NTNU
Kunnskap for en bedre verden

Department of Electronic Systems

TTT4275 - Estimation, Detection and Classification

# Classification project - 1

*Authors:*
Henrik Jordheim
Kristian Borgen

**Abstract**

This lab report presents the theory, implementation and results related to two classification tasks. The first task is about classification of Iris flowers using a linear discriminant classifier, achieving an error rate of 8.33%. The second task is about classification of handwritten numbers with variations of NN classifiers, achieving an error rate of 3.06%.

April, 2021

# Contents

# 1 Introduction

Classification could be seen as the way of systematically arrange certain elements into categorical groups based on some categorization criteria. While the open and private working-sectors of society is being build up of more and more information about industries, factories, processes, workers, and generally things that could be put into a structural, categorical way, the necessity to have the machines working along is urgent. Given a factory that mass produces chips that should be applied to critical systems, one should think that one chip could take many man-hours of work to ensure that the chip is fully operational and not defect. By having a machine that has, based on some given features, learnt how to detect such defects apart from the fully operational ones, one may believe that the production speed would drastically increase, and the error rate decrease. By training a machine to do a specific task - so called machine learning - would have huge benefits to optimize day to day work in industry and in the society in general.

The goal of the project is to get a glimpse of some of the techniques done by supervised learning. This is generally done by training up the machine - the classifier - by labeled, structured samples that contains information about certain features of given objects - classes - and then makes it possible to categorize future samples with the help of the trained classifier.

The project introduces how a linear classifier manages to separate different Iris flowers based on the length and width features of their leaves. Following this the concepts of a nearest neighbour based classifier and clustering of training data will be introduced, where hand written numbers from 0 to 9 is classified. This report will firstly introduce the theory needed to understand the Iris- and handwritten numbers tasks. Secondly, the iris task is presented, following by how it was implemented and the following results and discussion about it. Next, the handwritten numbers task is presented, and likewise as the Iris task, followed by how it was implemented, results and discussion regarding the results. Finally, some concluding words about the project will be stated.

# 2 Theory

This section presents the theory needed in order to understand the upcoming tasks, the implementation of the suggested solutions and the discussion of the obtained results.

## 2.1 Linear Separability

When classifying samples in a data set, one would try to distinguish the samples into different classes based on the different feature measurements that one sample is constructed of. The separability problem this causes is then solved by separating the samples into C different classes, which by constructing a hypersurface (or a line if $C = 2$) is separated into different regions in the feature space. One may then categorize these problems into three different problems: Linearly separable problems, non-linearly separable problems and non-separable problems. This categorization is dependent on how the hypersurface is splitting the classes in the feature space, where in the linear case the hypersurface becomes a hyperplane. Similarly, a problem is non-linearly separable if the classes can be separated by a non-linear hypersurface, while in the non-separable case the classes cannot be completely distinguished based on the given features.

## 2.2 The Linear Discriminant Classifier

When a classifier predicts the class of a sample it makes its decision based on a decision rule. This rule corresponds to a partitioning of the feature space of the samples. This partitioning further defines different decision regions such that if a sample is found to be in decision region X then its predicted class is also X. A linear classifier is a classifier that makes its decisions based on a linear partitioning of the sample space, i.e. the different decision regions are separated by hyperplanes. In the case of the project, the linear discriminant classifier ([1], section 2.4) is the classifier of interest. In such a classifier each class is described by a discriminant function:

$$g_i(x) = w_i^T x + w_{i0}, \qquad i = 1, ..., C \tag{1}$$

where $w_i$ is the weight of the class, $x$ the input sample, also known as a Fisher vector, and $w_{i0}$ is the discriminant offset. The decision rule for the linear discriminant classifier is then given by

$$x \in w_j \Leftrightarrow g_j(x) = \max_i g_i(x) \tag{2}$$

where $w_j$ is the predicted class of sample $x$ and $g_j(x)$ is the discriminant function with the highest value for that function. It is then possible to construct a more compact representation of the discriminates, by putting them in a matrix form

$$g = Wx + w_0 \tag{3}$$

where $g \in \mathbb{R}^C, W \in \mathbb{R}^{CXD}, x \in \mathbb{R}^C$ and $w_0 \in \mathbb{R}^C$.

To make the training of such a classifier less cumbersome, and easier to implement, the discriminant is transformed into a homogeneous form, where $\tilde{W} = [Ww_0]$ and $\tilde{x} = [x^T 1]$ which yields the new homogeneous discriminant function as $g = \tilde{W}\tilde{x},\ \tilde{W} \in \mathbb{R}^{CX(D+1)},\ \tilde{x} \in \mathbb{R}^{C+1}$.

The classifier is trained by tuning in the correct weights in $\tilde{\text{W}}$. Each $w_i$ is then the normal vector to one of the hyperplanes that separates the decision regions for the classifier. The training can be done by exposing the classifier for N numbers of samples, such that the hyperplanes are tuned in sufficiently to distinguish the classes based on the feature measurements given in a sample. One method to perform such a tuning is by minimum square error (MSE) based training and a gradient descent technique ([1], section 3.2).

## 2.3 Minimum Square Error & Gradiant Descent

MSE tuning is based on minimizing the square error between the discriminant values and the target vector for each training sample:

$$MSE = \frac{1}{2}\sum_{k=1}^{N}(g_k - t_k)^T(g_k - t_k) \tag{4}$$

where $N$ is the number of training samples, $g_k$ is the discriminant vector for sample $x_k$ (denoted only $g$ in the previous section) and $t_k \in \mathbb{R}^C$ is the target vector for that sample. The target vector $t_k$ is a zero-vector, except for the element on the index equal to the label of sample $x_k$, which has the value one. For instance, if sample $x_1$ has label 2 and $C = 3$ then the target vector for that sample is $t_1 = [0, 1, 0]$.

To be able to use the output of the discriminant, $g_k = \tilde{W}\tilde{x}_k$, one should be able to classify in a binary matter. I.e. one class should be chosen by giving that element of the discriminant vector the value 1 and the other classes should be disregarded, by giving the other elements of the discriminant vector the value 0. Since the output of the discriminant is a continuous valued vector, there should be used, ideally, a heaviside step function to map each discriminant vector into a corresponding target vector. However, in order to use gradient descent optimization algorithms the functions should be differentiable. A sigmoid function can therefore be used to approximate the heaviside function:

$$g_k \approx sigmoid(x_k) = \frac{1}{1 + \exp -z_k} \tag{5}$$

$$z_k = \tilde{W}\tilde{x}_k \tag{6}$$

To be able to optimize $\tilde{W}$ one could use the steepest descent algorithm, which is a basic gradient descent optimization algorithm where each iterative step $m$ is taken in the opposite direction of the gradient, i.e steepest descent:

$$\tilde{W}_m = \tilde{W}_{m-1} - \alpha\nabla_W MSE, \quad 0 < \alpha \leq 1 \tag{7}$$

which contains the previous iteration of $\tilde{W}$, a step factor $\alpha$ and the gradient of the MSE with respect to $\tilde{W}$. $\alpha$ is the step length of each iteration and is a tuning parameter for the optimization. The choice of $\alpha$ can be based on various techniques, such as the Wolfe conditions [3] or chosen by a divide & conquer strategy [4] where you try different values until you get a satisfactory result. $\nabla_W MSE$ is calculated as follows:

$$\nabla_W MSE = \sum_{k=1}^{N}\nabla_{g_k} MSE \nabla_{z_k} g_k \nabla_W z_k \tag{8}$$

where

$$\begin{aligned} \nabla_{g_k} MSE &= g_k - t_k \\ \nabla_{z_k} g_k &= g_k \circ (1 - g_k) \\ \nabla_W z_k &= x_k^T \end{aligned}$$

When implementing the steepest descent algorithm it is important to make sure the algorithm has a termination criterion. Normal termination criterions are when the current iterate is sufficiently close to a local optimum, the algorithm makes no particular progress or it has simply iterated too many times. In this way, the criterion could be implemented as one or more of these:

- $\|\nabla_W MSE\|_\infty \leq$ Some tolerance, $\epsilon$
- $\|x_{k+1} - x_k\|_\infty \leq$ Some tolerance, $\epsilon$
- $\|MSE_{k+1} - MSE_k\|_\infty \leq$ Some tolerance, $\epsilon$
- $k \geq N$, where $N$ being the maximum allowed iterations

## 2.4 Nearest Neighbour Classifier

Another type of classifier that will be tested in the tasks later on is the nearest neighbour classifier or NN classifier for short. This is another type of classifier known as a template based classifier ([1], section 2.5). When a new sample is to be classified the sample is compared to a set of templates, which in this case simply is labeled samples. The decision rule for the NN classifier is then to set the predicted class of the new sample equal to the label of the nearest template, i.e. the template that is the most similar to the new sample.

The first question is then how to calculate the distance between a sample and a template. Different metrics can be used for this, with each of their own properties. Two of the most common metrics are the Euclidian distance (9) and the Mahalanobis distance (10) ([1], section 2.5):

$$d_E^2(x_1, x_2) = (x_1 - x_2)^T (x_1 - x_2) = ||x_1 - x_2||^2 \tag{9}$$
$$d_M^2(x_1, x_2) = (x_1 - x_2)^T C^{-1} (x_1 - x_2) = ||x_1 - x_2||^2 \tag{10}$$

The Euclidian distance is the simplest one and it is this metric that is used in the tasks later on. The Mahalanobis distance uses the covariance matrix of the samples, $C$, to weight directions differently. One important difference between the two metrics is therefore that the Mahalanobis distance requires a probability distribution for the samples in order to calculate the covariance matrix. This often requires a lot of data samples to be able to estimate the stochastic parameters correctly. Note that the Mahalanobis distance equals the Euclidian one if the covariance matrix of the samples equals the unit matrix. Using the Euclidian distance for classification therefore brings with it the assumption that the data samples are perfectly uncorrelated.

An extension of the NN classifier is the KNN classifier. Instead of finding the single nearest template for each sample, the KNN classifier finds the K nearest templates, or K nearest neighbours to each sample. The main decision rule for the KNN classifier is that the predicted class of the sample is set equal to the most frequent label among the KNNs ([1], section 2.5). However, different decision rules exist when multiple classes are the most frequent one among the KNNs. This will be further discussed in section 4.2 about implementation.

## 2.5 Clustering

One way to look at a NN classifier is that each template sample defines a small decision region around it in the feature space. The size of this decision region will depend on how far it is to the nearest other templates. When multiple templates from the same class is next to each other in the feature space the decision regions surrounding these templates are joined in one decision region that covers both templates and is the union of the previous two decision regions. When the number of templates is increased the size of each little decision region decreases as the templates lie closer and closer together in the feature space. The overall goal and trend is then that the division of the feature space into decision regions more and more reflects the actual differences between the different phenomena that are to be classified. I.e. the larger number of templates the classifier uses the more detailed these decision regions will become. This will more correctly separate the real-world phenomena and give better classification results, given that the templates are a representative selection of samples from the phenomena at hand.

