NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTT4275 - Estimation, Detection and Classification

Project: Digit Classification

Authors:
544979
529312

30th April 2023

Summary

This report is an mandatory and graded part of the course TTT4275 - Estimation, Detection and Classification at NTNU. The course is an introduction to the three fields, and covers basic topics such as MVU estimators, the Neyman-Pearson lemma and linear classifiers.
The report presents, and discusses the findings of two tasks. The first concerning an linear classifier for the Iris data-set, the second concerning a nearest neighbor classifier for the MNIST data-set. In both cases variations of the classifiers are developed and compared.
For the Iris task one classifier was trained using a softmax activation function together with a cross-entropy loss function. This classifier generally outperformed the alternative training scheme - using a sigmoid activation function and the L2 loss function. At best, the classifier achieved only two miss-classifications on the whole data-set. It is hypothesized that this is the best a linear classifier can achieve on this data set. The MNIST data set was classified using different nearest neighbour schemes - unclustered NN, clustered NN and KNN. At best the classification achieved an error rate of 4.6%. Somewhat surprisingly the experiments showed that the error rate tended to increase when increasing the K value for the KNN.
Overall this project demonstrates that relatively simple classification algorithms - preferred for their low complexity - can yield great results, despite their simplicity.

Table of Contents

List of Figures

List of Tables

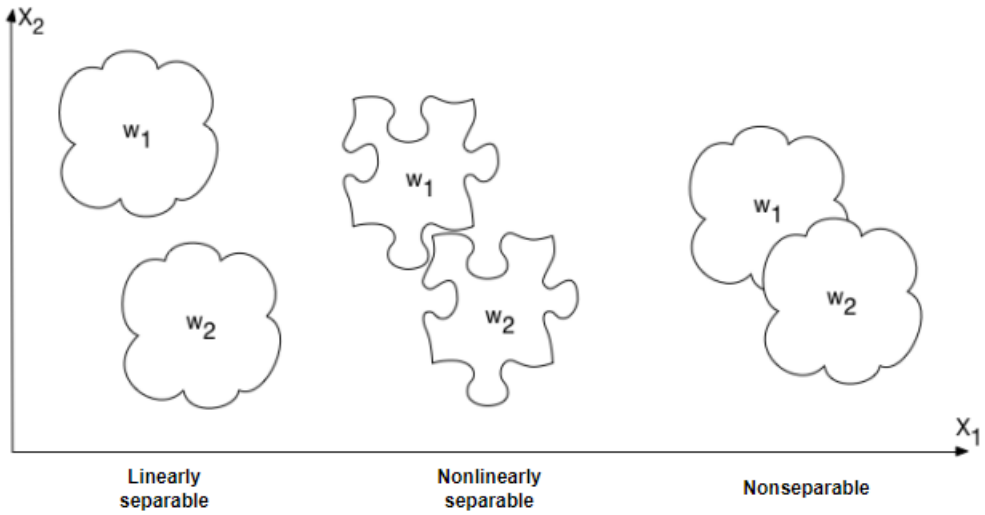


Figure 1: Types of classification problems, figure from Myrvoll et al. 2023

.

1 Introduction

The subject of classification is of great interest to modern society, with applications in technologies like autonomous vehicles, surveillance systems and voice assistants. The classic problem of handwriting recognition is an example of such a technology, with applications in features such as handwriting-to-text conversion.

This project covers the development of a Nearest Neighbor (NN) classifier for digit classification, as well as a linear classifier for Fisher's iris data-set.

In Section 2, the theory behind the linear and NN classifiers is described.
In Section 3, the problems of iris recognition and digit recognition are described. Sections 4 and 5 describes the implementation and analysis of the classifiers, with concluding remarks and further work in Section 6.

2 Theory

The following is a summary of the relevant theory for the classifiers applied in this project, namely Linear and Nearest Neighbor classifiers.

2.1 Classification

Classification can be described as assigning a class to a feature vector $\mathbf{x}$. One example is the determination of gender (class) based on height and body mass (features). The assignment of a class to an observation requires the application of a decision rule on the feature set, and the design of performant decision rules is the core problem of classification with main performance indicators being accuracy, computational cost and simplicity. A multitude of different classifiers exist, and the type of problem is an important factor in the design, with the main features being separability and linearity as seen in Figure 1.

A separable classification problem is characterized by a lack of overlap between classes in the feature space. For separable problems, there exists a set of decision borders that yields error-free classification. However, a lot of practical problems are non-separable, and there exists no error-free classifier. Myrvoll et al. 2023.

Important types of classifiers are the linear, nonlinear and template based classifiers. The main distinction between linear and nonlinear classifiers is the shape of the decision border, being a line, plane or hyperplane for the linear classifier and nonlinear for the nonlinear classifier. There is a trade-off between simplicity and accuracy, with the linear classifier being relatively simple at the cost of lower accuracy for nonlinear problems.

2.2 Linear Classifiers

A linear classifier is a type of classifier where a linear function of the feature vector $\mathbf{x}$ is used to decide between classes. I.e, in a two category case, suppose we have the two classes ω_1 and ω_2. For this case, a linear classifier is the linear function

$$g(\mathbf{x}) = \mathbf{a}^\top \mathbf{x} + a_0 \tag{1}$$

combined with the decision rule

$$\text{Choose } \omega_1 \text{ if } g(\mathbf{x}) > 0 \text{ and } \omega_2 \text{ if } g(\mathbf{x}) < 0.$$

This setup is fine, but it restricts us to the two category case, and the generalization to the multi category case is not obvious (Duda et al. 2000, Ch. 5)
We will follow in the footsteps of Duda et al. and briefly describe the linear machine. A linear machine is a type of linear classifier that can decide between multiple classes. It employs one linear function $g_i(\mathbf{x})$ for each class ω_i, and the decision rule is to choose the class with the highest value for its corresponding function. This formulation lends it self nicely for the formalism of linear algebra, especially so, if we employ

$$\tilde{\mathbf{x}} = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix} \tag{2}$$

$$\tilde{\mathbf{a}} = \begin{pmatrix} a_0 \\ \mathbf{a} \end{pmatrix} \tag{3}$$

With this Equation 1 just becomes $g(\mathbf{x}) = \tilde{\mathbf{a}}^\top \tilde{\mathbf{x}}$. Now a linear machine can be summed up as

$$\mathbf{x} \in \omega_i \Leftrightarrow \max \mathbf{g}(\tilde{\mathbf{x}}) = g_i(\tilde{\mathbf{x}}) \tag{4}$$

$$\text{where} \qquad \mathbf{g}(\mathbf{x}) = \begin{pmatrix} g_1(\mathbf{x}) \\ \vdots \\ g_N(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{a}}_1^\top \tilde{\mathbf{x}} \\ \vdots \\ \tilde{\mathbf{a}}_N^\top \tilde{\mathbf{x}} \end{pmatrix} = \tilde{\mathbf{W}} \tilde{\mathbf{x}} \tag{5}$$

(Duda et al. 2000, Ch. 5) So far we have seen how a linear classifier can be formulated, but still it has not been explained how to apply it, after all what should $\tilde{\mathbf{W}}$ be? Finding $\tilde{\mathbf{W}}$ is often called learning, though optimization is a more precise name, and for the optimization we will use gradient descent, described in the next subsection.

