Classification - Iris and Handwritten Numbers

Øyvind Reppen Lunde and Stian Jakobsen

April 2020

Contents

1 Summary **1**

2 Introduction **2**

3 The Iris Task **3**

3.1 Theory . . . 3

3.2 Design and training . . . 5

3.3 Features and linear separability . . . 6

3.4 Conclusion . . . 10

4 Classification of handwritten numbers **12**

4.1 Theory . . . 12

4.2 NN-classifier . . . 12

4.3 Clustering . . . 15

4.4 Conclusion . . . 17

1 Summary

The course TTT4275 - Estimation, detection and classification introduces the students to different basic theory and techniques of estimation, detection and classification, and their use for ICT systems and signal processing.

This report concerns the classification part of the course. The project is divided in two parts, one about linear classifiers for classifying different variants of the Iris flower, and the second is about nearest neighbour classifiers for classifying handwritten numbers.

In the Iris task we were to classify flowers into three different variants of the Iris flower. To do this we used a linear classifier, which is a classifier that is very simple to both design and train.

The linear classifier had a fairly good performance, considering its simple nature. For the first class it was almost perfect, whereas for the last two classes it wasn't quite as good. This is related to the matter of linear separability.

The handwritten numbers task was more complex, and for this task we used a nearest neighbour classifier, which classifies numbers by looking at other known handwritten numbers and chooses the one that the number is most similar to.

The classifier is quite accurate, predicting about 97 % of the numbers correctly, but for large data set it becomes very slow. To counter this we used clustering, which reduced the accuracy somewhat, but made the classifier much faster.

2 Introduction

In this report the goal is to gain some knowledge about the k-nearest neighbors algorithm as well as the linear classifier. These are two classification algorithms that can be used to assigns items in a collection to target categories or classes. In today's society where analysing large datasets becomes ever more important, we need tools to extract useful information and categorize it into something we can understand. To meet this challenge different methods has been developed, among them classification. Common uses are everything from predicting weather to predicting likely medical outcomes.

We start each chapter by introducing some relevant theory about the classification algorithm in question, before we go on to discuss its implementation and results. The performance of each algorithm is understood by analyzing confusion matrices for different training and test sets.

The Iris task is programmed in Python, whereas the Handwritten numbers task is programmed in Matlab.

3 The Iris Task

The Iris flower has several variants, and in this task we looked at the three variants Setosa, Versicolor and Virginica. Our goal was to design, implement and train a linear classifier to be able to classify different flowers into one of these variants correctly.

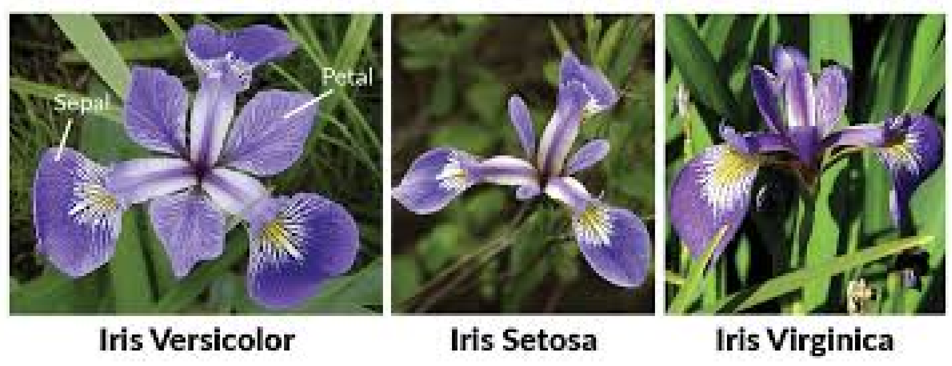


Figure 1: The three different variants of the Iris flower

3.1 Theory

We have three main types of classification problems: linearly separable, non-linear separable and non-separable. The ones that we are interested in are the linearly separable ones, defined as problems where the classes can be separated by a single decision surface, like a line in 2D or a plane in 3D. A graphical representation of the three problems is shown in fig. 2. The Iris task is close to linearly separable, and thus using a linear classifier is justified.

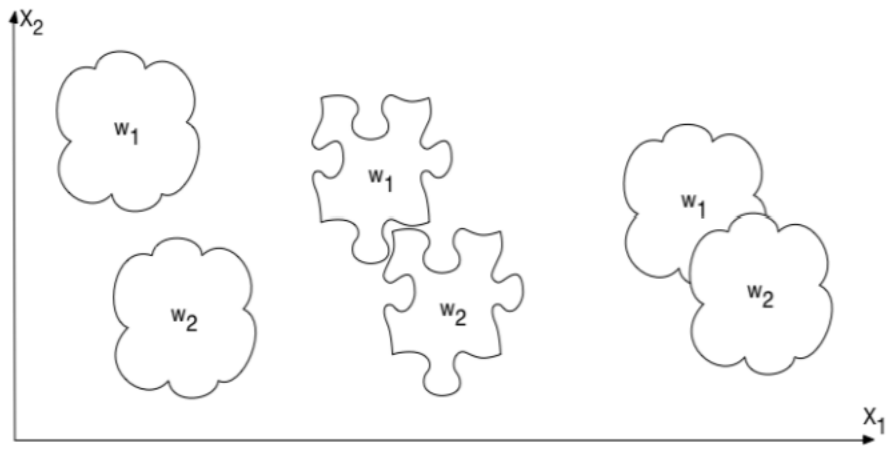


Figure 2: Three different classification problems [1]

The linear classifier is a discriminant classifier. In a discriminant classifier each class has a discriminant function $g_i(x)$, and the classifier has the following decision rule:

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

$\mathbf{x}$ is a vector consisting of a selection of attributes, and these are fed to the different discriminant functions. Equation (1) means that the classifier will choose the discriminant function that returns the highest value.

Concatenating all the discriminant functions gives us a system on the form $\mathbf{g} = \mathbf{Wx}$. Here $\boldsymbol{W}$ is a matrix containing weights for all the classes and system attributes, $\mathbf{x}$ is the attribute vector for a given set of data, and $\mathbf{g}$ is a discriminant vector with one value for each class, and as in eq. 1 we choose the largest element in $\mathbf{g}$ to classify the data.

In order to get our classifier to perform as good as possible, we need to train it. This means optimizing the matrix $\mathbf{W}$. In this project we are using the most popular variant, the MSE (minimum square error):

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

The output vector $\mathbf{g}_k = \mathbf{Wx}_k$ for a single input vector $\mathbf{x}_k$ is continuous, while we want a vector with binary values: one element should be 1, while all the other elements should be 0. A heavyside function would be the best for this, but since it is desirable to have a smooth function with a derivative, we use approximate it with a sigmoid function:

$$g_{ik} = sigmoid(x_{ik}) = \frac{1}{1+e^{-z_{ik}}}, i = 1, ..., C \tag{3}$$

with $\mathbf{z}_k = \mathbf{Wx}_k$.

