational efficiency, reduces feature redundancy, and allows for effective data visualization while maintaining good performance. It is particularly useful for tasks requiring a balance between simplicity, interpretability, and model performance. [8]

Performance Metrics - Evaluation with Custom PCA

To evaluate and compare the performance of the k-Nearest Neighbors (k-NN) classifier, I implemented Custom Principal Component Analysis (PCA), a dimensionality reduction technique that identifies key features while preserving as much variance as possible in the

dataset. The Custom PCA implementation involves centering the data, computing the covariance matrix, performing eigen decomposition, and projecting the data onto the principal components.

PCA from Scratch Implementation

```
class CustomPCA:
    def __init__(self, n_components=None):
        self.n_components = n_components

    def fit(self, X):
        # Required Dimensions
        if self.n_components is None:
            self.n_components = X.shape[1]

        # Centering Data
        self.mean = np.mean(X, axis=0)
        Xm = X - self.mean

        # Covariance Matrix
        covariance_matrix = np.cov(Xm, rowvar=False)

        # Eigenvalues and Eigenvectors
        eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)

        # Sort Eigenvalues and Eigenvectors in Descending Order
        indices = np.argsort(eigenvalues)[::-1]
        self.explained_variance =
eigenvalues[indices[:self.n_components]]
        self.components = eigenvectors[:, indices[:self.n_components]]

        # Explained Variance Ratio
        self.explained_variance_ratio_ = self.explained_variance /
np.sum(eigenvalues)

    def transform(self, Q):
        # Project Data
        Qm = Q - self.mean  # Centering
        return np.dot(Qm, self.components)  # Matrix multiplication

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

Centering the Data:

The mean of each feature is calculated and subtracted from the dataset to center the data around the origin, ensuring the principal components reflect the true variance.

```
self.mean = np.mean(X, axis=0)
Xm = X - self.mean
```

Covariance Matrix Calculation:

The covariance matrix is computed to capture the relationships (variances and covariances) between features. This matrix is used to determine the directions of maximum variance in

the data.

```
covariance_matrix = np.cov(Xm, rowvar=False)
```

Eigen Decomposition

Eigenvalues and eigenvectors of the covariance matrix are computed using `np.linalg.eigh`. The eigenvalues represent the variance explained by each component, while eigenvectors represent the directions (principal components).

```
eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
```

Sorting and Selecting Components:

Eigenvalues and their corresponding eigenvectors are sorted in descending order. The top n-components are selected based on the user-defined number of components (`n_components`). The explained variance ratio is also calculated.

```
indices = np.argsort(eigenvalues)[::-1]
self.explained_variance = eigenvalues[indices[:self.n_components]]
self.components = eigenvectors[:, indices[:self.n_components]]
self.explained_variance_ratio_ = self.explained_variance /
np.sum(eigenvalues)
```

Projecting Data:

The centered data is projected onto the principal components, reducing its dimensionality.

```
Qm = Q - self.mean  # Centering the test data
return np.dot(Qm, self.components)  # Matrix multiplication for
projection
```

Performance Metrics - Comparison Between Original Data, Custom PCA, and Sklearn PCA

Sklearn PCA Transformation:

Scikit-learn's PCA was used for comparison, similarly reducing the data to 2 components.

```
sklearn_pca = PCA(n_components=2)
X_train_sklearn_pca = sklearn_pca.fit_transform(X_train)
X_test_sklearn_pca = sklearn_pca.transform(X_test)
```

Training and Evaluating the k-NN Classifier:

For each k value from 1 to 15, the k-NN classifier was trained and evaluated on.

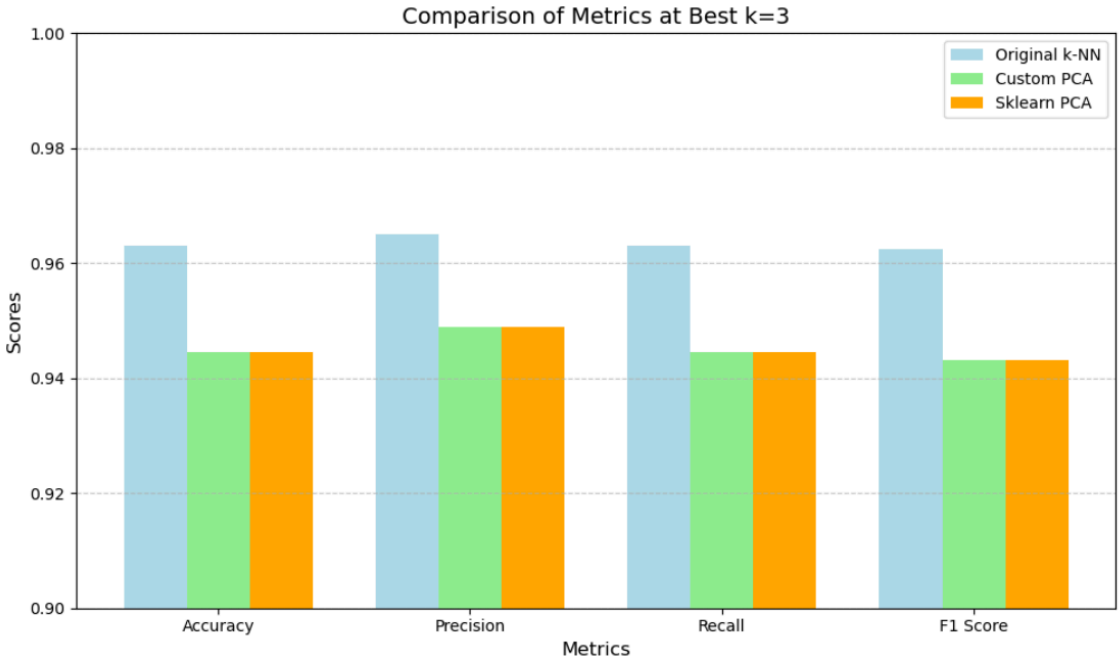
```
accuracy_results_original[k] = accuracy_score(y_test, y_pred_original)
accuracy_results_custom[k] = accuracy_score(y_test, y_pred_custom)
accuracy_results_sklearn[k] = accuracy_score(y_test, y_pred_sklearn)
```

Output

Metric	Original k-NN	Custom PCA	Sklearn PCA
Accuracy	0.96	0.94	0.94

Metric	Original k-NN	Custom PCA	Sklearn PCA
Precision	0.97	0.95	0.95
Recall	0.96	0.94	0.94
F1 Score	0.96	0.94	0.94

Visualisation



Summary

The Original k-NN outperforms both Custom PCA and Sklearn PCA across all metrics. Although PCA simplifies the data by reducing dimensions, this simplification results in a slight decline in performance metrics. Despite this, PCA provides computational benefits by reducing the feature space, making it useful for larger datasets where training time and efficiency are crucial. The consistency between Custom PCA and Sklearn PCA confirms the correctness of the manual implementation.

Naive Bayes Classifier

The Naive Bayes classifier was chosen for the Wine dataset due to its simplicity, computational efficiency, and Gaussian assumption, which aligns well with the continuous features of the dataset. It is particularly effective for multi-class classification tasks, as it models each class's feature distributions independently while remaining interpretable. [9]

Naive Bayes (Custom Implementation)