2.3 Gradient descent

The concept of gradient descent will now be explained and applied to the linear machine described above.
Gradient descent is a way of minimizing a function, typically referred to as a cost function $f(\mathbf{x})$. In general, optimization is concerned with minimizing the cost function, and so it is of interest to find a direction $\mathbf{p}$ such that the value of f decreases along $\mathbf{p}$. There are several ways of finding said direction and one of them is to use $\mathbf{p} = -\nabla f(\mathbf{x}_k)$, where $\mathbf{x}_k$ is the guess for the optimal argument in iteration k. The general procedure is therefore:

1. Start with an initial guess for $\mathbf{x}^*$ (optimal argument) $\mathbf{x}_0$

2. Set $k = 0$
3. While some optimality criteria is not fulfilled (or a max number of iteration are not done)
   (a) Calculate $\mathbf{p} = -\nabla f(\mathbf{x}_k)$
   (b) Choose $\alpha \in (0, 1]$
   (c) Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}$
   (d) Set $k = k + 1$

(Nocedal and Wright 2006, p.20-21)

$$S(x) = \frac{e^x}{e^x + 1} \tag{6}$$

$$S(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \tag{7}$$

For the case of the linear classifier this means that we will need to define a cost function. A popular choice is the L2 norm of the error. To formulate this function we need a set of labeled feature vectors. For each feature vector $\mathbf{x}_i$ the output of the linear classifier will be compared to the one-shot encoding of the correct label $\mathbf{y}_i$. Briefly explained, a one-shot encoding is a vector encoding of the class where class ω_i is encoded as $\mathbf{e_i}$ (the i'th unit vector). In order for each row of $\mathbf{g}(\mathbf{x})$ to be limited between 0 and 1 a function, typically called an activation function must be employed. Since we want our output to be limited to 1, our choice of activation functions is somewhat limited, the softmax(Equation 7), and sigmoid(Equation 6) functions being the most obvious choices. Sigmoid has the cons of simplicity, since the rows will remain independent of each other, while the softmax has the intuitive advantage, that the output vector will sum to one. Regardless of choice for now, let σ denote the activation function. The cost function can finally be stated as

$$E(\tilde{\mathbf{W}}) = \frac{1}{2}\sum_{i=1}^{N} \|\sigma(\tilde{\mathbf{W}}\tilde{\mathbf{x}}_i) - \mathbf{y}_i\|^2 \tag{8}$$

Using the sigmoid function as the activation function, some math shows that

$$\nabla E(\tilde{\mathbf{W}}) = \sum_{i=1}^{N} ((\sigma(\tilde{\mathbf{W}}\tilde{\mathbf{x}}_i) - \mathbf{y}_i) \circ \tilde{\mathbf{W}}\tilde{\mathbf{x}}_i \circ (\mathbf{1} - \tilde{\mathbf{W}}\tilde{\mathbf{x}}_i))\mathbf{x_i}^\top \tag{9}$$

where $\circ$ denotes element-wise multiplication(Myrvoll et al. 2023) For the softmax function the calculation becomes somewhat more involved, since the derivative of the softmax function is a jacobian matrix. Formally this is the case for the softmax function also, but since the output of row i only depends on row i of the input for this function the jacobian matrix becomes diagonal. For the softmax function $\mathbf{S}(\mathbf{x})$ one can find that the jacobian matrix is

$$\mathbf{DS}(\mathbf{x}) = \begin{pmatrix} S_1(\mathbf{x})(1 - S_1(\mathbf{x}) & -S_2(\mathbf{x})S_1(\mathbf{x}) & \dots & -S_N(\mathbf{x})S_1(\mathbf{x}) \\ -S_2(\mathbf{x})S_1(\mathbf{x}) & S_2(\mathbf{x})(1 - S_2(\mathbf{x}) & \dots & -S_N(\mathbf{x})S_2(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ -S_N(\mathbf{x})S_1(\mathbf{x}) & -S_N(\mathbf{x})S_2(\mathbf{x}) & \dots & S_N(\mathbf{x})(1 - S_N(\mathbf{x}) \end{pmatrix} \tag{10}$$

Using the L2 loss function is perfectly fine when using the softmax function as well, however a more suited loss function for classification problems is the cross entropy function. This loss function goes hand in hand with the softmax function, since it only cares about the output value at the correct index. That is the loss function is.

$$E(\tilde{\mathbf{W}}) = \sum_{i=1}^{N} \log(\sigma(\tilde{\mathbf{W}}\tilde{\mathbf{x}}_i))^\top \mathbf{y}_i \tag{11}$$

With the jacobian matrix for the softmax function the gradient is not to hard to obtain

$$\nabla E(\tilde{\mathbf{W}}) = -\sum_{i=1}^{N} \mathbf{DS}(\tilde{\mathbf{W}}\tilde{\mathbf{x}}) \cdot (\mathbf{y} \div \sigma(\tilde{\mathbf{W}}\tilde{\mathbf{x}})) \tag{12}$$

where $\div$ denotes element-wise division.

2.4 Template-based Classifiers

The template-based classifier is not statistically based, i.e it applies no probabilistic model in the design Myrvoll et al. 2023. The decision rule for this classifier type is based on the similarity between $\mathbf{x}$ and reference features (templates). For this type of classifier, critical design decisions are (Myrvoll et al. 2023):

- Which decision rule is used
- The kind of similarity measure used
- How the references are chosen and used

One decision rule is the NN rule, which classifies $\mathbf{x}$ as the most similar template. Another method is the KNN rule, which classifies $\mathbf{x}$ as the most frequent template among the $\mathbf{K}$ nearest neighbors.

Multiple distance measures exist. The Mahalanobis distance is defined by

$$d(x, ref_{ik}) = (x - \mu_{ik})^T \Sigma_{ik}^{-1} (x - \mu_{ik}) \tag{13}$$

where Σ is the covariance matrix and μ is a template. The Mahalanobis distance can be simplified by assuming $\Sigma = I$, which obtains the Euclidian distance

$$d(x, ref_{ik}) = (x - \mu_{ik})^T (x - \mu_{ik}). \tag{14}$$

One method for template generation is to choose a subset of the training data. One significant disadvantage of this approach is that only a limited amount of the training data is used for classification, which can lower performance. Another method is clustering, where the training data set is represented by a smaller set of references. For this project, the relevant clustering algorithm is Lloyd's algorithm (Lloyd 1982). A short description of the algorithm follows:

1. Generate K centroids, which define the clusters.
2. For each template, assign template to cluster
3. Compute average of templates in each cluster to obtain new centroids.
4. Repeat until cluster assignments do not change, or max. iterations is reached.

3 Task selection

Three classification topics were considered for this project.

- Speech recognition
- Music recognition
- Image recognition

The speech recognition topic considers the classification of vowels from recorded speech based on peaks in the frequency spectra. The problem is non-separable, as there is significant variation of the frequency peaks between different pronunciations of each vowel, resulting in overlaps between different classes (NTNU 2023).

The music recognition topic considers the assignment of a music genre to an audio track using a variety of classifiers, with a focus on the effect of separability and feature selection.

The task of digit classification considers handwritten digits. In this problem, a feature $\mathbf{x}$ is the pixel brightness of 28-by-28 image. The objective of the task is the design of NN and KNN classifiers

to classify the handwritten digits. This task compares the NN and KNN methods on the MNIST data-set, and evaluates the efficiency of clustering.

The chosen task is digit classification, due to the author' interest in image recognition and the focus on comparison of different classification methods. Additionally, as computational cost is a major design consideration for all practical estimation problems, this task is deemed to give the most insight into real-world application of classification.

3.1 The digit recognition task

The digit recognition task considers the MNIST dataset, which consists of 70 000 handwritten digits that are represented by 28-by-28 px greyscale images. The dataset consists of a training set of 60 000 digits and a test set of 10 000 digits. The objectives are:

1. Design of a NN classifier using Euclidian distance.
2. Design of a NN classifier using Euclidian distance and clustering
3. Design of a KNN classifier using Euclidian distance and clustering
4. Performance analysis and comparison of the classifiers, considering the confusion matrix and error rate

3.2 The Iris task

The project also required another task to be done. This task is the Iris classification task, where a linear classifier for the well-known Iris Fisher data-set is to be designed. The data-set consists of four different measurements of the leafs of three different species of the Iris flower. There are 50 instances of each class for a total of 150 labeled data-points.

In addition to training an linear machine for this data set, a set of experiments were to be conducted, mostly concerning the linear separability of the classes - specifically removing one or several of the measurements before doing the training and classification.

4 Digit recognition implementation

This section describes the implementation and performance of a digit recognition algorithm on the MNIST data-set. 3 variations of the NN approach are considered:

- Unclustered NN classifier
- Clustered NN classifier
- Clustered KNN classifier

Each classifier was evaluated using the code shown in Appendix C. Every script called the evaluation function, but as it is not a part of the classification itself it is only shown in the flowchart b) in Figure 3.