Using representative templates is therefore very important for the performance of the template based classifiers. As stated in the classification compendium, section 3.3: "Training of a template based classifier is the same as finding good references. The simplest approach is not to train at all but instead to use the whole training set as templates. The main objection with this is the amount of processing, i.e. one has to calculate N distances for every new input x" ([1], section 3.3). So the goal is to use few templates to reduce computation time, but keep the templates as representative as possible for the actual phenomena to avoid having faulty decision regions. This is the purpose of clustering. Clustering consists of creating new templates based on the "averages" of other templates. The clustering is normally performed on templates from the same class. Thus the number of templates is reduced, but the key features that makes the templates correspond to the same class are preserved as best as possible.

## 2.6 Confusion matrix

To be able to describe the performance of a classifier, one may create a table to illustrate how many times the classifier predicted the sample correctly in terms of what the actual sample is labeled as. This is the purpose of the confusion matrix, which looks like this:

| Predicted class / Actual class | Class 1 | Class 2 | ... | Class C |
|---|---|---|---|---|
| Class 1 | ✓ | X | ... | X |
| Class 2 | X | ✓ | ... | X |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Class C | X | X | ... | ✓ |

Figure 1: General confusion matrix: Each element (i,j) specifies the number of times the classifier predicted class j and the label was i. The confusion matrix of a perfect classifier is therefore a diagonal matrix.

where the dimensions of the confusion matrix is quadratic in terms with how many classes which are being classified.

The error rate is another much used attribute in evaluating classifiers, which can be calculated from the confusion matrix. The error rate states how many times the classifier failed to predict the actual class of the test sample compared to the number of test samples and is often given in percent.

# 3 Classifying Iris Flowers

## 3.1 Task

This task is about classifying three different Iris flowers (Setosa, Versicolor and Virginica) which is discriminated by the different lengths and widths of the sepal and petal leaves. The first part of the task is to design and evaluate a linear discriminant classifier. The second part has focus on features and linear separability, and the relative importance of each feature with respect to the linear separability of the different Iris classes.

### Data sets

The database (often called the "Fisher Iris data") consist of 50 samples of each Iris variant (150 in total). Each sample consist of four features and the class they belong to i.e. [Sepal length (cm), Sepal width (cm), Petal length (cm), Petal width, label]. More information could be found at https://en.wikipedia.org/wiki/Iris_flower_data_set [5].

### Task 1 - Design & training

In this task a linear discriminant classifier, as described in section 2.2, should be implemented, where the step factor $\alpha$ should be tuned in such that the training converges[1]. The first 30 samples in each class should be used for training and the remaining 20 samples in each class should be used for testing/evaluation. After getting a well tuned classifier, one should test the classifier with both the training- and test sample sets and from these find the confusion matrices (see section ) and error rates. The remaining of this task is to switch the order of samples used for training and testing. Now the last 30 samples is used for training and the first 20 samples for testing. Likewise, the classifier should evaluate both the training samples and test samples, and from these find the confusion matrices and error rates. At last, compare the confusion matrices and error rates given in the two cases.

### Task 2 - Features & Linear Separability

In the second part one should produce histograms for each feature and class, and analyse how the different features differ in terms of their respective class and as a whole. Next, one should take away the feature that correlate the most throughout the whole data set and train the classifier with the remaining features. Again, the first 30 samples should be used for training and the last 20 samples for tests. Evaluate the classifier by finding the confusion matrix and error rate for the test samples. Reduce the feature space once again by taking away the most correlated feature of the remaining three, and find the confusion matrix and error rate for the test samples. Do this one last time such that the only feature remaining is the least correlated feature, and find the confusion matrix and error rate for the test samples applied to the classifier. By comparing the different evaluations for the different feature spaces, one may see how the linear separability property, discussed in section 2.1, affects the feature reduction.

[1]With a rather good performance

## 3.2 Implementation

This subsection describes how the solutions of the tasks in section 3.1 were implemented. Firstly, the implementation of the design and the training of the linear classifier is presented, as being the first part of the Iris task. Following this is how the the different confusion matrices where calculated, and how the data sets were switched before training and testing the classifier again. Secondly, the same is done when using samples with reduced number of features, where histograms are used to analyse how the different features and classes coincide.

### 3.2.1 Design & Training

The implementation setup was heavily based on creating a labeled training set that could be used to train the classifier. To better utilize this in Matlab, the whole Fisher Iris data set was formatted from a .data file to a comma separated variable file (.csv). Loading this csv-file into Matlab was done by using the readtable('iris.csv') function, which made a 150x5 matrix where the five columns represented the four features, that represents the sample of an Iris, and the label of the Iris sample. The Iris-matrix was divided into three training sets, where each set contained the first 30 samples of one designated class. These training sets was then used to train the linear classifier. To be able to train the discriminant classifier with the MSE approach one must first generate the target vector for each training sample, as described in section 2.3. A cell array was then used to attach each target vector to its designated sample. Appendix A shows in detail how the training data was formatted.

Training the classifier then boils down to tuning the weighting matrix $W \in \mathbb{R}^{3X(4+1)}$, where W has been transformed such that the discriminant function, g, is at a homogeneous form. The training of the classifier is done by applying MSE and gradient descent which was described in section 2.3. The calculation of the MSE, and its gradient, is done by extracting each sample from the training set and applying the equations 4 and 8. The gradient descent was implemented as a "steepest descent" algorithm [3], where the MSE was used as the objective function. The essential part was then to have a good enough step length $\alpha$ such that the classifier converges with good enough performance i.e. it manages to classify most of the Iris' correctly. $\alpha$ was found by simple trial and error until the result did no longer improve. The fully tuned weighting matrix W was found when the steepest descent algorithm terminated, where the last iterated 3x5 matrix was the optimum matrix for the classifier. The implemented code can be seen in appendix C.

The confusion matrix was found by comparing the label of each test sample with the predicted class for that sample. The confusion matrix was initialized as a zero-matrix and for each test sample the corresponding element in the matrix was incremented. When fully evaluated, the error rate could be found by dividing the trace of the confusion matrix by the number of tests done, and then subtract this number from 1.

The final part of the task was then to change the training set to the last 30 samples and the test set to the 20 first. The implementation was equal to the previous part of the task, with minor changes to the initialization of some parameters. The diagram 2 is illustrating how the task flows.

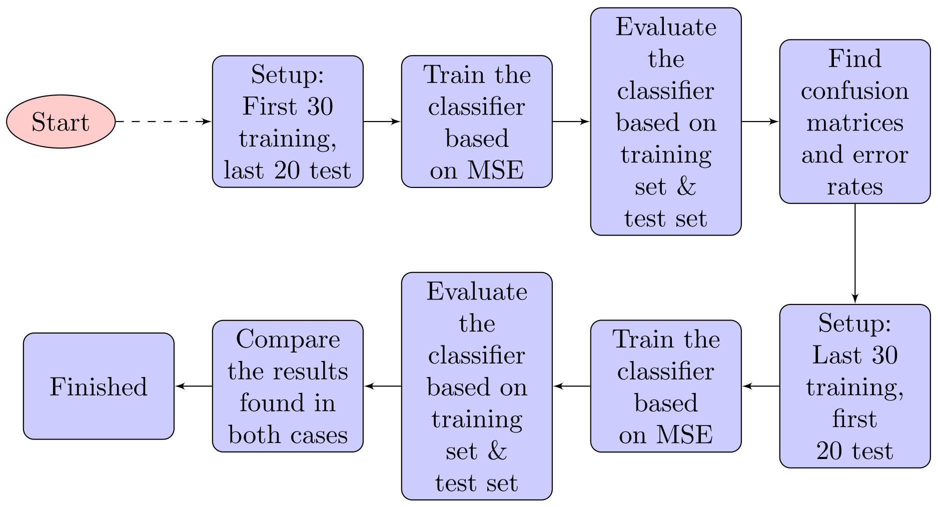

Start
Setup: First 30 training, last 20 test
Train the classifier based on MSE
Evaluate the classifier based on training set & test set
Find confusion matrices and error rates
Setup: Last 30 training, first 20 test
Train the classifier based on MSE
Evaluate the classifier based on training set & test set
Compare the results found in both cases
Finished

Figure 2: Diagram describing the flow of Iris subtask 1.

### 3.2.2 Features & Linear Separability

The second part of the Iris task was started by using the same Iris data divided into three class-separated sets. These was used to produce histograms for the different iris classes and for the different features, using the built-in Matlab function *histogram*. These histograms will be discussed further in the next section. The most overlapping feature was found and taken out from the feature space such that the classifier is trained, in the same matter as the previous task, with a reduced feature space. The confusion matrices and error rates was found as before, and this procedure was done until only one feature remained in the feature space - The least overlapping feature. The change done in regards to the previous task was primarily to pass the column indices of interest, namely the remaining features for each iteration, into the function get_training_data (see function in appendix B), such that the training data only contains the feature of interest. Diagram 3 illustrates how the flow of the algorithm goes. Code implemented is shown in the appendix B. Now, it is possible to see how the linear classifier manages to classify the different types of iris' based on the resulting confusion matrices and error rates.

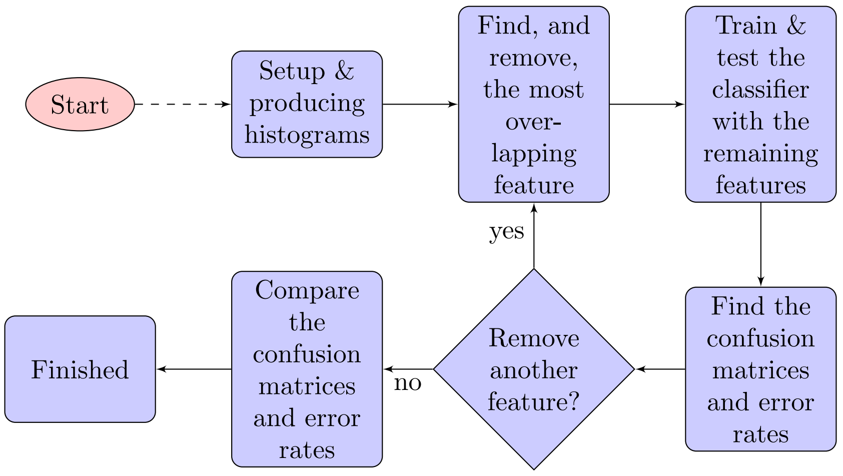

Start
Setup & producing histograms
Find, and remove, the most over-lapping feature
Train & test the classifier with the remaining features
Find the confusion matrices and error rates
Remove another feature?
yes
no
Compare the confusion matrices and error rates
Finished

Figure 3: Diagram describing the flow of Iris subtask 2.

## 3.3 Results

This section presents and discusses the obtained results from the solutions given in section 3.2 to the tasks given is section 3.1. The results are first presented for task 1 and secondly for task 2. In the end, the results are compared and the differences are discussed.

### 3.3.1 Design & Training

The optimal $\alpha$ in the gradient descent was found to be 0.00585. This yields then the following confusion matrices and error rates, where table 1 is the results when training the classifier with the first 30 samples of each class, and table 2 is the results when training the classifier with the last 30 samples of each class.

