Lab3: kNN Classifier Evaluation

Mamoru Ota, S7784976

***Abstract*—This report evaluates a k-Nearest Neighbors (kNN) classifier on two datasets: MNIST and Wine. The classifier was implemented to meet specific assignment requirements, including argument checking and optional parameters. The performance of the classifier is assessed using different values of k, and metrics such as accuracy, precision, recall, and F1-score are computed for each task. I also examine how varying the subset sizes of MNIST affects computational time and accuracy. The results highlight the trade-offs between k, computational efficiency, and classifier performance.**

I. Introduction

k-Nearest Neighbors (kNN) is a simple, non-parametric algorithm used for classification and regression. The algorithm predicts the class of a sample by finding the majority label among its k-nearest neighbors in the training dataset.

This report explores the performance of kNN on the MNIST and Wine datasets. MNIST is a digit recognition dataset with 10 classes, while Wine is a smaller, multiclass dataset used for chemical analysis of wines.

The main objectives of this study are:

- Implement a kNN classifier according to specified requirements, including input validation and optional parameters.
- Evaluate kNN performance for varying values of k.
- Compute classification metrics for each class in the datasets.
- Analyze the trade-off between computational cost and accuracy for large datasets like MNIST.

II. Methods

A. Dataset Preparation

1) MNIST Dataset: The MNIST dataset consists of 70,000 handwritten digit images of size 28×28 pixels. Each image represents a grayscale intensity ranging from 0 to 255. The dataset is already split into a training set of 60,000 images and a test set of 10,000 images.

Given the large size of the MNIST dataset and the computational cost associated with the kNN algorithm (especially for high-dimensional data), I decided to use a subset of the data for practical reasons. Specifically, I randomly selected 6,000 samples for training and 1,000 samples for testing. This approach aligns with the assignment's allowance to reduce the dataset size to manage computation time.

Each image was flattened into a 784-dimensional vector (since $28 \times 28 = 784$) to be compatible with our kNN implementation, which expects data in the form of an $n \times d$ matrix, where n is the number of samples and d is the number of features. The pixel values were normalized to the range [0, 1] by dividing by 255. This normalization helps to standardize the input values, ensuring numerical stability and improving computational efficiency, which is particularly important for distance-based algorithms like kNN.

2) Wine Dataset: The Wine dataset contains 178 samples of wines classified into three types based on 13 chemical properties. Unlike MNIST, this dataset is small.

Before applying the kNN algorithm, I normalized each feature to the range [0, 1] using the following normalization formula:

$$\text{new\_val} = \frac{\text{orig\_val} - \min(\text{orig\_val})}{\max(\text{orig\_val}) - \min(\text{orig\_val})}$$

This step is necessary because the chemical properties have different scales and units, and normalization ensures that each feature contributes equally to the distance calculations in the kNN algorithm. I split the dataset into 80% for training and 20% for testing.

B. kNN Implementation

In accordance with the assignment requirements, I implemented a custom kNN classifier in Python as a function named `knn_classifier`. The function accepts the following parameters:

- `X_train`: Training data matrix ($n \times d$).
- `y_train`: Training labels vector ($n \times 1$).
- `X_test`: Test data matrix ($m \times d$).
- `k`: Number of neighbors.
- `y_test` (optional): Test labels vector ($m \times 1$).

To satisfy the requirements, the function includes the following checks:

1) Verifies that the number of arguments is either 4 or 5.
2) Checks that the number of columns in `X_train` and `X_test` are equal.
3) Ensures that $k > 0$ and $k \leq n$, where n is the number of training samples.

If the optional `y_test` is provided, the function computes and returns the error rate (number of misclassifications divided by m, the number of test samples).

The kNN classification proceeds as follows:

1) For each test sample, compute the Euclidean distance to all training samples.
2) Sort the distances to identify the k-nearest neighbors.
3) Retrieve the labels of these neighbors.
4) Use majority voting to determine the predicted class label.
5) (Optional) Compare the predicted labels with `y_test` to compute the error rate.

