KNN CLASSIFIER

November 24, 2019

Alberto Grillo ID: 4229492
Alberto Ghiotto ID: 4225586

***Abstract*— Non-parametric statistics makes no assumptions about probability distributions, therefore, non-parametric models have a complexity that is not pre-defined but depends on data. They don't explicitly implement a model with parameters but directly build a discriminant rule from data. The k-nearest neighbors algorithm (kNN) is a simple, easy-to-implement supervised machine learning algorithm based on a non-parametric method, it can be used for both classification and regression problems. The kNN algorithm is based on the principle that similar things exist in close proximity, it assumes that similar things are near to each other. This algorithm captures the idea of similarity by calculating the distance between points. In the case examined in this project, this algorithm will be exploited for classification purposes. The classifier will give as an output a class label, obtained by majority voting from the k nearest neighbours, where k is a positive integer. In this laboratory the objective is to build and test a kNN classifier, computing the accuracy on the test set for several values of k and among different digits.**

I. INTRODUCTION

In this assignment some guidelines on how to develop the project were given, along with a dataset on which to test the kNN classifier to be built. The provided dataset, known as MNIST dataset, consists in a collection of handwritten digits in 28x28 greyscale images, and is a standard benchmark for machine learning tasks. It has a substantial dimension, with a training set of 60'000 samples and a test set of 10'000 samples. Together with the dataset, some functions to load the data on MATLAB were provided.
The first task was to simply exploit the above function to load the data on a MATLAB script.
The second task was to actually build the classifier by keeping in mind some guidelines on how to develop the classifier in a dedicated MATLAB function with the relative input parameters. It was requested to perform dimensional checking of the train and test set and check the cardinality of k.
Eventually, the classifier function would have to return the classification obtained and, in the case where the ground truth label vector was provided, it would also have to compute and return the computed error rate.
The third and last task consisted in testing the kNN classifier by computing the accuracy on the test set in two different scenarios: on 10 tasks with each digit vs the remaining 9 and for several values of k.

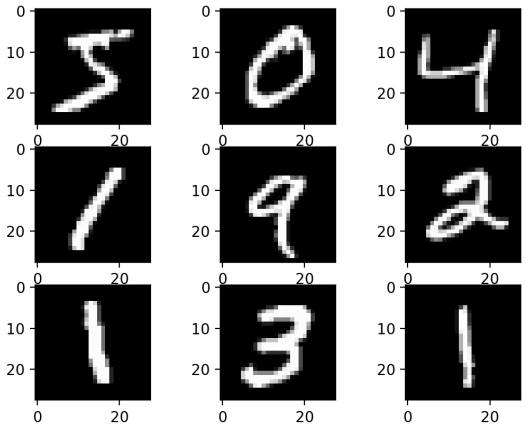

Fig. 1. Some example digit from the MNIST dataset

II. BACKGROUND

In this section, a brief explanation of the implemented principles and equations will be given. The detailed steps can be found in the class slides.

The kNN classifier algorithm works by implementing the following steps:

1) Compute the distances between two set of points: the query point, called $\bar{x}$ and the training set.
2) Take out the k smallest distance with relative class labels.
3) Compute the mode among those k values, the result will be the classifier's prediction.

III. Implementation and Results

A. Task 1 - Get the data

As anticipated in the introduction section, this task simply consisted in loading the data in the MATLAB script by using the provided function.

B. Task 2 - Build the kNN classifier

Task 2 consisted in developing a function which implements the classifier while following some directions.
In particular it was requested that the function would take as parameters:

1) The train set matrix
2) The test set matrix
3) The number of nearest neighbours to evaluate
4) The ground truth label vector (not mandatory)

Then it was requested to perform dimensional checking of the train and test set and check the cardinality of k (i.e the third parameter), in order to ensure that the input parameters were consistent.
After these tests the actual classification part takes place. For each element of the test set, i.e the current query point, the function computes the Euclidian distance from each element of the train set. From these quantities it will be necessary to extract the k smaller values, namely the k nearest neighbour, on which to compute the mode which will give out the predicted class label. Computing the Euclidian distances it is quite challenging computational-wise because it requires to iterate multiple times along the dataset: it's necessary to compute 60'000 distances for each one of the 10'000 elements of the test set, given a total number of iteration of 600'000'000.
The process of computing the distances and extracting the k nearest neighbours can be entirely done by exploiting the MATLAB function *"pdist2"*, which can compute the requested distances and return the indexes of the k smallest values. The mode among the nearest neighbours, i.e the predicted class, can be simply computed by using the MATLAB function *"mode"*. Before eventually returning the prediction vector, the function checks if the fourth parameters was indeed given in input and, if that is the case, it computes the classifier's error rate by comparing the inferred classification with the ground truth vector and returns this value along with the prediction vector.