Testing set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 20 | 0 | 0 |
| Versicolor | 0 | 15 | 5 |
| Virginica | 0 | 0 | 20 |
| Error rate | 8.33% | | |

Training set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 30 | 0 | 0 |
| Versicolor | 0 | 21 | 9 |
| Virginica | 0 | 0 | 30 |
| Error rate | 10% | | |

Table 1: Confusion matrices and error rates for both testing and training evaluation. The linear classifier is trained with the first 30 samples of each Iris-class. Features used: [Sepal length, Sepal width, Petal length, Petal width]

Testing set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 20 | 0 | 0 |
| Versicolor | 0 | 12 | 8 |
| Virginica | 0 | 0 | 20 |
| Error rate | 13.33% | | |

Training set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 30 | 0 | 0 |
| Versicolor | 0 | 20 | 10 |
| Virginica | 0 | 0 | 30 |
| Error rate | 11.11% | | |

Table 2: Confusion matrices and error rates for both testing and training evaluation. The linear classifier is trained with the last 30 samples of each Iris-class. Features used: [Sepal length, Sepal width, Petal length, Petal width]

### 3.3.2 Features & Linear Separability

The following histograms was produced to analyse the features of the iris'. The first histogram (figure 4) shows the features distributed in each class (row 2, 3 and 4) and all classes combined (row 1). The second histogram (figure 5) shows how the different features is distributed throughout the whole data set, coloured by which class the feature is labeled as.

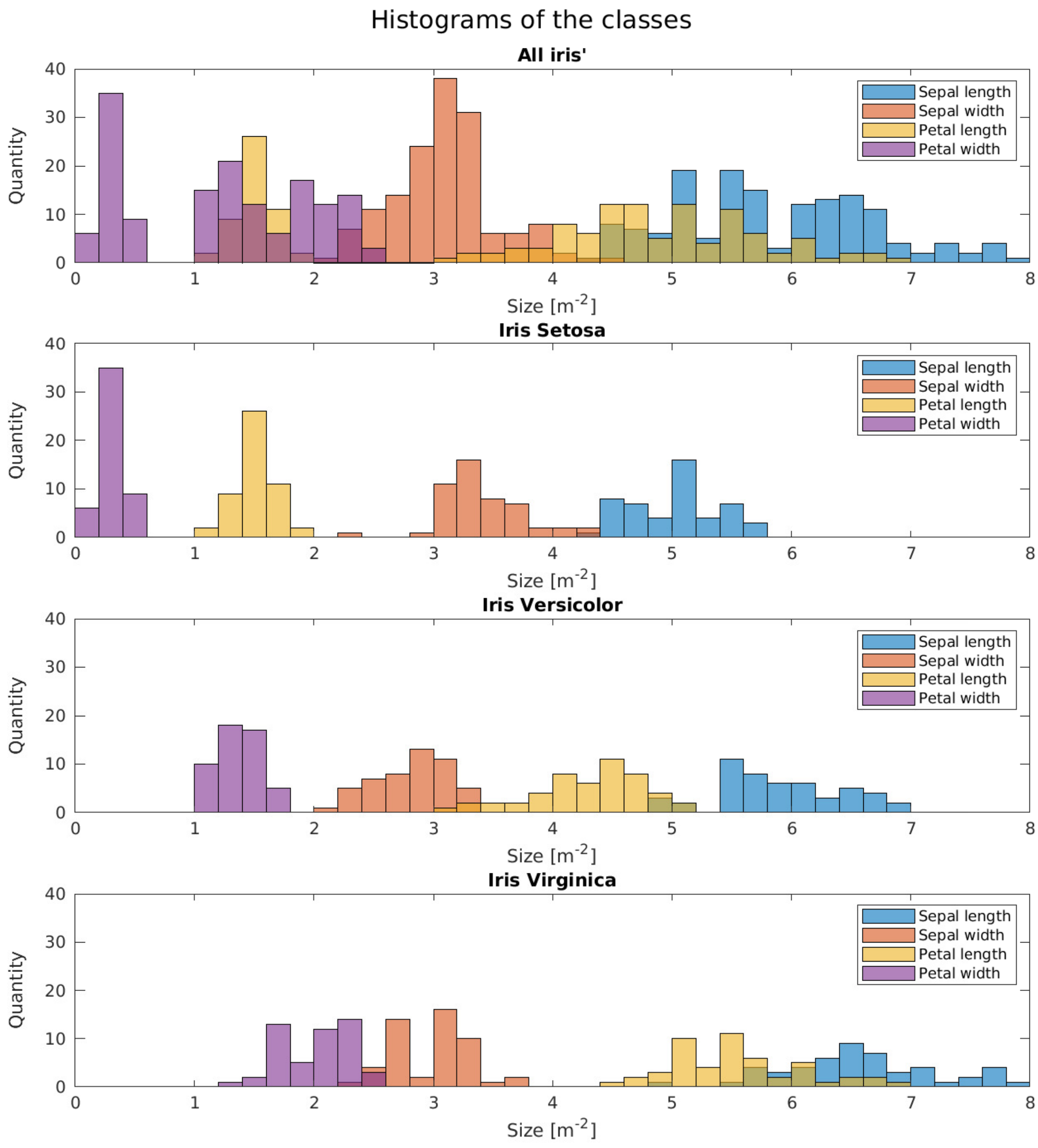


Figure 4: Histograms of each class

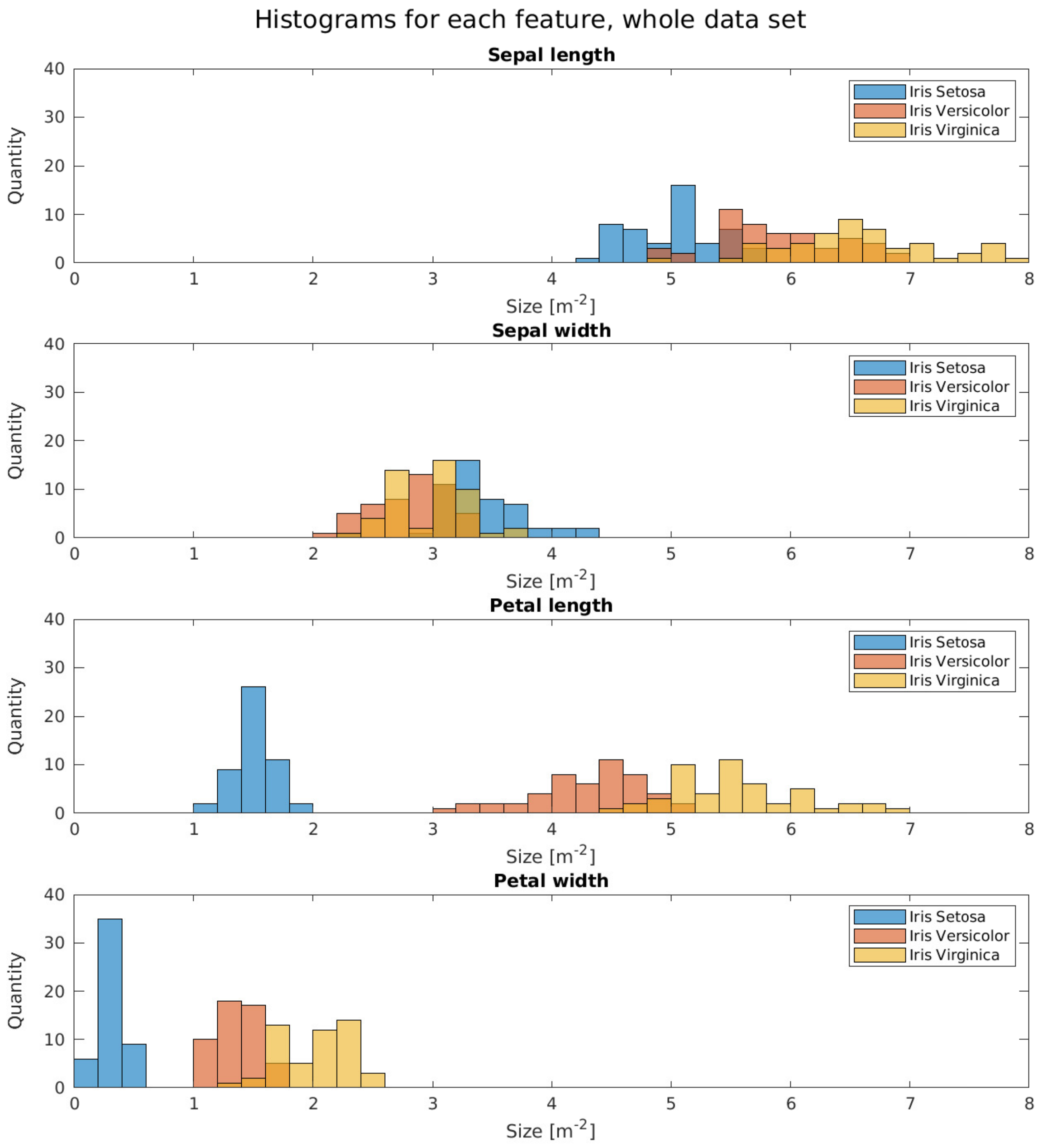


Figure 5: Histograms of each feature

The same tuned in $\alpha$ was used ($\alpha = 0.00585$) in the gradient descend, when training the classifier. This yields the following confusion matrices and error rates, where table 3 is the results from three features, table 4 is the results from two features, and table 5 is the results from one feature.

Testing set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 20 | 0 | 0 |
| Versicolor | 0 | 11 | 9 |
| Virginica | 0 | 0 | 20 |
| Error rate | 15% | | |

Training set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 30 | 0 | 0 |
| Versicolor | 0 | 17 | 13 |
| Virginica | 0 | 0 | 30 |
| Error rate | 14.44% | | |

Table 3: Confusion matrices and error rates for both testing and training evaluation. The linear classifier is trained with the first 30 samples of each Iris-class. Features used: [Sepal length, Petal length, Petal width]

Testing set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 20 | 0 | 0 |
| Versicolor | 0 | 0 | 20 |
| Virginica | 0 | 0 | 20 |
| Error rate | 33.33% | | |

Training set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 30 | 0 | 0 |
| Versicolor | 0 | 0 | 30 |
| Virginica | 0 | 0 | 30 |
| Error rate | 33.33% | | |

Table 4: Confusion matrices and error rates for both testing and training evaluation. The linear classifier is trained with the first 30 samples of each Iris-class. Features used: [Petal length, Petal width]

Testing set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 20 | 0 | 0 |
| Versicolor | 0 | 0 | 20 |
| Virginica | 0 | 0 | 20 |
| Error rate | 33.33% | | |

Training set used for testing the classifier

| Predicted Iris / Actual Iris | Setosa | Versicolor | Virginica |
|---|---|---|---|
| Setosa | 30 | 0 | 0 |
| Versicolor | 0 | 0 | 30 |
| Virginica | 0 | 0 | 30 |
| Error rate | 33.33% | | |

Table 5: Confusion matrices and error rates for both testing and training evaluation. The linear classifier is trained with the first 30 samples of each Iris-class. Features used: [Petal length]

### 3.3.3 Discussion

By comparing the error rates, found from the confusion matrices in tables 1 and 2 one could observe that when using the testing set to evaluate the classifier then the error rate gets five percent worse performance when switching the training set from the 30 first samples to the 30 last samples. In the case when the training set is used to evaluate the classifier, the performance got a little bit worse (1.11% worse). This could be because the last 20 samples of each class doesn't represent the respected classes in the best way (Where the best way could be having more samples of the average type). This then illustrates the importance of having enough representative samples of the given class, or at least have the majority of the expected values of the features. Another interesting behaviour is that in figure 1 the trained classifier is having a harder time classifying the samples from the training set than the testing set. An important observation is that the only time the classifier is making an error is when the sample is classified as Virginica when it is labeled as Versicolor. This could be reasoned with that the two classes may be more alike than Setosa is to any of them.