4.1 Unclustered NN classifier

For the unclustered NN (UNN) classifier, the template set was chosen to be a randomized sample of N_t data-points from the training data. Two different implementations were considered; one with $N_t = 1000$, and one with $N_t = 2000$. The randomization was performed to eliminate any

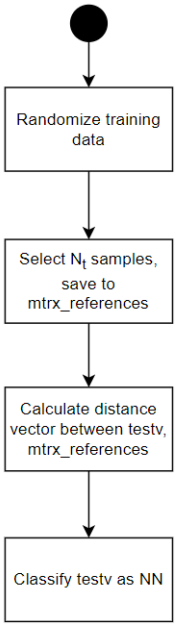


Figure 2: Flowchart: UNN

pattern in the training data and ensure class representation in the template set. A flowchart of the classifier design process and classification is given in Figure 2. The code was implemented as a Matlab script as shown in Appendix D. To evaluate the performance of the classifier, a function to calculate the confusion matrix was implemented as shown in Appendix B.

To calculate Euclidian distance, a custom function was implemented in Matlab as shown in in Appendix A. It was later discovered that the built-in Matlab function *dist()* provides the same functionality, and thus *dist()* was used for the CNN and KNN.

4.2 Clustered NN classifier

The clustered NN (CNN) classifier built on the principle of the UNN classifier. Additionally, a routine for clustering was required to sort the training set into classes and perform the clustering algorithm on each class. This routine could be run once, and the resulting template set used for all subsequent classification. The program flow is shown in Figure 3. The clustering algorithm was provided by the Matlab-function *kmeans()*.

4.3 Clustered KNN classifier

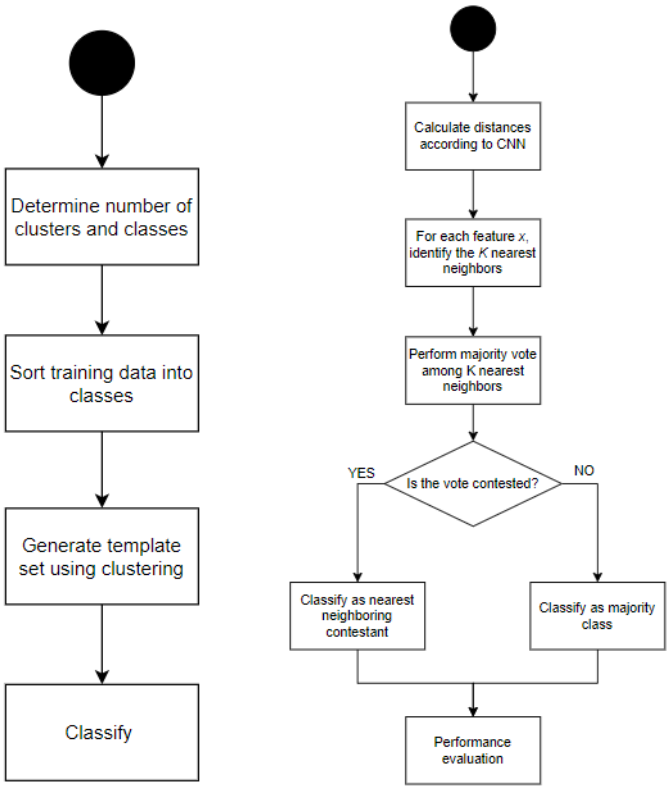
The clustered KNN classifier built on the principle of the CNN classifier, but required an additional routine for identifying the **K** nearest neighbors and performing a "majority vote" among them. The majority vote is the specified KNN decision rule, and it chooses the most common class among the $K = 7$ closest neighbors. If the vote is contested, i.e there is no "clear winner", the closest neighboring contestant is chosen. The program flow is shown in Figure 3, and the complete code is given in Appendix F.

Classifier: → Nr. Templates/Error rate(%):↓	UNN	CNN	KNN
1k templates	13.1	N/A	N/A
2k templates	9.72	N/A	N/A
320 clusters	N/A	5.26	7.52
640 clusters	N/A	4.6	5.9

Table 1: Error rates

Method: → Templates/Runtime (s)↓	UNN	CNN	KNN
1k	135	N/A	N/A
2k	475	N/A	N/A
320	N/A	5.85	6.77
640	N/A	11.4	12.1

Table 2: Run times



(a) Program flow: clustered NN (b) Program flow: KNN

Figure 3: Flowcharts: clustered approaches

4.4 Results and discussion

The error rates are shown in Table 1. Additionally, the algorithm run-times in wall clock seconds are given in Table 2. Note that for the clustered methods, the run-time of the clustering algorithm is excluded. The reason for this is that this run-time is a one-time cost to generate the template set, which can then be reused for subsequent classification. The clustering algorithm took 98.3 seconds to generate 640 templates, and 99.4 seconds to generate 320 templates. The confusion matrices are given in Figure 6. Some misclassified pictures of the UNN classifier is given in Figure 4. Some correctly classified pixtures are given in Figure 5.