```
class NaiveBayesClassifier:
    def __init__(self):
        self.class_priors = {}  # Prior probabilities of each class
        self.feature_stats = {}  # Mean and variance for each feature by
class
```

```
    def fit(self, X, y):
        """
        Train the Naive Bayes classifier.
        """
        self.classes = np.unique(y)
        self.class_priors = {c: np.mean(y == c) for c in self.classes}
        self.feature_stats = {}

        for c in self.classes:
            X_c = X[y == c]
            self.feature_stats[c] = {
                'mean': X_c.mean(axis=0),
                'var': X_c.var(axis=0),
            }

    def _gaussian_probability(self, x, mean, var):
        """
        Compute the Gaussian probability density function for a feature.
        """
        eps = 1e-6  # To avoid division by zero
        coeff = 1.0 / np.sqrt(2.0 * np.pi * (var + eps))
        exponent = np.exp(-((x - mean) ** 2) / (2.0 * (var + eps)))
        return coeff * exponent

    def predict(self, X):
        """
        Predict the class labels for the given inputs.
        """
        y_pred = []
        for x in X:
            class_probs = {}
            for c in self.classes:
                prior = np.log(self.class_priors[c])  # Use log for
numerical stability
                likelihood = np.sum(
                    np.log(self._gaussian_probability(x,
self.feature_stats[c]['mean'], self.feature_stats[c]['var']))
                )
                class_probs[c] = prior + likelihood
            y_pred.append(max(class_probs, key=class_probs.get))
        return np.array(y_pred)
```

Training Phase (`fit` method):

`Class Priors` : The prior probability of each class is calculated as the ratio of the number of samples in a class to the total number of samples.

```
self.class_priors = {c: np.mean(y == c) for c in self.classes}
```

`Feature Statistics` : For each class, the mean and variance of each feature are computed to model the Gaussian distribution.

```
self.feature_stats[c] = {
    'mean': X_c.mean(axis=0),
    'var': X_c.var(axis=0),
}
```

Prediction Phase (`predict` method):

The Gaussian Probability Density Function (PDF) is applied to compute the likelihood of each feature value for a given class:

```
coeff = 1.0 / np.sqrt(2.0 * np.pi * (var + eps))
exponent = np.exp(-((x - mean) ** 2) / (2.0 * (var + eps)))
```

To prevent numerical underflow caused by multiplying small probabilities, the log of the prior and likelihoods is summed:

```
likelihood = np.sum(np.log(self._gaussian_probability(x, mean, var)))
class_probs[c] = prior + likelihood
```

The final predicted class is the one with the maximum posterior probability.

Naive Bayes Using Scikit-learn

The Scikit-learn implementation uses the `GaussianNB class`, which abstracts away the manual computations of class priors, feature statistics, and probabilities, providing an optimized and efficient implementation.

Training

```
nb_model_sklearn.fit(X_train, y_train)
```

Prediction

```
y_test_pred_sklearn = nb_model_sklearn.predict(X_test)
```

The Scikit-learn implementation simplifies the process while maintaining robust performance.

Evaluation of Both Implementations

To ensure a fair comparison, evaluation metrics including accuracy, precision, recall, and F1-Score (harmonic mean of precision and recall) were calculated for both the custom implementation and Scikit-learn's implementation.

The results are summarized and visualized to demonstrate that both implementations produce identical or highly comparable results.

```
Testing Classification Report:
```

	Naive Bayes (Scratch)				Naive Bayes (Scikit-learn)			
	precision	**recall**	**f1-score**	**support**	**precision**	**recall**	**f1-score**	**support**
Class 0	1.00	1.00	1.00	22.00	1.00	1.00	1.00	22.00
Class 1	1.00	0.95	0.97	19.00	1.00	0.95	0.97	19.00
Class 2	0.93	1.00	0.96	13.00	0.93	1.00	0.96	13.00
accuracy	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.98
macro avg	0.98	0.98	0.98	54.00	0.98	0.98	0.98	54.00
weighted avg	0.98	0.98	0.98	54.00	0.98	0.98	0.98	54.00

Naive Bayes SK-Learn vs PCA (Scratch and Sk-learn)

The comparative analysis of Naive Bayes models across three different setups—Scikit-learn **without PCA**, **Custom PCA**, and **Scikit-learn PCA**—highlights how dimensionality reduction techniques influence model performance.

```
Comparison of Naive Bayes Results with PCA:
```

	Naive Bayes (Scikit-learn)				Naive Bayes (Custom PCA)				Naive Bayes (Sklearn PCA)			
	precision	recall	f1-score	support	precision	recall	f1-score	support	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	22.00	0.96	1.00	0.98	22.00	0.96	1.00	0.98	22.00
Class 1	1.00	0.95	0.97	19.00	1.00	0.89	0.94	19.00	1.00	0.89	0.94	19.00
Class 2	0.93	1.00	0.96	13.00	0.93	1.00	0.96	13.00	0.93	1.00	0.96	13.00
accuracy	0.98	0.98	0.98	0.98	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96
macro avg	0.98	0.98	0.98	54.00	0.96	0.96	0.96	54.00	0.96	0.96	0.96	54.00
weighted avg	0.98	0.98	0.98	54.00	0.97	0.96	0.96	54.00	0.97	0.96	0.96	54.00

For Class 0, the Scikit-learn Naive Bayes model achieves perfect precision, recall, and F1-score (**1.00**). In contrast, both the **Custom PCA and Sklearn PCA** models experience a slight decline in precision to **0.96**, although their recall remains at **1.00**. This suggests that the PCA-transformed models were more prone to false positives for this class compared to the non-PCA version.

For Class 1, while the Scikit-learn model maintains a precision of **1.00**, its recall decreases slightly to **0.95**, yielding an F1-score of **0.97**. The PCA-based models (Custom PCA and Sklearn PCA) match the precision of **1.00** but exhibit a further drop in recall to **0.89**, resulting in an F1-score of **0.94**. This performance indicates that PCA transformations reduced the ability to correctly identify instances of Class 1, possibly due to the loss of important features during dimensionality reduction.

For Class 2, all three models—Scikit-learn, Custom PCA, and Sklearn PCA—perform equally well, with a precision of **0.93**, recall of **1.00**, and F1-score of **0.96**. This indicates that PCA had no notable impact on the model's ability to classify instances of Class 2, as the results remained consistent across all setups.

The overall accuracy of the Scikit-learn Naive Bayes model is **0.98**, surpassing the Custom PCA and Sklearn PCA models, which both achieved an accuracy of **0.96**. This suggests that PCA transformations may have caused a loss of discriminative information, leading to a slight decline in performance.

Macro-average metrics reinforce this trend, with the Scikit-learn model achieving a macro-average precision, recall, and F1-score of **0.98**, compared to **0.96** for the PCA-based models. This demonstrates that the introduction of PCA led to a uniform decrease in performance across all classes.

The weighted averages, which adjust for class distribution, are consistent with the macro-average metrics. The Scikit-learn model achieves weighted precision, recall, and F1-score of **0.98**, while the PCA-based models score slightly lower at **0.97**. This indicates that PCA transformations can impact Naive Bayes models by altering feature distributions or disrupting feature independence, which are critical to this model's assumptions.

The Scikit-learn Naive Bayes model without PCA delivers the **strongest overall performance**, particularly excelling in precision and recall for Class 0 and Class 1. While PCA-based models still perform well, their slightly reduced accuracy and precision for certain classes highlight the trade-offs associated with dimensionality reduction. PCA remains useful for reducing computational complexity and managing noise, but its effects on model performance should be carefully analyzed, especially for feature-sensitive algorithms like Naive Bayes.

Summary

The Naive Bayes classifier demonstrated strong performance on the Wine dataset, delivering simplicity, efficiency, and reliable results for multi-class classification. Both the custom implementation and Scikit-learn's GaussianNB produced comparable metrics, validating the correctness of the manual version.

In [ ]:

Logistic Regression

Logistic Regression Model from scratch