We optimize $\mathbf{W}$ using the steepest descent method, meaning that we move (update $\mathbf{W}$) in the opposite direction of the gradient. Calculating the gradient and updating $\mathbf{W}$ is done with the following equations:

$$\nabla_W MSE = \sum_{k=1}^{N} \nabla_{g_k} MSE \nabla_{z_k} g_k \nabla_W z_k \tag{4a}$$

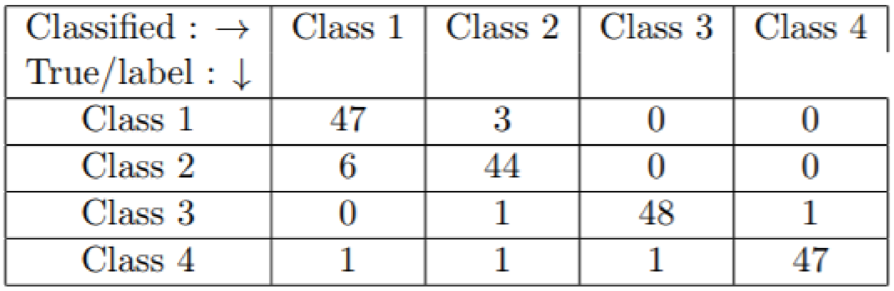
$$W(m) = W(m-1) - \alpha \nabla_W MSE \tag{4b}$$

where m is the iteration number, and α is the step length.

To graphically show the performance of a classifier, we often use a confusion matrix. For a classification problem with n classes, the confusion matrix is

[1]The figure is taken from the course compendium

an $n \times n$ matrix that shows the relation between predicted and actual values for a data set. The better the classifier, the more values are one the diagonal. Figure 3 shows an example of a confusion matrix.



Classified : $\rightarrow$ True/label : $\downarrow$	Class 1	Class 2	Class 3	Class 4
Class 1	47	3	0	0
Class 2	6	44	0	0
Class 3	0	1	48	1
Class 4	1	1	1	47

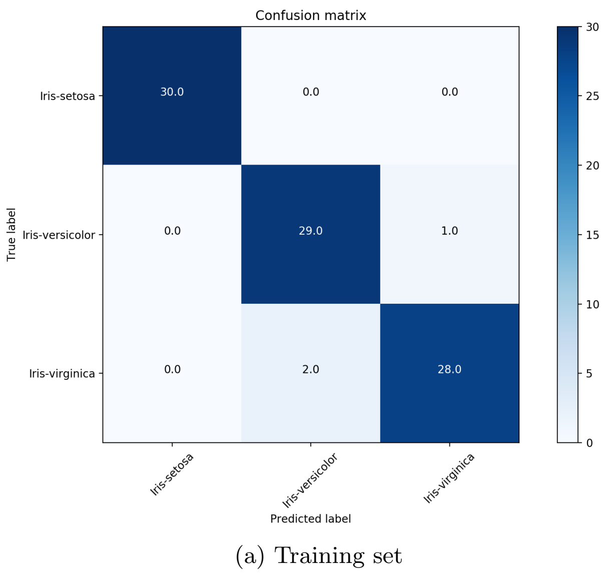
Figure 3: Example of a confusion matrix for 4 classes, with 50 samples in each class[2]

3.2 Design and training

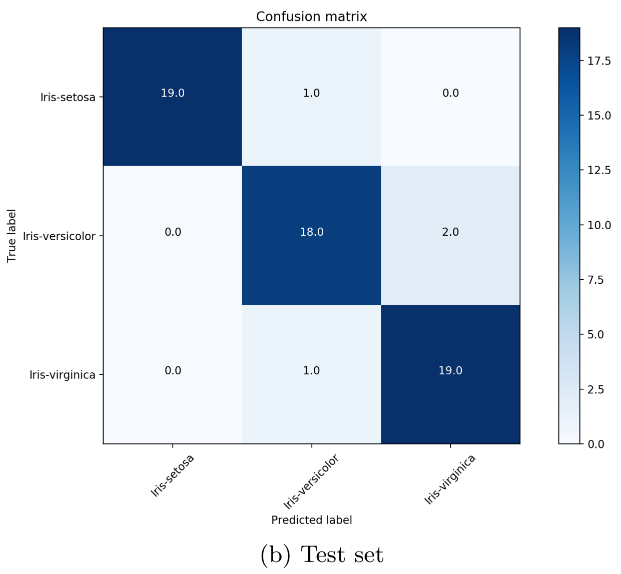
In this part we designed and trained the linear classifier, based on the methods explained in the previous chapter. But before we could do that, we needed to split the data into a training set containing the 30 first samples, and a test set containing the 20 last. Then after implementing the mathematical functions needed to calculate MSE, W and sigmoid as described in section 3.1, we were ready to start training the classifier. We started testing with $\alpha = 0.1$, which yielded an unaccepted error rate of 66 %, meaning that it needed some tuning. To do this we implemented a bactracking algorithm that updates α by a factor of 0.8 for each iteration until either the error rate for the training set goes below 5 % or α becomes too small. If α is too small more iterations are demanded to even get close to the minimum, and because we achieved good results by just imposing a lower bound for alpha, we decided that the slight decrease in error rate did not justify the increase of computational cost. After running the algorithm we found that the optimal step size was $\alpha = 0.041$, giving an error rate of 3.3 % for the training set, and an error rate of 6.7 % for the test set. Figure 4 shows the corresponding confusion matrices confirming that the classifier performs pretty well.

Doing the same procedure for a training set containing the last 30 samples and a test set containing the 20 first samples provided similar results. The error rate was 4.4 % and 1.6 % on the training and test set respectively. The small differences might be a result of using a fairly small dataset making the classifier particularly vulnerable to outliers.

[2]The figure is taken from the course compendium



(a) Training set



(b) Test set

Figure 4: Confusion matrices for the training set (30 first samples) and the test set (20 last samples)

3.3 Features and linear separability

Using the first 30 samples for training and the last 20 for testing gave us the following histograms:

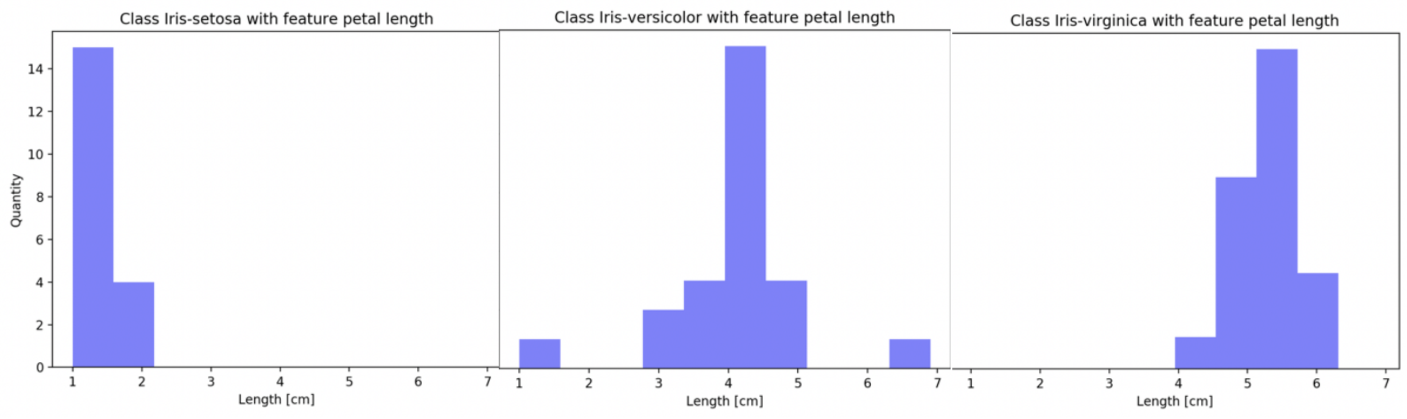


Figure 5: Histogram of petal length for the three classes

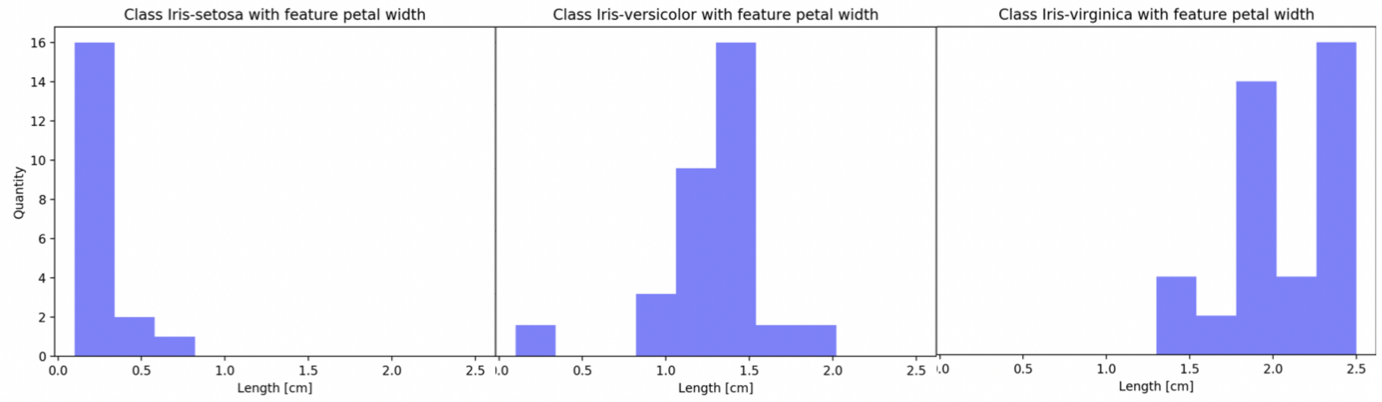


Figure 6: Histogram of petal width for the three classes

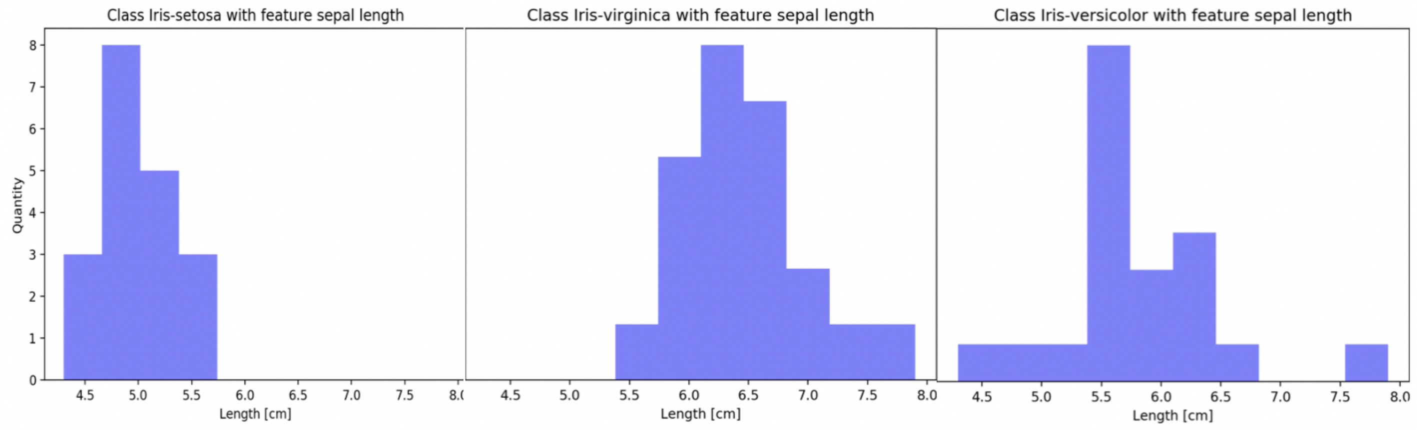


Figure 7: Histogram of sepal length for the three classes

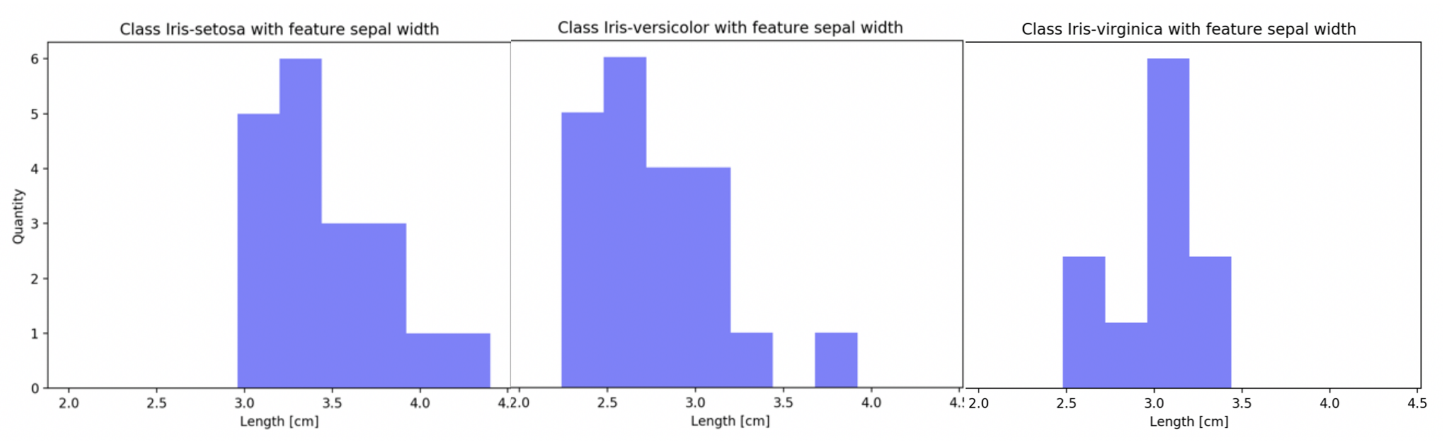
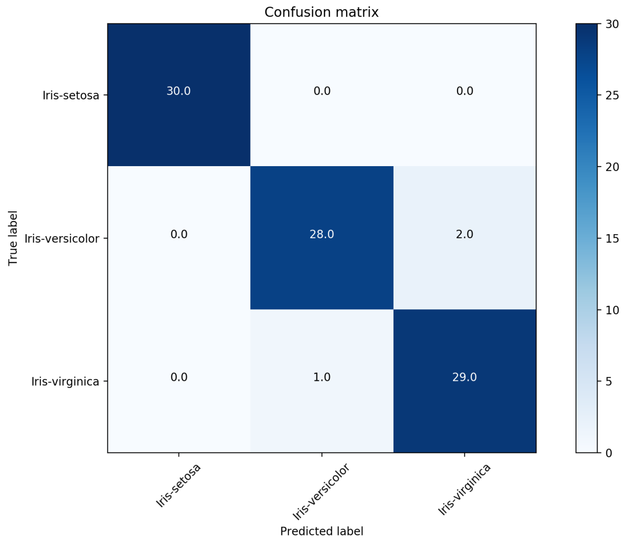
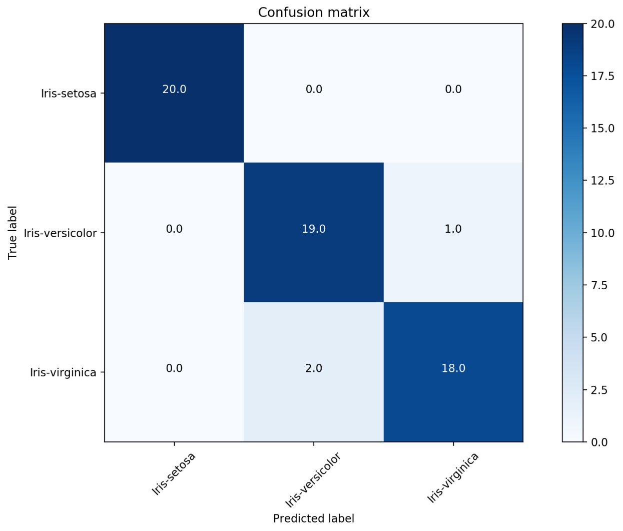