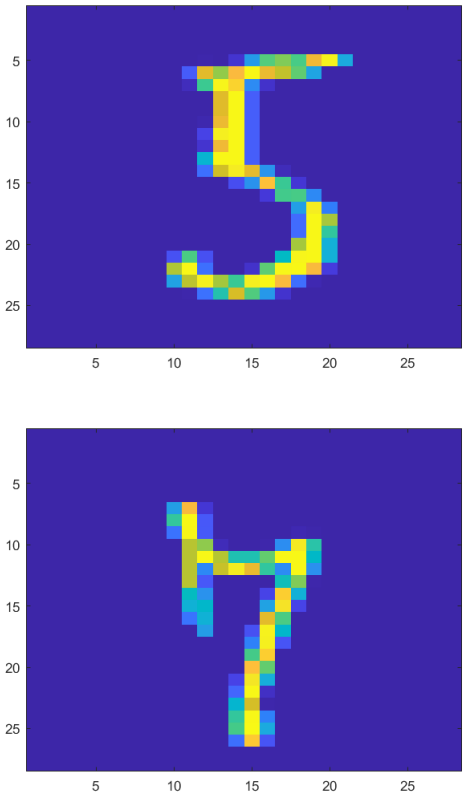


Figure 4: UNN misclassified pixtures. 5 classified as 6, 7 classified as 4

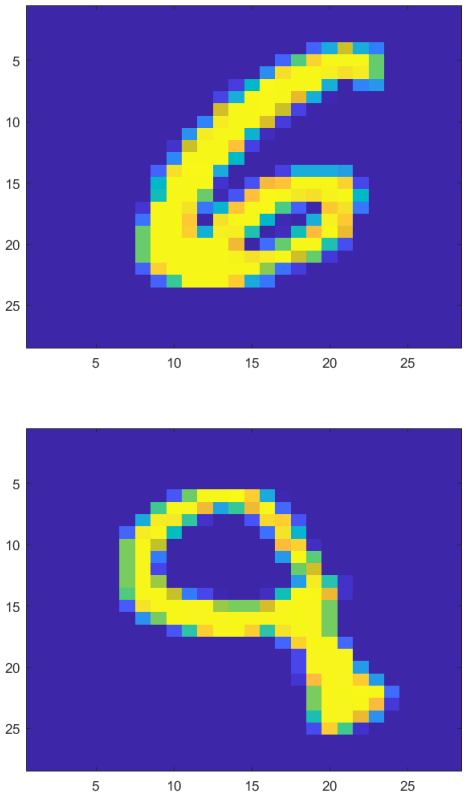


Figure 5: Correctly classified pixtures. 6 and 9

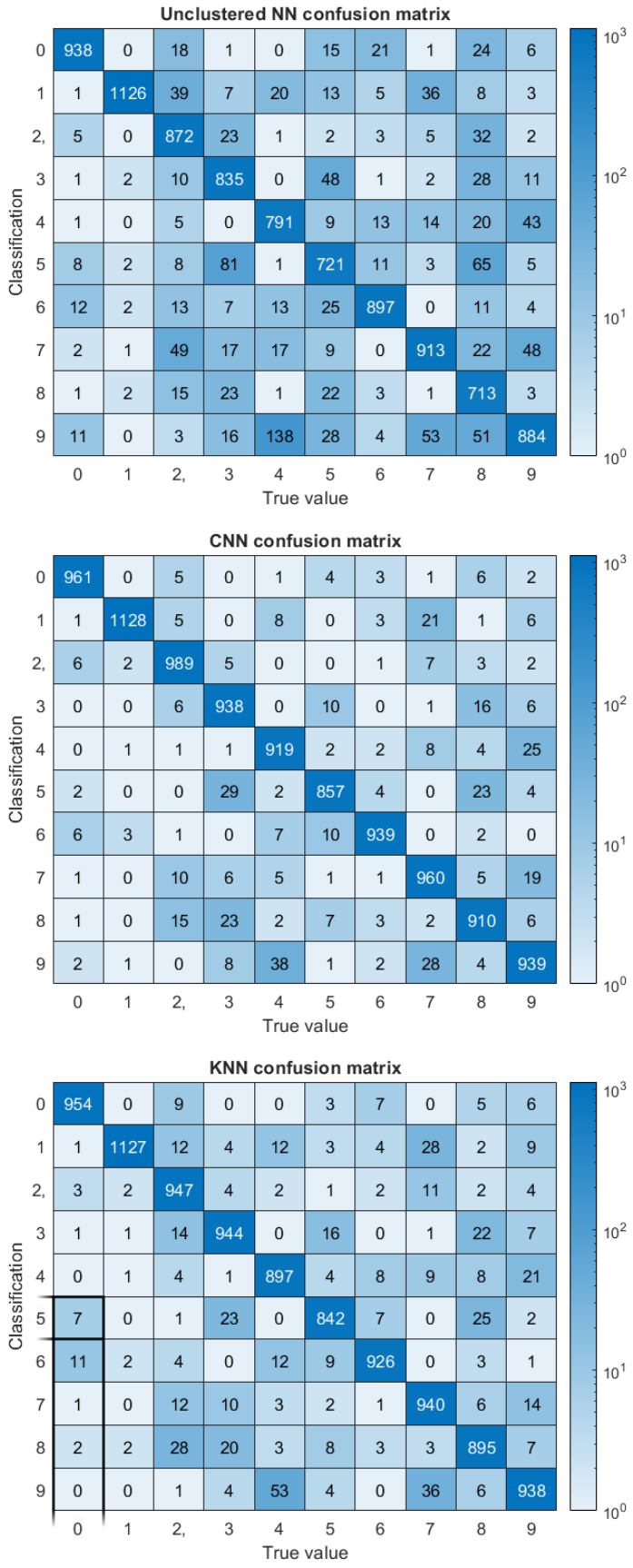
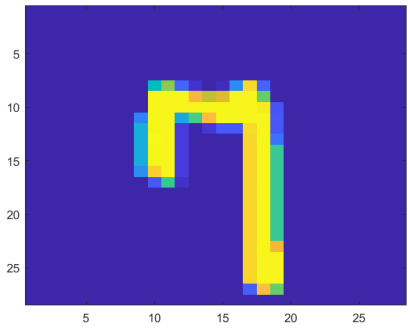


Figure 6: Confusion matrices



81
7
9
7
6
9
9
9

Figure 7: KNN misclassification: Correct class (7) is "outvoted" by wrong class(9)

The clustering approach improved both the error rate and the execution time of the classification. The error rate was reduced from 13.1% to 4.6% when going from using a subset of 1000 training features as templates to using 640 templates generated by clustering. Considering the confusion matrices, it was found that the improvements were especially large for "similar" numbers such as "4" and "9". The improved error rate was likely a result of the template set taking account of the entire training set instead of a small subset. Additionally, clustering reduced the classification time by more than 90%. The reduced execution time is a significant advantage for any practical implementation, as the more efficient algorithm can provide higher data throughput or run on less complex and costly hardware.

The execution time and error rate sensitivity to a change in template set size was investigated. For the UNN, doubling the size of the template set resulted in an error rate reduction of 3.38% (flat) at the cost of a 252% increase in execution time. For the the CNN, doubling the amount of templates from 320 to 640 resulted in an error rate reduction of 0.66% (flat) at the cost of a 95% increase in run-time. The KNN showed similar results. This suggests that there are diminishing returns in the trade-off between accuracy and execution time, and large improvements in execution time can be achieved at small reductions in accuracy by decreasing the number of templates.

Perhaps surprisingly, the KNN was outperformed by the CNN approach. The performance difference increases with the value of K, as shown in Figure 8. In other words: choosing the nearest neighbor yields better performance than the majority vote among the K closest neighbors. Considering the cases where the KNN misclassified a feature that the CNN correctly classified, the low performance of the KNN can be attributed to the correct class being "outvoted" by an erroneous class. One example of this is illustrated in Figure 7. The effect is particularly pronounced with pairs of number that "look the same", such as "4" and "9", and "7" and "9", as shown in Figure 6. Given that the KNN is a more complex classifier, there is no reason to choose it over the CNN unless its accuracy is better. One approach to improving the KNN accuracy is to modify the decision rule so that it considers the similarity measure, for example by assigning higher voting power to the closest neighbors. This would effectively place more importance on the closest neighbors, while retaining rejection of edge-cases where the wrong class has a single template that is very similar.

The CNN was the best performing classifier in terms of error rate. Using 640 clusters it correctly classified 95.4% of the pictures. For reference, the human accuracy on the NIST-set is 98-99% (Simard et al. 1992). As such, the CNN does not match human accuracy, but exhibits decent performance given its simplicity and demonstrates the viability of the NN approach for digit recognition.

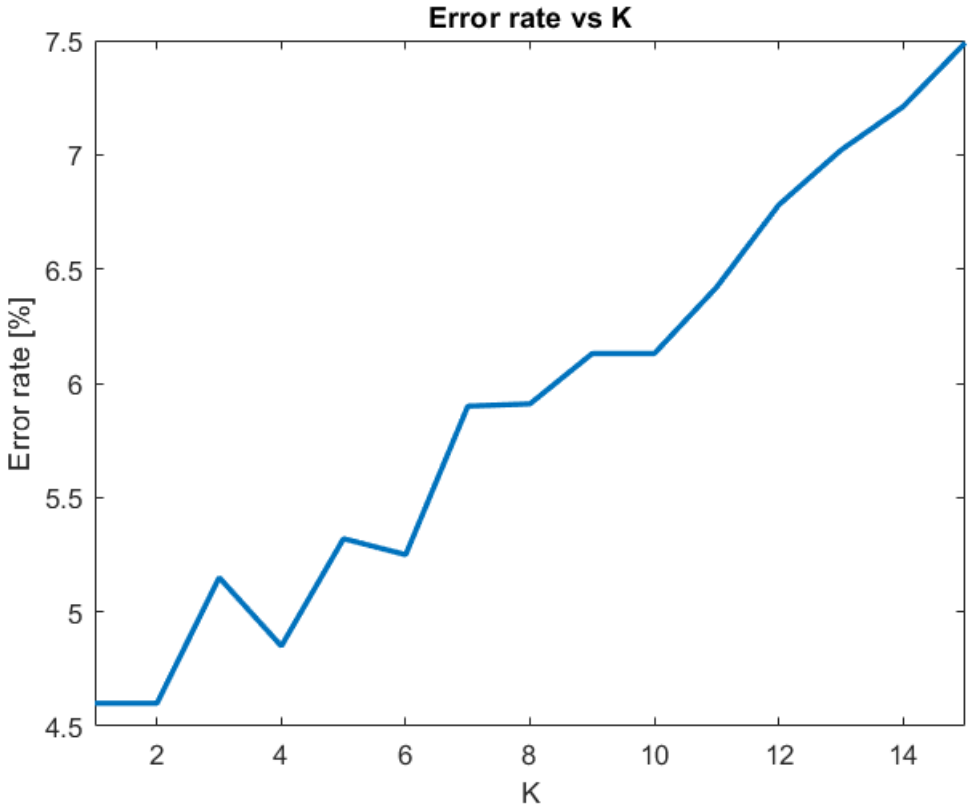


Figure 8: KNN classifier: Error rate vs K

5 Iris classification

This section describes the implementation of a linear classifier for the Iris data set, presents the obtained results, and briefly discusses the findings. All the necessary theory is presented in the theory chapter, specifically sub-chapters 3.1, 3.2 and 3.3.