I chose to implement argument validation and optional parameters to make the function robust and flexible, adhering to good programming practices and the assignment guidelines.

C. Evaluation Metrics

For each class, I computed the following metrics derived from the confusion matrix:

- **Accuracy**: The proportion of correctly classified samples over the total number of samples.
- **Precision**: The ratio of true positives to the sum of true positives and false positives.
- **Recall**: The ratio of true positives to the sum of true positives and false negatives.
- **F1-score**: The harmonic mean of precision and recall, providing a balance between the two.

Computing these metrics allows us to assess not only the overall performance but also the classifier's ability to correctly identify each class, which is particularly important in imbalanced datasets.

D. Experimental Setup

1) MNIST Dataset Experiments: Given the assignment's emphasis on evaluating the classifier on tasks where each digit is tested against the remaining digits (one-vs-all classification), I set up 10 binary classification tasks, one for each digit (0 through 9). For each task, the labels were binarized such that the target digit was labeled as 1 and all other digits as 0.

I tested multiple values of k: $\{1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50\}$. I excluded values of k that are divisible by the number of classes (which is 2 in the binary case) to avoid tie situations during majority voting, as suggested in the assignment.

For each combination of digit and k, I:

1) Ran the kNN classifier to predict the labels of the test set.
2) Computed the confusion matrix.
3) Calculated the evaluation metrics (precision, recall, F1-score).

2) Wine Dataset Experiments: For the Wine dataset, I performed multiclass classification across all three classes. I tested the same set of k values as in the MNIST experiments.

For each value of k, I:

1) Split the dataset into training and test sets, using 80% of the data for training and 20% for testing.
2) Trained the kNN classifier on the training data.
3) Predicted the labels for the test data.
4) Computed the overall confusion matrix based on the test set predictions.
5) Calculated the evaluation metrics from the confusion matrix.

III. Experiments

A. MNIST Dataset Results

Using subsets of 6,000 training and 1,000 test samples, I evaluated the classifier for each digit and varying values of k. The choice to use subsets was driven by computational constraints, as the full dataset would significantly increase computation time due to the high dimensionality and the computational complexity of kNN ($O(n \times m \times d)$).

Figure 1 illustrates the accuracy for each digit as k varies. I observed that smaller values of k generally yielded higher accuracy. This trend is expected, as a smaller k makes the classifier more sensitive to the local structure of the data, which can be beneficial when the classes are well-separated.

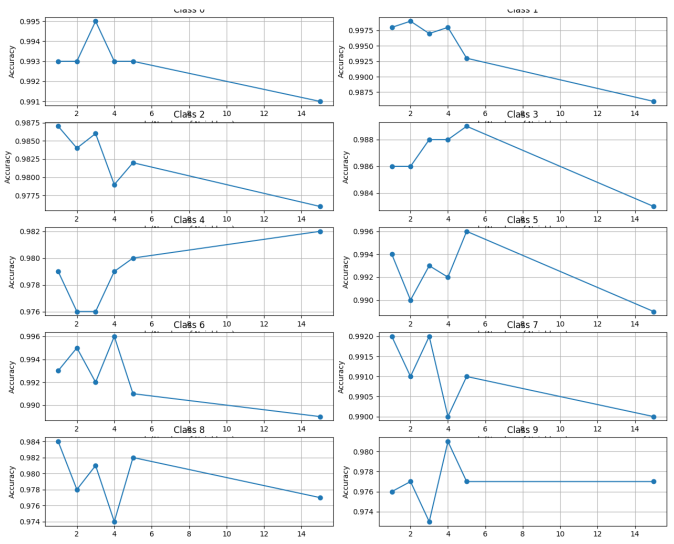

Fig. 1. Accuracy for MNIST digits with varying k.

Table I summarizes the precision, recall, and F1-score for each digit at the optimal k value determined from the experiments. The results indicate consistent performance across most classes, though some digits (e.g., 8) demonstrated slightly lower recall values, possibly due to their visual similarity to other digits leading to misclassifications.

