Department of Electronic systems

TTT4275 - Estimation, detection and classification

Classification Project

Classifying the Iris flower and handwritten digits

Authors:
Aria Alinejad
Thea Charlotte Andersen Brusevold

April, 2022

NTNU
Norwegian University of
Science and Technology

1 Summary

This report is made as a part of the course Estimation, detection and classification taught at NTNU. The classification part of the course will be inspected where both a linear classifier and some template based classifiers will be used. Both classifiers are made using supervised learning. Subsequently,they are tested and evaluated.

The linear discriminant classifier will classify three different variants of the Iris flower. The classification is based on four features coming from measurements of the leaves. It achieved an error rates between 0.00% and 2.22% when all the features were used. The run time of the code was ca. 12 seconds. When using less features the error rates increased, but the run time remained approximately the same. It is therefore preferred to use all the features.

Nearest neighborhood (NN) with and without clustering, as well as k-nearest neighborhood (KNN) with clustering are used to classify the handwritten digits using gray scale pixel values of images. NN without clustering gave an error rate of 3.09%. Using clustering, the error rates increased to 4.63% and 5.67% for NN and KNN respectively. However, clustering reduced the run time significantly, meaning that NN with clustering in most cases are to be preferred.

Table of Contents

List of Figures

List of Tables

2 Introduction

Classification has become an integral part of the development of new technology over the resent years. E.g. it is used in medicine, navigation, monitoring of control systems. This report will deal with two classification problems and give insight into how these can be solved. The report will firstly present relevant theory. Then the implementation and the results for the two problems will be shown and discussed. The report ends with a conclusion that sums up the most relevant findings.

3 Theory

This section introduces and elaborates the central concepts and theory needed for understanding the implementation. Classification is a method for discriminating between data based on features of the data, so that it later can be labeled. Supervised learning is when the classification model is trained using data that is already correctly classified. This is what we will do in this project, and is also very similar to how humans learn to classify objects based on earlier experiences.

3.1 Linear Classifiers

If a set of data is linearly separable we can use a linear classifier to correctly classify the data, see figure 1. One such linear classifier is the linear discriminant classifier (LDC). The LDC uses weighting nodes w to emphasize the importance of each features for each class. If we were to design a classifier to classify men and women based on hair length and height, the waiting functions for women would have high values for hair length and low values for height. Figure 2 shows the structure of an LDC.

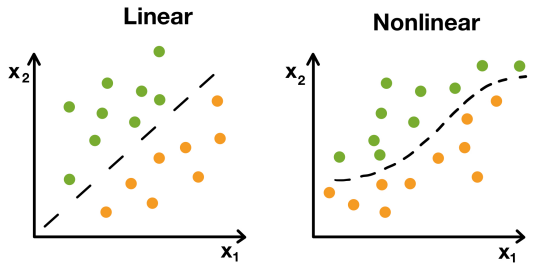


Figure 1: Linearly separable vs nonlinear separable.

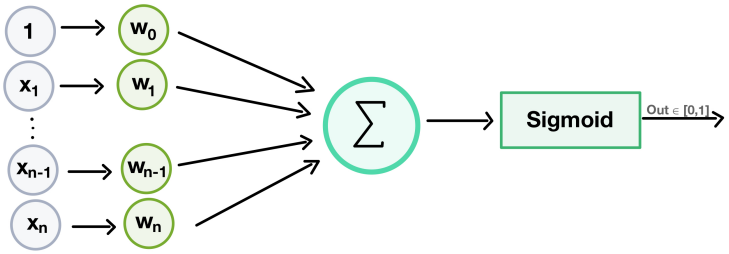


Figure 2: Structure of a LDC.

The LDC finds the right values for each w through an iterative process where the weights are given a random value to start with. For each iteration m a cost function quantifies how well the weights work, the weights are then changed to reduce the cost function for the next m. This is the main idea behind the LDC.

3.1.1 LDC

The theory for constructing the LDC is taken from [1]. The linear combination of the features x and the weighting functions w give a discriminant function $z_i(x)$ for each class i.

$$z_i(x) = \mathbf{w}_i^T \mathbf{x} + w_{io} \quad i = 1, \ldots, C \tag{1}$$

Where C is the number of classes, $\mathbf{x}$ is the vector containing the feature values, $\mathbf{w}_i$ is the vector containing the weighting functions for class i, and w_{io} is the offset of $\mathbf{w}_i$. The offset effectively increases or decreases the sum of weights as a whole. If we have $C > 2$ we can define W as a matrix holding all the weighting functions w_i. It will have the dimension $C \times D$, where D is the number of features. We can then give the weighting function as

$$\mathbf{z}(x) = \mathbf{W}\mathbf{x} + w_o. \tag{2}$$

$\mathbf{z}(x)$ is now a vector containing the discriminate function for each class. By defining $\mathbf{z}(x) = [\mathbf{W}w_0][\mathbf{x}^T 1]^T$ and using that $\mathbf{W} = [\mathbf{W}w_0]$ and $\mathbf{x} = [\mathbf{x}^T 1]^T$, equation (2) is simplified to

$$\mathbf{z}(x) = \mathbf{W}\mathbf{x}. \tag{3}$$

In $\mathbf{z}(x)$, each $z_i(x)$ scales with how well the classifier thinks the values of the features match with the class. The decision rule is then

$$z_k(x) = \max_i z_i(x), \tag{4}$$

where $z_k(x)$ is the discriminant function we choose for a single sample k. To be able to evaluate the effectiveness of the classifier we need a cost function. One such cost function is the Minimum Square Error (MSE). This is defined as

$$MSE = \frac{1}{2} \sum_{k=1}^{N} (\mathbf{g}_k - \mathbf{t}_k)^T (\mathbf{g}_k - \mathbf{t}_k), \tag{5}$$

where N is the number of samples. $\mathbf{t}_k$ is a vector that describes the correct class for each sample k, giving 1 if it is truly that class and 0 if it is not. For example $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$ if the sample is truly the second class. $\mathbf{g}_k$ is defined as

$$g_{ik} = sigmoid(x_{ik}) = \frac{1}{1 + e^{-z_{ik}}}, \quad i = 1, ..., C \tag{6}$$

so that the weighting function obtains a value between 0 and 1. This is needed to compare it with $\mathbf{t}_k$. Using the sigmoid function is suitable because it has a derivative.

MSE is used to train the classifier. By using the derivative of the MSE, we can evaluate in what direction we need to change W to minimize the MSE. This method is called gradient descent (GD) and is used to find a local minimum of the MSE. This is given by the following equation

$$\nabla_W MSE = \sum_{k=1}^{N} \left[(\mathbf{g}_k - \mathbf{t}_k) \circ \mathbf{g}_k \circ (1 - \mathbf{g}_k) \right] \mathbf{x}_k^T, \tag{7}$$

where $\nabla_W MSE$ is the gradient of the MSE function. Using

$$\mathbf{W}(m) = \mathbf{W}(m-1) - \alpha \nabla_W MSE, \tag{8}$$

we can change $\mathbf{W}$ in a favorable direction based on $\nabla_W MSE$. α is the step factor and decides by what scale $\mathbf{W}$ is changed for each iteration. A too large α will result in an inaccurate minima found by GD and a too small α will need a large number of iterations to find the minima.

3.2 Overfitting

Overfitting happens when a classifier gets too specialized on the training data instead of becoming a more general model. For example using too many iterations when training a LDC can result in a classifier that has the perfect weighting functions for that exact data, but can not handle other data. Large deviations between results for testing with training data and testing with testing data is an indication of overfitting.