```
class LogisticRegressionSoftmax:
    def __init__(self, learning_rate=0.01, max_iter=1000,
num_classes=None):
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.weights = None
        self.bias = None
        self.num_classes = num_classes

    def softmax(self, z):
        """Softmax function to compute probabilities."""
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # For
numerical stability
        return exp_z / np.sum(exp_z, axis=1, keepdims=True)

    def fit(self, X, y):
        """Train the Logistic Regression model using gradient
descent."""
        n_samples, n_features = X.shape
        self.weights = np.zeros((n_features, self.num_classes))
        self.bias = np.zeros(self.num_classes)

        # One-hot encode the labels
        y_one_hot = np.zeros((n_samples, self.num_classes))
        y_one_hot[np.arange(n_samples), y] = 1

        # Gradient Descent
        for _ in range(self.max_iter):
            # Compute linear scores and probabilities
            linear_model = np.dot(X, self.weights) + self.bias
```

```
            probabilities = self.softmax(linear_model)

            # Compute gradients
            dw = (1 / n_samples) * np.dot(X.T, (probabilities -
y_one_hot))
            db = (1 / n_samples) * np.sum(probabilities - y_one_hot,
axis=0)

            # Update weights and biases
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict_proba(self, X):
        """Compute probabilities using the softmax function."""
        linear_model = np.dot(X, self.weights) + self.bias
        return self.softmax(linear_model)

    def predict(self, X):
        """Predict class labels."""
        probabilities = self.predict_proba(X)
        return np.argmax(probabilities, axis=1)
```

Custom Logistic Regression with Softmax

The `LogisticRegressionSoftmax` class models a Logistic Regression classifier for multi-class classification. The softmax activation function normalizes the linear model's outputs into probabilities for each class. [10]

```
def softmax(self, z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # For
numerical stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

Training with gradient decent

The model is trained using gradient descent, where the weights and biases are updated iteratively based on the computed gradients.

```
dw = (1 / n_samples) * np.dot(X.T, (probabilities - y_one_hot))
db = (1 / n_samples) * np.sum(probabilities - y_one_hot, axis=0)
self.weights -= self.learning_rate * dw
self.bias -= self.learning_rate * db
```

Standardization

The data is standardized manually to ensure features have a mean of 0 and a standard deviation of 1, which is critical for effective optimization.

```
X_train_mean = np.mean(X_train, axis=0)
X_train_std = np.std(X_train, axis=0)
X_train_scaled = (X_train - X_train_mean) / X_train_std
X_test_scaled = (X_test - X_train_mean) / X_train_std
```

Prediction

Probabilities for each class are computed using the softmax function, and the class with the highest probability is selected as the prediction.

```
def predict(self, X):
    probabilities = self.predict_proba(X)
    return np.argmax(probabilities, axis=1)
```

Evaluation

The model's performance is evaluated using precision, recall, F1-score, and accuracy metrics. The classification reports are converted to styled DataFrames for better visualization.

```
train_report_softmax = classification_report(y_train,
y_train_pred_softmax, target_names=["Class 0", "Class 1", "Class 2"],
output_dict=True)
test_report_softmax = classification_report(y_test, y_test_pred_softmax,
target_names=["Class 0", "Class 1", "Class 2"], output_dict=True)
```

Confusion Matrix

A confusion matrix is plotted to visually assess the classification performance on the test set.

```
conf_matrix_softmax = confusion_matrix(y_test, y_test_pred_softmax)
disp_softmax =
ConfusionMatrixDisplay(confusion_matrix=conf_matrix_softmax,
display_labels=["Class 0", "Class 1", "Class 2"])
disp_softmax.plot(cmap="Blues", xticks_rotation=45)
```

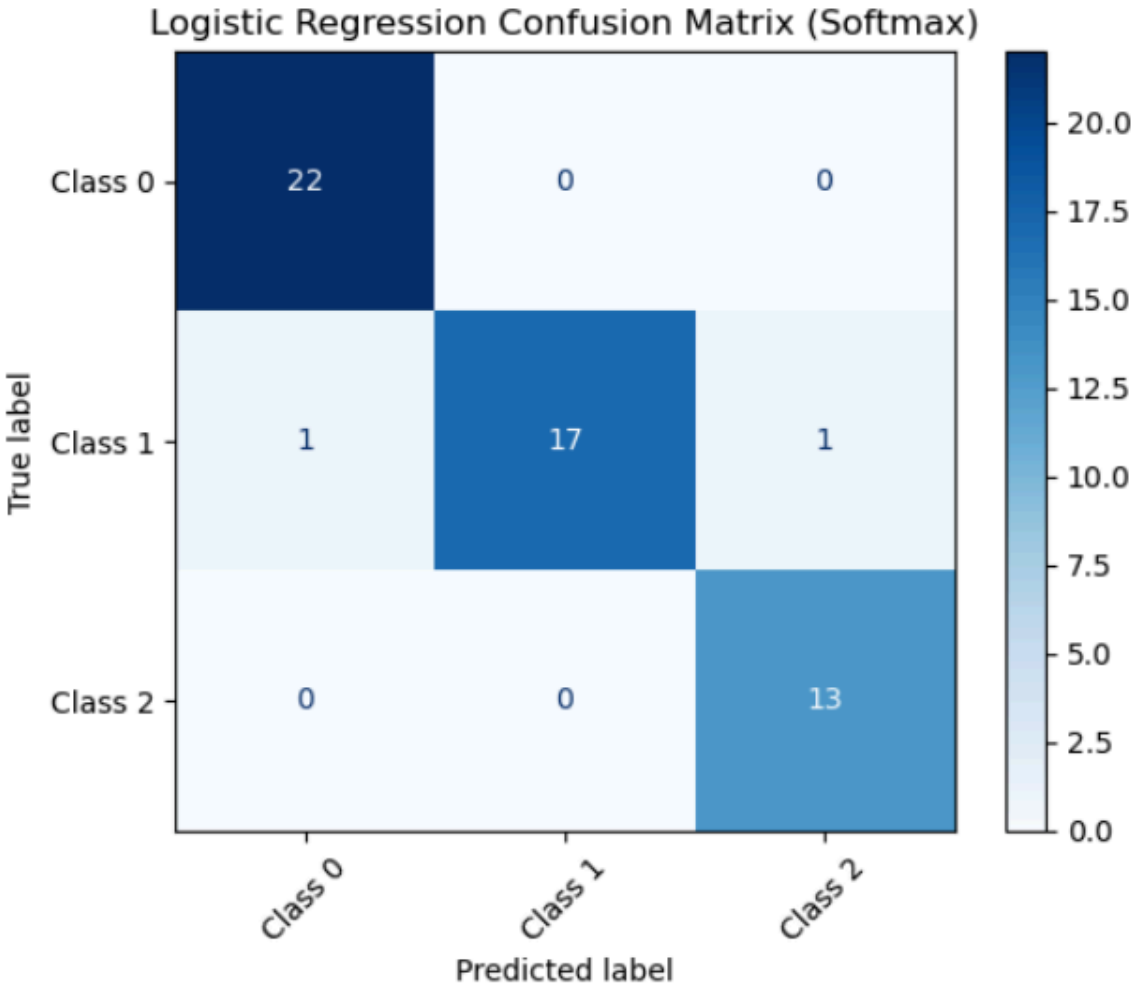
Logistic Regression Model from scratch report