5.1 Implementation

The classifier was implemented in Matlab, see the attached code. The classifier was trained both using sigmoid and softmax activation functions. When using the sigmoid function the L2 error loss function was used. When using the softmax function the cross entropy loss function was used. Both are presented in the theory section, and the results are compared below. A few key points became clear during implementation. Firstly, the importance of input normalization became transparent. A first implementation used non-normalized data, which resulted in unpredictable behaviour. Sometimes the classifier would converge nicely, other times it wouldn't converge at all. The practice of input normalization is in theory not strictly necessary, however experiments like this show that input normalization gives more robust gradient descent optimization. The code uses min-max normalization, where each feature gets scaled by the following formula:

$$x_{\text{norm}} = \frac{x - \min x}{\max x - \min x}$$

This makes all the elements of each feature vector range between 0 and 1.

The code normalizes the input, applies gradient descent to train the classifier, and finally evaluates the classifier, see the flowchart Figure 9. The code also includes a section concerning an exploratory

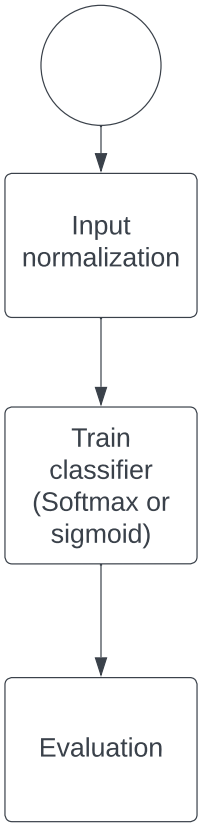


Figure 9: Flow chart for iris classifier code

data analysis to inspect the separability properties of the data set.

5.2 Results

In this section the results and main finding of the linear classifier on the Iris data set are presented. The training process became very stable after input normalization was implemented. Regardless of loss and activation function combination used, the loss function seemed to converge quite quickly and robustly even with a large learning rate ($\alpha = 0.3$). A plot of the loss functions for the different schemes is found in Figure 10. To facilitate analysis, the same number of epochs was employed in the two schemes, even though it seemed that the softmax-cross entropy pairing converged somewhat faster than the sigmoid-L2 error pairing. On the other hand the training time with the softmax activation function was on average around 5 times longer compared to using the sigmoid activation function. Another difference between the two plots, is that the sigmoid scheme seems to be more stable compared to the softmax scheme which has large spikes before it converges.
From Figure 10 it may seem that the error was smaller for the softmax scheme, however this comparison is not entirely fair, since the two schemes use different loss functions. Still the two loss functions should be somewhat related as the underlying problem is the same, and optimizing one should also decrease the other. The results shown in Figure 12 corroborates this hypothesis. The softmax scheme does a better job at minimizing the L2 error than the sigmoid scheme.
This result is reflected in the evaluation of the classifiers as well. As we can see from the tables 3, 4, 5 and 6 the softmax scheme generally performed better than the sigmoid training scheme, and it achieved an error rate of 0 on the training set.

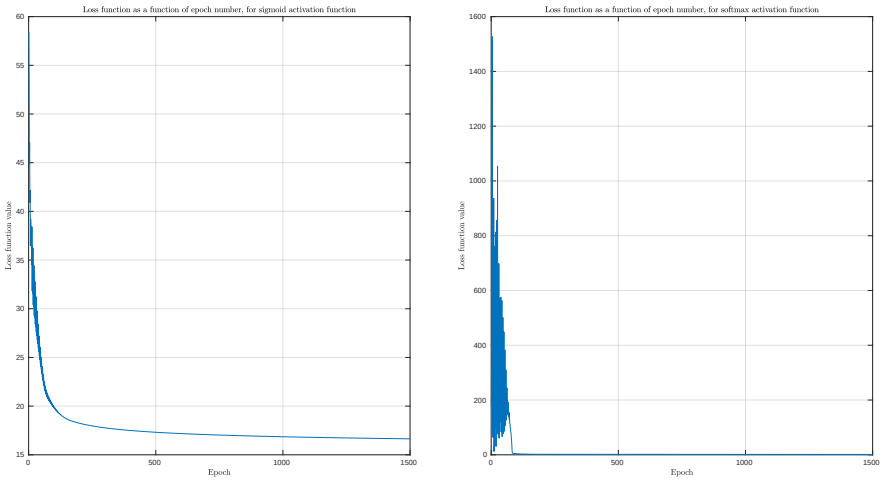


Figure 10: Loss function for sigmoid(left) vs. softmax(right) activation functions

Classified: → True/label:↓	Iris Setosa	Iris Versicolour	Iris Virginica
Iris Setosa	30	0	0
Iris Versicolour	0	28	2
Iris Virginica	0	1	29

Table 3: Confusion matrix on the training set for sigmoid scheme. Error rate: 0.03

In the above-mentioned results the first 30 samples of each class were used for training while the last 20 were used for testing. The results of the opposite split, that is using the last 30 samples from each class for training and the first 20 for testing, are presented in Table 7. The only noticeable difference in this case is that for the softmax-scheme the classifier was unable to get 0 error on the training data, while it was able to do that on the test data. The number of total errors made by the classifier for the softmax-scheme remained the same however.

Next the separability properties of this data set were examined. Firstly histograms of the features for each class were produced (Figure 11). From this, one feature after another was removed from the data before the classifier was re-trained. The features were picked in increasing order of separability; the features showing the most overlap were removed first. The removed features were sepal width, sepal length and petal length in that order. The evaluation results of these experiments are presented in Table 8, Table 9 and Table 10. We see that the sigmoid-scheme was more robust to losing features in comparison to the softmax-scheme. However, the softmax-scheme generally provides higher accuracy. When classifying using only one feature, both schemes have worse performance, but the result is still relatively good, exemplified by the classification of the Iris Setosa class being error free.

Classified: → True/label:↓	Iris Setosa	Iris Versicolour	Iris Virginica
Iris Setosa	20	0	0
Iris Versicolour	0	19	1
Iris Virginica	0	1	19