3.3 Template Based Classifiers

The templates of a class is a set of values that is used to represent the class when classifying. The templates can be derived form a set of training samples or from using the samples directly. The templates need to have the same format as the test samples so that they can be compared and classified. The decision rule for template based classifiers is nearest-neighborhood (NN). The test samples are classified as the class of the template closest to it. A more advanced decision rule is called k-nearest-neighborhood (KNN). Instead of classifying based on the closest template, KNN decides based on the majority of the $K > 1$ nearest templates. If there are several modes, the nearest neighbor of the modes is chosen. In cases where there are an equal amount of closest templates between classes, the class with the closest sample is chosen. KNN may achieve better performance than NN, differing on the choice of K. The classification approaches are illustrated in figure 3.

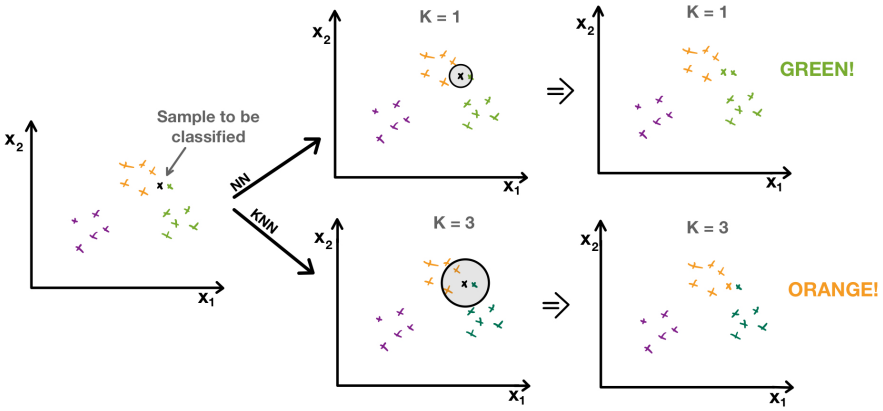


Figure 3: Classifying based on features x_1 and x_2 with NN and KNN.

Euclidean distance gives the shortest path between two points, and can therefore be used to determine the nearest neighbor. The Euclidean distance between $\mathbf{p}$ and $\mathbf{q}$ in an n-dimensional space is found by

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}, \tag{9}$$

where $\mathbf{p} = (p_1, p_2, ..., p_n)$ and $\mathbf{q} = (q_1, q_2, ..., q_n)$ [3].

By using each sample in the training data as templates we use the most information. However, having many templates results in a long running time. The running time can be reduced by separating the samples in each class into M clusters. This gives significantly fewer templates to compare the test values to, and thus less running time.

K-means clustering is a method that divides the training data into M clusters and finds the mean value of each cluster. The means are used as the templates. This is illustrated in figure 4.

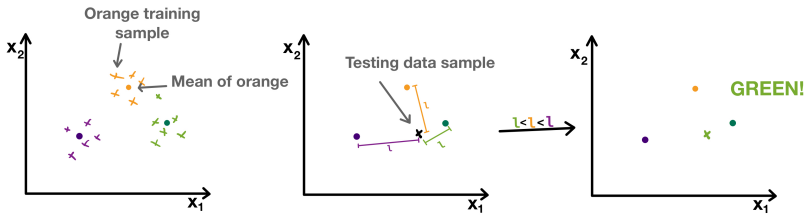


Figure 4: Clustering based on two features x_1 and x_2.

4 The Task

The section describes the two tasks as well as giving insight about the data used to classify the variants of the Iris flower and the handwritten digits. In addition it presents how the results will be displayed.

4.1 Iris

Iris

The Iris flower has several variants, three of these being Septosa, Versicolor and Virginica which we will reefer to as the three classes. All of these variants have two types of leaves called Sepal and Petal, see figure 5. By measuring the width and length of each of these leaves one has good a foundation for classifying what variant of the flower one has. Using the four features the data is close to linearly separable. It is therefore suitable to use a linear classifier.

We will make use of a database consisting of 50 examples of each class [2]. We will reefer to this database as *Iris Data* in this rapport. *Iris Data* will be used to train and test a linear classifier. The classifier will be evaluated using different separations of the *Iris Data* and display the results using confusion matrices and error rates. We will use histograms to evaluate the importance of each feature and evaluate the classifier with some features removed.

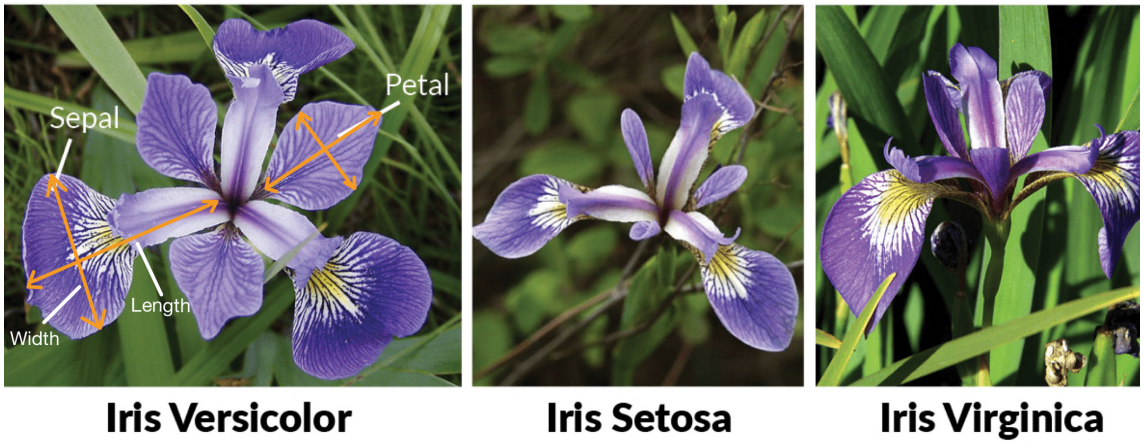


Figure 5: The three variants of Iris to be classified.

4.2 Classification of handwritten numbers

Handwritten digits 0-9 are to be classified using different variants of KNN. The classification will be done using the $MNIST$ database, consisting of 70000 handwritten numbers. Of these, 60000 are training digits written by 250 people, and 10000 are test digits written by another group of 250 people. All the digits are centered and scaled in the dimension 28x28 pixels, where each pixel is grayscaled with values between 0 to 255.

The task is separated in two. Firstly, the digits will be classified using NN with all the training samples as templates. Euclidean distance is to be used. Secondly, clustering will be used to produce fewer templates for each class. These templates are then used to classified with NN and KNN with $K = 7$. In all the tasks the confusion matrix and the error rate will be found.

5 Implementation

This section describes the approach for implementing the classifiers using flow charts. Python was used for the Iris task, while Matlab was used to classify the handwritten digits.

5.1 Iris

5.1.1 Python implementation

The code is implemented in python using VSCode as the text editor, except for the confirmation matrix that was implemented in Matlab. The choice of language and text editor made it easy to collaborate on the code using the VSCode extension Live share and GitHub. Python has several libraries for handling data and for doing matrix operations. We used the following libraries:

- pandas: used for splitting the data.
- numpy: used for matrix operations.
- matplotlib: used for plotting.
- scipy: used for saving the data in .mat format.
- timeit: used to time the runing time of the code.

The data was split into testing and training, we then used the LDC method to train the classifier. After training, the classifier is tested using the test data. The result is displayed with confusion matrices and error rates. The code was implemented as shown in figure 6.