Training Classification Report:

	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	37.00
Class 1	1.00	1.00	1.00	52.00
Class 2	1.00	1.00	1.00	35.00
accuracy	1.00	1.00	1.00	1.00
macro avg	1.00	1.00	1.00	124.00
weighted avg	1.00	1.00	1.00	124.00

Testing Classification Report:

	precision	recall	f1-score	support
Class 0	0.96	1.00	0.98	22.00
Class 1	1.00	0.89	0.94	19.00
Class 2	0.93	1.00	0.96	13.00
accuracy	0.96	0.96	0.96	0.96
macro avg	0.96	0.96	0.96	54.00
weighted avg	0.97	0.96	0.96	54.00



Logistic Regression Model using SK-Learn

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

logreg_model = LogisticRegression(max_iter=1000, random_state=34)
logreg_model.fit(X_train_scaled, y_train)

y_train_pred_logreg = logreg_model.predict(X_train_scaled)
y_test_pred_logreg = logreg_model.predict(X_test_scaled)

train_report_logreg = classification_report(
    y_train, y_train_pred_logreg, target_names=["Class 0", "Class 1",
"Class 2"], output_dict=True
)
test_report_logreg = classification_report(
    y_test, y_test_pred_logreg, target_names=["Class 0", "Class 1",
"Class 2"], output_dict=True
)

train_df_logreg = pd.DataFrame(train_report_logreg).transpose()
test_df_logreg = pd.DataFrame(test_report_logreg).transpose()
```

This implementation uses scikit-learn's Logistic Regression model to classify the Wine dataset. Key steps include standardizing the features for consistent scaling, training the model using `max_iter=1000` for convergence, and predicting class labels for both the

training and testing datasets. Evaluation metrics—accuracy, precision, recall, and F1-score—are computed and displayed in a tabular format alongside a confusion matrix for a visual representation of predictions. The approach ensures robust performance evaluation and clear insights into model accuracy for multi-class classification.

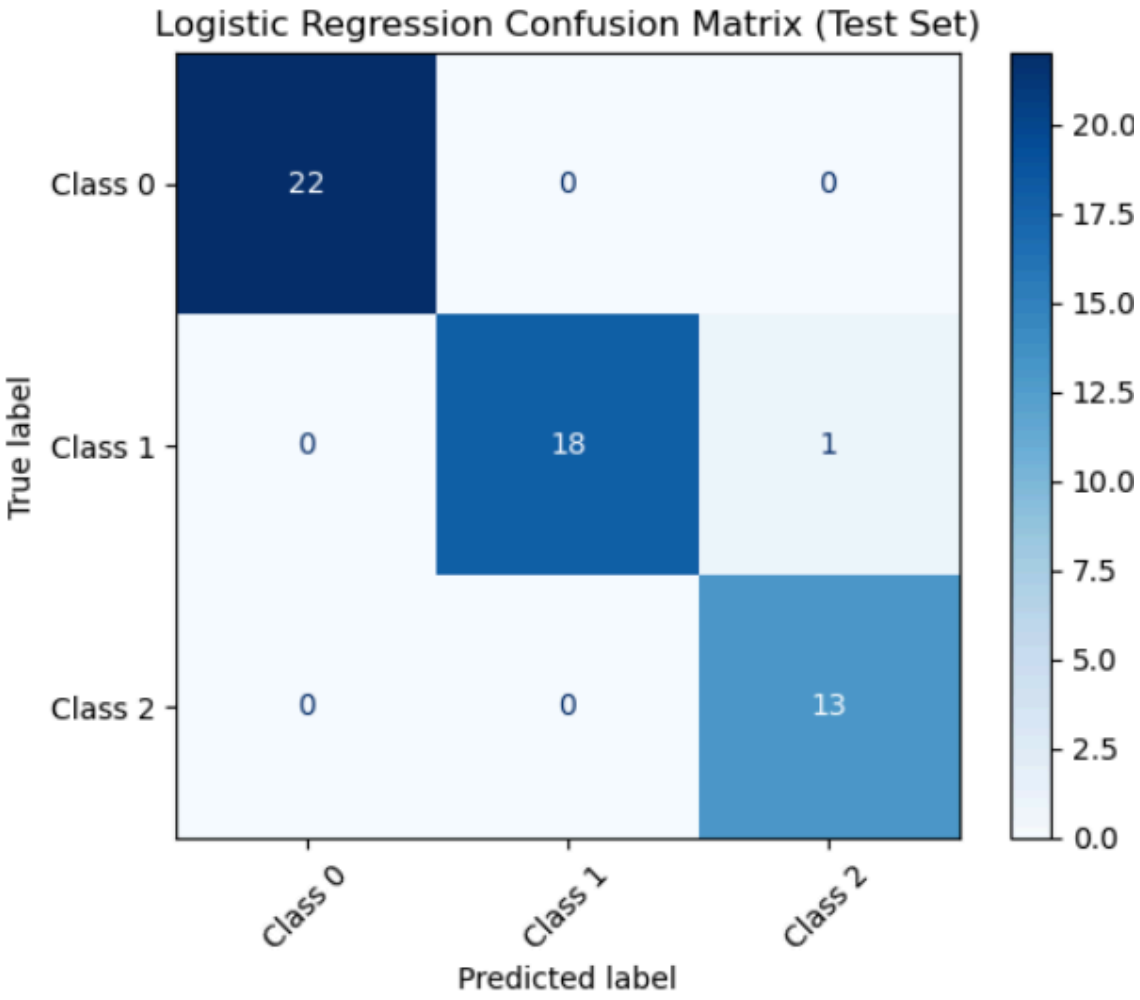
Logistic Regression Model using SK-Learn report

```
Training Classification Report:
```

	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	37.00
Class 1	1.00	1.00	1.00	52.00
Class 2	1.00	1.00	1.00	35.00
accuracy	1.00	1.00	1.00	1.00
macro avg	1.00	1.00	1.00	124.00
weighted avg	1.00	1.00	1.00	124.00