Figure 5 shows how each feature is related to eachother in terms of class. One could see that the Setosa is clearly separated on the petal features. When looking at the two other classes one may observe that they are much more coinciding, and would make it harder for the classifier to determine which class the sample represent. Based on this, there could be of interest to see how well the classifier would evaluate by reducing the feature space by taking away the most overlapping feature.

The selection was based on which feature that had the most overlapping and the least variance/spread of size. The order of the feature that was taken away was sepal width, sepal length and then petal width, which lead to the results in table 3, 4 and 5 respectively.

By comparing the four error rates in table 1, 3, 4 and 5 one could see that the performance is worsened for each feature reduction, and ending up at an error rate of 33.33%. An interesting behaviour of the classifier is that after reducing the feature space to only contain petal length and petal width (table 4 and 5), the error rate doesn't get worse for the next feature reduction. By observing their confusion matrices, one could see in both cases that all the Versicolor samples is classified as Virginica, but all the Setosa samples are classified as Setosa. This indicates that Iris Setosa is linearly separable from the two other classes, by only looking at this single feature. The fact that the classifier does not manage to separate the two other classes implies that they are not linearly separable from each other based on this single feature only. This could also be seen in the third row in figure 5, where the Setosa petal length is clearly separated from Versicolor and Virginica, but the two latter coincide a bit. This was also the case when using all four features, where the classifier didn't manage to separate all the Versicolor from the Virginica. By looking again at figure 5 this makes sense as the two classes always has a bit of overlapping features.

The second, third and forth row in figure 4 illustrates how the different features differ in size, isolated for each class. For Iris Setosa, one could see that the petal length and petal width are completely separable from each other and the other features, while the sepal length and sepal width are overlapping a bit. For Iris Versicolor one could see that petal width is fully separable, while sepal length, sepal width and petal length are overlapping a bit. For Iris Virginica one could see that sepal width and petal width are almost fully separable, while sepal length and petal length are coinciding for the most part. This makes it reasonable to believe that the features within Setosa and Versicolor to be, isolated, linearly separable, while there is less believable to think that the features in Virginica is linearly separable.

# 4 Classifying Handwritten Numbers

This section of the report will discuss classification of handwritten numbers. Firstly, section 4.1 will describe the task in detail. Secondly, section 4.2 explains the chosen implementation of the classifier based on the theory in section 2.4 and 2.5, before section 4.3 presents and discusses the obtained results.

## 4.1 Task

This task is about classifying pictures of handwritten numbers $0-9$. The first part of the task uses a nearest neighbour (NN) classifier to classify the numbers. The second part applies clustering on the training set before both NN and KNN classifiers are evaluated with the clustered data.

#### Data sets

For both training and test data the MNIST database created by National Institute of Standards and Technology, NIST, in the U.S. was used. The database consists of 70000 labeled samples of hand-written numbers, 60000 training samples and 10000 test samples, each set written by 250 different persons. Half of the samples in each set is written by high-school students while the other half is written by Census Bureau employees. These data sets are preprocessed from other NIST databases to prepare them for classification. As stated by the database website: "The original black and white images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels (8-bit pixel values between $0-255$) as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field" [2]. Additional information about the MNIST database can be found at http://yann.lecun.com/exdb/mnist/

#### Task 1 - Nearest Neighbour Classifier

In this task a nearest neighbour based classifier using the Euclidian distance, as described in section 2.4, was implemented to classify the numbers. The whole training set was used as templates for finding the nearest neighbour for each test sample. To evaluate the classifier the confusion matrix and the error rate is calculated and discussed upon. Also, some of the correctly and incorrectly classified pictures are displayed to better visualize the discussion.

#### Task 2 - Clustering and KNN Classifier

In the second task clustering is used to produce a smaller set of templates for each class. The first part of the task the templates consisted of roughly 6000 samples from each class. The training data is then clustered into 64 clusters for each class. Then, both a NN and KNN classifier with $K = 7$ is tested on the reduced set of templates. Again, to evaluate the classifiers, and also evaluate the computation times, confusion matrices and error rates are computed and compared. Some of the clustered templates are also displayed to better illustrate the effects of clustering.

## 4.2 Implementation

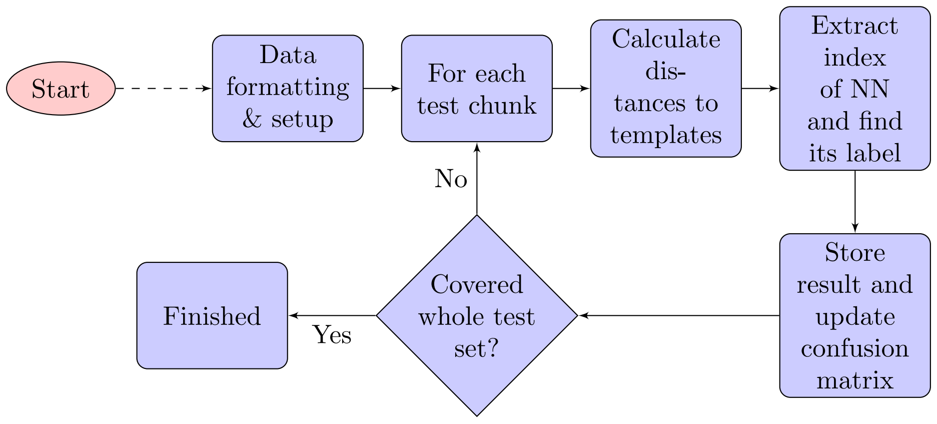
This subsection describes in detail how the solutions to the tasks in section 4.1 were implemented. First, the setup and implementation of the NN classifier from task 1 is presented. Then the similar is done for task 2 about clustering and a KNN classifier. The implementations are based on the theory in section 2.4 and 2.5.

### 4.2.1 Nearest Neighbour Classifier

To better prepare the data from the MNIST database for usage in Matlab, the data was reformatted into a Matlab data file using the script in appendix A which was handed out together with the project tasks. The produced data file can be loaded into the Matlab workspace which gives four main variables, two matrices which hold the test and training samples and two vectors which hold the test and training labels. The samples are stored as 1x784 image vectors, where each element is the grey-scale value of each pixel in the sample, with one image vector per row in the matrices.

The Matlab code which implements the classifier and calculates its confusion matrix and error rate can be seen in appendix B. As seen in the script the function $dist(template, test\_sample)$, from the Deep Learning Toolbox, was used to calculate the Euclidean distance from each test sample to each template. The task stated that in order to reduce computation time it is desirable to use a matrix of test samples instead of just a single test sample in each *dist*-call, thus calculating the distance to all templates for several test samples at once. On the other hand, using all training samples in one single function call will result in a very large distance matrix, 60000x10000 of doubles, which takes $4.8 * 10^9$ bytes $\approx$ 4.5GB of memory. It is therefore preferable to split the test data into chunks of samples to avoid using too much memory, but still reducing excess computation time. In the task, chunks of 1000 test samples were used in each *dist*-call.

When the distance from the 1000 test samples to all the templates were calculated, the *min*-function was used to find the index of the template, which then represent the nearest neighbour to each sample. This set of NN-indices was then used to extract the labels of these nearest neighbours, which then again equals the estimated classes of the corresponding test samples. To be able to later display some correctly and incorrectly classified numbers, the estimated class of each sample was stored and the confusion matrix was updated. This process was then repeated for each chunk of test samples and can be summarized with the following flow-diagram:


Start
Data formatting & setup
For each test chunk
Calculate dis-tances to templates
Extract index of NN and find its label
Store result and update confusion matrix
Covered whole test set?
No
Yes
Finished

### 4.2.2 Clustering and KNN Classifier

The Matlab code which performs the clustering of the training set, implements the NN and KNN classifiers and calculates their confusion matrices and error rates can be seen in appendix C.

Firstly, it could be noted that the matrix that holds the training data, generated by the reformatting script in appendix A, is not sorted, in the sense that all the samples from one class do not come one after the other. Also, the 64 clusters that are to be created for each class are meant to be a clustering of the whole training set for that class. Therefore, the training set must be sorted by class before it could be clustered. This was done by creating a 2 dimensional cell array holding all the training vectors in column 1 and the corresponding labels in column 2. The rows of the cell array was then sorted with the *sortrows* function, sorting on the label value in column 2. The sorted training vectors were then extracted and converted back to a normal 60000x784 matrix. In order to perform the clustering all the training samples for each class must be extracted from this matrix, however, the number of samples from each class is not equal. To avoid using a bunch of if-sentences to sort out the samples from each class, the number of samples from each class was counted and the start index of each class was calculated.

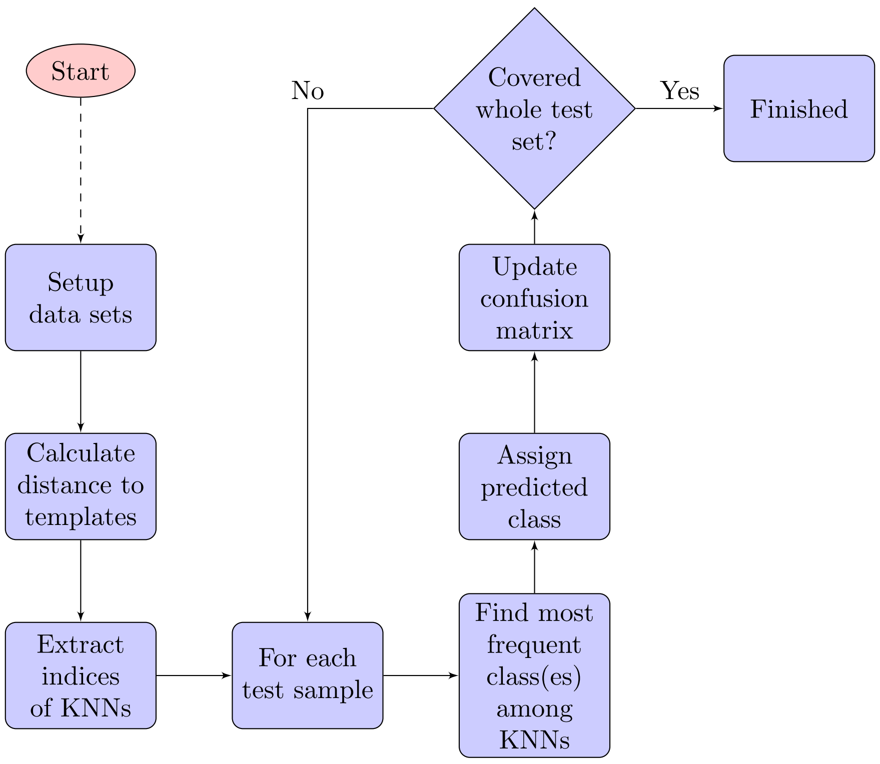
For creating the clusters for each class the function $kmeans(class\_samples, num\_clusters)$ from the Statistics and Machine Learning Toolbox was used. The new clusters were then inserted directly into the new training set. The NN classifier was implemented exactly as for the previous task, but this time it used the clustered training set as templates and the entire test set in one run without splitting it into chunks. To implement the KNN classifier the $mink(dist\_matrix, K)$-function was used instead of the $min(dist\_matrix)$-function, with $K = 7$, to extract the indices of the $K$ nearest templates to each test sample. The labels of these templates were extracted and the estimated class of each test sample was set to the most frequent class of the $K$ nearest templates. For doing so, three different implementations were tested. The difference between the methods lies in which class the algorithm chooses when there are several most frequent classes among the NNs:

The first method simply uses the *mode*-function to extract the most frequent class of the NNs. This is definitely the easiest option as it consists of one simple line of code. However, when there are several most frequent classes among the NNs the *mode*-function returns the lowest categorized class of the most frequent classes. This means that in these cases the classifier would have a small bias towards classifying the samples as lower numbers, which there is no logical reason for doing.

