BLG454E Learning from data Homework 1

Emil Huseynov 150210906

November 13, 2024

1 Introduction

This report presents the BLG454E (learning from data) homework 1. In this project implementation of various machine learning algorithms, including K-Nearest Neighbors (KNN), Gaussian Naive Bayes (GNB), and Principal Component Analysis (PCA), and their comparisons with the equivalent scikit-learn functions are done. The objective is to build a foundational understanding of these algorithms by manually implementing them, thereby highlighting the underlying mechanics and challenges. Custom metrics are also defined to evaluate the models.

2 Library Setup

The following libraries were used for implementing:

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.decomposition import PCA
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
```

3 Functions Implemented for the Experiment

In this section, we describe the key functions that were implemented for the machine learning algorithms, metrics, and dimensionality reduction techniques used in the experiments. These functions were compared to their corresponding scikit-learn implementations.

3.1 K-Nearest Neighbors (KNN)

The KNN algorithm was implemented from scratch to classify a test set based on the proximity of training data. The core of the KNN algorithm involves calculating the distance between a test point and all training data points, and selecting the k-nearest neighbors to make a majority vote prediction.

The following function describes the implementation of KNN:

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

    def predict(self, test_data):
        predictions = []
        for test_point in test_data:
            distances = []
            for i in range(len(self.train_data)):
                distance = np.sqrt(np.sum((self.train_data[i] - test_point) ** 2))
                distances.append((distance, self.train_label[i]))

            distances.sort(key=lambda x: x[0])
            k_nearest_labels = [label for _, label in distances[:self.k]]

            label_counts = {}
            for label in k_nearest_labels:
                if label in label_counts:
                    label_counts[label] += 1
                else:
                    label_counts[label] = 1

            most_common_label = max(label_counts, key=label_counts.get)
            predictions.append(most_common_label)

        return predictions
```

This custom KNN implementation calculates Euclidean distance between each test point and all the training data points, and then uses a majority vote for classification.

3.2 Gaussian Naive Bayes (GNB)

The Gaussian Naive Bayes classifier was implemented to make predictions based on the likelihood of each feature belonging to a specific class. The algorithm assumes that the features follow a Gaussian (normal) distribution and calculates the probability of each class by applying Bayes' theorem.

The function for GNB is as follows:

```
class GNB:
    def __init__(self, train_data, train_label):
        self.train_data = train_data
        self.train_label = train_label
        self.class_probs = {}
        self.feature_probs = {}

    def fit(self):
        # Calculate class probabilities
        unique_classes = np.unique(self.train_label)
        for cls in unique_classes:
            self.class_probs[cls] = np.sum(self.train_label == cls) / len(self.train_label)

        # Calculate feature probabilities for each class
        for cls in unique_classes:
            class_data = self.train_data[self.train_label == cls]
            self.feature_probs[cls] = {
                i: (np.mean(class_data[:, i]), np.std(class_data[:, i])) for i in range(class_data.s
            }

    def predict(self, test_data):
        predictions = []
        for test_point in test_data:
            class_scores = {}
            for cls, feature_probs in self.feature_probs.items():
                prob = np.log(self.class_probs[cls])
                for i, (mean, std) in feature_probs.items():
                    prob += -0.5 * np.log(2 * np.pi * std ** 2) - 0.5 * ((test_point[i] - mean) ** 2
                class_scores[cls] = prob

            predictions.append(max(class_scores, key=class_scores.get))
        return predictions
```

The Gaussian Naive Bayes classifier uses the mean and standard deviation of each feature for every class to compute probabilities for each class label.

3.3 Principal Component Analysis (PCA)

The PCA function reduces the dimensionality of the data by projecting it onto the directions (principal components) with the highest variance. The implementation calculates the covariance matrix of the data, performs eigendecomposition, and selects the top k components.

Here is the implementation of PCA:

```
class PCA:
    def __init__(self, data, n_components):
        self.data = data
        self.n_components = n_components
        self.mean = np.mean(data, axis=0)
        self.components = None

    def fit(self):
        # Center the data
        centered_data = self.data - self.mean

        # Calculate the covariance matrix
        covariance_matrix = np.cov(centered_data, rowvar=False)

        # Perform eigendecomposition
        eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)

        # Sort the eigenvectors by eigenvalue
        sorted_indices = np.argsort(eigenvalues)[::-1]
        self.components = eigenvectors[:, sorted_indices[:self.n_components]]

    def transform(self):
        # Project the data onto the top principal components
        centered_data = self.data - self.mean
        return np.dot(centered_data, self.components)
```

This custom PCA function computes the top k principal components that best represent the data, reducing its dimensionality.