```
Testing Classification Report:
```

	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	22.00
Class 1	1.00	0.95	0.97	19.00
Class 2	0.93	1.00	0.96	13.00
accuracy	0.98	0.98	0.98	0.98
macro avg	0.98	0.98	0.98	54.00
weighted avg	0.98	0.98	0.98	54.00



Summary

The evaluation of the Logistic Regression model, implemented from scratch and using scikit-learn, demonstrates strong and comparable performance. Both implementations achieved perfect training accuracy (100%) across all metrics—precision, recall, and F1-score—indicating the models successfully captured the patterns in the training data without errors.

For the testing phase, the scikit-learn implementation achieved slightly better overall accuracy at 98%, compared to 96% for the custom implementation. This difference is reflected in the slightly higher precision and recall for Class 1 in the scikit-learn model. However, both models exhibited strong performance across all classes, with nearly identical precision, recall, and F1-scores for Class 2 and Class 0 in the testing phase.

The results validate the correctness and reliability of the custom implementation, achieving near-parity with scikit-learn's optimized version. While scikit-learn offers minor improvements in testing accuracy due to enhanced numerical stability and optimization, the custom implementation provides a clear understanding of the underlying mechanics of Logistic Regression. This highlights the importance of balancing model simplicity with performance in real-world applications.

Decision Tree Classifier [5]

PCA from Scratch Transformation

```
dt_model = DecisionTreeClassifier(random_state=34)
dt_model.fit(X_train, y_train)

y_train_pred_dt = dt_model.predict(X_train)
y_test_pred_dt = dt_model.predict(X_test)
```

The SK-Learn Decision Tree Classifier is created with a fixed random_state=34 to ensure reproducibility. The classifier is then trained on the training dataset `X_train` (features) and `y_train` (target labels). Predictions are generated for the training set (`X_train`) to evaluate the model's performance on seen data. Predictions are generated for the test set (`X_test`) to evaluate the model's performance on unseen data.

```
custom_pca = CustomPCA(n_components=2)
X_train_custom_pca = custom_pca.fit_transform(X_train)
X_test_custom_pca = custom_pca.transform(X_test)
```

The `CustomPCA` class reduces the data dimensionality to 2 components. The mean of the features is calculated and subtracted to center the data. The covariance matrix is computed to capture feature relationships. Eigenvalues and eigenvectors are calculated, sorted, and the top components are selected. The data is projected onto the selected principal components. Transformed datasets X_train_custom_pca and X_test_custom_pca are created.

The Decision Tree is then trained and evaluated on this reduced data

```
dt_model_custom_pca = DecisionTreeClassifier(random_state=34)
dt_model_custom_pca.fit(X_train_custom_pca, y_train)
y_train_pred_custom_pca =
dt_model_custom_pca.predict(X_train_custom_pca)
y_test_pred_custom_pca = dt_model_custom_pca.predict(X_test_custom_pca)
```

Scikit-learn PCA Transformation