Figure 8: Histogram of sepal width for the three classes

By inspecting the histograms, we see that there is very little overlap between the classes for both petal length and petal width. For sepal length and sepal width, however, there was quite a bit of overlap. We decided that sepal length was the one with the most overlap, and removed it. We then repeated the training and testing process with the remaining three features.



(a) Training set

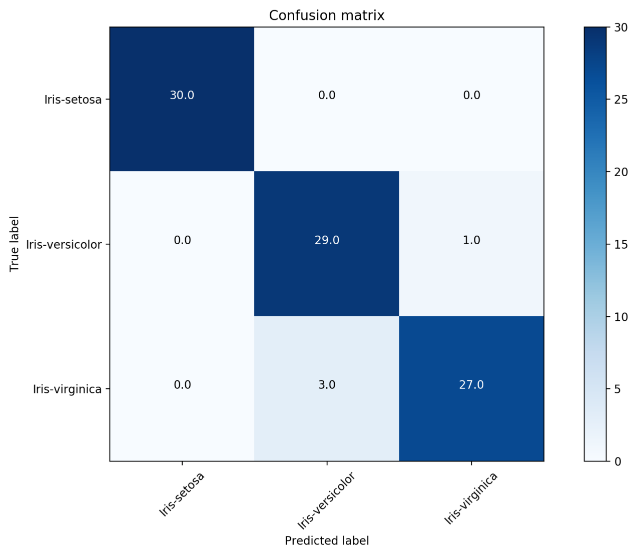


(b) Test set

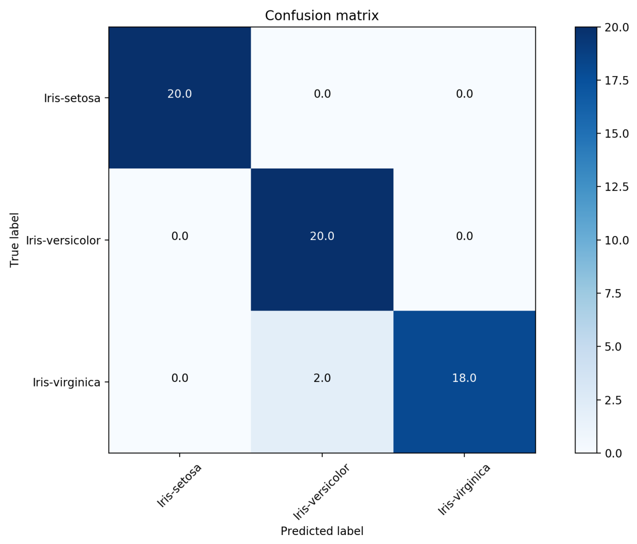
Figure 9: Confusion matrices for the training and test sets, with the sepal length feature removed

We see in fig. 9 that the classifier is still very accurate, with an error rate of 3.3 % for the training set, and 5 % for the test set. Like before, it classifies Iris-Setosa perfectly, but has some misclassifications for Iris-Versicolor and Iris-Virginica.

Next, we removed the sepal width feature and repeated the experiment with the two remaining features (petal length and petal width). That yielded the following results:



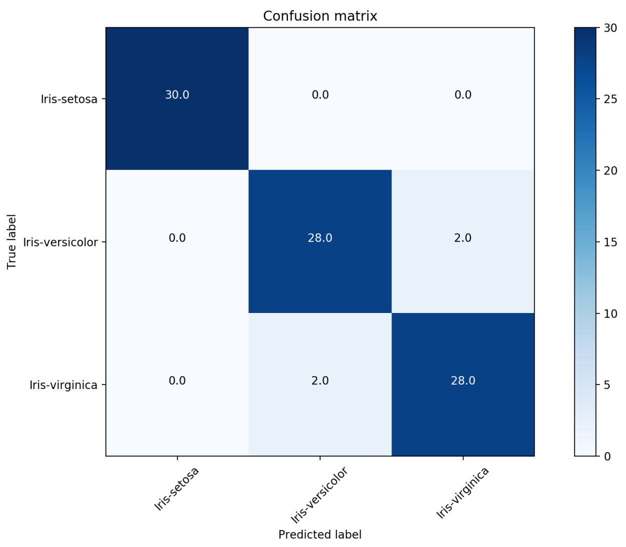
(a) Training set



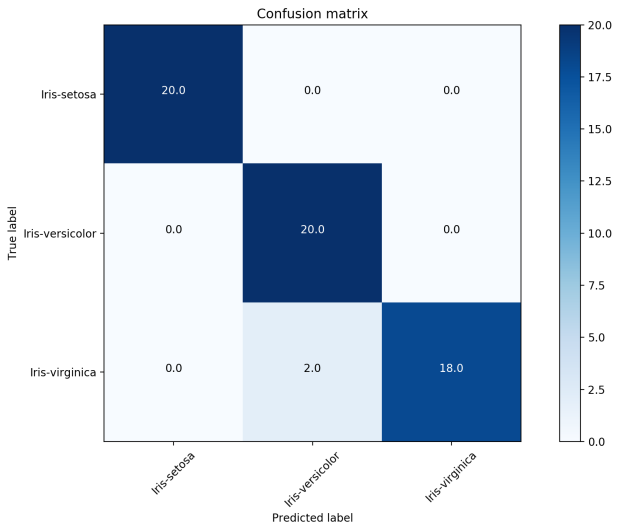
(b) Test set

Figure 10: Confusion matrices for the training and test sets, with the sepal length and sepal width feature removed

This time the classifier had an error rate of 4.4 % for the training set, and 3.3 % for the test set, meaning that this classifier performed slightly better on the test set than the previous classifier that used 3 features. Lastly, we removed the petal length feature and trained and tested a classifier that only used one feature: petal width. This gave us the following results:



(a) Training set



(b) Test set

Figure 11: Confusion matrices for the training and test sets, with the classifier only using the petal width feature

To our surprise, the classifier still had a very good performance, considering how simple it now had become. Error rates of 4.4 % for the training set and 3.3 % for the test set is very impressive for a linear classifier that only uses one feature to classify.

