BLG454E-Learning from data HW1 report

Akbar Bunyadzade
Student No: 912400214

1 Introduction

In this homework, we are providing an in-depth report on implementing various machine learning algorithms from scratch, including K-Nearest Neighbours (KNN), Logistic Regression, Naive Bayes, and Principal Component Analysis (PCA). Each algorithm is implemented manually and compared with Scikit-Learn's pre-built implementations, allowing for an understanding of model mechanics, key parameters, computational requirements, and the nuances of each approach.

2 Libraries Used

The libraries used in this project provide essential functionality for implementing and evaluating machine learning algorithms:

- **scikit-learn**: Used for accessing established machine learning models, metrics, and data manipulation utilities.
- **NumPy**: Provides efficient array handling and mathematical operations essential for computations.
- **Matplotlib**: A plotting library used for visualizing datasets and model performance.
- **Pandas**: Facilitates dataset handling, manipulation, and preprocessing.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.decomposition import PCA as sklearnPCA
from sklearn.metrics import accuracy_score, f1_score,
    root_mean_squared_error
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
```

Listing 1: Library Imports

3 K-Nearest Neighbours (KNN) - 5 Points

The K-Nearest Neighbours (KNN) algorithm is a straightforward, non-parametric method used for both classification and regression tasks. KNN works by locating the K nearest neighbors of a given point and classifying the point based on the majority class among the neighbors.

3.1 Theory of KNN

KNN is based on the assumption that similar data points are located near each other. When classifying a data point, KNN finds the K closest neighbors in the dataset and assigns the class that is most frequent among these neighbors. This approach can handle multi-class classification but can become computationally expensive as it scales linearly with the dataset size.

3.2 Custom Implementation

The code below demonstrates a custom implementation of KNN, where the Euclidean distance is used to find the nearest neighbors.

```
K-Nearest Neighbour (5 points)
```

3.3 Comparison with Scikit-Learn

Scikit-Learn provides an optimized version of KNN, which uses efficient algorithms like KD-Trees or Ball-Trees for faster nearest neighbor searches. The comparison with Scikit-Learn's implementation allows us to evaluate both accuracy and computational efficiency.

```
K-Nearest Neighbour (5 points)
```

4 Logistic Regression

Logistic Regression is a popular algorithm for binary classification tasks. It models the probability of a binary outcome using a logistic function, making it suitable for predicting probabilities.

4.1 Theory of Logistic Regression

Logistic regression uses a linear combination of input features, passed through a sigmoid function to map predictions to probabilities between 0 and 1. This technique assumes linear separability between classes and is sensitive to feature scaling.

4.2 Custom Implementation

In our custom logistic regression implementation, we use gradient descent to minimize the binary cross-entropy loss. The sigmoid function is used to transform the linear output into probabilities.

```
Principal Component Analysis (5 points)
```

4.3 Comparison with Scikit-Learn

Scikit-Learn's Logistic Regression includes optimization techniques like stochastic gradient descent for efficiency. The comparison here highlights differences in convergence speed and performance.

```
Principal Component Analysis (5 points)
```

5 Naive Bayes

Naive Bayes is a probabilistic classifier based on Bayes' theorem with an assumption of feature independence. It is highly effective for large datasets and commonly used in text classification tasks.

5.1 Theory of Naive Bayes

Naive Bayes calculates the probability of each class given the input features, assuming that the features are conditionally independent. Despite its simplicity, Naive Bayes often performs well on classification tasks with high-dimensional data.

5.2 Custom Implementation

Below is the custom implementation of the Naive Bayes classifier. This implementation calculates class probabilities and updates them using Bayes' theorem.

```
Visualization Tools (5 points)
```

5.3 Comparison with Scikit-Learn

Scikit-Learn's Naive Bayes implementation is optimized for speed, especially on sparse data. Our comparison assesses performance differences in accuracy and speed.

```
Visualization Tools (5 points)
```

6 Principal Component Analysis (PCA)

PCA is a technique for dimensionality reduction, particularly useful when working with large datasets containing many correlated variables. PCA identifies new uncorrelated variables (principal components) that maximize the variance.

6.1 Theory of PCA

PCA works by finding the eigenvectors of the covariance matrix of the data, projecting the data onto these principal components. It aims to reduce dimensionality while preserving as much variance as possible.

6.2 Custom Implementation

The custom PCA implementation calculates covariance, finds eigenvalues and eigenvectors, and projects data onto principal components. This implementation allows us to explore dimensionality reduction without using pre-built functions.

```
#loading Digits dataset

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
digits = load_digits()
X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)

#validating GNB
gnb = GNB(X_train, y_train)
gnb.fit()
y_pred_custom_gnb = gnb.predict(X_test)
accuracy_custom_gnb, f1_score_custom_gnb = accuracyNf1_score(y_test
    , y_pred_custom_gnb)

gnb_sklearn = GaussianNB()
gnb_sklearn.fit(X_train, y_train)
y_pred_sklearn_gnb = gnb_sklearn.predict(X_test)
accuracy_custom_gnb_sklearn, _ = accuracyNf1_score(y_test,
    y_pred_sklearn_gnb)

print("Custom GNB Accuracy:", accuracy_custom_gnb)
print("Sklearn GNB Accuracy:", accuracy_custom_gnb_sklearn)