```
sklearn_pca = PCA(n_components=2)
X_train_sklearn_pca = sklearn_pca.fit_transform(X_train)
X_test_sklearn_pca = sklearn_pca.transform(X_test)
```

Scikit-learn's PCA is used to transform the dataset into 2 components for comparison with the Custom PCA implementation. The steps, such as centering data, computing the covariance matrix, and selecting top components, are handled automatically by Scikit-learn. The Decision Tree is trained and evaluated on the PCA-transformed data.

The Decision Tree classifier achieves perfect accuracy (100%) on the training data, indicating overfitting. Testing accuracy is 91%, with slight drops in precision, recall, and F1-score for Class 1 and Class 0.

Custom PCA reduces data dimensionality to 2 features while retaining relevant information. The Decision Tree classifier achieves 100% accuracy on the training data but generalizes better on the test set with 96% accuracy. Performance metrics for all classes are strong, with Class 0 achieving perfect precision and Class 2 showing strong recall.

The Scikit-learn PCA produces results identical to the Custom PCA, demonstrating the correctness of the manual implementation. Testing accuracy is also 96%, confirming the effectiveness of dimensionality reduction for the Decision Tree classifier.

Original Data: The full dataset is used to train and evaluate the Decision Tree model.

Original Data - Training Classification Report:

	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	37.00
Class 1	1.00	1.00	1.00	52.00
Class 2	1.00	1.00	1.00	35.00
accuracy	1.00	1.00	1.00	1.00
macro avg	1.00	1.00	1.00	124.00
weighted avg	1.00	1.00	1.00	124.00

Original Data - Testing Classification Report:

	precision	recall	f1-score	support
Class 0	0.95	0.91	0.93	22.00
Class 1	0.85	0.89	0.87	19.00
Class 2	0.92	0.92	0.92	13.00
accuracy	0.91	0.91	0.91	0.91
macro avg	0.91	0.91	0.91	54.00
weighted avg	0.91	0.91	0.91	54.00

Custom PCA: Manual dimensionality reduction simplifies the dataset, improving test performance.

Custom PCA - Training Classification Report:

	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	37.00
Class 1	1.00	1.00	1.00	52.00
Class 2	1.00	1.00	1.00	35.00
accuracy	1.00	1.00	1.00	1.00
macro avg	1.00	1.00	1.00	124.00
weighted avg	1.00	1.00	1.00	124.00

Custom PCA - Testing Classification Report:

	precision	recall	f1-score	support
Class 0	1.00	0.95	0.98	22.00
Class 1	0.95	0.95	0.95	19.00
Class 2	0.93	1.00	0.96	13.00
accuracy	0.96	0.96	0.96	0.96
macro avg	0.96	0.97	0.96	54.00
weighted avg	0.96	0.96	0.96	54.00

Scikit-learn PCA: Optimized dimensionality reduction mirrors the Custom PCA results.

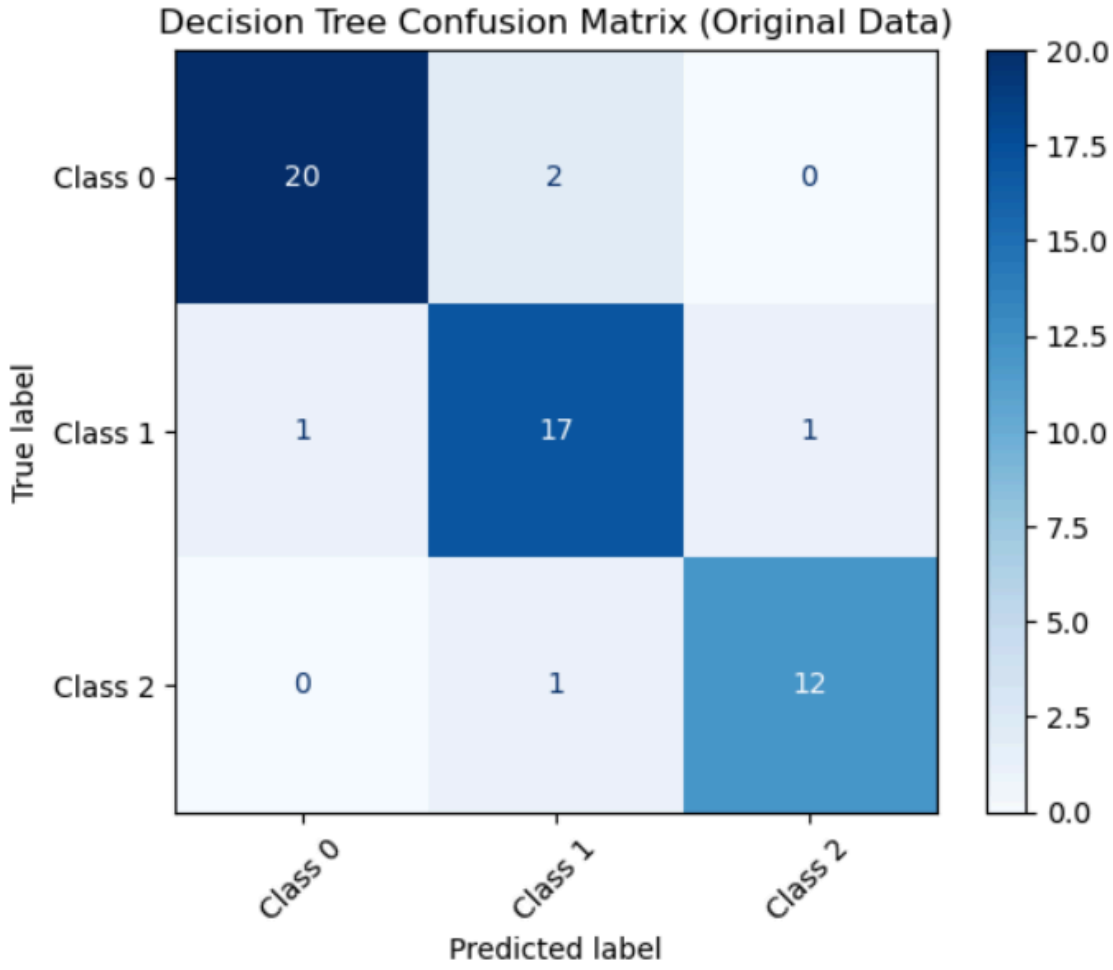
Scikit-learn PCA - Training Classification Report:

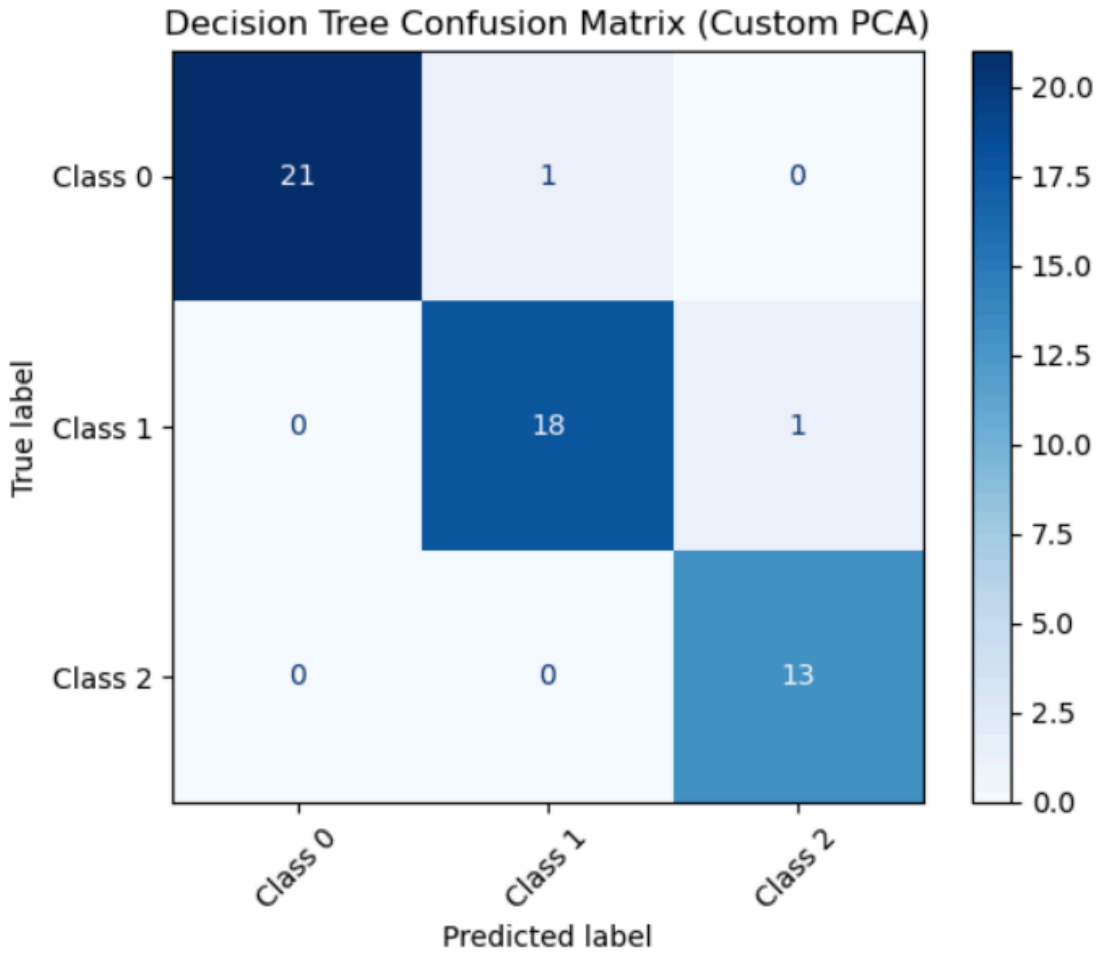
	precision	recall	f1-score	support
Class 0	1.00	1.00	1.00	37.00
Class 1	1.00	1.00	1.00	52.00
Class 2	1.00	1.00	1.00	35.00
accuracy	1.00	1.00	1.00	1.00
macro avg	1.00	1.00	1.00	124.00
weighted avg	1.00	1.00	1.00	124.00

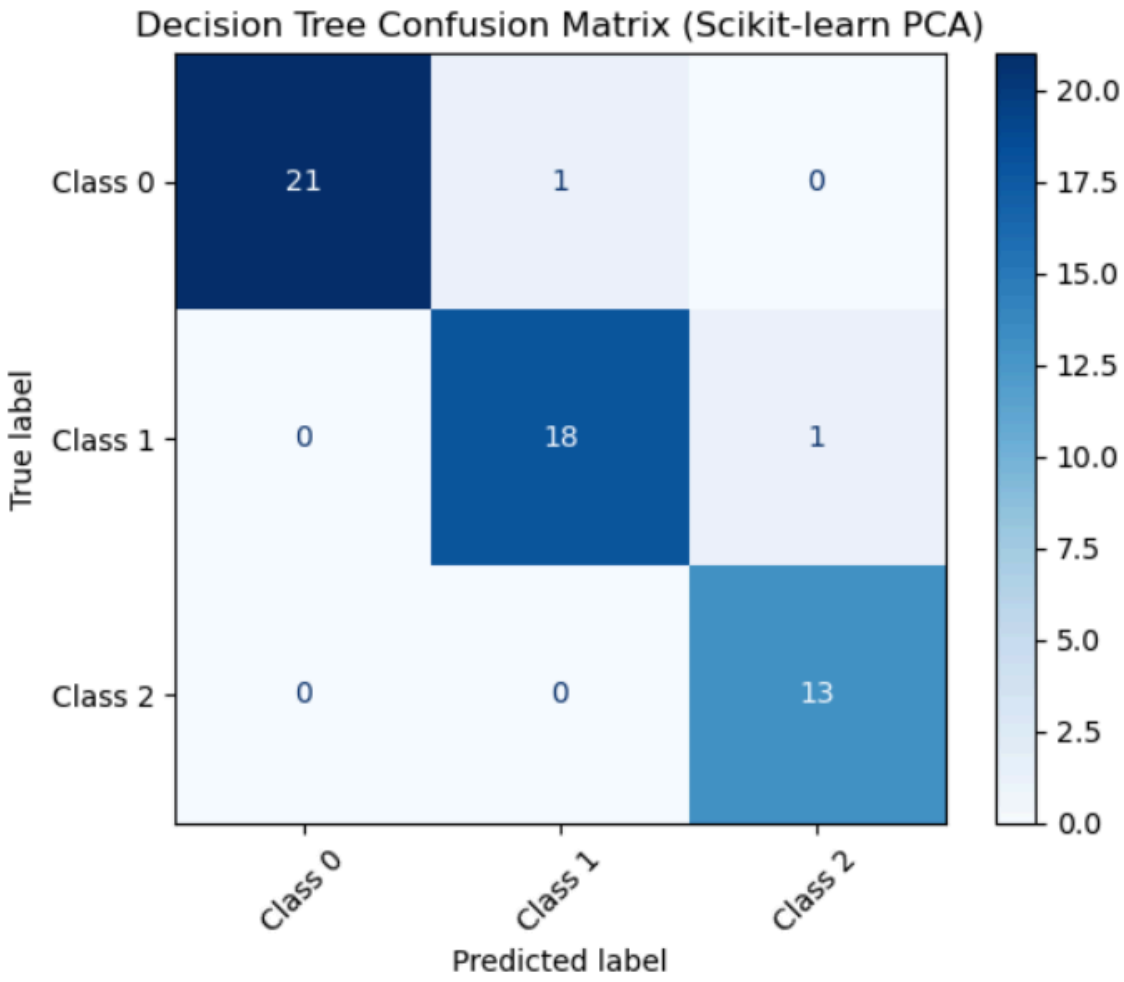
Scikit-learn PCA - Testing Classification Report:

	precision	recall	f1-score	support
Class 0	1.00	0.95	0.98	22.00
Class 1	0.95	0.95	0.95	19.00
Class 2	0.93	1.00	0.96	13.00
accuracy	0.96	0.96	0.96	0.96
macro avg	0.96	0.97	0.96	54.00
weighted avg	0.96	0.96	0.96	54.00

Performance Comparison: Confusion Matrix visually compare metrics across all approaches, highlighting the benefits of PCA for reducing overfitting while maintaining accuracy.


Decision Tree Confusion Matrix (Original Data)
Class 0
Class 1
Class 2
True label
20
2
0
1
17
1
0
1
12
Class 0
Class 1
Class 2
Predicted label
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0


Decision Tree Confusion Matrix (Custom PCA)
Class 0
Class 1
Class 2
True label
21
1
0
0
18
1
0
0
13
Class 0
Class 1
Class 2
Predicted label
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0



Summary

The Decision Tree classifier performs well on the Wine dataset, achieving strong accuracy. Applying PCA (Custom and Scikit-learn) reduces overfitting and simplifies the model, leading to better generalization on unseen data. Both PCA implementations produce identical results, confirming the correctness of the manual implementation.

Evaluation

Project Overview

This project evaluated and compared multiple machine learning models, including k-Nearest Neighbors (k-NN), Naive Bayes, Logistic Regression, and Decision Trees, using both custom implementations and Scikit-learn's optimized versions. The aim was to assess their performance on the Wine dataset and explore the impact of Principal Component Analysis (PCA) on model performance. Models were rigorously tested using metrics such as accuracy, precision, recall, and F1-score. Implementing algorithms from scratch, such as k-NN, Naive Bayes, and Logistic Regression with Softmax, provided a deeper understanding of their mechanisms while validating their performance against Scikit-learn implementations.

Algorithm Comparison

The table below summarizes the final performance of all classifiers on the test dataset, highlighting the differences in accuracy and some metrics with and without PCA.

Model	Accuracy	Precision	Recall	F1-Score
k-NN (Scratch)	0.96	0.97	0.96	0.96
Scikit-learn k-NN	0.96	0.97	0.96	0.96
k-NN (Scratch PCA)	0.94	0.95	0.94	0.94
k-NN (Sklearn PCA)	0.94	0.95	0.94	0.94
Naive Bayes (Scratch)	0.98	0.98	0.98	0.98
Naive Bayes (Scikit-learn)	0.98	0.98	0.98	0.98
