Designing classifiers for hand written numbers and iris-species

Project in TTT4275 - Estimation, Detection and Classification

S. L. Breivik[a], M. Vårdal[a]

[a] *Department of Engineering Cybernetics, Norwegian University of Science and Technology, 7491 Trondheim.*

Summary

The project source code: [1]

Contents

1. Introduction

The goal of this project is to develop an understanding of different types of classifiers and how they are implemented. Part of the task is to create a classifier for the classification of images of handwritten digits. This is a tool that can be used to automate tasks that would otherwise be done by human beings, for example in check processing or in digit recognition for the postal service.

In chapter 2, the theory needed to understand the tasks is discussed. In chapter 3, we take a look at the task that we chose. Chapter 4 focuses on how we implemented our solution, in addition to discussion of the results. Finally, in chapter 5, we write a conclusion for the report.

2. Theory

This chapter is a presentation of the theory needed to understand the task. We will first look at the theory necessary for understanding the linear classifier. Then, we will take a look at the theory underlying the nearest-neighbor classifier. This will also include the concepts of clustering and majority vote for the k-nearest neighbors. Most of the equations and theory is collected from a compendium about classification[2], handed out in the subject TTT4275 Estimation, detection and Classification at NTNU. The figures in the theory part are made using geogebra classic.

2.1. The linear classifier

A classifier is trying to answer the question of which class a given instance belongs to. This is practically done by comparing discriminant functions[1]. If we partition (divide into parts) the whole input space according to which class is most probable, then we have a classifier. Said mathematically this is given by the decision rule below 1,

$$x \in \omega_j \iff g_j(x) = \max_i g_i(x) \tag{1}$$

x (the thing we want to classify)is in the class (ω_j) that has the highest value for the discriminant function (g_j). The linear discriminant classifier is defined by the function[2, p. 9]

$$g_i(x) = \omega_i^T x + \omega_{io}, \qquad i = i, \ldots, C. \tag{2}$$

Here, ω_{io} is the offset for the class i. In the 2d-case this classifier is given by straight lines. If the number of classes C is greater than 2, then this can be written in the more compact form

$$g = Wx + \omega_o \tag{3}$$

where g and w_0 are column vectors of dimension C. The matrix W then has C rows and D columns, where D is the number of features in x. [2] An example of a linear classifier is given in Figure 1

2.2. MSE based training of a linear classifier.

Before it is possible to talk about training of a linear classifier, we need to first define how the performance of the classifier is to be measured. There are several ways of doing this, and one common way is to measure the performance in terms of the Minimun Square Error (MSE) of the classifier. In other words, we want to minimize the sum of the square of the difference between a prediction from the classifier and the target value. Only now we can start determining what the classifier should look like. But first, let us rewrite

[1] functions which gives the probability that an instance belongs to a given class.

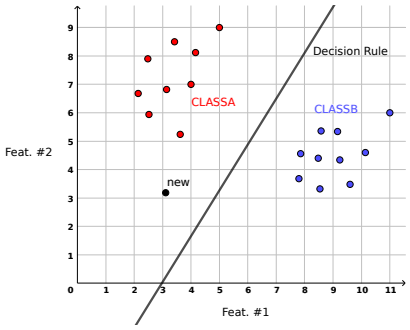


Figure 1: An example of a linear classifier, splitting the input space of two features, and deciding which classes new inputs belong to. For example the black point, showing a new instance would be classified as Class A. A matrix W is specifying the decision rule. Feat. stands for feature and is a defining property of classes.

equation 3. We can make it more compact by performing the following two redefinitions,

$$\begin{bmatrix} W & \omega_o \end{bmatrix} \rightarrow W$$

$$\begin{bmatrix} x^T & 1 \end{bmatrix} \rightarrow x$$

The equation offset is now embedded in the feature-vector 'x'.