The second method implements the decision rule that when there are several most frequent classes among the NNs, the most frequent class with the lowest total distance to the test sample should be chosen. This is a much more logical method than the first one and was implemented by counting the number of occurrences of each class among the NNs. Further, when a NN from class X was counted the distance from that NN to the test sample was added to the total distance between class X and the sample, making it easy to extract the class with the lowest distance when needed.

The third method implements a very similar decision rule as the second one: When there are several most frequent classes among the NNs, the most frequent class with the lowest single NN distance to the test sample should be chosen. Naturally, this implementation does not differ much from method number two. Simply, instead of adding the distance between each NN and the test sample to the total class distance, the class distance is replaced by the NN distance if the NN distance is lower than the class distance.

All three methods are implemented in the script in appendix C. The algorithm for the KNN classifier can be summarized with the following flow-diagram, where the differences between the three methods are in the first two steps of the loop:


Start
Setup data sets
Calculate distance to templates
Extract indices of KNNs
For each test sample
Find most frequent class(es) among KNNs
Assign predicted class
Update confusion matrix
Covered whole test set?
No
Yes
Finished

## 4.3 Results

This section presents and discusses the obtained results from the solutions given in section 4.2 to the tasks given is section 4.1. The results are first presented and discussed for task 1 and secondly for task 2. In the end, the results are compared and the differences between the classifiers are discussed.

### 4.3.1 Nearest Neighbour Classifier

When the NN classifier was run with the entire training set as templates and tested with the entire test set it produced the following confusion matrix and error rate:

| Predicted class Label | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 973 | 1 | 1 | 0 | 0 | 1 | 3 | 1 | 0 | 0 |
| 1 | 0 | 1129 | 3 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 7 | 6 | 992 | 5 | 1 | 0 | 2 | 16 | 3 | 0 |
| 3 | 0 | 1 | 2 | 970 | 1 | 19 | 0 | 7 | 7 | 3 |
| 4 | 0 | 7 | 0 | 0 | 944 | 0 | 3 | 5 | 1 | 22 |
| 5 | 1 | 1 | 0 | 12 | 2 | 860 | 5 | 1 | 6 | 4 |
| 6 | 4 | 2 | 0 | 0 | 3 | 5 | 944 | 0 | 0 | 0 |
| 7 | 0 | 14 | 6 | 2 | 4 | 0 | 0 | 992 | 0 | 10 |
| 8 | 6 | 1 | 3 | 14 | 5 | 13 | 3 | 4 | 920 | 5 |
| 9 | 2 | 5 | 1 | 6 | 10 | 5 | 1 | 11 | 1 | 967 |
| Error rate | 3.09% | | | | | | | | | |

Table 6: Confusion matrix and error rate for the NN classifier with the entire training set as templates, tested on the whole training set.

The confusion matrix shows that the classifier performs very well, with a quite low error rate. This is very likely due to the large set of templates that is used, which contains all the 60000 training samples. The table also shows that the most common misclassification is classifying the number 4 as 9. This is not very surprising as when written by hand these number can look very similar. Especially if the 4 is written with a diagonal line on the top which closes the figure, like in the upper right picture of figure 6.

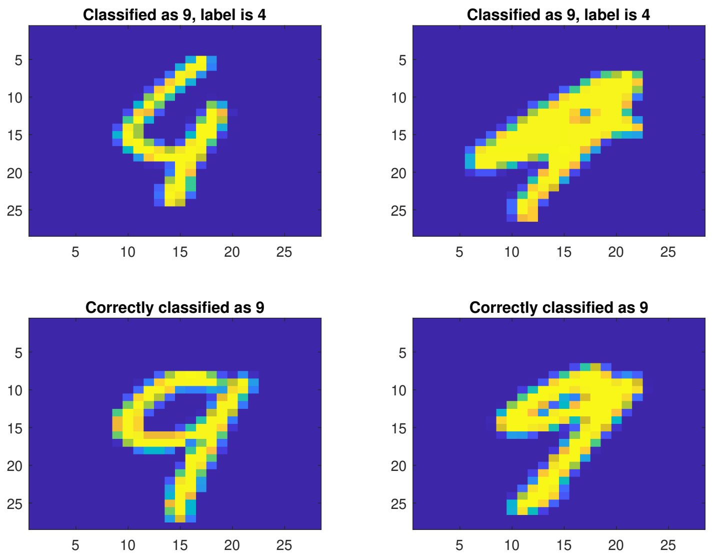

Classified as 9, label is 4
Classified as 9, label is 4
Correctly classified as 9
Correctly classified as 9
5
10
15
20
25

Figure 6: Some correctly and incorrectly classified numbers, illustrating the similarities between 4 and 9. The displayed samples have indices 13, 21, 116 and 448 respectively in the test set matrix, counted from left to right starting at the top.

Figure 6 further illustrates the similarities between the numbers 4 and 9. Even though a human would probably classify the top right image as 4 it is easily mistaken for 9. It is a bit more surprising that the top left image was misclassified, as it clearly resembles 4. However, also this image has clear resemblances to the number 9, like in the image to the bottom left, which makes it more difficult for the classifier to predict the correct class.

Further, figure 7 shows some chosen samples that most humans would probably agree are difficult to classify. Yet, the NN classifier has predicted the correct class. Figure 8 on the other hand, shows some chosen samples that were misclassified. Looking at the upper images in the figure, it can be understandable that the classifier misclassified these numbers, while the bottom images should be seemingly easy to classify based on the human intuition. One aspect that separates the difficult samples in figure 7 from the difficult samples in figure 8 is that the ones in figure 7 are hard because they, to some extent, don't resemble any number at all. While the ones in figure 8 are hard because the image could resemble two numbers at once. This illustrates that the classifier works quite well on ill-drawn numbers as long as the image does not start to resemble multiple numbers.

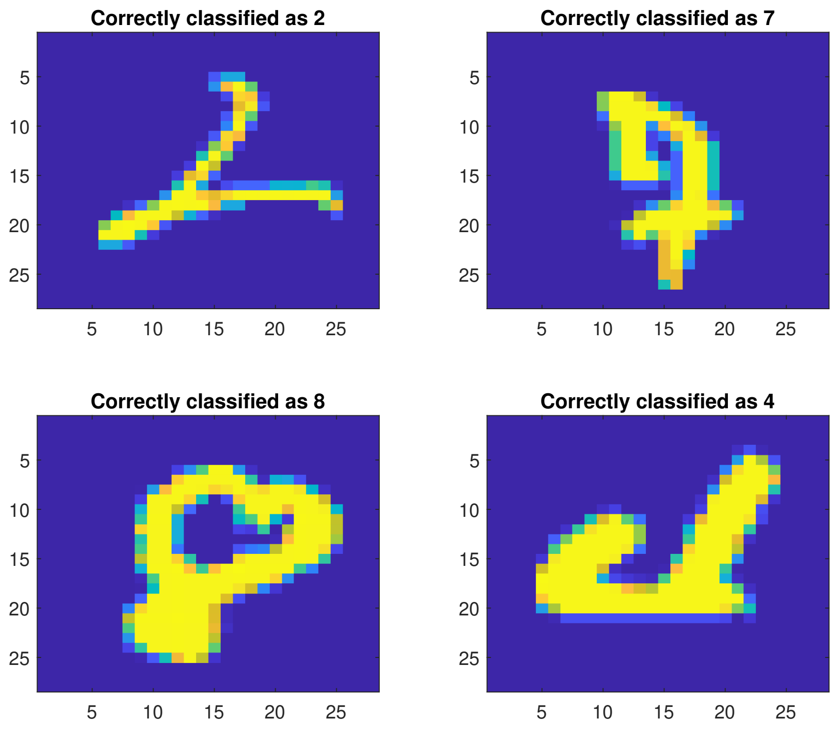


Figure 7: Some chosen difficult test samples that were correctly classified. The displayed samples have indices 44, 4008, 8000 and 8001 respectively in the test set matrix, counted from left to right starting at the top.