Table 4: Confusion matrix on the test set for sigmoid scheme. Error rate: 0.03

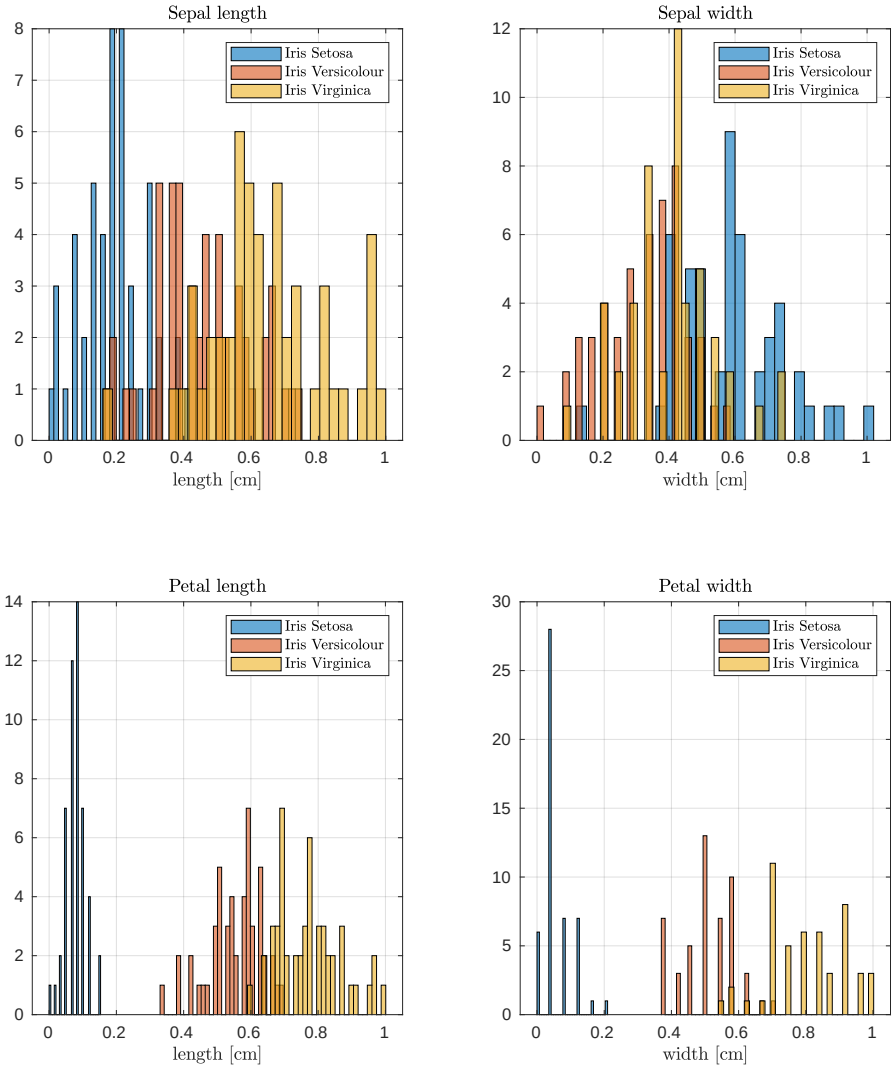


Figure 11: Histograms of Iris data

Classified: → True/label:↓	Iris Setosa	Iris Versicolour	Iris Virginica
Iris Setosa	30	0	0
Iris Versicolour	0	30	0
Iris Virginica	0	0	30

Table 5: Confusion matrix on the training set for softmax scheme. Error rate: 0

Classified: → True/label:↓	Iris Setosa	Iris Versicolour	Iris Virginica
Iris Setosa	20	0	0
Iris Versicolour	0	19	1
Iris Virginica	0	1	19

Table 6: Confusion matrix on the test set for softmax scheme. Error rate: 0.03

Sigmoid						Softmax					
Training data			Test data			Training data			Test data		
30	0	0	20	0	0	30	0	0	20	0	0
0	28	2	0	19	1	0	29	1	0	20	0
0	1	29	0	2	18	0	1	29	0	0	20
Error rate: 0.03			Error rate: 0.05			Error rate: 0.02			Error rate: 0		

Table 7: Confusion matrices for the linear classifier trained on the last 30 samples, and tested on the 20 first

Sigmoid						Softmax					
Training data			Test data			Training data			Test data		
30	0	0	20	0	0	30	0	0	20	0	0
0	28	2	0	19	1	0	29	1	0	19	1
0	1	29	0	2	18	0	0	30	0	0	20
Error rate: 0.03			Error rate: 0.05			Error rate: 0.01			Error rate: 0.02		

Table 8: Confusion matrices for the linear classifier on data set without sepal width data

Sigmoid						Softmax					
Training data			Test data			Training data			Test data		
30	0	0	20	0	0	30	0	0	20	0	0
0	28	2	0	20	0	0	28	2	0	19	1
0	1	29	0	2	18	0	2	28	0	1	19
Error rate: 0.03			Error rate: 0.03			Error rate: 0.04			Error rate: 0.03		

Table 9: Confusion matrices for the linear classifier on data set without sepal width and sepal length data

Sigmoid						Softmax					
Training data			Test data			Training data			Test data		
30	0	0	20	0	0	30	0	0	20	0	0
0	27	3	0	18	2	0	28	2	0	20	0
0	1	29	0	2	18	0	2	28	0	2	18
Error rate: 0.04			Error rate: 0.07			Error rate: 0.04			Error rate: 0.03		

Table 10: Confusion matrices for the linear classifier on data set without sepal width, sepal length and petal length data

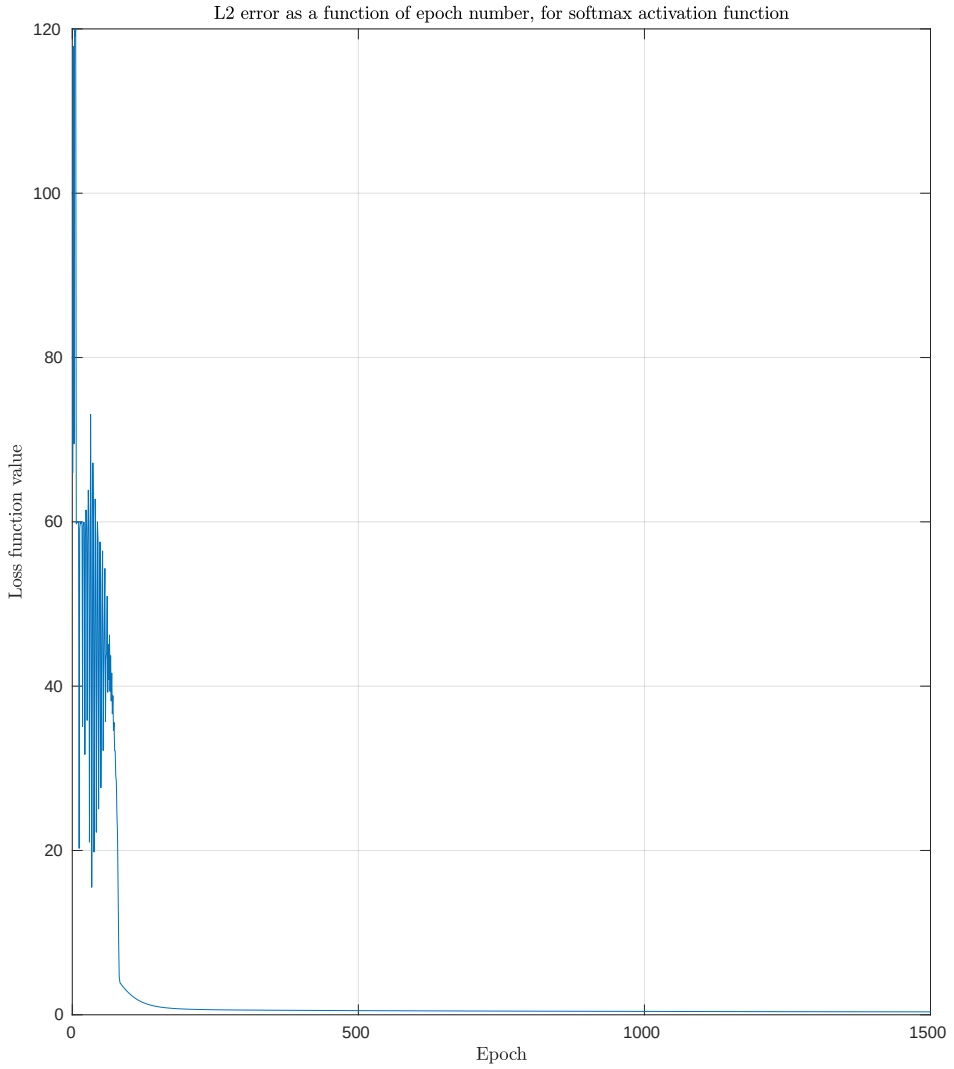


Figure 12: L2 error for classifier with softmax activation function, optimized on the cross entropy loss function

5.3 Discussion

In this section some of the results from the previous section are discussed and attempted explained.

The generally better performance of the softmax-scheme is an anecdote that this scheme is better suited for classification problems. That restricting the output to a set generally closer to the desired output gave improved results also makes intuitive sense, as it eliminated functions that were wrong. This is also a result of choosing one-shot encoding, where in this case the sum of the correct output vector is always one.

An interesting result showed up when comparing using the last 30 samples as training data to when the 30 first samples were used. For the softmax-scheme the classifier achieved 0 misclassifictions for the first 30 samples when used for training, and for the first 20 samples when used as test data. The total number of misclassifications remained constant for the two splits of the data. This suggests that there are exactly two samples that make the two classes not linearly separable, and moreover that these two samples are in the last 20 samples of the two classes. When using the sigmoid scheme the difference was small, but present. Also the total number of miss-classifications were different in this case for the two splits. This is perhaps simply because the sigmoid-scheme was not able to achieve the best possible linear classifier in either of the two cases. That the number of miss classification on the data set remained constant for the two splits when using the softmax scheme allows one to hypothesize that this linear classifier is the best one achievable and that the misclassifications were a "problem" in the data-set and not the classifier.