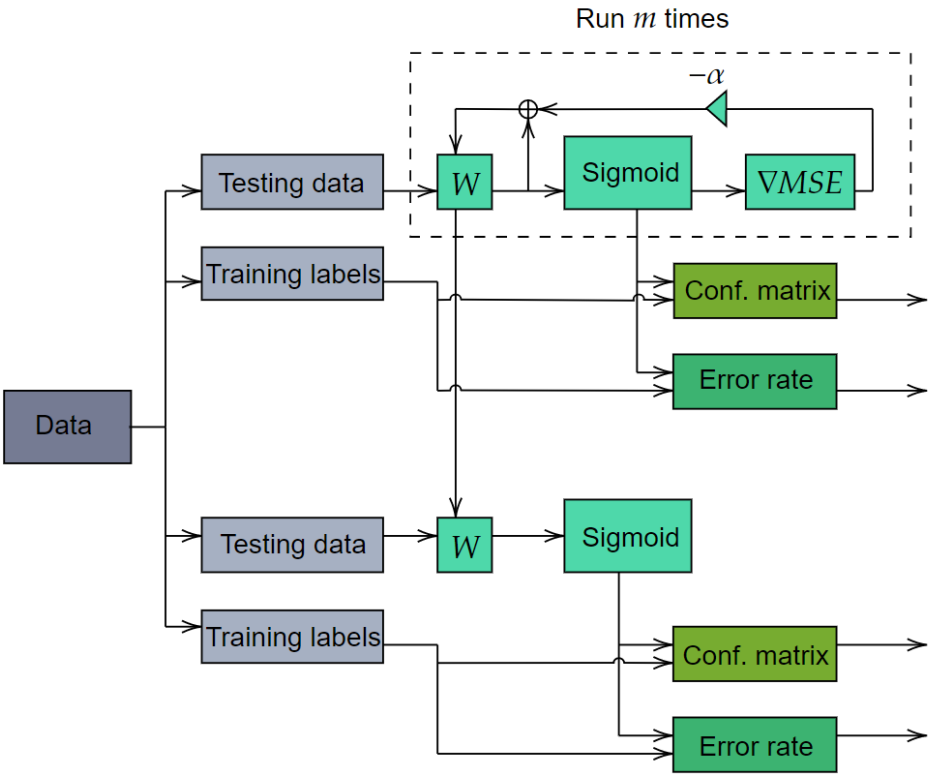


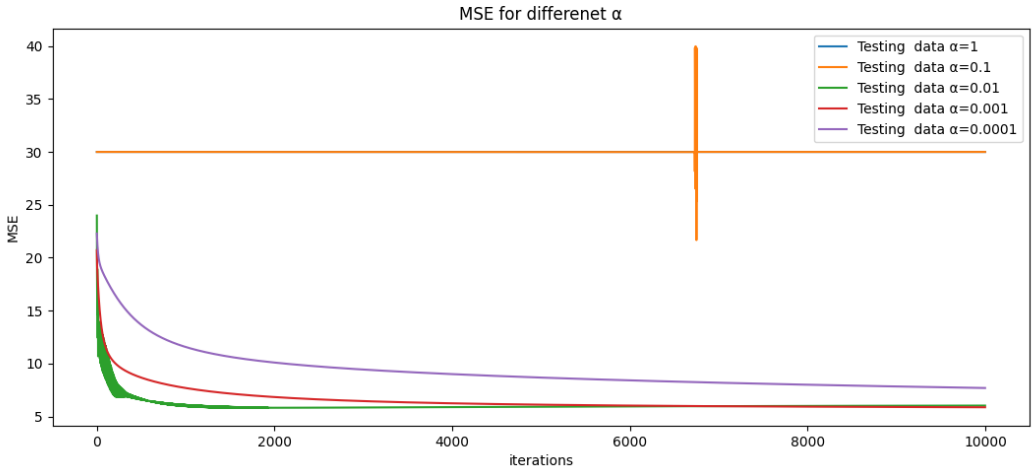
Figure 6: Flow chart of Iris classification.

5.1.2 Splitting of the data

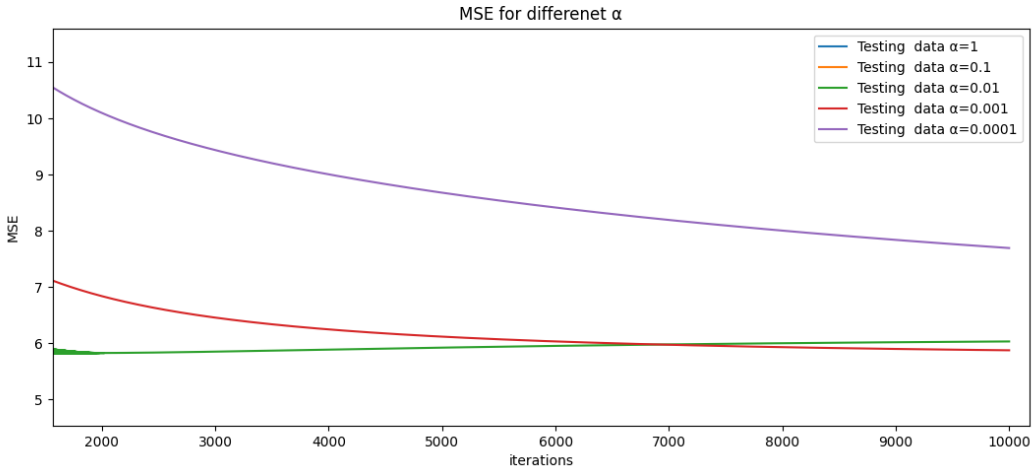
The data is split into training data and testing data. We split the data using 30 training samples and 20 testing samples. The system was tested for both using the first 30 for training and the last 20 for testing, and for using the last 30 for training and the first 20 for testing. We used the *pandas* python library to read from the data files and split the data.

5.1.3 Choosing the right α and number of iterations

To choose the right α we tested for different α to see what value gave the lowest MSE within a reasonable number of iterations. Figure 7 shows the MSE for the test data using 10000 iterations with five different values of α. One can observe that $\alpha = 0.001$ reaches the lowest MSE at the expense of using more than 7000 iterations. From figure 7a and 7b one can observe that choosing $\alpha = 0.01$ with 2000 iterations gives a good middle-ground between processing time/power and precision.



(a) MSE for testing data.



(b) MSE for testing data zoomed inn.

Figure 7: MSE for different values of α using 10000 iterations.

5.2 Digits

5.2.1 Matlab implementation

The digit task is implemented in Matlab. This is because it handles matrices and data processing well without having to include extra packages. The Deep Learning Toolbox was needed for clustering. Also, the MNIST data had to be run in Matlab. Functions for computing NN and KNN was developed. A function for computing the error rate and making the confusion matrix was programmed as well.

5.2.2 The NN function

The NN function finds the euclidean distances between each train sample and test sample using the Matlab function $dist(trainSet, testSet^T)$. Which returns the distances as a matrix. To avoid a large distance matrix, the training samples were divided into 60 chunks of 1000 samples. Too many chunks would results in using excessive time. For each chuck the distances between the templates and the test samples are found. The flow chart 8a show the sequence of how the code is run. The built-in *dist()* function is used to calculate the Euclidean distance and the built-in *min()* function is then used to find the closest template to each test sample.