3.4 Conclusion

This task showed that a linear classifier works very well for classifying different variants of the Iris flower. This also supports the initial statement in the task that this problem is close to linearly separable, as that is a requirement for a linear classifier to be accurate. Especially the class Iris-Setosa was clearly linearly separable from the other two classes, as all our classifiers predicted it perfectly.

When inspecting the histograms in section 3.3 we see that it is in particular the petal length and petal width features that makes the Iris problem linearly separable, as their histograms have almost no overlap between the classes, and this explains why the classifiers that only used one or two feature perform so well.

In fact, the classifier actually performed the worst on the test set when it used all four features, and performed the best when using only one or two features. This could be because two sepal features had the most overlap, and their presence might ”cloud” the classifiers judgement a bit. But one should also keep in mind that the data sets are fairly small (90 samples for training and 60 samples for testing). This makes the classifier less accurate than it could have been with a larger training set, and makes the testing more vulnerable to fluctuations and coincidences in the test set, such as outliers (e.g. flowers with unusual sizes).

4 Classification of handwritten numbers

In this task we designed a nearest neighbour (NN) classifier, using the Euclidean distance, to classify handwritten numbers 0-9 from the MNIST database. MNIST consists of 60 000 training examples and 10 000 testing examples, written by 250 persons.

4.1 Theory

In a nearest neighbour classifier, the input $\boldsymbol{x}$ (test data) is compared to a set of references (training data). The classifier finds the reference that is closest (most similar) to $\boldsymbol{x}$, and assumes that it belongs to the same class as the reference.

A common way of finding the distance between the input and a reference is by using the Euclidean distance. The Euclidean distance, or the L2 norm, is the standard straight-line distance between two points. For two points $\mathbf{p} = (p_1, p_2, ..., p_n)$ and $\mathbf{q} = (q_1, q_2, ..., q_n)$ in an n-dimensional space, the Euclidean distance is defined like this:

$$d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{5}$$

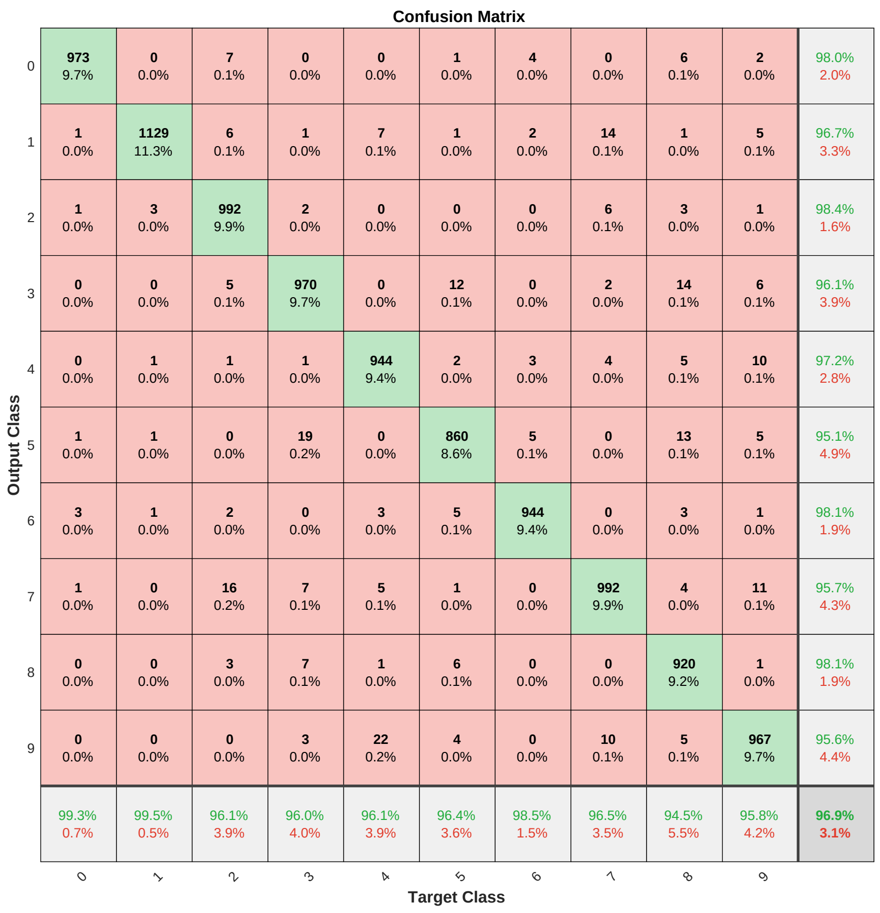
A more advanced decision rule is the k-nearest neighbours (KNN). Here you find the k closest references, and choose the class based on a majority vote from the k references.

The NN (or KNN) classifier is very simple to implement, but can be very slow for large data sets, such as the MNIST database. A way to prevent this while maintaining the classifiers accuracy is to cluster the training data. This means dividing all the training data for a given class into M clusters, where each cluster represents the data it contains. This results in far fewer references to compare each test data to, reducing the computation time significantly.

4.2 NN-classifier

In this task we implemented the nearest neighbour classifier as explained in the previous chapter. All the handwritten numbers were to be represented on a 28x28 matrix form (or a 1x784 vector). The numbers had already been preprocessed, centred and scaled, and were thus ready to be classified. Running the delivered matlab file "read09.m" gave us all the training and test data on a 1x784 vector form, as well as lists with labels for the data.

From fig. 12 we see that our classifier performs very well, with an error rate of only 3.1 %. However, it is very slow as it has to compare every test pictures to all 60 000 training pictures, pixel by pixel.