$$g = Wx. \tag{4}$$

We define the loss function as follows,

$$MSE = \frac{1}{2}\sum_{k=1}^{N}(g_k - t_k)^T(g_k - t_k) \tag{5}$$

The output vector $g_k = Wx_k$ corresponding to a single input x_k. We need to match this output to the target vector (which is the correct label) with the values 0 and 1. Since we need a smooth function, with a derivative, we use the sigmoid function. This is given in equation 6 and shown in Figure 2. This is basically just a more curved step function.[2].

$$g_k = \frac{1}{1+\exp^{-z_{ik}}}, \qquad z_{ik} = Wx_k. \tag{6}$$

In order for us to train our model, we need to know the steepest direction of decrease for the MSE. Therefore we calculate the gradient of the MSE with respect to the matrix W. This calculation uses the chain rule,

$$\nabla_W MSE = \sum_{k=1}^{N} \nabla_{g_k} MSE \, \nabla_{z_k} g_k \, \nabla_W z_k.$$

And if we use the following simplifications

$$\nabla_{g_k} MSE = g_k - t_k$$
$$\nabla_{z_k} g = g_k \circ (1 - g_k)$$
$$\nabla_W z_k = x_k^T,$$

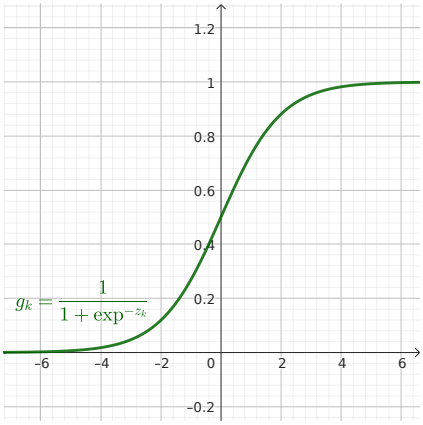


Figure 2: Plot of the sigmoid function, that converts a continous value to a nearly binary value. The reason why we need a differentiable function is so that the gradient is defined.

we get the following result

$$\nabla_W MSE = \sum_{k=1}^{N}[(g_k - t_k) \circ g_k \circ (1 - g_k)]x_k^T. \tag{7}$$

Where $\circ$ stands for elementwise multiplication. We can now move the matrix W an appropriate step length α in the direction opposite of the gradient,

$$W(m) = W(m-1) + \alpha\dot{\nabla}_W MSE, \tag{8}$$

where m is the iteration number. If the step length is chosen wisely and we update W according to equation 8, the solutions will converge towards a local (and possibly global) minimum. This way we end up describing a classifier based upon our trained up "probability"-like matrix W

2.3. The nearest neighbor classifier

The nearest neighbor - NN classifier is based on a very intuitive idea. The idea is that we measure the "distance" between a set of features with unknown label to all the samples from the training set with known labels. Then, we simply classify the unknown sample as the class of its nearest neighbor.

The "distance" from one featureset to the next can be defined in a number of different ways. One of those ways is to define it as the Euclidean space.

The Euclidean space between two vectors in $\mathbb{R}^n$, $p_1 = (x_1, x_2, \ldots, x_n)^T$ and $p_2 = (y_1, y_2, \ldots, y_n)^T$ is defined as follows,

$$d(p_1, p_2) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

3. The task

We chose the task where we were supposed to design classifiers for images of different iris-species and of different handwritten digits, respectively. The reason we chose

this task in particular was because we both thought that classification was the most interesting topic. Furthermore, we thought that the one about classification of images of handwritten digits seemed like the most exciting one at first glance.

To describe the task more specifically, the first main task was to design a linear classifier that could classify a set of features of a given Iris flower as one of three subspecies: Iris-setosa, Iris-versicolor or Iris-virginica. The features were sepal length, sepal width, petal length and petal width. To do this, we were given a dataset consisting of 150 sets of the aforementioned features together with the corresponding subspecies - or target value. There were 50 sets of features and targets for each of the three subspecies.

First, we were going to train the classifier on the first 30 samples in the dataset for each subspecies, and test it on the remainig 20, and find the confusion matrix and error rate. After this, we were going to repeat what we had done, but with the last 30 samples as the training set and the first 20 as the testing set.