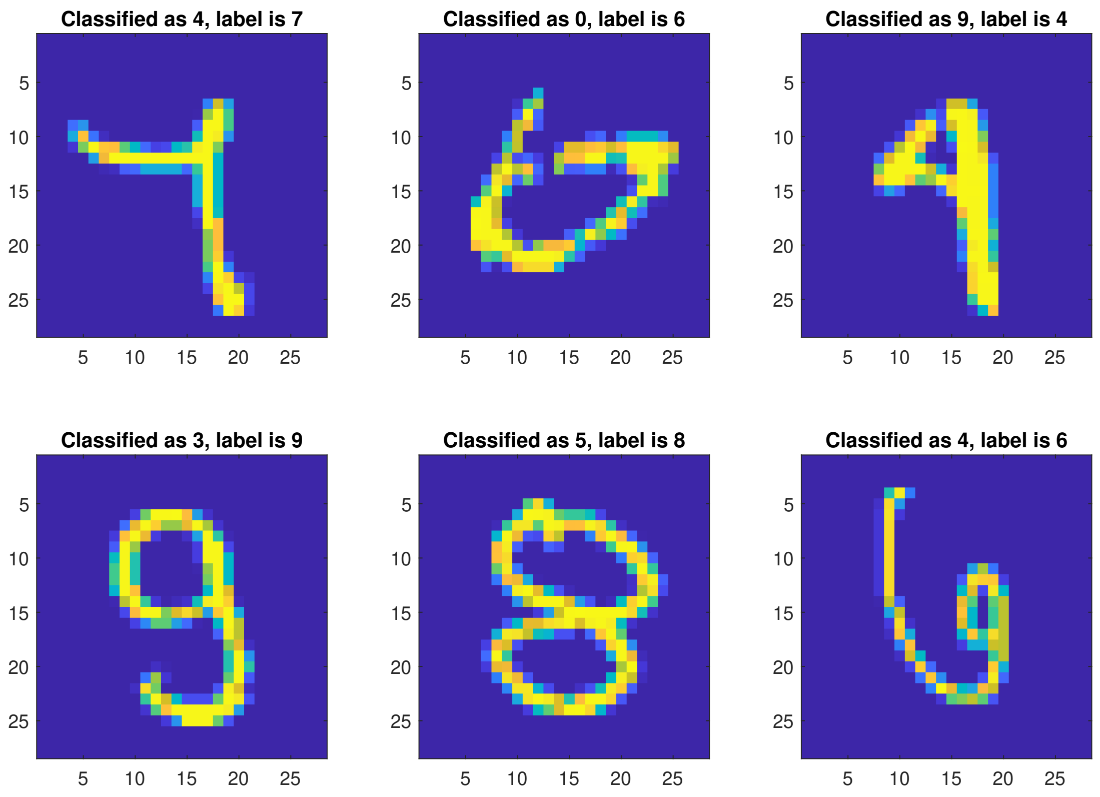


Figure 8: Some chosen misclassified samples. The samples at the top can be mistaken by most humans, while people will easily recognize the images at the bottom. The displayed samples have indices 359, 446, 741, 480, 269 and 342 respectively in the test set matrix, counted from left to right starting at the top.

The misclassification of the bottom images in figure 8 are a bit harder to explain. It is somewhat understandable that the leftmost sample is misclassified as you only have to remove parts of the upper circle of the 9er to make it look like 3. So if this part of the image is more weakly drawn, as may be the case here, the image could more easily be misclassified as 3. However, how the 8 and 6 could become 5 and 4 is a bit harder to grasp.

One method to improve the performance of the classifier could perhaps be to change the way the images are preprocessed. As stated on the MNIST website: "With some classification methods (particuarly template-based methods, such as SVM and K-nearest neighbours), the error rate improves when the digits are centered by bounding box rather than center of mass" [2]. However, as indicated by the error rate in table 6, the NN classifier performs quite well when it uses the entire test set of 60000 samples as templates. The drawback of this implementation is that it becomes quite computationally heavy to calculate the distance from each sample to all the templates, and thus it takes a long time to classify the samples. The task recommended to gather the test data into chunks, and not calculate the distance to each template for only one sample at a time, to reduce computation time. To test this the classifier was run with different chunk sizes on the 10000 test samples and the computation time was measured. The results are summarized in table 7:

| Chunk size | Computation time |
|---|---|
| 1 (single samples) | 35min 21s |
| 1000 (10 chunks) | 34min |
| 10000 (1 chunk) | 32min 52s |

Table 7: Computation time for the NN classifier when using different chunk sizes on the test data. The test data consists of 10000 samples and all the 60000 training samples are used as templates.

The table shows that the computation time is reduced with increasing chunk size, as expected. However, the effect is marginal as the classification takes a long time in either case. It is here important to remember that large chunk sizes uses a lot of memory, which in some cases pose a larger issue that computation time. Nonetheless, this is why the next task discusses the effects of clustering the training data to reduce the number of templates, and thus the computation time.

### 4.3.2 Clustering and KKN Classifier

After clustering the training samples into 64 clusters per class the training set was reduced from 60000 samples to 640 clustered samples. When the NN classifier was run using the clustered samples as templates and tested with the entire test set it provided the following confusion matrix and error rate:

| Label \ Predicted class | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 963 | 1 | 4 | 0 | 0 | 4 | 3 | 1 | 3 | 1 |
| 1 | 0 | 1129 | 2 | 0 | 0 | 0 | 3 | 0 | 0 | 1 |
| 2 | 8 | 7 | 978 | 7 | 2 | 0 | 2 | 13 | 14 | 1 |
| 3 | 1 | 0 | 4 | 942 | 0 | 21 | 0 | 11 | 23 | 8 |
| 4 | 0 | 6 | 2 | 0 | 921 | 0 | 9 | 5 | 2 | 37 |
| 5 | 6 | 0 | 0 | 13 | 2 | 852 | 10 | 1 | 3 | 5 |
| 6 | 9 | 3 | 1 | 0 | 6 | 4 | 934 | 0 | 1 | 0 |
| 7 | 0 | 16 | 12 | 1 | 9 | 0 | 0 | 957 | 1 | 32 |
| 8 | 4 | 1 | 5 | 12 | 5 | 21 | 1 | 7 | 913 | 5 |
| 9 | 4 | 4 | 5 | 8 | 23 | 8 | 1 | 27 | 5 | 924 |
| Error rate | 4.87% | | | | | | | | | |

Table 8: Confusion matrix and error rate for the NN classifier with the clustered training set as templates, tested on the whole training set. The clustered training set consists of 64 clusters per class.

Table 8 shows that the error rate increases when using clustered training data. This is expected as the classifier now has fewer templates to compare each test sample with. With fewer templates the classifier may no longer recognizes as many variations and characteristic features of each class, which leads to poorer classification. This can be seen in figure 9 as the numbers being more blurred and diffuse than in the original training samples. The confusion matrices in tables 6 and 8 also show that two of the largest differences between using the clustered and unclustered training data is that when the clustered templates were used the numbers 9 and 4 as well as 9 and 7 got mixed up a lot more. This is likely due to the fact that the numbers in the new templates were more blurred out.

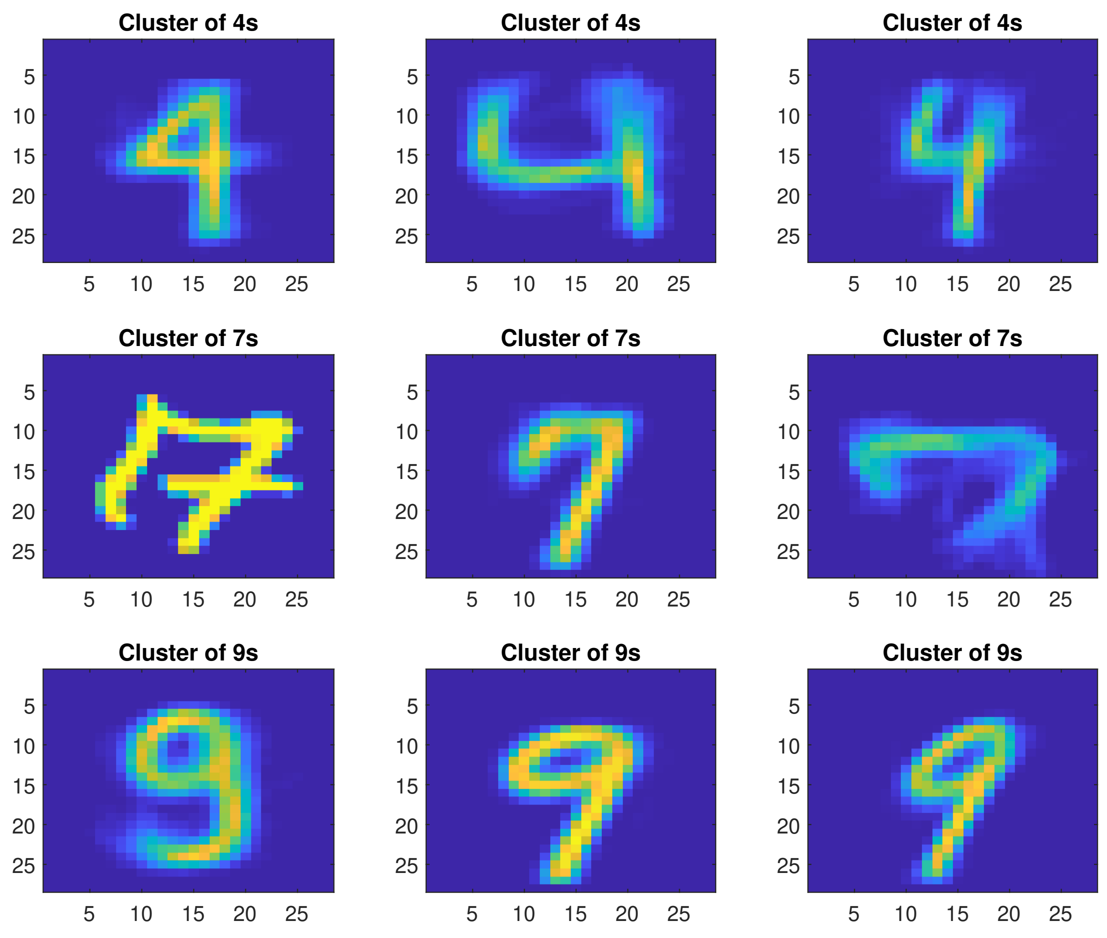


Figure 9: Some chosen clusters. Each cluster is composed of roughly 90 samples. Note how the clustered samples are more smeared than the ones in previous figures, except for the one cluster of 7s which looks very weird. The displayed samples have indices 277, 314, 274, 451, 503, 488, 588, 633 and 628 respectively in the clustered training set matrix, counted from left to right starting at the top.

On the other hand, the benefit of using clustered data as templates is illustrated in table 9. This shows that clustering the training set into 64 clusters per class reduced the computation time of the classification by a factor of almost 50:

| Classifier | Computation time |
|---|---|
| NN, not clustered | 34min |
| Clustering | 30sec |
| NN, clustered | 13sec |
| 7NN, clustered | 15-17sec[2] |

Table 9: The NN classifier without clustering is run with 10 test chunks of 1000 samples and the entire training set as templates. The clustering is the computation time for performing the clustering itself, from 10000 $\rightarrow$ 640 samples, which is only needed once. The computation time for the classifiers using the clustered data as templates is therefore measured excluding the clustering time.

In this case, clustering the training data introduces therefore a trade-off between reduced computation time and increased error rate. Which degree of clustering that is the most beneficial one will depend on the application in which the classifier is used and the constraints and requirements it imposes on the classification process.

[2]Computation time depends on which of the three implementation methods are chosen.

One possible method to improve the error rate, while keeping the benefit of using the clustered training set as templates, could be to use a KNN classifier instead. When a KNN classifier was run with the clustered data as templates, $K = 7$ and tested with the whole test set, it produced the following confusion matrix and error rate:

| Label \ Predicted class | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 953 | 1 | 3 | 0 | 0 | 7 | 13 | 1 | 1 | 1 |
| 1 | 0 | 1127 | 2 | 1 | 0 | 0 | 2 | 0 | 2 | 1 |
| 2 | 10 | 12 | 945 | 12 | 8 | 2 | 4 | 13 | 26 | 0 |
| 3 | 0 | 1 | 5 | 951 | 0 | 24 | 0 | 10 | 14 | 5 |
| 4 | 1 | 9 | 2 | 0 | 887 | 0 | 9 | 8 | 4 | 62 |
| 5 | 4 | 1 | 3 | 28 | 0 | 826 | 7 | 2 | 15 | 6 |
| 6 | 8 | 4 | 1 | 0 | 5 | 11 | 925 | 0 | 3 | 1 |
| 7 | 0 | 25 | 11 | 1 | 5 | 0 | 0 | 948 | 1 | 37 |
| 8 | 5 | 2 | 1 | 24 | 8 | 22 | 2 | 5 | 896 | 9 |
| 9 | 3 | 8 | 3 | 8 | 21 | 3 | 1 | 25 | 6 | 931 |
| Error rate | 6.11% | | | | | | | | | |

Table 10: Confusion matrix and error rate for the KNN classifier with the clustered training set as templates, $K = 7$ and tested on the whole training set. The second implementation of the KNN classifier is chosen, as discussed in the implementation section.