C. Task 3 - Test the kNN classifier

In the last task it was requested to test the classifier with the provided MNIST dataset in two different ways:

1) On 10 tasks: each digit vs the remaining 9 (i.e: recognize whether an observation is or not a given number).
2) For several values of k (e.g: k = 1,2,3,4,5,10,15,20,30,40,50) by taking into account that k should not be divisible by the number of classes to avoid ties.

The kNN function was developed following the given guidelines, which require to compute the distances every time a new instance is made. In order to avoid having to modify the former and maintain an acceptable computation time preventing a lot of redundant calculation, two addition matlab script were developed for the testing phase. This two program are entirely similar to the kNN function with the only exception that they compute the distances matrix using *"pdist2"* only one time while computing the error rates for several values of k. This is done by extracting the k nearest neighbours from the computed distances matrix using the MATLAB function *"mink"*. In order to at best evaluate the result, it was chosen to compute the error rates for every odd value of k ranging from 1 to 201 for both tests. The values were chosen to be odd in order have k not divisible by the number of classes to avoid ties in the voting mechanism (i.e the mode).
The results of the first test are shown in figures 1 to 10 in the appendix section, where it can be seen the obtained accuracy (i.e the number of correct classification over the total number of samples) of each digit vs the remaining nine.
These results highlight that for big values of k the accuracy drops down, due to the fact that too many neighbours are considered and the computation of the mode takes into account values that are not actually similar to the one in evaluation. On the other hand, a too small value of k can also lead to errors in the case of an observation for which the assumption about the proximity of similar elements doesn't holds.
In figure 11 it can be seen the result of test 2 where, as seen before, the accuracy drops down as k increases.

IV. Conclusions

The kNN classifier has the main disadvantage of becoming slower as the volume of data increases. This is due to the fact that there isn't an actual training phase in which the model gets developed, hence it's not suitable for applications were the prediction needs to be fast. The lack of a model to be trained can be somehow seen as an advantage: there is no need to tune several parameters or make additional assumptions. This feature makes this classifier very simple and easy to implement, provided you have sufficient computing resources to speedily handle the data. The kNN can still be useful in solving problems which have solutions that depend on identifying similar objects, such as recommender system.