Confusion Matrix

Output Class (rows) / Target Class (columns)

	0	1	2	3	4	5	6	7	8	9	
0	**973** 9.7%	**0** 0.0%	**7** 0.1%	**0** 0.0%	**0** 0.0%	**1** 0.0%	**4** 0.0%	**0** 0.0%	**6** 0.1%	**2** 0.0%	98.0% 2.0%
1	**1** 0.0%	**1129** 11.3%	**6** 0.1%	**1** 0.0%	**7** 0.1%	**1** 0.0%	**2** 0.0%	**14** 0.1%	**1** 0.0%	**5** 0.1%	96.7% 3.3%
2	**1** 0.0%	**3** 0.0%	**992** 9.9%	**2** 0.0%	**0** 0.0%	**0** 0.0%	**0** 0.0%	**6** 0.1%	**3** 0.0%	**1** 0.0%	98.4% 1.6%
3	**0** 0.0%	**0** 0.0%	**5** 0.1%	**970** 9.7%	**0** 0.0%	**12** 0.1%	**0** 0.0%	**2** 0.0%	**14** 0.1%	**6** 0.1%	96.1% 3.9%
4	**0** 0.0%	**1** 0.0%	**1** 0.0%	**1** 0.0%	**944** 9.4%	**2** 0.0%	**3** 0.0%	**4** 0.0%	**5** 0.1%	**10** 0.1%	97.2% 2.8%
5	**1** 0.0%	**1** 0.0%	**0** 0.0%	**19** 0.2%	**0** 0.0%	**860** 8.6%	**5** 0.1%	**0** 0.0%	**13** 0.1%	**5** 0.1%	95.1% 4.9%
6	**3** 0.0%	**1** 0.0%	**2** 0.0%	**0** 0.0%	**3** 0.0%	**5** 0.1%	**944** 9.4%	**0** 0.0%	**3** 0.0%	**1** 0.0%	98.1% 1.9%
7	**1** 0.0%	**0** 0.0%	**16** 0.2%	**7** 0.1%	**5** 0.1%	**1** 0.0%	**0** 0.0%	**992** 9.9%	**4** 0.0%	**11** 0.1%	95.7% 4.3%
8	**0** 0.0%	**0** 0.0%	**3** 0.0%	**7** 0.1%	**1** 0.0%	**6** 0.1%	**0** 0.0%	**0** 0.0%	**920** 9.2%	**1** 0.0%	98.1% 1.9%
9	**0** 0.0%	**0** 0.0%	**0** 0.0%	**3** 0.0%	**22** 0.2%	**4** 0.0%	**0** 0.0%	**10** 0.1%	**5** 0.1%	**967** 9.7%	95.6% 4.4%
	99.3% 0.7%	99.5% 0.5%	96.1% 3.9%	96.0% 4.0%	96.1% 3.9%	96.4% 3.6%	98.5% 1.5%	96.5% 3.5%	94.5% 5.5%	95.8% 4.2%	**96.9% 3.1%**

Figure 12: Confusion matrix for the NN classifier

Figure 13 shows three different pictures that were misclassified by the NN classifier. Both the picure on the left and the one on the right are quite hard to classify, and while we don't agree with our classifier, we don't agree with the actual number either. Especially the one one the left looks more like a 2 than a 4 to us.

For the picture in the middle, however, we don't agree with our classifier, as it pretty clearly is a 3. One reason for the misclassification could be that we are using an NN-classifier (instead of a KNN-classifier), which is more vulnerable to outliers, in this case e.g. a 5 in the training set that looks like the 3 we were to classify.

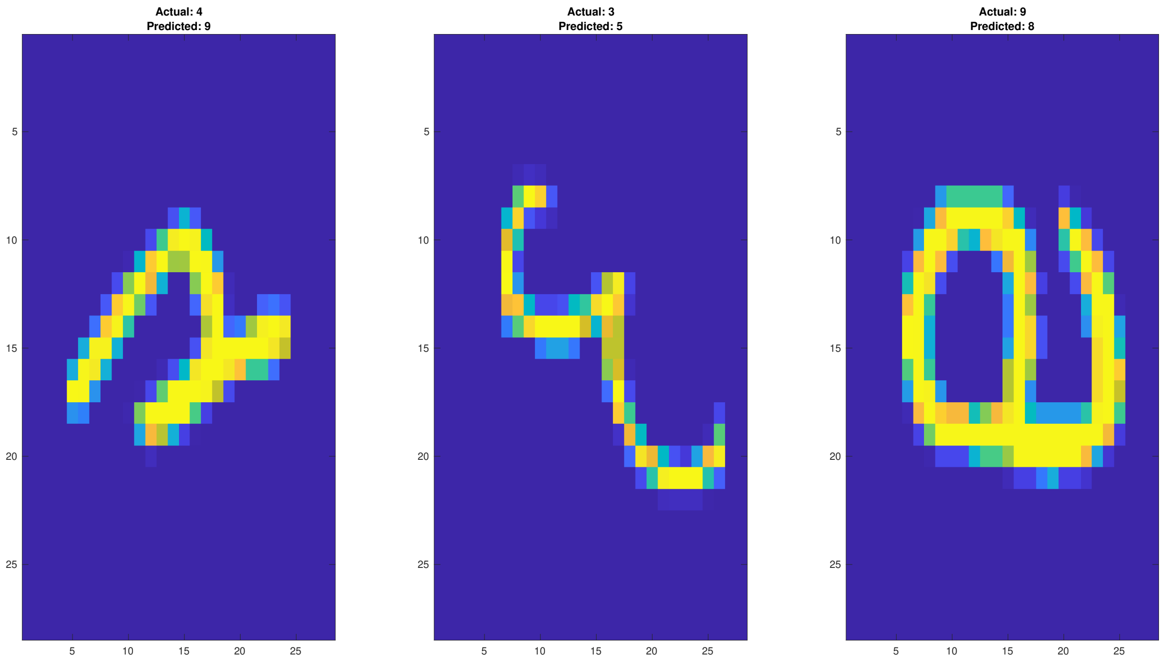


Figure 13: Three examples of misclassified numbers

Looking at fig. 14, we were quite amazed that the classifier managed to correctly classify these pictures. However, we don't necessarily agree with it for the two leftmost pictures. The 7 on the left doesn't really look like a number at all, and the 2 in the middle look more like a 4 or a 9 (unless you rotate it clockwise and then mirror it).

For the 1 to the right, though, we agree with the classifier.