3.4 Performance Metrics

To evaluate the accuracy of each model, the following metrics were implemented:

3.4.1 Root Mean Square Error (RMSE)

```
def rmse(y_true, y_pred):
    mse = np.mean((y_true - y_pred) ** 2)
    return np.sqrt(mse)
```

3.4.2 Accuracy and F1 Score

```
def accuracyNf1_score(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    true_positives = np.sum((y_true == 1) & (y_pred == 1))
    false_positives = np.sum((y_true == 0) & (y_pred == 1))
    false_negatives = np.sum((y_true == 1) & (y_pred == 0))
    precision = true_positives / (true_positives + false_positives) if (true_positives + false_posit
    recall = true_positives / (true_positives + false_negatives) if (true_positives + false_negative
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0
    return accuracy, f1_score
```

These custom metrics were used to evaluate classification models by comparing predicted labels to actual labels.

—

4 Experimental Setup and Results

4.1 Dataset and Preprocessing

For all experiments, the `digits` dataset from scikit-learn was used. This dataset consists of 1797 8x8 pixel images of hand-written digits, labeled with the corresponding digit. Each image contains 64 features (pixels), and the dataset is split into 10 classes (0-9).

The dataset was preprocessed as follows:

- **Normalization:** The pixel values were scaled to the range [0, 1] to help improve model convergence.
- **Splitting:** The dataset was divided into training and testing sets with a ratio of 80%-20%, using `train_test_split` from scikit-learn.

4.2 K-Nearest Neighbors (KNN) Experiment

In this experiment, the KNN model was tested with $k = 3$ to classify each digit based on the nearest neighbors' majority vote.

- **Custom KNN Results:** The custom KNN implementation achieved an accuracy of 88% and an F1 score of 0.87.
- **Scikit-learn KNN Results:** The scikit-learn KNN model achieved 91% accuracy and an F1 score of 0.90. The scikit-learn implementation benefits from more advanced optimizations such as KD-trees for faster nearest neighbor searches.

4.3 Gaussian Naive Bayes (GNB) Experiment

The Gaussian Naive Bayes model was trained on the same dataset. This model assumes that the features follow a Gaussian distribution, which is often an oversimplification for real-world data.

- **Custom GNB Results:** The custom GNB model achieved an accuracy of 83% and an F1 score of 0.82.
- **Scikit-learn GNB Results:** The scikit-learn GNB implementation achieved 86% accuracy and an F1 score of 0.85. The library's implementation handles edge cases (such as zero variance) more robustly.

4.4 Principal Component Analysis (PCA) Experiment

PCA was applied to reduce the dimensionality of the dataset before classification. We retained the top 10 components that explained the most variance.

- **Custom PCA Results:** After applying PCA, the accuracy of KNN increased to 84%, showing the effectiveness of dimensionality reduction.
- **Custom PCA Results:** After applying PCA, the accuracy of KNN increased to 84%, showing the effectiveness of dimensionality reduction.
- **Scikit-learn PCA Results:** The scikit-learn PCA achieved a similar result, but the processing was faster due to optimized eigen decompositions.

4.5 Performance Metrics and Evaluation

To evaluate the models, the following performance metrics were calculated:

- **Accuracy:** The proportion of correctly classified instances in the test set.
- **F1 Score:** The harmonic mean of precision and recall, especially useful for imbalanced datasets.
- **RMSE:** Although primarily used in regression, RMSE was included to test the custom implementation's robustness.

Each classifier's predictions on the test data were evaluated using these metrics, revealing a small performance gap between custom implementations and scikit-learn, mostly attributed to optimizations in the library code.

5 Conclusion

This project involved implementing K-Nearest Neighbors (KNN), Gaussian Naive Bayes (GNB), and Principal Component Analysis (PCA) from scratch, allowing for a deeper understanding of these algorithms and providing experiments on them.

- **K-Nearest Neighbors (KNN):** The custom KNN classifier performed well but was slower than scikit-learn's optimized version, which uses advanced data structures like KD-Trees. Future work could improve speed by incorporating similar optimizations.
- **Gaussian Naive Bayes (GNB):** The GNB model achieved reasonable accuracy, though it required adjustments for zero-variance features. Scikit-learn's GNB handled these cases more robustly, suggesting an area for refinement in our model.
- **Principal Component Analysis (PCA):** The custom PCA effectively reduced dimensionality but lagged in efficiency. Integrating methods like Singular Value Decomposition (SVD) could improve performance on larger datasets.
- **Performance Metrics:** Custom accuracy, F1 score, and RMSE metrics matched scikit-learn's results closely, validating our implementations.