#validating KNN
knn = KNN(X_train, y_train, k = 3)
y_pred_custom_knn = knn.predict(X_test)
accuracy_custom_knn, f1_score_custom_knn = accuracyNf1_score(y_test
    , y_pred_custom_knn)

knn_sklearn = KNeighborsClassifier(n_neighbors=3)
knn_sklearn.fit(X_train, y_train)
y_pred_sklearn_knn = knn_sklearn.predict(X_test)
```

```
accuracy_custom_knn_sklearn, _ = accuracyNf1_score(y_test,
    y_pred_sklearn_knn)

print("Custom KNN Accuracy:", accuracy_custom_knn)
print("Sklearn KNN Accuracy:", accuracy_custom_knn_sklearn)

#validating PCA
pca_custom = PCA(X_train, n_components = 3)
pca_custom.fit()
X_train_custom_pca = pca_custom.transform(X_train)

pca_sklearn = sklearnPCA(n_components = 3)
X_train_sklearn_pca = pca_sklearn.fit_transform(X_train)

print("Custom PCA Transformed X_train (first few rows):",
    X_train_custom_pca[:5])
print("Sklearn PCA Transformed X_train (first few rows):",
    X_train_sklearn_pca[:5])

#validating metrics defined above for further clarification
accuracy_custom, f1_custom = accuracyNf1_score(y_test,
    y_pred_custom_gnb)

accuracy_sklearn = accuracy_score(y_test, y_pred_sklearn_gnb)
f1_sklearn = f1_score(y_test, y_pred_sklearn_gnb, average = '
    weighted')

print(f"Custom Accuracy: {accuracy_custom}, Sklearn Accuracy: {
    accuracy_sklearn}")
print(f"Custom F1 Score: {f1_custom}, Sklearn F1 Score: {f1_sklearn
    }")
```

6.3 Comparison with Scikit-Learn

Scikit-Learn's PCA implementation is highly optimized, making it more efficient for large datasets. By comparing the custom PCA with Scikit-Learn's, we can understand computational differences in dimensionality reduction.

```
#loading Digits dataset

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
digits = load_digits()
X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)

#validating GNB
gnb = GNB(X_train, y_train)
gnb.fit()
y_pred_custom_gnb = gnb.predict(X_test)
accuracy_custom_gnb, f1_score_custom_gnb = accuracyNf1_score(y_test
    , y_pred_custom_gnb)

gnb_sklearn = GaussianNB()
gnb_sklearn.fit(X_train, y_train)
```

```
y_pred_sklearn_gnb = gnb_sklearn.predict(X_test)
accuracy_custom_gnb_sklearn, _ = accuracyNf1_score(y_test,
    y_pred_sklearn_gnb)

print("Custom GNB Accuracy:", accuracy_custom_gnb)
print("Sklearn GNB Accuracy:", accuracy_custom_gnb_sklearn)

#validating KNN
knn = KNN(X_train, y_train, k = 3)
y_pred_custom_knn = knn.predict(X_test)
accuracy_custom_knn, f1_score_custom_knn = accuracyNf1_score(y_test
    , y_pred_custom_knn)

knn_sklearn = KNeighborsClassifier(n_neighbors=3)
knn_sklearn.fit(X_train, y_train)
y_pred_sklearn_knn = knn_sklearn.predict(X_test)
accuracy_custom_knn_sklearn, _ = accuracyNf1_score(y_test,
    y_pred_sklearn_knn)

print("Custom KNN Accuracy:", accuracy_custom_knn)
print("Sklearn KNN Accuracy:", accuracy_custom_knn_sklearn)

#validating PCA
pca_custom = PCA(X_train, n_components = 3)
pca_custom.fit()
X_train_custom_pca = pca_custom.transform(X_train)

pca_sklearn = sklearnPCA(n_components = 3)
X_train_sklearn_pca = pca_sklearn.fit_transform(X_train)

print("Custom PCA Transformed X_train (first few rows):",
    X_train_custom_pca[:5])
print("Sklearn PCA Transformed X_train (first few rows):",
    X_train_sklearn_pca[:5])

#validating metrics defined above for further clarification
accuracy_custom, f1_custom = accuracyNf1_score(y_test,
    y_pred_custom_gnb)

accuracy_sklearn = accuracy_score(y_test, y_pred_sklearn_gnb)
f1_sklearn = f1_score(y_test, y_pred_sklearn_gnb, average = '
    weighted')

print(f"Custom Accuracy: {accuracy_custom}, Sklearn Accuracy: {
    accuracy_sklearn}")
print(f"Custom F1 Score: {f1_custom}, Sklearn F1 Score: {f1_sklearn
    }")
```

7 Performance Evaluation Metrics

Evaluating model performance is critical in assessing both model accuracy and efficiency. The metrics used in this project include:

- **Accuracy Score**: Measures the proportion of correct predictions out of total predictions.

- **F1 Score**: A harmonic mean of precision and recall, useful for imbalanced datasets.
- **Root Mean Squared Error (RMSE)**: Computes the square root of the average squared differences between predicted and actual values, often used in regression tasks.

8 Conclusion

In this report, we have explored the manual implementations of several machine learning algorithms and compared them with Scikit-Learn's implementations. Each algorithm's theoretical foundation, parameterization, and computational challenges were discussed, providing a comprehensive understanding of these essential machine learning methods.