K-Nearest Neighbours Classifier
Assignment 3 of the Machine Learning 1

Basit Akram
5161322

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
Università Degli Studi di Genova

1 Introduction

One of the most common classifiers is the *k-Nearest Neighbor classifier* (kNN), a supervised learning algorithm that estimates how likely a data point belongs to one class or another depending on which class its 'k' nearest instances (neighbors) belong to.
It is a non-parametric model, so it does not make assumptions about the data beforehand like in linear regression where the data must be linear. kNN is referred to as a *Lazy learner* because it doesn't do anything in the training phase.

2 Theory of KNN Classifier

Given an input point ($\bar{x}$), called *query point*, the goal is to classify this point using all the 'k' points around him. The classifier then returns the predicted class y as output, based on a certain decision rule.
In kNN this rule is based on **majority vote**, hence $\bar{x}$ will be classified with the class y_1 if the majority of the k-nearest objects belong to that class, otherwise it will be classified with the other class.
Let's take **x** as the input of the kNN algorithm, and x_i as a single observation of the training set:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix} \tag{1}$$

To determine the nearest neighbors we need to choose a distance metrics. For this purpose we choose the *Euclidean distance*, and we compute it between the query point $\bar{x}$ and each observation x_i:

$$N = (\|x_1 - \bar{x}\|, ..., \|x_n - \bar{x}\|) \tag{2}$$

Then we consider the first k closest points to $\bar{x}$:

$$\{n_1, ..., n_n\} = top_k \|x_i - \bar{x}\| \tag{3}$$

And finally, the class given to $\bar{x}$ is the one that appears the most:

$$y = mode\{t_{n_1}, ..., t_{n_k}\} \tag{4}$$

Where t_{n_k} is the class of the class *k*.
To observe the performances of our model we can use a *Confusion Matrix* and other performance metrics.

2.1 Confusion matrix

Here we will take a brief view of how is composed a confusion matrix, it is a 2×2 matrix:

- *True Positives (TP)*: the actual value is positive and the predicted value is also positive.
- *True negatives (TN)*: the actual value is negative and the predicted value is also negative.
- *False positives (FP)*: the actual is negative but predicted value is positive.
- *False negatives (FN)*: the actual is positive but the predicted value is negative.

To compute these values we need to use other performance indices:

- *Precision*: ratio of the total number of correctly classified positive classes divided by the total number of predicted positive classes.
- *Accuracy*: ratio between the number of correct predictions and the total number of predictions;
- *Sensitivity*: total number of correctly classified positive classes divide by the total number of true positive classes;
- *Specificity*: tells us what fraction of negative samples are correctly predicted as negative;
- *F1-score*: is a combination of precision and recall with β=1.

3 The assignment

The problem is composed of three tasks:

- Obtain the data set
- Build a kNN classifier
- Test the kNN classifier

3.1 Obtain the data set

For this assignment, we used the MNIST data set, consisting of 70000 handwritten digits (from 0 to 9 representing the classes) in $28 \times 28px$ greyscale images, already split 60000 images for the training set and 10000 images for the test set.
Since we are dealing with a lot of data and the kNN needs to do a lot of computations, we use 5% (random) elements of the training set and the test set.
Instead of the usual *Pandas* library, this time we use the *Keras* library alongside *Python*.

3.2 Build a kNN classifier

We implement a k-Nearest Neighbour classifier as a function for the second task. Before classifying the data, we check the correctness of the input with the function: *check_data()*. This function check that we select a valid number for the parameter *k* and that the train and test sets have the same number of columns.
Now, after the preliminary checks, we can classify the data set according to the theory in section 2, and return the classification obtained.
The function that makes the classification is called *kNN()*, and operates as follows:

1. calculates the distances between the test images and the training images;
2. finds the k nearest neighbors;
3. Decides the most common class in the k nearest neighbors via majority vote.

Finally, the function returns the error rate if we provided the true output as a parameter of the function otherwise, it returns the prediction made.