In the second and last part of the Iris task, we were going to produce histograms of all the features for the different subspecies. This was done so that we could try to eliminate the feature(s) with the most overlap between the subspecies in order to see if this increased linear separability and thereby also the performance of the classifier. First, we eliminated one feature, then another, and after that, another one. Then we compared the results for the classifier based on four, three, two and one feature(s), respectively.

The second main part of the task was to design some classifiers for classifying handwritten digits, based on the MNIST database. The MNIST database is a database containing 70'000 images of handwritten digits and the corresponding target values. The dataset is divided into a training set of 60'000 images and a test set of the remaining 10'000 images. The first classifier we were going to desing for this dataset, was the simple Nearest-Neighbour classifier. Then, we investigated its performance via a confusion matrix and the error rate. Furthermore, we plottet a few of the correctly classified digits and a few of the misclassified digits, and discussed these results.

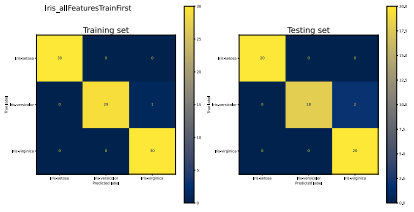
4. Implementation and results

In this chapter, we will discuss the implementation of our solutions to the different tasks as well as the results. All of our solutions were written in Python. The way we implemented the linear classifier for the Iris task was by first organizing all the data in a class that we called "Dataset". This allowed us to structure the training and testing data in a very clean way. The training data was put in the Dataset.trainig dictionary where Dataset.training["features"] contained the features and Dataset.training["targets"] contained the targets. The same was done for the testing samples. After the initial oranization of the data was done, we used the function "TrainClassifier" to train our classifier. This function started with a matrix W that contained random values and updated it 15000 times with a step length of 0.01 in a accordance with 7 and 8.

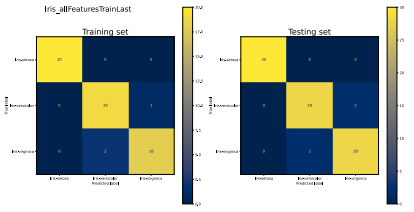
After this, what remained was to test the classifier that we had trained. This was done by using our "TestClassifier" function. This function tested the classifier on the test samples, created confusion matrices and calculated the error rate.

In the MNIST task, we started by organizing the code in the same way as with the Iris data. That is, we downloaded the MNIST dataset through the Keras library in Python, and added it to an instance of our Dataset class. The Dataset class in this task was defined very similarly to the one from the Iris task. Then, we defined an NN_predict function that takes as input a set of images and returns a list of the prediction for each of the input images. The way it does this is to iterate through the image_predict input variable, which contains an array of all the images we want to predict. For each of the images, it calculates the Euclidian distance from this image to all the others, and classifies it as the class of that image for which the distance is the smallest. To avoid cumbersome calculations with very big distance matrices, we divide the training set into chucks of 1000 and do the calculations for each of the chunks respectively, and save the nearest neighbor in each chunk. After all the calculations are done for all the chunks, we simply find the absolute nearest neighbor by comparing all of the nearest neighbors from the different chunks.

To test this classifier, we made confusion matrices and calculated error rates, similar to what we did in the Iris task. This was done with the "conf_matrix_test" function which used the "NN_predict" function to make a prediction for the test set of images from the MNIST dataset.



(a) Training first



(b) Training last

Figure 3: Confusion matrices for all features

5. Conclusion

test

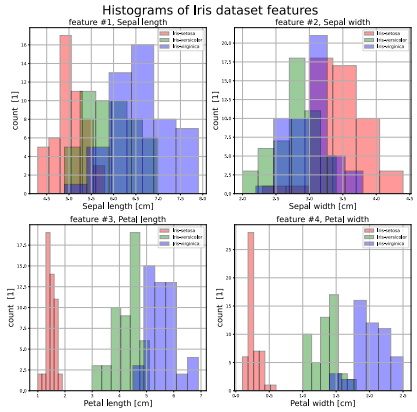
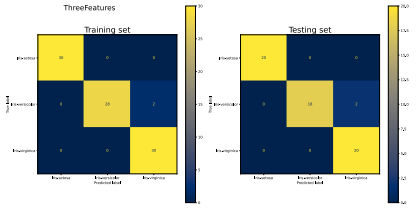
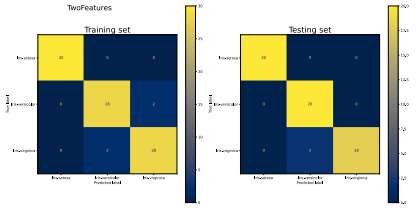