Considering removing features, the experiments revealed interesting qualities of the underlying classification problem. Firstly it seems that the addition of more features does not help, as the results with just one feature were surprisingly good. However it is clear that better performance is achieved with more features. Intuitively this makes sense, but it is surprising how well the classification worked with just one feature. This suggests that the features are strongly correlated, and provide little extra information. It is also interesting that in all cases the classification for the Setosa class was error free, and that no other samples are misclassified as the Setosa. All errors occured in between the two other classes. This suggests that the first class is linearly separable from the other two, but that these are not linearly separable from each other.

6 Conclusion

In this project, classifiers of handwritten digits and iris species were implemented and tested.
For the digit recognition task, an Unclustered NN, Clustered NN and KNN (K=7) classifiers were implemented, tested and compared. The Clustered Nearest Neighbor (CNN) classifier showed the highest performance in both execution time and accuracy, achieving an error rate of 4.6%. The implementation demonstrated the efficiency of clustering, with a significant improvement in error rate and execution time compared to using a subset of training data as the template set. The relatively low accuracy of the KNN classifier was discussed, and it was deduced that the simple majority vote decision rule is prone to "sabotage" by similar classes. Further work on the KNN should explore modified decision rules, for example with the introduction of distance-based weighting of neighbors.

For the Iris classifier two implementations were tested and compared. The softmax-cross-entropy scheme seemed to work somewhat better, which was expected as it is tailored for classification problems. At best the linear classifier achieved 2 misclassifications on the whole data set. It was observed that for the softmax scheme different splits of the data between test data and training data resulted in the same error rate, suggesting single data points were making the problem linearly inseparable, and the classifier optimal. Further experiments concerning the exact misclassification would have to be conducted to investigate the hypothesis.
The linear separability properties of the data set were further studied, and it was found that with just one (strategically chosen) feature the classifier was still able to achieve good results.

Bibliography

Duda, Richard O., Peter E. Hart and David G. Stork (2000). *Pattern classification*. John Wiley Sons.

Lloyd, Stuart P. (1982). 'Least Squares Quantization in PCM'. In: *IEEE Transactions on Information Theory* 28, pp. 129–137.

Myrvoll, T., S. Werner and M. Johnsen (2023). *Estimation, Detection and Classification theory*.

Nocedal, Jorge and Stephen J. Wright (2006). *Numerical Optimization*. Springer.

NTNU (2023). 'Classification of pronounced vowels'. In: *TTT4275 task descriptions*.

Simard, Cun and Denker (1992). 'Efficient Pattern Recognition Using a New Transformation Distance'. In: *NIPS*, p. 56.

Appendix

A Euclidian distance

```
function dist_eucl = calc_distance_euclidian(x,mu)
%CALC_DISTANCE_EUCLIDIAN Calculates the euclidian distance between a feature
↪  %vector x and template vector mu
assert(size(x,1) == size(mu,1),'dist_eucl: feature length =! reference length');
dist_eucl = (x-mu)'*(x-mu);
```

B Confusion matrix

```
function mtrx_confusion =
↪  calc_confusion_matrix(vec_true_classes,vec_estimated_classes)
%CALC_CONFUSION_MATRIX Calculates the confusion matrix from a set of
%classification results
mtrx_confusion = zeros(10);

for i=1:length(vec_estimated_classes)
    mtrx_confusion(vec_estimated_classes(i)+1, vec_true_classes(i)+1) =
↪  mtrx_confusion(vec_estimated_classes(i)+1, vec_true_classes(i)+1) + 1;
end
end
```

C Performance evaluation

```
function classifier_evaluate(class_estimated,class_true)
%CLASSIFIER_EVALUATE Calculates and dislays the error rate and confusion matrix
↪  given a class estimate and class ground truth

num_samples = length(class_estimated);

% Find confusion matrix and error rate for the test set.
mtrx_confusion = calc_confusion_matrix(class_true(1:num_samples),
↪  class_estimated);
is_equal = class_estimated == class_true(1:num_samples);
num_correct = sum(is_equal);
error_rate = (num_samples-num_correct)/num_samples * 100;

% Display
disp(strcat("The error rate for the classifier is ", num2str(error_rate), "%."));

figure;
hmo = heatmap({'0', '1', '2,' '3', '4', '5', '6', '7', '8', '9'}, {'0', '1', '2,'
↪  '3', '4', '5', '6', '7', '8', '9'}, mtrx_confusion);
hmo.Title = "Classifier confusion matrix";
hmo.XLabel = "True value";
hmo.YLabel = "Classification";
hmo.ColorScaling = 'log';
end
```

D MNIST task 1

```
%% Design NN-based classifier using Euclidian distance

% Use a subset of num_samples samples as reference. For each feature vector x,
% find the training reference with the smallest distance and classify as
% that target
%% Init
num_references = 1000;
num_samples = 10000;
%% Run classification using the first num_references references and the first
↪  num_samples test features
tic;

mtrx_references = trainv(1:num_references,:)';
vec_targets = trainlab(1:num_references);

classes = zeros(num_samples,1);
for test_samp = 1:num_samples
    x_test = testv(test_samp,:)';
    mtrx_dist = calc_distance_euclidian(x_test,mtrx_references);
    distances = diag(mtrx_dist);
    [dist_min,ind_min] = min(distances);
    classes(test_samp) = vec_targets(ind_min);
end
toc;

%% Evaluate performance
classifier_evaluate(classes, testlab);

disp('..done');
```

E MNIST task 2

```
%% Task 2
% Use clustering to produce a small(er) set of templates for each class.
N_clusters = 64;
N_classes = 10;

%% Perform clustering of the 6000 training vectors for each class into M = 64
↪  clusters.
% Separate training data into matrix based on class
tic;

load('class_clusters.mat');
% arr_class_clusters = cell(1,10);
% for i = 1:length(trainlab)
%     arr_class_clusters{trainlab(i)+1}(end+1,:) = trainv(i,:);
% end

load('arr_clusters.mat');
% arr_clusters = cell(1,10);
% for i =1:length(arr_clusters)
%     [idx,C] = kmeans(arr_class_clusters{i}, N_clusters);
%     arr_clusters{i} = C;
% end
```

```
toc;


%% Perform classification
% Run classification on one class, find the
% minimum distances for all feature vectors
tic;
for i = 1:N_classes
    distances = dist(arr_clusters{i}, testv');
    min_distances(i,:) = min(distances,[],1);
end

% Classify by choosing the class with the minimum distance
[~,class] = min(min_distances,[],1);
class = class'-1;
toc;

%% Evaluate performance
classifier_valuate(class, testlab)

disp('..done')
```

F MNIST task 3

```
%% Design a KNN classifier with K=7. Find the confusion matrix and the error
↪  rate
tic;
K_knn = 7;

% Collect all distances in a single matrix
distances = zeros(N_classes*N_clusters, length(testlab));
for i = 1:N_classes
    distances(N_clusters*(i-1)+1:N_clusters*i,:) = dist(arr_clusters{i}, testv');
end

%% Classify
% Identify the K nearest neighbours of each feature vector
[~,I] = mink(distances, K_knn, 1);
classes_neighbors = floorDiv(I-1,N_clusters);

classes = zeros(length(testlab), 1);
% Majority vote; identify the class with the highest number of neighbors,
% if two classes get equal amount of votes, choose the closest neighbor
for i=1:length(testlab)
    % Determine the count of each class
    [votes, class_val] = groupcounts(classes_neighbors(:,i));
    % Find the class(es) with the most votes
    [max_votes, ind] = max(votes);
    % Find number of contestants
    is_contestant = votes == max_votes;
    N_contestants = sum(is_contestant);
    if N_contestants > 1
        % Determine what classes are contestants
        contestants = class_val(is_contestant);
        % Find the shortest distance to each of the classes
        contestant_distance = zeros(N_contestants, 1);
```

```
        for j=1:N_contestants
            contestant_distance(j) = min(dist(arr_clusters{contestants(j)+1},
↪   testv(i,:)'));
        end
        % Classify as contestant class with the lowest distance
        [~, ind] = min(contestant_distance);
        classes(i) = contestants(ind);
    else
        % Classify as class with most votes
        classes(i) = class_val(ind);
    end
end
toc;

%% Evaluate performance
classifier_valuate(class, testlab)

disp('..done')
```