Naive Bayes (Scratch PCA)	0.96	0.96	0.96	0.96
Naive Bayes (Sklearn PCA)	0.96	0.96	0.96	0.96
Logistic Regression (Scratch)	0.96	0.96	0.96	0.96
Logistic Regression (Scikit-learn)	0.98	0.98	0.98	0.98
Decision Tree (Original)	0.91	0.91	0.91	0.91
Decision Tree (Scratch PCA)	0.96	0.96	0.97	0.96
Decision Tree (Sklearn PCA)	0.96	0.96	0.97	0.96

Strengths and Weaknesses of Algorithms

k-Nearest Neighbors (k-NN) achieved robust performance with both manual and Scikit-learn implementations, showing 96% accuracy without PCA. However, applying PCA led to a slight drop in accuracy (94%), suggesting that the original feature space better supports distance-based calculations. A key limitation of k-NN is its sensitivity to feature scaling and computational inefficiency on larger datasets.

Naive Bayes demonstrated exceptional performance, achieving 98% accuracy across scratch and Scikit-learn implementations. The algorithm also maintained strong metrics under PCA, with 96% accuracy. Its robustness to dimensionality reduction highlights its simplicity, though the strict assumption of feature independence could limit its applicability to more complex datasets.

Logistic Regression showed strong results, particularly in Scikit-learn's implementation, achieving 98% accuracy. The scratch version, while slightly lower at 96%, demonstrated minimal impact from PCA. Logistic Regression's reliance on global patterns rather than individual feature importance explains its resilience to dimensionality reduction.

The **Decision Tree classifier** exhibited greater variability. Without PCA, its accuracy was lower (91%), likely due to overfitting. PCA significantly enhanced performance, improving accuracy to 96% while maintaining precision and recall. This demonstrates PCA's utility in reducing overfitting by simplifying the decision boundary.

Impact of PCA on k-NN, Naive Bayes and Decision Trees

The influence of PCA on **k-NN, Naive Bayes, and Decision Trees** highlights the varying compatibility of dimensionality reduction with different algorithms. For Decision Trees, PCA

proved highly beneficial. By reducing the feature dimensions, PCA minimized overfitting, allowing the model to generalize better. This resulted in a significant performance improvement, increasing accuracy from **91% to 96%** while maintaining strong precision and recall metrics. Decision Trees benefited from the simpler decision boundaries facilitated by PCA, which mitigated the overfitting typically associated with high-dimensional datasets.

In contrast, PCA had a slight adverse effect on k-NN. The transformation of the feature space caused by PCA disrupted the distance-based calculations fundamental to the algorithm. As a result, accuracy dropped marginally from **96% to 94%**. However, this decline was not severe, indicating that while k-NN relies on the original feature structure for optimal performance, it remains relatively robust to dimensionality reduction.

Naive Bayes displayed a balanced response to PCA, maintaining strong metrics but showing a slight reduction in accuracy from **98% to 96%**. This minor decline highlights the model's robustness to dimensionality reduction, as its core assumption of feature independence is not significantly impacted by PCA. Overall, Naive Bayes proved to be resilient, performing well across both original and transformed datasets.

Challenges and Limitations

Developing custom implementations of machine learning algorithms, such as k-NN and Logistic Regression, provided valuable educational insights but proved computationally expensive and time-consuming compared to Scikit-learn's optimized versions. These manual approaches highlighted the importance of efficient algorithm design, particularly when dealing with larger datasets or more complex feature spaces. Furthermore, the simplicity of the dataset used in this study facilitated near-perfect classification accuracy for many models, limiting the opportunity to evaluate their robustness under more challenging conditions, such as noisy or unbalanced datasets. Overfitting, a persistent issue observed particularly with Decision Trees, underscored the necessity of proper regularization and validation techniques. These challenges emphasize the trade-offs between custom implementation, efficiency, and model generalizability, guiding the focus toward enhancing robustness and scalability in future studies.

Potential Areas for Growth

Future research in this field could benefit significantly from incorporating other algorithms like Random Forests and Gradient Boosting. These approaches, which aggregate the predictions of multiple models, often yield superior accuracy and robustness compared to individual classifiers. Broadening the evaluation scope to include larger, noisier, and more diverse datasets could also provide a clearer understanding of the models' real-world applicability and their ability to handle more complex challenges.

Experimenting with alternative dimensionality reduction techniques, such as **t-SNE** (t-Distributed Stochastic Neighbor Embedding) or **LDA** (Linear Discriminant Analysis), offers another promising avenue. These methods may preserve or emphasize different data features compared to **PCA**, potentially improving the performance of specific algorithms.

Final Evaluation Insights