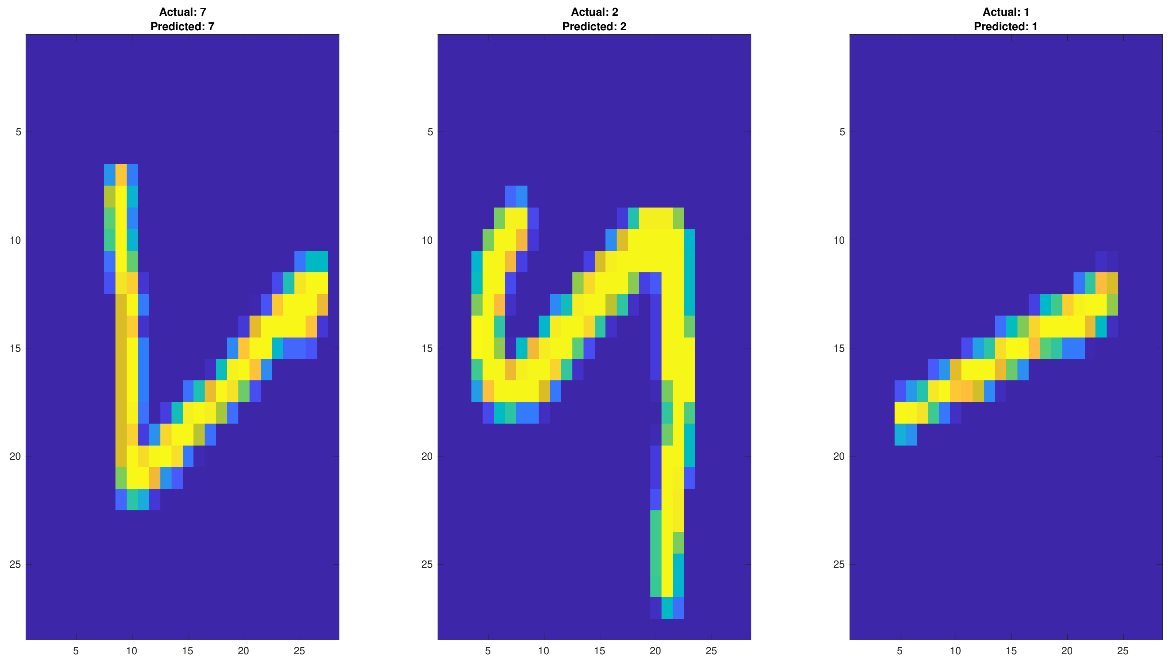
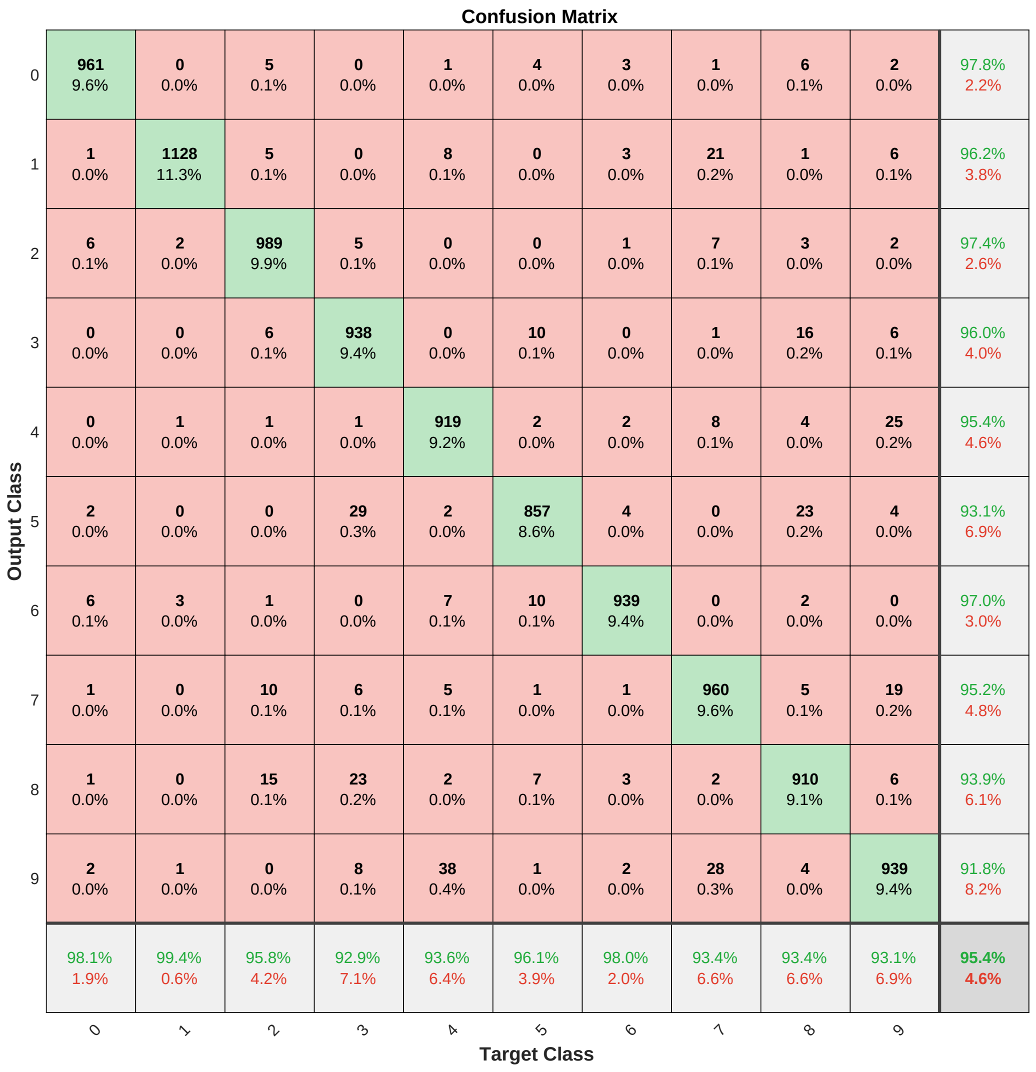


Figure 14: Three examples of correctly classified numbers

4.3 Clustering

After clustering each class and running the code with our new, clustered training data, we obtained the following confusion matrix:



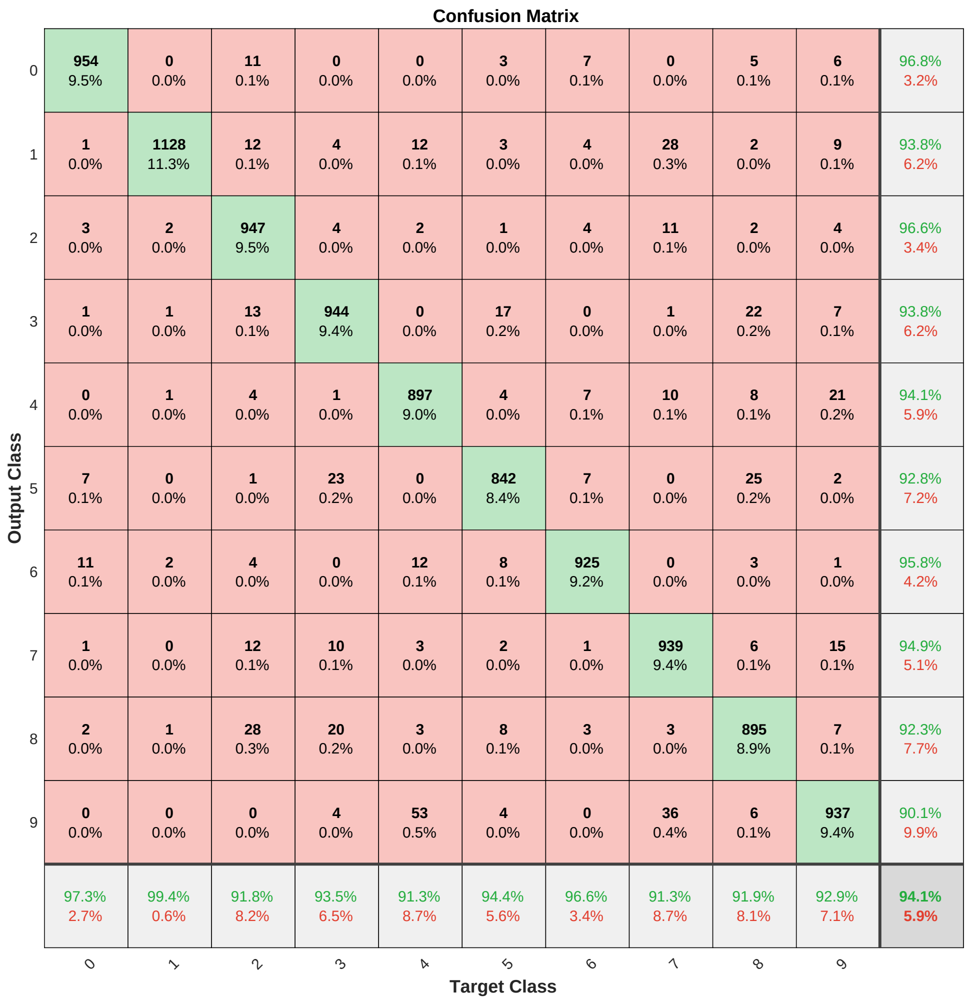
Confusion Matrix

Output Class \ Target Class	0	1	2	3	4	5	6	7	8	9	
0	961 9.6%	0 0.0%	5 0.1%	0 0.0%	1 0.0%	4 0.0%	3 0.0%	1 0.0%	6 0.1%	2 0.0%	97.8% 2.2%
1	1 0.0%	1128 11.3%	5 0.1%	0 0.0%	8 0.1%	0 0.0%	3 0.0%	21 0.2%	1 0.0%	6 0.1%	96.2% 3.8%
2	6 0.1%	2 0.0%	989 9.9%	5 0.1%	0 0.0%	0 0.0%	1 0.0%	7 0.1%	3 0.0%	2 0.0%	97.4% 2.6%
3	0 0.0%	0 0.0%	6 0.1%	938 9.4%	0 0.0%	10 0.1%	0 0.0%	1 0.0%	16 0.2%	6 0.1%	96.0% 4.0%
4	0 0.0%	1 0.0%	1 0.0%	1 0.0%	919 9.2%	2 0.0%	2 0.0%	8 0.1%	4 0.0%	25 0.2%	95.4% 4.6%
5	2 0.0%	0 0.0%	0 0.0%	29 0.3%	2 0.0%	857 8.6%	4 0.0%	0 0.0%	23 0.2%	4 0.0%	93.1% 6.9%
6	6 0.1%	3 0.0%	1 0.0%	0 0.0%	7 0.1%	10 0.1%	939 9.4%	0 0.0%	2 0.0%	0 0.0%	97.0% 3.0%
7	1 0.0%	0 0.0%	10 0.1%	6 0.1%	5 0.1%	1 0.0%	1 0.0%	960 9.6%	5 0.1%	19 0.2%	95.2% 4.8%
8	1 0.0%	0 0.0%	15 0.1%	23 0.2%	2 0.0%	7 0.1%	3 0.0%	2 0.0%	910 9.1%	6 0.1%	93.9% 6.1%
9	2 0.0%	1 0.0%	0 0.0%	8 0.1%	38 0.4%	1 0.0%	2 0.0%	28 0.3%	4 0.0%	939 9.4%	91.8% 8.2%
	98.1% 1.9%	99.4% 0.6%	95.8% 4.2%	92.9% 7.1%	93.6% 6.4%	96.1% 3.9%	98.0% 2.0%	93.4% 6.6%	93.4% 6.6%	93.1% 6.9%	**95.4% 4.6%**

Figure 15: Confusion matrix for the NN classifier, using clustered training data

We see that the accuracy of the classifier went down somehow, as the error rate increased from 3.1 % to 4.6 %. But when you take into consideration that the processing time was reduced to nearly 1/100 of the original time, we think that this is overall a better classifier than the one without clustering.

Next, we designed a KNN classifier as described in section 4.1, with K=7. The resulting confusion matrix is seen in fig. 16. The error rate is 5.9 %, meaning that the KNN classifier performs worse than the two previous ones. This might be because we have already clustered the training data, and when we look at 7 clusters we might begin to get clusters that aren't very close to the test data, and thus we can get unlucky and classify wrongly more often.



Confusion Matrix

Output Class \ Target Class	0	1	2	3	4	5	6	7	8	9	
0	**954** 9.5%	**0** 0.0%	**11** 0.1%	**0** 0.0%	**0** 0.0%	**3** 0.0%	**7** 0.1%	**0** 0.0%	**5** 0.1%	**6** 0.1%	96.8% 3.2%
1	**1** 0.0%	**1128** 11.3%	**12** 0.1%	**4** 0.0%	**12** 0.1%	**3** 0.0%	**4** 0.0%	**28** 0.3%	**2** 0.0%	**9** 0.1%	93.8% 6.2%
2	**3** 0.0%	**2** 0.0%	**947** 9.5%	**4** 0.0%	**2** 0.0%	**1** 0.0%	**4** 0.0%	**11** 0.1%	**2** 0.0%	**4** 0.0%	96.6% 3.4%
3	**1** 0.0%	**1** 0.0%	**13** 0.1%	**944** 9.4%	**0** 0.0%	**17** 0.2%	**0** 0.0%	**1** 0.0%	**22** 0.2%	**7** 0.1%	93.8% 6.2%
4	**0** 0.0%	**1** 0.0%	**4** 0.0%	**1** 0.0%	**897** 9.0%	**4** 0.0%	**7** 0.1%	**10** 0.1%	**8** 0.1%	**21** 0.2%	94.1% 5.9%
5	**7** 0.1%	**0** 0.0%	**1** 0.0%	**23** 0.2%	**0** 0.0%	**842** 8.4%	**7** 0.1%	**0** 0.0%	**25** 0.2%	**2** 0.0%	92.8% 7.2%
6	**11** 0.1%	**2** 0.0%	**4** 0.0%	**0** 0.0%	**12** 0.1%	**8** 0.1%	**925** 9.2%	**0** 0.0%	**3** 0.0%	**1** 0.0%	95.8% 4.2%
7	**1** 0.0%	**0** 0.0%	**12** 0.1%	**10** 0.1%	**3** 0.0%	**2** 0.0%	**1** 0.0%	**939** 9.4%	**6** 0.1%	**15** 0.1%	94.9% 5.1%
8	**2** 0.0%	**1** 0.0%	**28** 0.3%	**20** 0.2%	**3** 0.0%	**8** 0.1%	**3** 0.0%	**3** 0.0%	**895** 8.9%	**7** 0.1%	92.3% 7.7%
9	**0** 0.0%	**0** 0.0%	**0** 0.0%	**4** 0.0%	**53** 0.5%	**4** 0.0%	**0** 0.0%	**36** 0.4%	**6** 0.1%	**937** 9.4%	90.1% 9.9%
	97.3% 2.7%	99.4% 0.6%	91.8% 8.2%	93.5% 6.5%	91.3% 8.7%	94.4% 5.6%	96.6% 3.4%	91.3% 8.7%	91.9% 8.1%	92.9% 7.1%	**94.1%** **5.9%**

Figure 16: Confusion matrix for the KNN classifier, using clustered training data and K=7

4.4 Conclusion

In this task, the nearest neighbour classifier proved to be very accurate for classifying numbers, as well as being fairly simple to design and implement. However, due to the large size of our data sets, the classifier was very slow (took about 30 minutes to classify all 10 000 pictures), which showed the need for clustering. Clustering the training data gave us a classifier with "the best of both worlds" as it was now much faster, while still being very accurate (though a bit worse than before the clustering).

When looking at some of the misclassified pictures, we saw how some of the numbers were pretty horribly written. Thus, it is understandable that the classifier makes some mistakes as there will be several numbers that even humans won't be able to classify correctly. These numbers are rather a problem of terrible handwriting than poor classifying.