The error rate of the different classifiers can be summarized as follows:

| Classifier | Error rate |
|---|---|
| NN, not clustered | 3.09% |
| NN, clustered | 4.87% |
| 7NN-1, clustered | 6.21% |
| 7NN-2, clustered | 6.11% |
| 7NN-3, clustered | 6.13% |
| 7NN-2, not clustered | 3.06% |

Table 11: Comparison of the error rates of the different classifiers. The "(not) clustered" specifies whether the templates for the classifier were clustered or not and the "-X" for the KNN classifier separates the different implementations of the classifier as discussed in the implementation part.

Table 11 shows that the 7NN classifier actually has a larger error rate than the NN classifier. At first thought, this can be a bit surprising as one could expect that comparing the class of several NNs would lead to better justified class predictions. But this is clearly not the case. The reason for this is probably due to the clustering of the training data. As already illustrated in figure 9 the clustering has the effect of averaging out the training samples, creating blurry templates that sometimes tend towards similar shapes, compared to the more sharply-drawn non-clustered samples with more distinct features. In this way, letting the predicted class depend on the class of several nearest neighbours may lead to worse classification if templates with different labels are relatively closer together than before. Note that the average distance between the templates probably increases when applying clustering, simply due to the fact that there are much fewer templates to cover the sample space with. But due to the averaging effects the distance between two templates with different classes may increase less than the distance between two templates of the same class, which relatively speaking moves the templates with different labels closer together. This explanation is supported by the fact that the 7NN classifier performs slightly better than the NN classifier when it uses the entire training set as templates. Table 11 summarizes this, which further indicates that the problem with the higher error rate for the KNN classifier lies in the clustering of the training data.

## 5 Conclusion

After gotten to know the theory behind the tasks ahead, the classification of the Iris flowers and handwritten numbers gave both expected and unexpected behaviours. When the first task showed the importance of how the classifier was trained based on representative samples it surely sat the baseline for the rest of the report. An increase of the error rate occurred when choosing to train the classifier by the last 30 samples rather than choosing the first 30 samples. It was further illustrated, based on that the classifier was constructed to solve linear separable problems, that the Iris Setosa could be linearly separated from the other two classes based only on one feature, while the other two classes was not linearly separable. Not even when choosing all four features. The best error rate achieved was 8.33%.

When it comes to classifying the handwritten numbers it has been shown that a simple nearest neighbor classifier can perform quite good, with error rates down to almost 3%. The classifier was sometimes able to classify hard samples, while other times failing at rather easy samples, based on the human intuition. Further, it has also been seen that getting good performance could cost lots of computational time. This could be traded off by clustering of the data, where the computational cost would decrease drastically while also loosing some accuracy due to less, and more averaged, templates. This project has, by some means, illustrated some techniques that could be useful to classify labeled classes based on features that is measurable and coexisting throughout multiple of classes. By working with such a project there has been some great experiences with getting a classifier to work and select correctly labeled samples, and have also raised thoughts of what could happen if one could make a classifier train when not having labeled sample, but rather creating a decision rule based on its own.

This project has, by some means, illustrated some techniques that could be useful to classify labeled classes based on features that is measurable and coexisting throughout multiple of classes. By working with such a project there has been some great experiences with getting a classifier to work and select correctly labeled samples. The importance of representative and large enough data sets have also been thoroughly illustrated. In the end the project has also raised thoughts of how one can train a classifier without having labeled samples and thus inspired interest in unsupervised learning.

# Bibliography

[1] Magne H. Johnsen. ‘Classification’. 2017.

[2] Yann LeCun, Corinna Cortes and Christopher J.C. Burges. *THE MNIST DATABASE - of handwritten digits.* URL: http://yann.lecun.com/exdb/mnist/ (visited on 30/04/2021).

[3] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization - Second Edition.* Springer, 2006, pp. 21–35.

[4] Wikipedia. *Divide-and-conquer algorithm.* URL: https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm (visited on 30/04/2021).

[5] Wikipedia. *Iris flower data set.* URL: http://yann.lecun.com/exdb/mnist/ (visited on 30/04/2021).

# Appendix

## A Matlab Code - Iris

All scripts can be downloaded from https://github.com/hjordheimen/EDC-project.

### A Task 1 Implementation

```
clc
clear

%% Set up
x1all = load('class_1','-ascii');
x2all = load('class_2','-ascii');
x3all = load('class_3','-ascii');

N          = size(x1all, 1); % Number of samples from each class
N_training = 30;             % Number of training samples from each class
N_test     = N - N_training; % Number of test samples from each class
C = 3;
D = 4;

% Loading data
xall_labeled = readtable('iris.csv');

% Set up training
x1_training_labeled = xall_labeled(1:N_training, :);
x2_training_labeled = xall_labeled(N+1:N + N_training, :);
x3_training_labeled = xall_labeled(2*N + 1:2*N + N_training, :);

% Target vectors
t1 = [1 0 0]';
t2 = [0 1 0]';
t3 = [0 0 1]';

% Storing each training sample and its target vector in a cell array
training_data = cell(N_training*C, 2);
for i = 1:N_training
   x1_i = [x1_training_labeled.Var1(i) x1_training_labeled.Var2(i)
   ↪  x1_training_labeled.Var3(i) x1_training_labeled.Var4(i)]';
   x2_i = [x2_training_labeled.Var1(i) x2_training_labeled.Var2(i)
   ↪  x2_training_labeled.Var3(i) x2_training_labeled.Var4(i)]';
   x3_i = [x3_training_labeled.Var1(i) x3_training_labeled.Var2(i)
   ↪  x3_training_labeled.Var3(i) x3_training_labeled.Var4(i)]';
   training_data(i, :)                = {x1_i, t1};
   training_data(i + N_training, :)   = {x2_i, t2};
   training_data(i + 2*N_training, :) = {x3_i, t3};
end


%% Linear classifier - Training
% Classifier consists of discriminant
% functions on homogenous matrix form
% g = W*x
```

```
W = train_classifier(training_data, C, D);


%% Testing

[confusion_matrix_testing, confusion_matrix_training] =
↪  get_confusion_matrices(x1all, x2all, x3all, 1:D, W, N_training, N_test, C);

disp('Testing')
disp(confusion_matrix_testing)
disp('Training')
disp(confusion_matrix_training)

testing_error_rate  = 1 - (trace(confusion_matrix_testing)/(N_test*C));
training_error_rate = 1 - (trace(confusion_matrix_training)/(N_training*C));


disp('Error rate - testing')
disp(testing_error_rate)

disp('Error rate - training')
disp(training_error_rate)

disp('-------------------------')
disp('--Reversed sample Order--')
disp('-------------------------')

%% Last 30 samples for training, 20 first as tests
clear

%% Set up
x1all = load('class_1','-ascii');
x2all = load('class_2','-ascii');
x3all = load('class_3','-ascii');

N          = size(x1all, 1); % Number of samples from each class
N_training = 30;             % Number of training samples from each class
N_test     = N - N_training; % Number of test samples from each class
C = 3;
D = 4;

% Loading data
xall_labeled = readtable('iris.csv');

% Set up training
x1_training_labeled = xall_labeled(N_test + 1:end, :);
x2_training_labeled = xall_labeled(N+ N_test + 1:N + N_test+ N_training, :);
x3_training_labeled = xall_labeled(2*N + N_test + 1:2*N + N_test + N_training,
↪  :);

% Target vectors
t1 = [1 0 0]';
t2 = [0 1 0]';
t3 = [0 0 1]';

% Storing each training sample and its target vector in a cell array
training_data = cell(N_training*C, 2);
for i = 1:N_training
```

```
   x1_i = [x1_training_labeled.Var1(i) x1_training_labeled.Var2(i)
   ↪  x1_training_labeled.Var3(i) x1_training_labeled.Var4(i)]';
   x2_i = [x2_training_labeled.Var1(i) x2_training_labeled.Var2(i)
   ↪  x2_training_labeled.Var3(i) x2_training_labeled.Var4(i)]';
   x3_i = [x3_training_labeled.Var1(i) x3_training_labeled.Var2(i)
   ↪  x3_training_labeled.Var3(i) x3_training_labeled.Var4(i)]';
   training_data(i, :)                = {x1_i, t1};
   training_data(i + N_training, :)   = {x2_i, t2};
   training_data(i + 2*N_training, :) = {x3_i, t3};
end

% Extracting training data without labels
x1_training = x1all(N_test + 1:end, :);
x2_training = x2all(N_test + 1:end, :);
x3_training = x3all(N_test + 1:end, :);

% Set up tests
x1_test = x1all(1:N_test, :);
x2_test = x2all(1:N_test, :);
x3_test = x3all(1:N_test, :);



%% Linear classifier - Training
% Classifier consists of discriminant
% functions on homogenous matrix form
% g = W*x

W = train_classifier(training_data, C, D);

%% Testing

confusion_matrix_testing  = zeros(C, C);
confusion_matrix_training = zeros(C, C);

for i = 1:N_test
    % Extract samples
    x1 = x1_test(i, :)';
    x2 = x2_test(i, :)';
    x3 = x3_test(i, :)';

    % Classify samples
    x1_class = classifier(x1, W);
    x2_class = classifier(x2, W);
    x3_class = classifier(x3, W);

    % Update confusion matrix
    confusion_matrix_testing(1, x1_class) = confusion_matrix_testing(1, x1_class)
    ↪  + 1;
    confusion_matrix_testing(2, x2_class) = confusion_matrix_testing(2, x2_class)
    ↪  + 1;
    confusion_matrix_testing(3, x3_class) = confusion_matrix_testing(3, x3_class)
    ↪  + 1;
end

for i = 1:N_training
    % Extract samples
    x1 = x1_training(i, :)';
    x2 = x2_training(i, :)';
    x3 = x3_training(i, :)';

    % Classify samples
    x1_class = classifier(x1, W);
    x2_class = classifier(x2, W);
    x3_class = classifier(x3, W);

    % Update confusion matrix
    confusion_matrix_training(1, x1_class) = confusion_matrix_training(1,
    ↪  x1_class) + 1;
    confusion_matrix_training(2, x2_class) = confusion_matrix_training(2,
    ↪  x2_class) + 1;
    confusion_matrix_training(3, x3_class) = confusion_matrix_training(3,
    ↪  x3_class) + 1;
end

disp('Testing')
disp(confusion_matrix_testing)
disp('Training')
disp(confusion_matrix_training)


testing_error_rate  = 1 - (trace(confusion_matrix_testing)/(N_test*C));
training_error_rate = 1 - (trace(confusion_matrix_training)/(N_training*C));

disp('Error rate - testing')
disp(testing_error_rate)

disp('Error rate - training')
disp(training_error_rate)
```