5.2.3 The KNN function

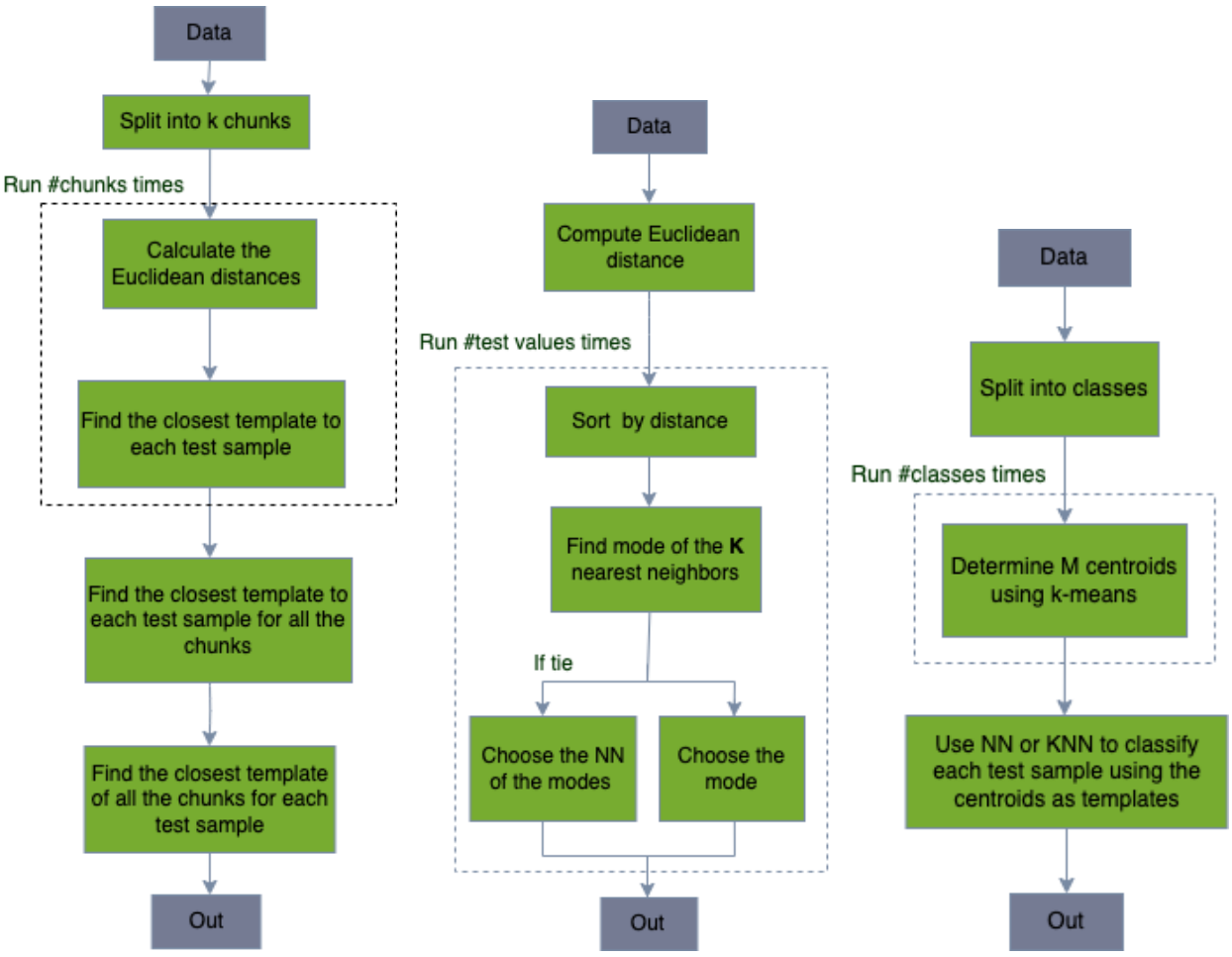
The KNN function is similar to the NN function, but instead of deciding on the single nearest neighbor the function decides the mode of the K nearest neighbors. If there are several modes, the NN of the modes is chosen. The built-in *sort()* function is used to sort the distances and the built-in *mode()* function is used to find the mode. The flow chart of the KNN function is shown in figure 8b.

5.2.4 Clustering

In the second part of the MNIST task, the training data was split into all the ten classes. Using the built-in function *kmeans()*, $M = 64$ centroids, i.e. templates, were found for each class. Using KNN or NN the test samples were then classified using the templates. This sequence can be seen in the flow chart in figure 8c.

5.2.5 The confusion matrix

The confusion matrix was found by using Matlab's *confusionchart()*-function.



(a) NN flow chart. (b) KNN flow chart. (c) Clustering flow chart.

Figure 8: NN, KNN and clustering flow charts.

6 Results

The section presents the results by showing confusion matrices, histograms, plots of digits and tables showing error rates and run times. The results will be discussed.

6.1 Iris

We use $\alpha = 0.01$ and 2000 iterations for all of the tests. After training the classifier we get the following weighting matrix:

$$W = \begin{array}{c} \begin{array}{ccccc} \textit{sepal length} & \textit{sepal width} & \textit{petal length} & \textit{petal width} & w_0 \end{array} \\ \begin{pmatrix} 0.4282 & 1.6800 & -2.5005 & -1.1575 & 0.3035 \\ 1.4720 & -2.9180 & -0.1945 & -1.1662 & 1.5932 \\ -2.9538 & -2.4927 & 4.3144 & 3.7621 & -1.8878 \end{pmatrix} \end{array} \begin{array}{c} \textit{Setosa} \\ \textit{Versicolour} \\ \textit{Virginica} \end{array} \tag{10}$$

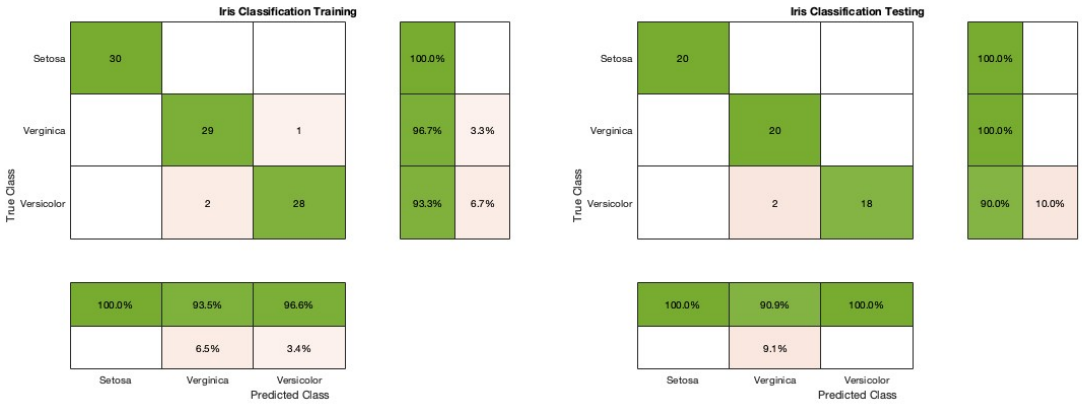
The weighting matrix shows importance of each feature for each class according to the trained classifier. E.g. it can be observed that *Virginica* has a high value for *petal length* in comparison to the other flowers. Thus, the classifier thinks that *Virginica* has much bigger petal length than the other flowers.

Figure 9a and figure 9b show the confusion matrices for the training data and the test data respectively. The first 30 samples was used for training and the last 20 for testing. We see that the classifier has most difficulty discriminating between $Virginica$ and $Versicolor$. The error rate for the training data is

$$error\ rate = 3.33\%, \tag{11}$$

and error rate for the test data is

$$error\ rate = 2.22\%. \tag{12}$$



(a) Confusion matrix for training data.