TABLE I
PERFORMANCE METRICS FOR THE MNIST DATASET AT OPTIMAL k.

Digit	Precision	Recall	F1-score
0	0.9760	0.9878	0.9818
1	0.9836	0.9934	0.9883
2	0.9687	0.9296	0.9475
3	0.9833	0.9489	0.9652
4	0.9717	0.9217	0.9447
5	0.9781	0.9685	0.9729
6	0.9707	0.9837	0.9770
7	0.9652	0.9768	0.9708
8	0.9821	0.8988	0.9353
9	0.9375	0.9486	0.9426

B. Wine Dataset Results

Figure 2 shows the accuracy for varying k values. The accuracy remained relatively stable across different values of k, suggesting that the classifier's performance is less sensitive to k in this dataset.

Table II presents the precision, recall, and F1-score for each class. The metrics are consistent across classes, indicating balanced performance. The use of normalization ensured that

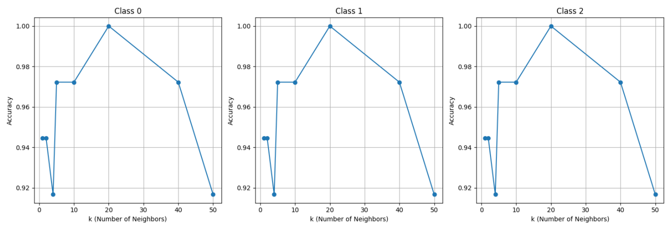

Fig. 2. Accuracy for Wine classes with varying k.

all features contributed equally to the distance calculations, which is crucial in datasets with features on different scales.

TABLE II
PERFORMANCE METRICS FOR THE WINE DATASET.

Class	Precision	Recall	F1-score
0	0.9582	0.9589	0.9563
1	0.9582	0.9589	0.9563
2	0.9582	0.9589	0.9563

C. Computational Time Analysis

I analyzed the computational time required for the kNN algorithm on both datasets. For MNIST, using the full dataset would have resulted in impractical computation times due to the $O(n \times m \times d)$ complexity. By using subsets, I significantly reduced the computation time (e.g., approximately 10 seconds for $k = 5$ and 1,000 test samples), making the experiments feasible.

For the Wine dataset, the small size allowed us to perform without significant computational burden.

IV. Discussion

A. Impact of k on Performance

Our experiments showed that smaller values of k tend to yield better performance on the MNIST dataset. This could be because a smaller k allows the classifier to be more sensitive to the local data structure, which is beneficial when classes are well-separated. However, too small a k may make the classifier sensitive to noise. In the Wine dataset, the performance was relatively stable across different k values, possibly due to the dataset's small size.

B. Normalization Importance

The normalization of the Wine dataset was essential due to the varying scales of its features. Without normalization, features with larger numeric ranges could dominate the distance calculations, adversely affecting the classifier's performance.

C. Implementation Choices

Implementing the kNN classifier with input validation and optional parameters made the function robust and user-friendly. The argument checks ensure that the function is used correctly and that errors are caught early. Providing an optional parameter for test labels allows for flexibility in usage, accommodating scenarios where ground truth labels may or may not be available.

D. Limitations and Future Work

The primary limitation of this study is the reduced subset size for the MNIST dataset, which may not fully capture the classifier's performance on the entire dataset. Future work could explore more efficient algorithms for kNN, such as using KD-trees or ball trees to reduce computational complexity, enabling the use of larger datasets.

V. Conclusions

The kNN classifier performed effectively on both the MNIST and Wine datasets. Key observations include:

- Smaller k values generally yield better performance on datasets like MNIST.
- Normalization is crucial for distance-based methods, especially when features have different scales.
- Implementing argument validation and optional parameters enhances the robustness and usability of the classifier.

This study demonstrates the importance of careful implementation and evaluation in machine learning tasks, ensuring that algorithms are both effective and efficient.