G Softmax gradient

```
function gradSoft = softmaxGradient(x)
    sigma = softmax(x);
    gradSoft = zeros(size(x));
    for i = 1:length(x)
        for j = 1:length(x)
            if i == j
                gradSoft(i,j) = sigma(i)*(1-sigma(i));
            else
                gradSoft(i,j) = -sigma(i)*sigma(j);
            end
        end
    end
end
```

H Iris classification code

```
clear all
close all
x1all = load('class_1','-ascii');
x2all = load('class_2','-ascii');
x3all = load('class_3','-ascii');
%% Exploritary data analysis
% Histograms
% Sepal length
fig = figure;
subplot(2,2,1);
nbins = 30;
samples = 1:50;
histogram(x1all(samples,1),nbins);
hold on;
histogram(x2all(samples,1),nbins);
hold on;
histogram(x3all(samples,1),nbins);
```

```
legend("Iris Setosa", "Iris Versicolour", "Iris
↪  Virginica",'interpreter','latex');
title("Sepal length",'Interpreter','latex');
xlabel("length [cm]",'Interpreter','latex');
grid on;

%Sepal width
subplot(2,2,2);
nbins = 30;
histogram(x1all(samples,2),nbins);
hold on;
histogram(x2all(samples,2),nbins);
hold on;
histogram(x3all(samples,2),nbins);
legend("Iris Setosa", "Iris Versicolour", "Iris
↪  Virginica",'interpreter','latex');
title("Sepal width",'Interpreter','latex');
xlabel("width [cm]",'Interpreter','latex');
grid on;

%Petal length
subplot(2,2,3);
nbins = 30;
histogram(x1all(samples,3),nbins);
hold on;
histogram(x2all(samples,3),nbins);
hold on;
histogram(x3all(samples,3),nbins);
legend("Iris Setosa", "Iris Versicolour", "Iris
↪  Virginica",'interpreter','latex');
title("Petal length",'Interpreter','latex');
xlabel("length [cm]",'Interpreter','latex');
grid on;

%Petal width
subplot(2,2,4);
nbins = 30;
histogram(x1all(samples,4),nbins);
hold on;
histogram(x2all(samples,4),nbins);
hold on;
histogram(x3all(samples,4),nbins);
legend("Iris Setosa", "Iris Versicolour", "Iris
↪  Virginica",'interpreter','latex');
title("Petal width",'Interpreter','latex');
xlabel("width [cm]",'Interpreter','latex');
grid on;

%% Remove petal lenght
x1all(:,1) = [];
x2all(:,1) = [];
x3all(:,1) = [];

%% Remove sepal width
x1all(:,2) = [];
x2all(:,2) = [];
x3all(:,2) = [];
%% Remove sepal length
```

```
x1all(:,1) = [];
x2all(:,1) = [];
x3all(:,1) = [];
%% Data pre-processing
% Input normalization
Max = max([x1all;x2all;x3all]);
%max finds the biggest element in each colum(i.e. feature)
Min = min([x1all;x2all;x3all]);
%min finds the smallest element in each colum(i.e. feature)
x1all = (x1all-Min)./(Max-Min);
x2all = (x2all-Min)./(Max-Min);
x3all = (x3all-Min)./(Max-Min);

% Homogeneus form
trainSamples = 1:30;
testSamples = 31:50;

x1train = [ones(length(trainSamples),1), x1all(trainSamples,:)];
x2train = [ones(length(trainSamples),1), x2all(trainSamples,:)];
x3train = [ones(length(trainSamples),1), x3all(trainSamples,:)];

x1test = [ones(length(testSamples),1), x1all(testSamples,:)];
x2test = [ones(length(testSamples),1), x2all(testSamples,:)];
x3test = [ones(length(testSamples),1), x3all(testSamples,:)];


% Merge training data for ease of training
xtrain = [x1train; x2train; x3train]';
% One-hot encoding
encodingClass1 = [1; 0; 0];
encodingClass2 = [0; 1; 0];
encodingClass3 = [0; 0; 1];
% Create correct output values for training
ytrain = [repmat(encodingClass1,1,length(trainSamples))...
    , repmat(encodingClass2,1,length(trainSamples))...
    , repmat(encodingClass3,1,length(trainSamples))];

% Group test data in cells for easy evaluation
numClasses = 3;


xtest = cell(1,numClasses);
xtest{1} = x1test';
xtest{2} = x2test';
xtest{3} = x3test';

%clearvars -except xtrain ytrain xtest numClasses
numFeatures = height(xtrain)-1;


%% Training

% Hyperparameters
numEpochs = 1500;
alpha = 0.3;
eta = alpha;
decay = 0.1;
%Random initial values
```

```
W = randn(numClasses, numFeatures+1);% +1 for bias term
Errors = zeros(1,numEpochs);
tic;
for j = 1:numEpochs
    %alpha = eta /(1+decay*j);
    gradW = zeros(numClasses, numFeatures+1);
    E = 0;
    for i = 1:length(xtrain)
        x = xtrain(:,i);
        y = ytrain(:,i);
        g = sig(W*x);
        %g = softmax(W*x);
        gradW = gradW + (((g-y).*g).*(1-g))*x';
        %gradW = gradW + (softmaxGradient(W*x)*(g-y))*x';
        %gradW = gradW -(softmaxGradient(W*x)*(y./g))*x';
        %E = E - log(g)'*y;
        E = E + (g-y)'*(g-y);
    end
    Errors(j) = E;
    W = W - alpha*gradW;
    fprintf("Epoch: %d,Learning rate: %f, Error: %f\n", j, alpha, E);
end
t = toc;
fprintf("Training took %.2f seconds",t);
fig = figure;
plot(Errors);
grid on;
xlabel("Epoch",'Interpreter','latex');
ylabel("Loss function value",'Interpreter','latex');
title("Loss function as a function of epoch number, for sigmoid activation
↪  function",'Interpreter','latex');
%exportgraphics(fig,"L2error_softmax.pdf")

%% Evaluation
confMatrixtest = zeros(numClasses,numClasses);
confMatrixtrain = zeros(numClasses,numClasses);
for j = 1:length(xtrain)
    x = xtrain(:,j);
    y = find(ytrain(:,j)==max(ytrain(:,j)));
    g = W*x;
    class = find(g == max(g));
    % Count classifications on training dataset
    confMatrixtrain(y,class) = confMatrixtrain(y,class) + 1;
end
errorRateTrain =
↪  (sum(confMatrixtrain,'all')-sum(diag(confMatrixtrain)))/length(xtrain);

for i = 1:numClasses
    for j = 1:length(xtest{i})
        x = xtest{i}(:,j);
        g = W*x;
        class = find(g == max(g));
        % Count classifications on training dataset

        confMatrixtest(i,class) = confMatrixtest(i,class) + 1;
    end
end
```

```
errorRateTest =
↪  (sum(confMatrixtest,'all')-sum(diag(confMatrixtest)))/(3*length(xtest{1}));

%confMatrixtest = confMatrixtest/length(xtest{1})*100;
%confMatrixtrain = confMatrixtrain/(length(xtrain)/numClasses)*100;

disp("Confusion matrix training set:");
disp(confMatrixtrain);
fprintf("Error rate training set: %.2f \n",errorRateTrain);
disp("Confusion matrix test set:");
disp(confMatrixtest);
fprintf("Error rate test set: %.2f\n",errorRateTest);




function sig = sig(x)
    sig = 1./(1+exp(-x));
end

function soft = softmax(x)
    soft = exp(x)/sum(exp(x));
end
```