(b) Confusion matrix for testing data.

Figure 9: Confusion matrices for trained classifier using the first 30 samples for training and the last 20 for testing.

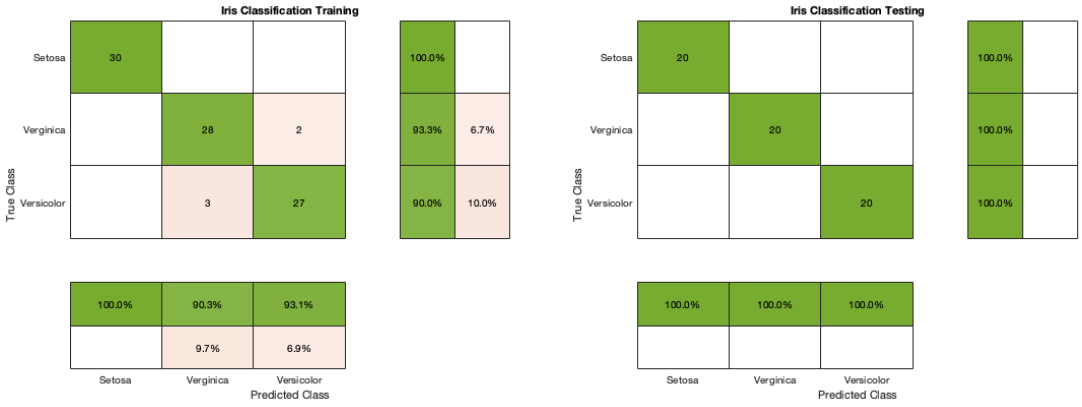
Figure 10a and figure 10b show the confusion matrices for the training data and the test data respectively. Now, the first 20 samples was used for testing and the last 30 for training. The error rate for the training data is then

$$error\ rate = 5.56\%, \tag{13}$$

and error rate for the test data is

$$error\ rate = 0.00\%. \tag{14}$$

We see that the *error rate* for the training data increases, but the *error rate* for the test data decreases. Looking at figure 7 one can observe that the MSE is at its lowest at ca. 2000 iterations, but converges to a higher value as the iterations increase. Thus the low error rate that is achieved with $\alpha = 0.01$ and 2000 iterations might be misleading. This underlines how much randomness is at play when dealing with such a small data set.



(a) Confusion matrix for training data.

(b) Confusion matrix for testing data.

Figure 10: Confusion matrices for trained classifier using the first 20 samples for testing and the last 30 for training.

6.1.1 The importance of each feature

From now on, the first 30 samples will be used for training and the last 20 for testing. When dividing the data into 16 bins we get the histograms that are shown in figure 11. Here we see the values of each feature for each class. We can see that *Setosa* generally has less overlap than the other classes, thus it is easier to classify. This was also shown in the confusion matrices in figure 9 and 10. Looking at *Petal length* of *Virginica* in figure 11c it has high values as predicted by (10). If we calculate the number of bins that overlap between features for the classes we get the results that are shown in table 1.

Table 1: Number of overlapping bins for the different features.

	#bins
Sepal Length	42
Sepal Width	78
Petal Length	8
Petal Width	6

We see that the *Sepal Width* has the most overlap, meaning that it gives the least useful information when trying to separate the three classes. Removing this feature we get the confusion matrices that is shown in figure 12a and figure 12b, for training data and test data respectively. We see that the confusion matrix is the same for training data, but that it misses with one extra for test data. This indicates the feature was of little significance.

When continuing further with removing the least useful features until there only is one feature left, we get the confusion matrices that are shown in figure 12. Furthermore, we get the error rates that are shown in table 2, the run times are shown in table 3. We see that the only errors are between $Virginica$ and $Versicolor$, even when three features are removed. This is because *Setosa* is linearly separable from $Virginica$ and $Versicolor$ using only *Petal width*. This can be seen in figure 11d. In addition, the error added when removing each feature is small. This is because both *Petal length* and *Petal Width* have little overlap between $Virginica$ and $Versicolor$ while *Sepal length* and *Sepal width* overlaps a lot, as shown in figure 11. Thus, removing the two with

the most overlap has little significance. The same yields for removing the third since the remaining feature has little overlap.

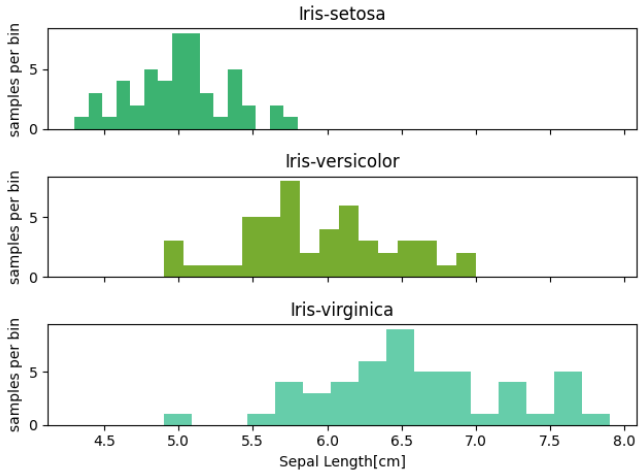
Generally removing features increases the error rate of the classifier. However the positive side of removing features is that the classifier takes less time and power to run. In addition, using less features requires less data, which in turn often makes it less costly. The time difference is not that noticeable when dealing with relatively small data sets as this one. It is therefore preferred to use all the features in this case.

Table 2: Error rates using 3, 2 and 1 features for training data and test data.

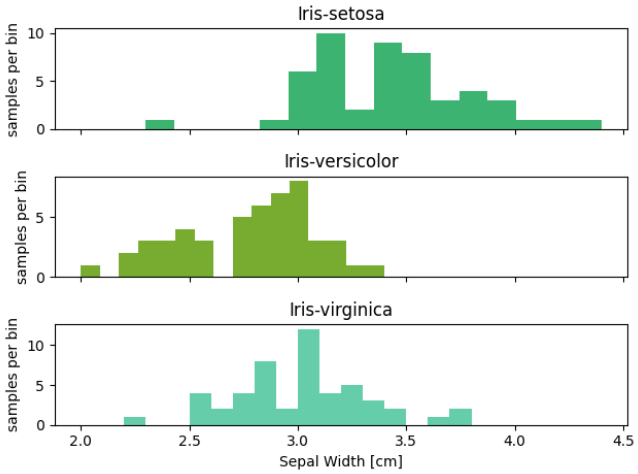
	D = 4	D = 3	D = 2	D = 1
Training data error rate	3.33%	3.33%	6.67%	4.44%
Test data error rate	2.22%	3.33%	3.33%	4.44%

Table 3: Run times using 3, 2 and 1 features for training data and test data.