V. Appendix

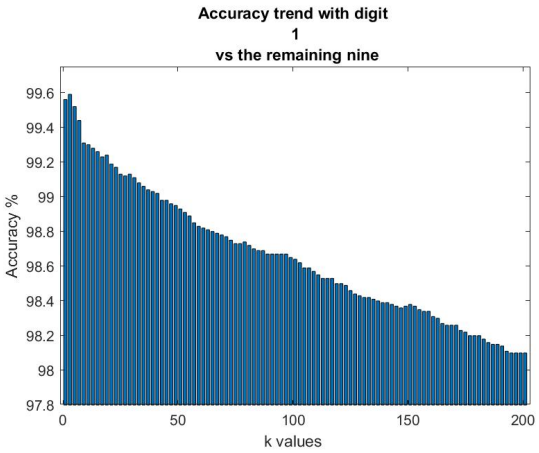

Fig. 2. Test 1 - 1 vs other

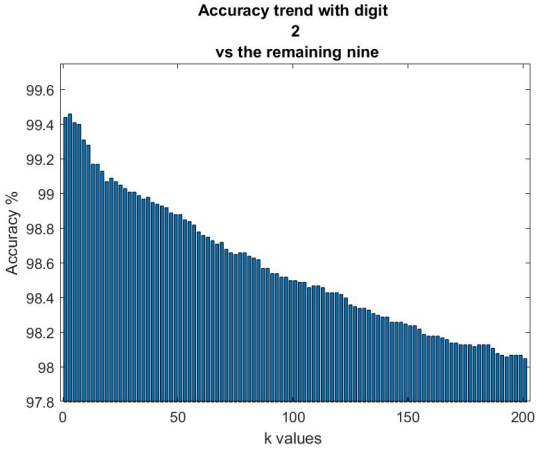

Fig. 3. Test 1 - 2 vs other

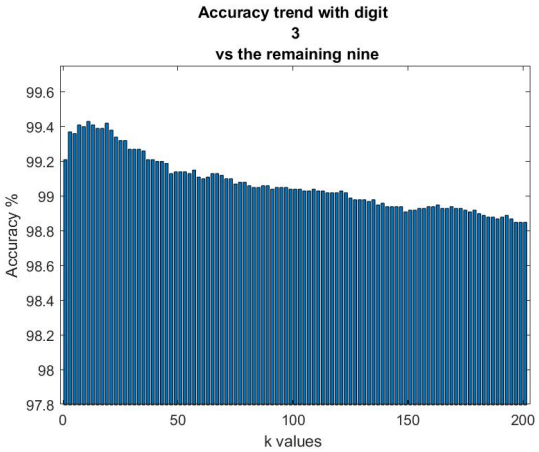

Fig. 4. Test 1 - 3 vs other

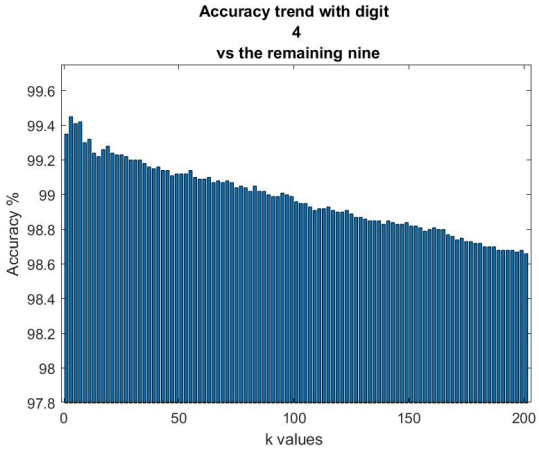

Fig. 5. Test 1 - 4 vs other

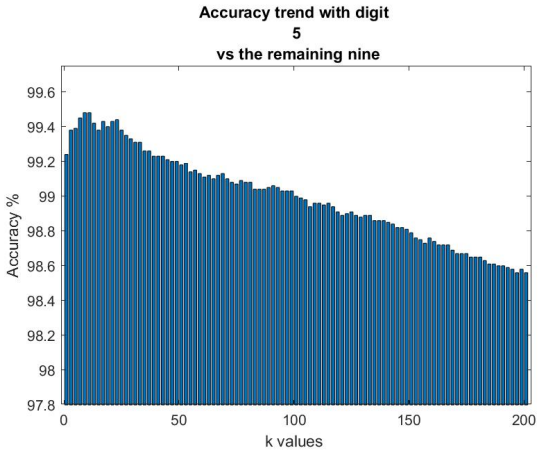

Fig. 6. Test 1 - 5 vs other

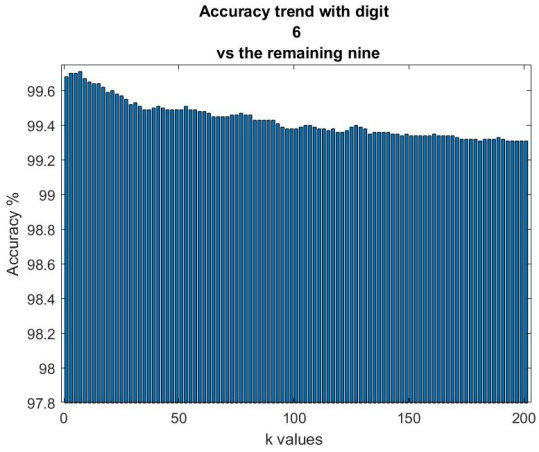