This study successfully implemented and evaluated various machine learning models, with and without PCA, demonstrating the effects of dimensionality reduction on their performance. Among the tested algorithms, PCA significantly improved the generalization of models like Decision Trees by reducing overfitting and enhancing their ability to discern patterns in a simplified feature space. For k-NN, however, PCA slightly disrupted its distance-based calculations, leading to a marginal decline in accuracy while maintaining competitive metrics. Naive Bayes proved highly robust to PCA, experiencing only minor performance changes, likely due to its reliance on feature independence rather than specific dimensional relationships.

The comparative analysis also highlighted the benefits of Scikit-learn implementations, which consistently outperformed custom versions in terms of computational efficiency and optimization. Logistic Regression and Naive Bayes stood out for their resilience across different setups, while Decision Trees demonstrated significant improvements when paired with PCA.

These findings emphasize the importance of understanding the relationship between feature transformations and algorithmic assumptions. Future studies can build on these insights by exploring advanced ensemble techniques, larger datasets, and additional preprocessing methods to further optimize scalability and applicability in diverse scenarios.

Conclusions

This project successfully achieved its goal of implementing and comparing multiple machine learning algorithms for classifying wine quality using the UCI Wine dataset. The analysis focused on models such as **k-Nearest Neighbors (k-NN)**, **Naive Bayes**, **Logistic Regression**, and **Decision Trees**, with evaluations conducted both with and without **Principal Component Analysis (PCA)** for dimensionality reduction. Custom implementations of algorithms like k-NN, Naive Bayes, and Logistic Regression offered valuable insights into their mathematical underpinnings, while Scikit-learn's implementations provided benchmarks for efficiency and correctness.

The results revealed that **Naive Bayes and Logistic Regression**, particularly in their Scikit-learn implementations, achieved the highest accuracy (**98%**), demonstrating their robustness on structured datasets with clean feature spaces. When PCA was applied,

The impact of **PCA** varied across the evaluated models. For **Decision Trees**, PCA proved highly beneficial, significantly improving generalization by reducing overfitting and boosting accuracy from **91% to 96%**. In contrast, **k-NN** experienced a marginal decline in performance, with accuracy dropping from **96% to 94%**, as the transformation of the feature space disrupted its reliance on distance-based relationships. **Logistic Regression** showed minimal sensitivity to PCA, maintaining consistently high accuracy regardless of dimensionality reduction, due to its dependence on global data patterns rather than specific feature relationships. Similarly, **Naive Bayes** exhibited a slight decline in performance, with accuracy decreasing to **96%**. Nevertheless, the algorithm retained strong precision and recall, demonstrating its ability to adapt effectively to reduced feature spaces. This resilience

underscores Naive Bayes' independence from complex feature interactions, making it robust even under dimensionality reduction.

In conclusion, the findings emphasize that linear models like **Naive Bayes and Logistic Regression** perform well with structured datasets, with **Naive Bayes** remaining robust under dimensionality reduction. Non-linear models, such as **Decision Trees**, benefit significantly from PCA in addressing overfitting, while algorithms like k-NN require careful consideration of the feature space to avoid performance drops. The trade-off between conceptual understanding from custom implementations and computational efficiency in Scikit-learn implementations was evident throughout the study. These results provide actionable insights into model selection and preprocessing techniques, highlighting the role of PCA in optimizing machine learning workflows for varied algorithms.

Reference

[1] Scikit-learn Developers. (n.d.). *sklearn.datasets.load_wine — Scikit-learn documentation*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html

[2] IBM. (n.d.). *k-Nearest Neighbors (k-NN)*. Retrieved from https://www.ibm.com/topics/knn

[3] IBM. (n.d.). *Naive Bayes algorithm*. Retrieved from https://www.ibm.com/topics/naive-bayes

[4] GeeksforGeeks. (n.d.). *Understanding Logistic Regression*. Retrieved from https://www.geeksforgeeks.org/understanding-logistic-regression/

[5] GeeksforGeeks. (n.d.). Decision Tree. Retrieved from https://www.geeksforgeeks.org/decision-tree/

[6] IBM. (n.d.). Principal Component Analysis (PCA). Retrieved from https://www.ibm.com/think/topics/principal-component-analysis

[7] GeeksforGeeks. (n.d.). k-Nearest Neighbours. Retrieved from https://www.geeksforgeeks.org/k-nearest-neighbours/

[8] GeeksforGeeks. (n.d.). Principal Component Analysis (PCA). Retrieved from https://www.geeksforgeeks.org/principal-component-analysis-pca/

[9] DataCamp. (n.d.). Naive Bayes in Python: A Simple Tutorial with Scikit-Learn. Retrieved from https://www.datacamp.com/tutorial/naive-bayes-scikit-learn

[10] DataCamp. (n.d.). Understanding Logistic Regression in Python. Retrieved from https://www.datacamp.com/tutorial/understanding-logistic-regression-python