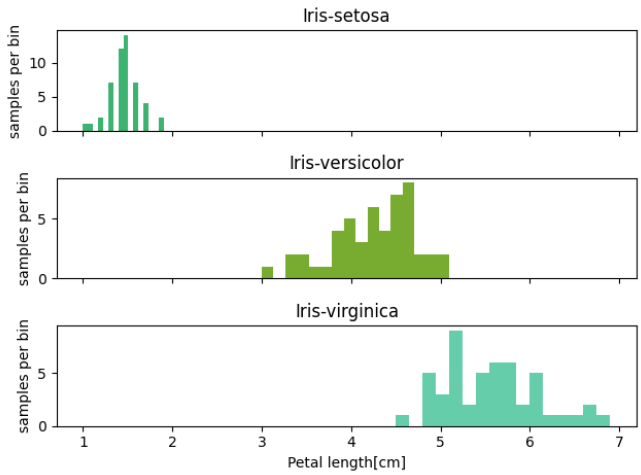
	D = 4	D = 3	D = 2	D = 1
Training data run time [s]	11.3	12.5	11.8	11.4
Test data run time [s]	13.3	11.8	11.8	11.7



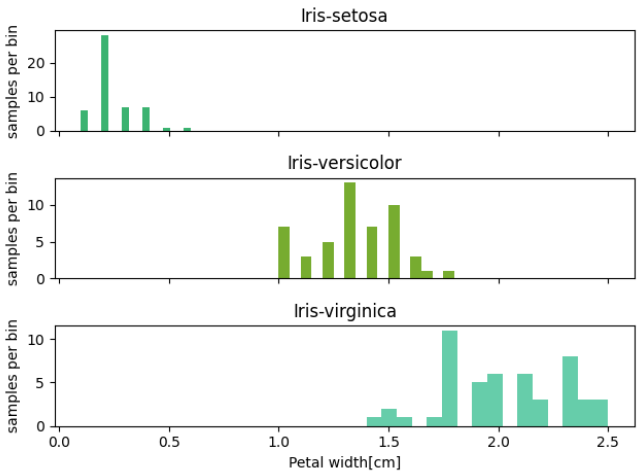
(a) Length of the sepal leave for the different flower types.



(b) Width of the sepal leave for the different flower types.

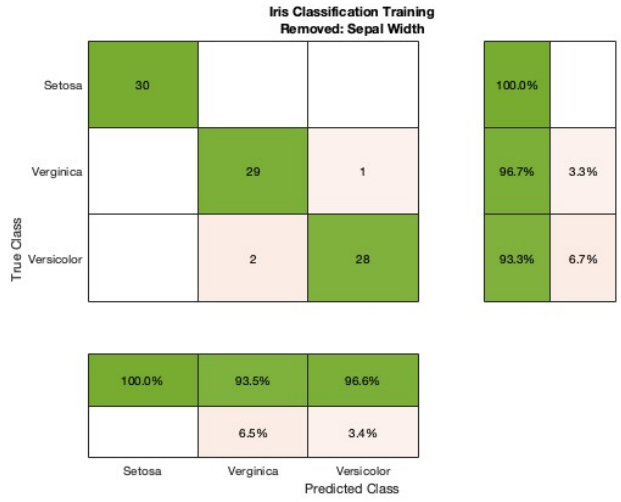


(c) Length of the petal leave for the different flower types.

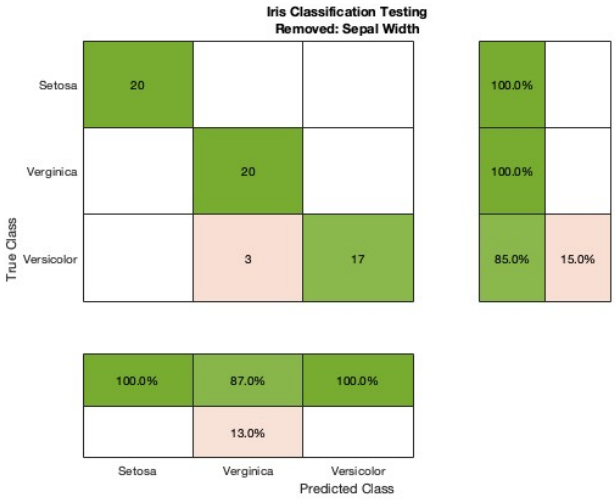


(d) Width of the petal leave for the different flower types.

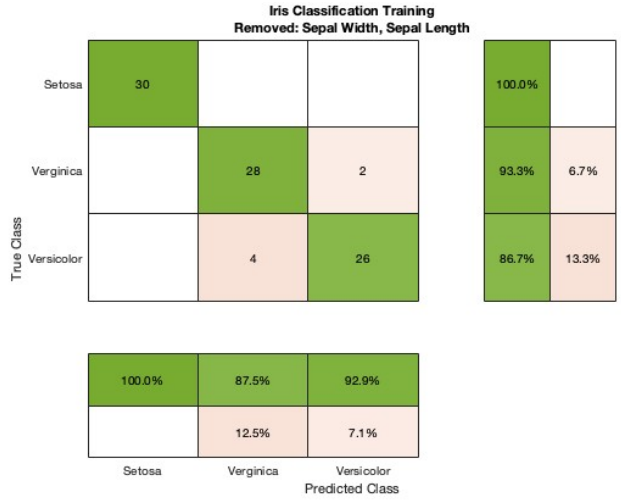
Figure 11: The four feature values of the the three classes sorted by value.



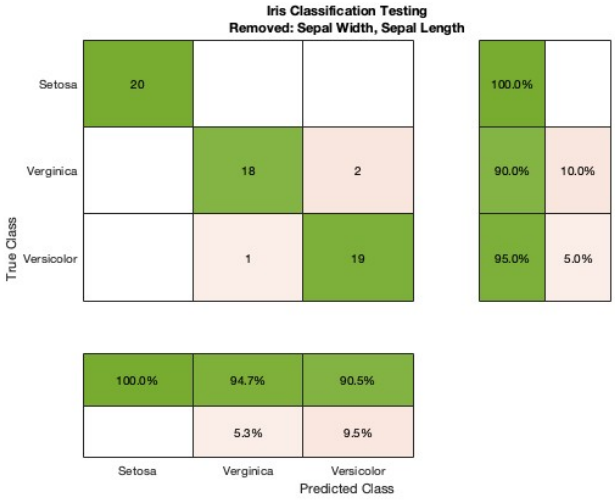
(a) Confusion matrix for training data using *Sepal Length*, *Petal Length* and *Petal Width* as features.



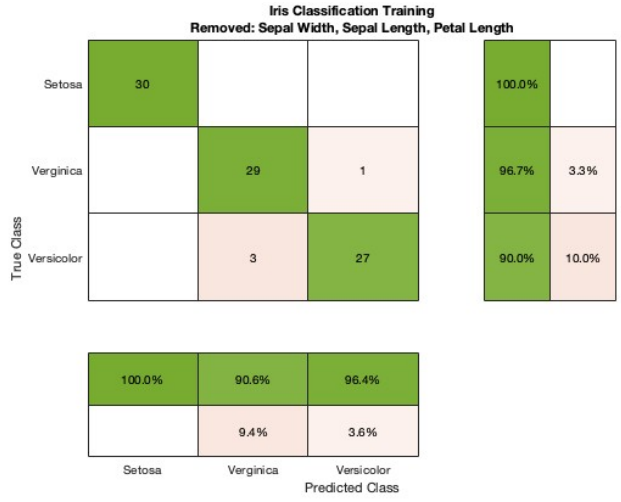
(b) Confusion matrix for test data using *Sepal Length*, *Petal Length* and *Petal Width* as features.