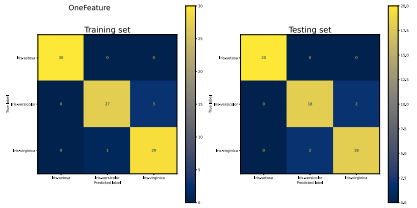
Figure 4: Histograms



(a) Three features



(b) Two features



(c) One Feature

Figure 5: Confusion matrices for different number of features

References

[1] Breivik, S. L. and M. Vårdal: *our classifier*. https://github.com/simon-cmyk/Classifier, 2023.

[2] TTT4275 Estimation, Detection and Classification, NTNU, M. H. Johnsen: *Classification*. 12.18.2017.

Table 1: Iris classifier error rates. The different features are abbreviated. So 'S' stands for Sentipetal, 'P' for petal, 'L' for length and 'W' for width. This way PW stands for Petal width. The last two columns stands for error rates, given in percent. The subtext specifies which training set it belongs to. The first row belongs to all features with the first 30 examples for training, while the second belongs to all features with the last 30 examples for training

SL	SW	PL	PW	$Err_{train}[\%]$	$Err_{test}[\%]$
✓	✓	✓	✓	1	3
✓	✓	✓	✓	5	4
✓		✓	✓	2	3
		✓	✓	4	3
			✓	4	7

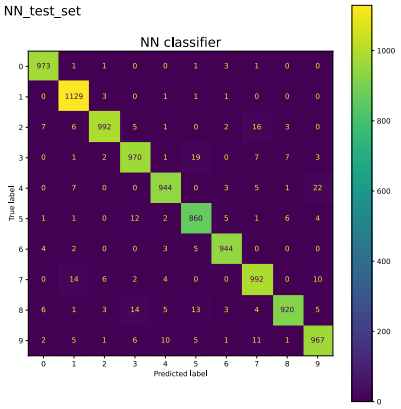
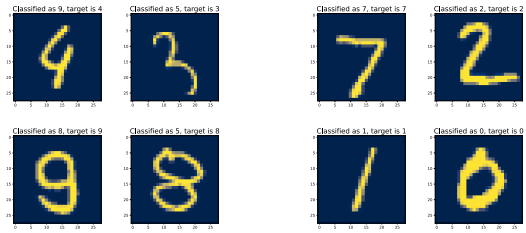


Figure 6: Confusion matrix for nearest neighbour test set



(a) Examples of errors **(b)** Examples of successes

Figure 7: Examples of classifications

Table 2: results from MNIST Tasks. Displayed are the type of classifier (if it is using clustering and majority vote. TODO, change this.). Also shown are the error rates and execution time. It should be noted that the clustering-time (210 seconds) is not included. Also this time is completely dependent on which system it runs on.

Cluster	k-value	$Err[\%]$	$t[s]$
	1	0.04	739
✓	1	5	95
✓	7	7	93

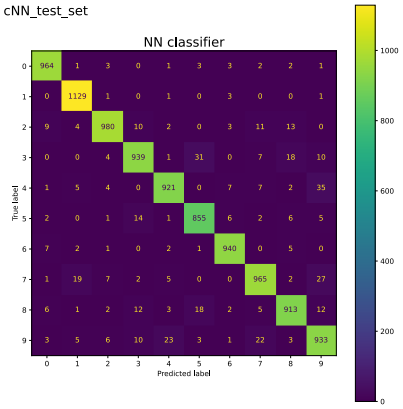


Figure 8: Confusion matrix for nearest neighbour test set with clustering