3.3 Test the kNN classifier

In the last task, we run our kNN classifier using different values of k for each digit against all the others:

$$K_1 = \big[1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50\big]$$

This time we take 10% of the data.
We save the accuracy scores for each k and for each digit, then plot a graph (figure 1). The figure shows that above $k = 10$ the performances suffer a lot, so an optimal solution could be to take $6 < k < 11$. Of course, the performances also depend on the dimension of the data used.
Then, we compute the confusion matrix for each digit and for each k. Afterward, we use these matrices to compute the sensitivity and the F1 score for each digit and each k. The result can be observed in the graph in figure 2 and 3.
Note that we don't compute specificity and precision, the first can't be computed and the latter is always 1.
We also print histograms to show the average sensitivity and F1 score for each digit. For how we selected the data (10% random) we can see that digits 0 and 1 perform better.

4 Results

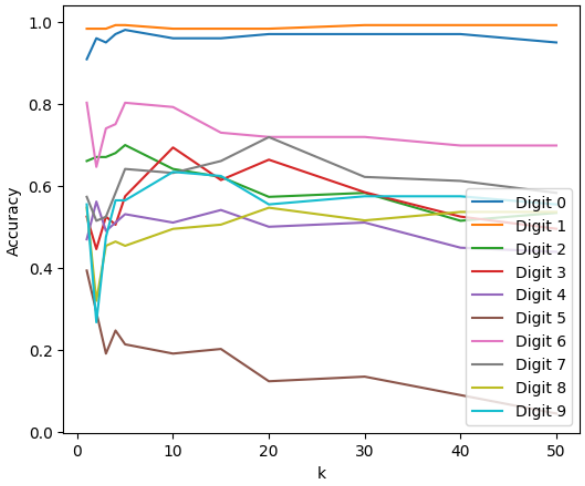

Figure 1. Accuracy and Error rate for several values of k.

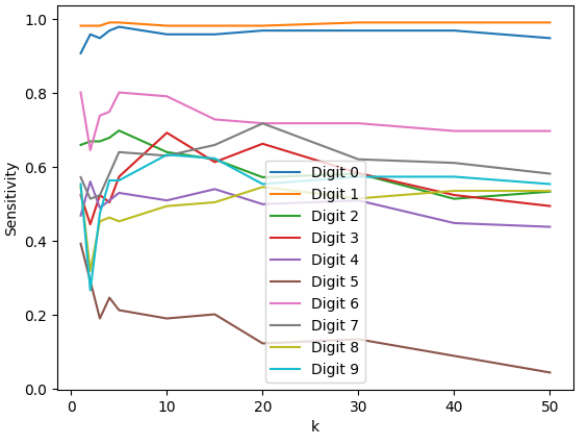

Figure 2. Confusion matrix for different values of k.

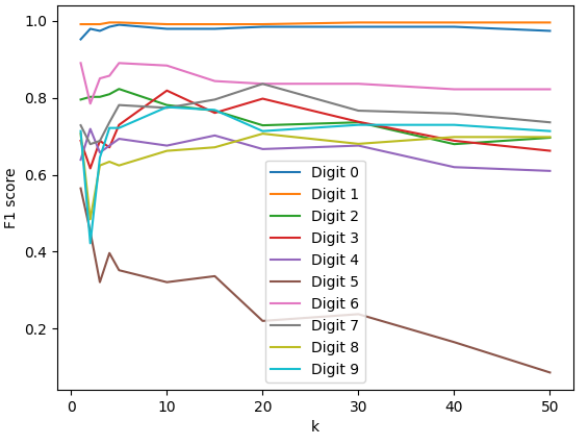

Figure 3. Quality indexes for different values of k.

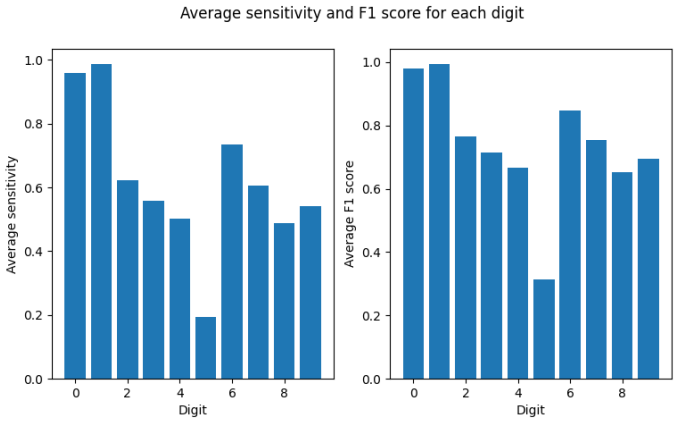

Figure 4. Accuracy for each digit vs remaining nine.