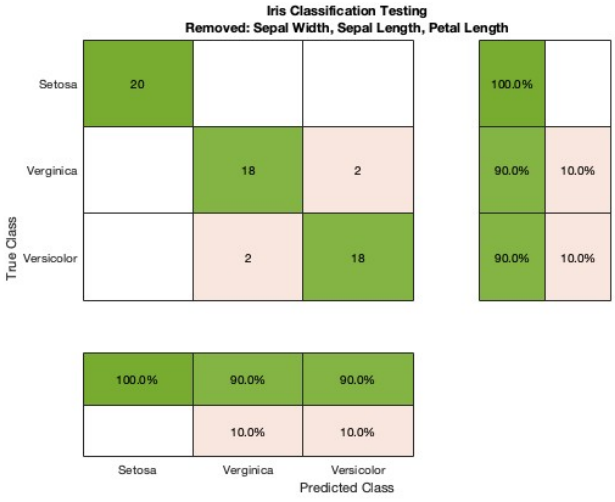
(c) Confusion matrix for training data using *Petal Length* and *Petal Width* as features.



(d) Confusion matrix for training data using *Petal Length* and *Petal Width* as features.



(e) Confusion matrix for training data using only *Petal Width* as a feature.



(f) Confusion matrix for test data using only *Petal Width* as a feature.

Figure 12: Confusion matrix using 3, 2 and 1 features for training data and test data.

6.2 Digits

The confusion matrix for NN without clustering is shown in figure 13. The overall error rate is found to be

$$error\ rate = 3.09\%. \tag{15}$$

One can observe that the majority of the digits were classified correctly. It seems that the zeros were the easiest to classify since it predicted correct 99.3% when the true value was *zero*. Catching that the digit is *eight* seems to be the hardest with an error rate of 5.5%. This is not surprising since *eight* has much overlap with other digits. In the confusion matrix 13 one can observe that predicting *eight* to be *three* and *five* occurs relatively many times. The most common mismatch is predicting a *nine* when *four* is the true digit. Again not surprising since the digits look very much the same.

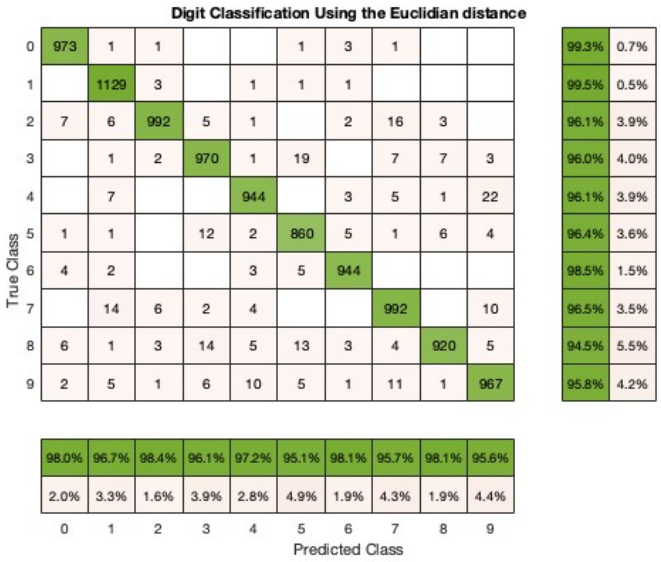


Figure 13: Confusion matrix for NN without clustering.

Figure 14 shows four digits that was misclassified and their nearest neighbor. The digit *four* on the top was classified as a *nine*. This represents a common humanly mistake, thinking a *nine* is a *four* when *nine* is written sloppy. Not surprising that they were misclassified since they have much overlap, the same goes for the rest of the pictures. Some humans would probably classify the first, third and fourth test digit wrongly. The second, however, is clearly a nine.

Figure 15 shows an example of each digit that was classified correctly. Seeing that some are sqewed and sloppy, e.g. the digits five and three, as well as the classification success being 96.91%, one can conclude that the classification gave a satisfactory response. However, in the paper "Efficient Pattern Recognition Using a New Transformation Distance" a human error rate of 2.5% were found [4]. Which shows that the classifier gives a higher error rate then if a human were to classify the digits. The elapsed time of the code is 1880 seconds, i.e. ca 30 minutes. Clustering is a method for decreasing this time.

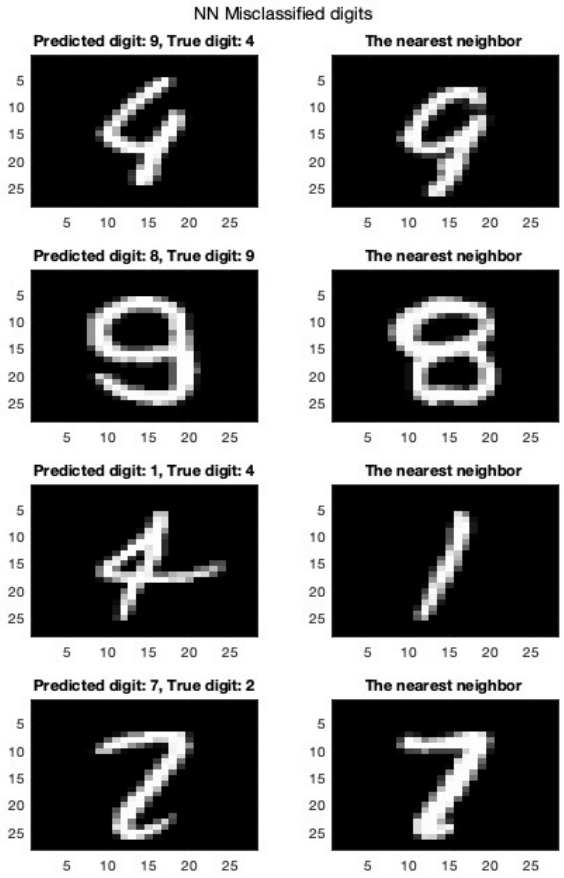


Figure 14: Misclassified digits using NN without clustering.

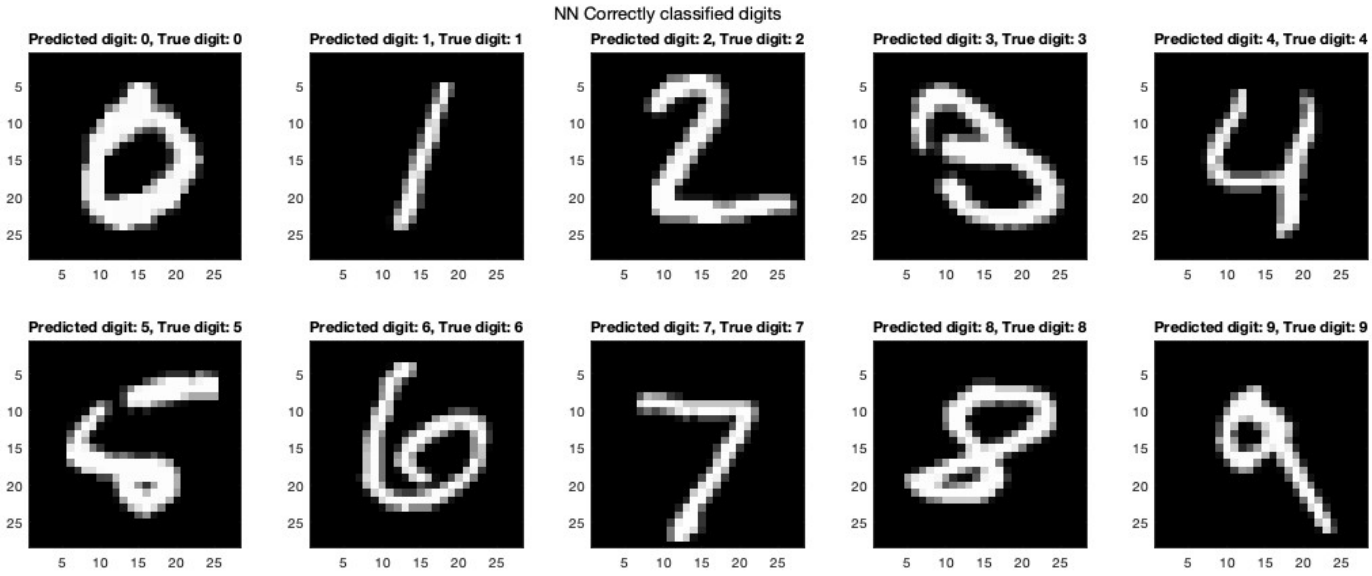


Figure 15: Correctly classified digits using nearest neighbor.