Fig. 7. Test 1 - 6 vs other

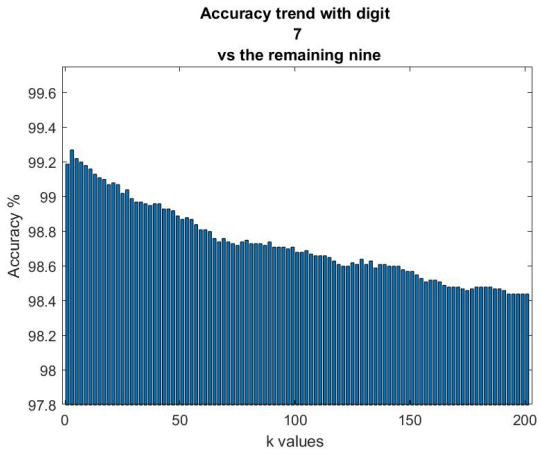

Fig. 8. Test 1 - 7 vs other

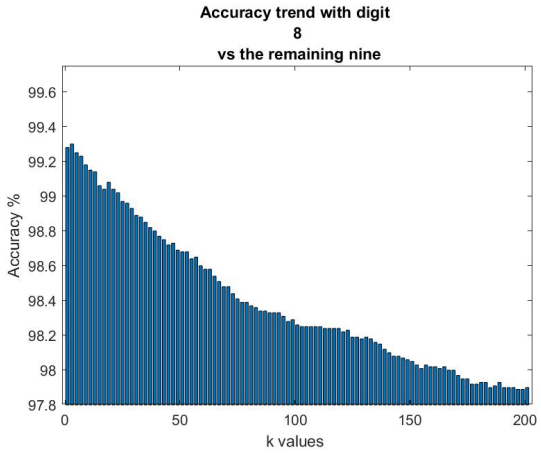

Fig. 9. Test 1 - 8 vs other

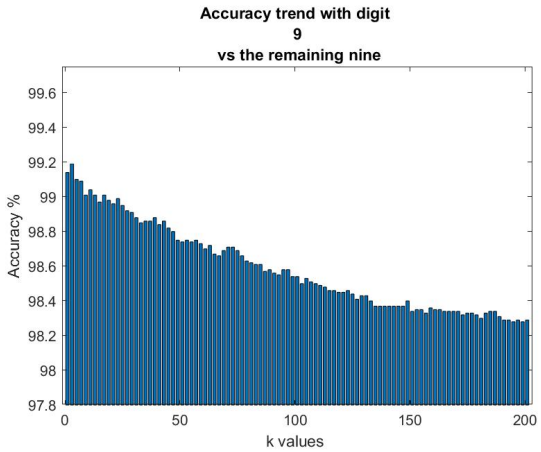

Fig. 10. Test 1 - 9 vs other

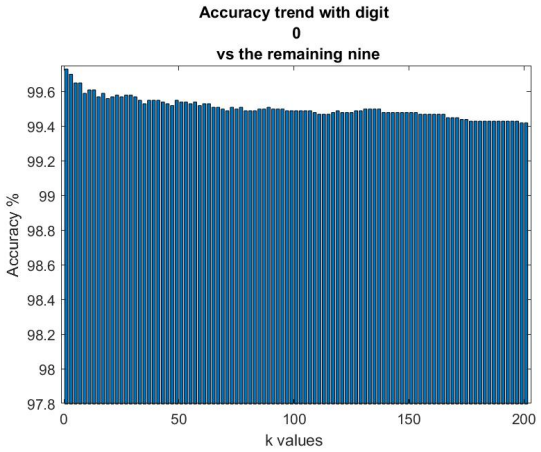

Fig. 11. Test 1 - 0 vs other

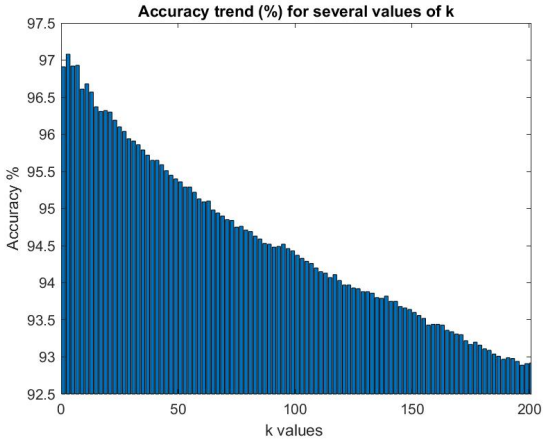

Fig. 12. Test 2