The confusion matrices for NN and KNN using clustering is shown in figure 16 and 17 respectively. The error rates and running time for all classifications done are shown in table 4.[1] One can observe that NN classifying without clustering gave the lowest error rate. This is because the number of templates are reduced drastically with clustering, meaning loss of information. Also, many of the training samples are skewed and sloppy giving noisy centroids. Two of these centroids are shown in figure 18. From the confusion matrices in figure 16 and figure 17 one can see that the digit *two* was sometimes predicted as a *seven*. This can also be seen in figure 18 where the centroid of the digit *two* has overlapping pixels as an average *seven*. However, many of the centroids are not so noisy, e.g. the centroid *five* in figure 18, which underlines that the classifiers are correct for about 95%.

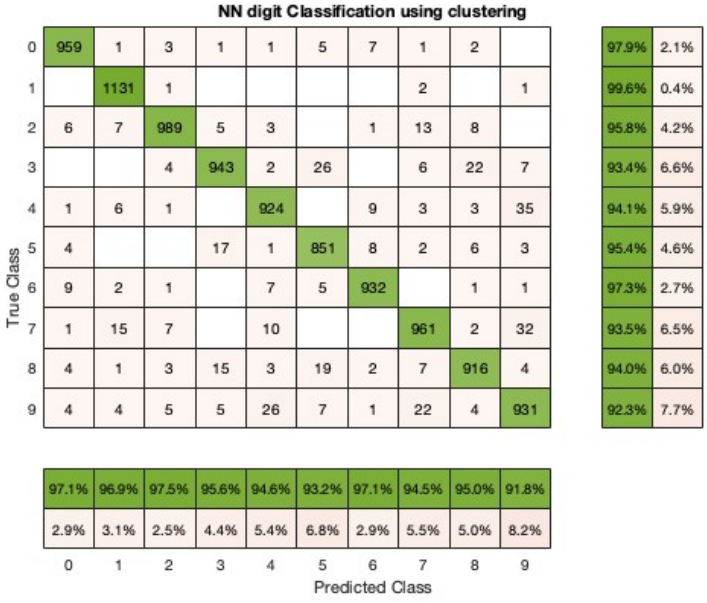


Figure 16: Confusion matrix for NN with clustering.

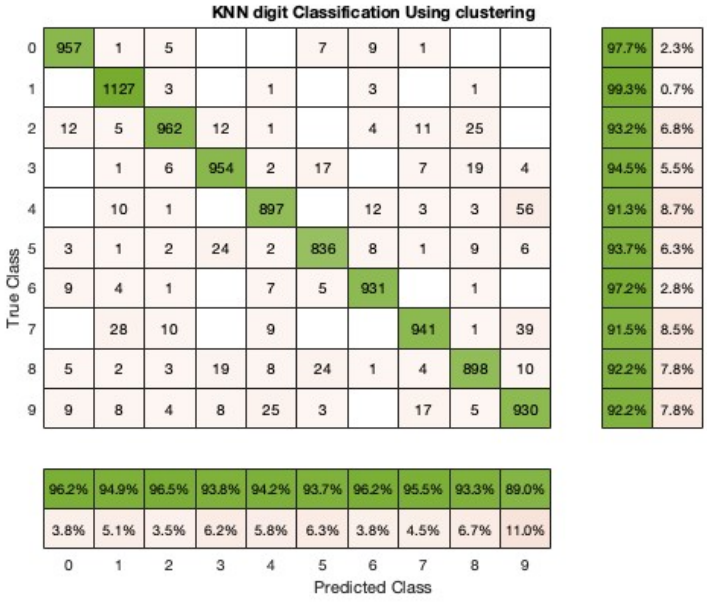


Figure 17: Confusion matrix for KNN with clustering.

[1]The running time is included clustering of the original training data.

Table 4: Error rates and running times for three different template based classifiers.

Classification type	**Time [s]**	**Error rate [%]**
NN	1880	3.09
NN with clustering	80	4.63
KNN with clustering	67	5.67

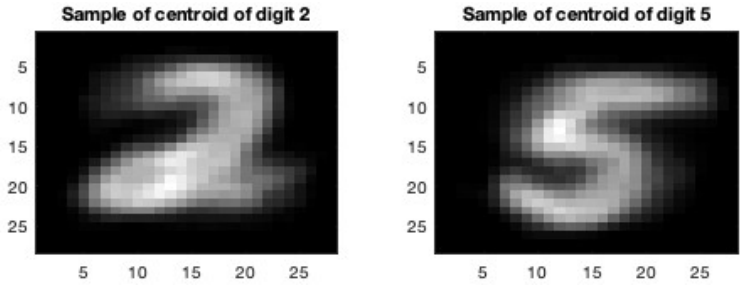


Figure 18: Example of centroid digits.

From table 4 one can see that KNN has a higher error rate then NN with clustering. From the available data, it is not clear why this is the case. The difference in run time is negligible.

Even though using NN without clustering classified most digits right, it used much more time then classifying with clustering. Therefore one must decide what is most important, minimal error rate or running time. If one shall grade tests, it might be more important to keep the error rate down then when digitalizing lecture notes. For most cases, the performance time outweighs the error rate meaning that the NN classifier with clustering is preferred.

7 Conclusion

In conclusion, we used a LDC and a template based classifier to classify flower types and digits respectively. The classifiers were implemented in Python and Matlab, and the data was taken from $IrisData$ and the $MNIST$ database.

The LDC used four features and 90 samples to train the classifier. It achieved an error rates between 0.00% and 2.22% for the 60 samples used for testing. The run time of the code was ca. 12 seconds. When using less features the error rates increased, but the run time remained approximately the same. It is therefore preferred to use all the features. The results are satisfactory taking into account the amount of data used.

To classify handwritten digits three different template based classifiers were made. The database consisted of 60 000 training samples and 10 000 samples for testing. NN without clustering resulted in an error rate of 3.09% and a run time of 1880 seconds. When clustering the data into 64 templates for each class and using NN as the decision rule, an error rate of 4.63% was achieved with a run time of 80 seconds. KNN with clustering achieved an error rate of 5.67% and a run time of 67 seconds. Taking into account the simplicity of the classifiers these are satisfactory results.

References

[1] Tor A. Myrvoll, Stefan Werner and Magne H. Johnsen, "ESTIMATION, DETECTION AND LASSIFICATION THEORY", Compendium TTT4275, NTNU.

[2] "Iris flower data set", Wikipedia, 2022, https://en.wikipedia.org/wiki/Iris_flower_data_set.

[3] "DISTANCES IN CLASSIFICATION", CAFÉ SCIENTIFIQUE, 2016.

[4] P. Simard, Y. Le Cun and J. Denker "Efficient Pattern Recognition Using a New Transformation Distance", 1993, https://papers.nips.cc/paper/1992/file/26408ffa703a72e8ac0117e74ad46f33-Paper.pdf