## B Task 2 Implementation

```
clc
clear

%% Initializing and setup

x1all = load('class_1','-ascii'); % Iris Setosa
x2all = load('class_2','-ascii'); % Iris Versicolor
x3all = load('class_3','-ascii'); % Iris Virginica

num_plots    = 4;
num_features = 4;


%% Producing histograms of the classes
figure(1);
subplot(4,1,1);
for i = 1:num_features
    histogram([x1all(:, i); x2all(:, i); x3all(:, i)], 'BinWidth', 0.2);
    hold on;
end
title("All iris'");
xlabel("Size [m^{-2}]");
ylabel("Quantity");
legend("Sepal length", "Sepal width", "Petal length", "Petal width");
xlim([0, 8]);
ylim([0, 40]);

subplot(4,1,2);
for i = 1:num_features
    histogram(x1all(:, i), 'BinWidth', 0.2);
    hold on;
end
title('Iris Setosa')
xlabel("Size [m^{-2}]");
ylabel("Quantity");
legend("Sepal length", "Sepal width", "Petal length", "Petal width");
xlim([0, 8]);
ylim([0, 40]);

subplot(4,1,3);
for i = 1:num_features
    histogram(x2all(:, i), 'BinWidth', 0.2);
    hold on;
end
title('Iris Versicolor')
xlabel("Size [m^{-2}]");
ylabel("Quantity");
legend("Sepal length", "Sepal width", "Petal length", "Petal width");
xlim([0, 8]);
ylim([0, 40]);

subplot(4,1,4);
for i = 1:num_features
    histogram(x3all(:, i), 'BinWidth', 0.2);
    hold on;
end
```

```
title('Iris Virginica');
xlabel("Size [m^{-2}]");
ylabel("Quantity");
legend("Sepal length", "Sepal width", "Petal length", "Petal width");
xlim([0, 8]);
ylim([0, 40]);

sgtitle("Histograms of the classes");

%% Producing histograms for each feature
feature_names = ["Sepal length", "Sepal width", "Petal length", "Petal width"];
figure(2);
for i = 1:num_features
   subplot(4,1,i);
   histogram(x1all(:, i), 'BinWidth', 0.2);
   hold on;
   histogram(x2all(:, i), 'BinWidth', 0.2);
   hold on;
   histogram(x3all(:, i), 'BinWidth', 0.2);
   hold on;
   title(feature_names(i));
   xlabel("Size [m^{-2}]");
   ylabel("Quantity");
   legend("Iris Setosa", "Iris Versicolor", "Iris Virginica");
   xlim([0, 8]);
   ylim([0, 40]);
end

sgtitle("Histograms for each feature, whole data set");



%% Take away the feature with most overlap between classes

% Setting up
N          = size(x1all, 1); % Number of samples from each class
N_training = 30;             % Number of training samples from each class
N_test     = N - N_training; % Number of test samples from each class
C = 3;
D = 3;

% Loading data
xall_labeled = readtable('iris.csv');

% Set up training
x1_training_labeled = xall_labeled(1:N_training, :);
x2_training_labeled = xall_labeled(N+1:N + N_training, :);
x3_training_labeled = xall_labeled(2*N + 1:2*N + N_training, :);



x_training = [ x1_training_labeled.Var1 x1_training_labeled.Var2
↪  x1_training_labeled.Var3 x1_training_labeled.Var4;
               x2_training_labeled.Var1 x2_training_labeled.Var2
               ↪  x2_training_labeled.Var3 x2_training_labeled.Var4;
               x3_training_labeled.Var1 x3_training_labeled.Var2
               ↪  x3_training_labeled.Var3 x3_training_labeled.Var4;];
```

```
%% Taking away feature: Sepal width
% Features left: Sepal length, Petal length, Petal width

features_needed               = [1,3:4];
training_data_3f              = get_training_data(x_training, features_needed,
↪  N_training, C);
W_3                           = train_classifier(training_data_3f, C,
↪  size(features_needed, 2));
[conf_testing, conf_training] = get_confusion_matrices(x1all, x2all, x3all,
↪  features_needed, W_3, N_training, N_test, C);
disp("Features: [Sepal length (cm), Petal length (cm), Petal width (cm)]")
disp('Testing')
disp(conf_testing)
disp('Training')
disp(conf_training)

testing_error_rate  = 1 - (trace(conf_testing)/(N_test*C));
training_error_rate = 1 - (trace(conf_training)/(N_training*C));

disp('Error rate - testing')
disp(testing_error_rate)

disp('Error rate - training')
disp(training_error_rate)

disp("-------------------")
disp("-------------------")

%% Taking away features: Sepal length, Sepal width
% Features left: Petal length, Petal width

features_needed               = 3:4;
training_data_2f              = get_training_data(x_training, features_needed,
↪  N_training, C);
W_2                           = train_classifier(training_data_2f, C,
↪  size(features_needed, 2));
[conf_testing, conf_training] = get_confusion_matrices(x1all, x2all, x3all,
↪  features_needed, W_2, N_training, N_test, C);
disp("Features: [Petal length (cm), Petal width (cm)]")
disp('Testing')
disp(conf_testing)
disp('Training')
disp(conf_training)

testing_error_rate  = 1 - (trace(conf_testing)/(N_test*C));
training_error_rate = 1 - (trace(conf_training)/(N_training*C));

disp('Error rate - testing')
disp(testing_error_rate)

disp('Error rate - training')
disp(training_error_rate)

disp("-------------------")
disp("-------------------")

%% Taking away features: Sepal length, Sepal width, Petal width
% Features left: Petal length
```

```
features_needed                = 3;
training_data_1f               = get_training_data(x_training, features_needed,
↪  N_training, C);
W_1                            = train_classifier(training_data_1f, C,
↪  size(features_needed, 2));
[conf_testing, conf_training] = get_confusion_matrices(x1all, x2all, x3all,
↪  features_needed, W_1, N_training, N_test, C);
disp("Features: [Petal length (cm)]")
disp('Testing')
disp(conf_testing)
disp('Training')
disp(conf_training)

testing_error_rate = 1 - (trace(conf_testing)/(N_test*C));
training_error_rate = 1 - (trace(conf_training)/(N_training*C));

disp('Error rate - testing')
disp(testing_error_rate)

disp('Error rate - training')
disp(training_error_rate)

disp("-------------------")
disp("-------------------")


function training_data = get_training_data(x_training, features_needed,
↪  N_training, C)

   % Targets
    t1 = [1 0 0]';
    t2 = [0 1 0]';
    t3 = [0 0 1]';

    % Storing each training sample and its target vector in a cell array
    training_data = cell(N_training*C, 2);
    for i = 1:N_training
       x1_i = x_training(i, (features_needed))';
       x2_i = x_training(i + N_training, (features_needed))';
       x3_i = x_training(i + 2*N_training, (features_needed))';
       training_data(i, :) = {x1_i, t1};
       training_data(i + N_training, :) = {x2_i, t2};
       training_data(i + 2*N_training, :) = {x3_i, t3};
    end
end
```

## C Train classifier - functions

```
function W = train_classifier(training_data, C, D)
% Classifier consists of discriminant
% functions on homogenous matrix form
% g = [W w_0]*[x' 1]'

    % Initial guess at weights
    W_0 = zeros(C,D);
    w_0 = [0 0 0]';

    % Homogenous form
    W_0 = [W_0 w_0];

    % Object function to be optimized
    f      = @(W) MSE(training_data, W);
    grad_f = @(W) grad_MSE(training_data, W);

    % Run steepest descent algorithm
    % Extract final iterate
    W_iterates = SteepestDescent(f, grad_f, W_0);
    W = W_iterates(:, end - D: end);
end

function mse = MSE(training_data, W)
% Evaluates MSE function for the linear discriminant classifier
% given the labeled training data and the weights W.

    N = size(training_data, 1);

    mse  = 0;
    for i = 1:N
        x_i = cell2mat(training_data(i, 1));
        t_i = cell2mat(training_data(i, 2));
        x_i = [x_i' 1]';     % Homogenous form

        z_i = W*x_i;
        g_i = 1./(1+exp(-z_i));

        mse = mse + 0.5*(g_i - t_i)'*(g_i - t_i);
    end
end

function grad_mse = grad_MSE(training_data, W)
% Evaluates the gradient of the MSE function w.r.t W
% for the linear discriminant classifier given
% the labeled training data and the weights W.

    N = size(training_data, 1);

    grad_mse  = 0;
    for i = 1:N
        x_i = cell2mat(training_data(i, 1));
        t_i = cell2mat(training_data(i, 2));
        x_i = [x_i' 1]';     % Homogenous form

        z_i = W*x_i;
        g_i = 1./(1+exp(-z_i));

        grad_mse = grad_mse + ((g_i - t_i).*g_i.*(1 - g_i))*(x_i');
    end
end


% Simple optimization algorithm for nonlinear unconstrained optimization
% Finds local minimum of object function
function [x,f_opt] = SteepestDescent(f,gradf,x0,tol,N)
% f          function handle for object function
% gradf      function handle for gradient of object function

% Returns all iterates of x and value
% of object function for final iterate

    row = size(x0, 1);
    col = size(x0, 2);

    if nargin < 5
        N = 100;
    end
    if nargin < 4
        tol = 1e-8;
    end

    iterate = true;
    x = NaN(row,(N+1)*col);
    x(:,1:col) = x0;

    xk = x0;
    i = 1;

    while iterate
        % Perform next iterate step
        pk = -gradf(xk);
        ak = 0.00585; %BLS(f,f(xk),xk,gradf(xk),pk);
        xk_p1 = xk + ak*pk;

        % Check termination criterions
        iterate = (norm(gradf(xk), inf) > tol & norm(xk_p1 - xk, inf) > tol &
        ↪  norm(f(xk_p1) - f(xk), inf) > tol & i < N); % Checking all
        ↪  termination criterias

        % Store and update iterate
        x(:,i*col + 1:i*col + col) = xk_p1;
        xk = xk_p1;
        i = i + 1;
    end

    x = x(:,1:i*col);
    f_opt = f(x(:,end-col + 1:end));
end
```

## D Confusion matrix - functions

```
function [confusion_matrix_testing, confusion_matrix_training]
↪ = get_confusion_matrices(x1all, x2all, x3all, features_needed, W, N_training,
↪ N_test, C)

    confusion_matrix_testing = zeros(C, C);
    confusion_matrix_training = zeros(C, C);

    % Set up training
    x1_training = x1all(1:N_training, features_needed);
    x2_training = x2all(1:N_training, features_needed);
    x3_training = x3all(1:N_training, features_needed);

    % Set up tests
    x1_test = x1all(N_training + 1:end, features_needed);
    x2_test = x2all(N_training + 1:end, features_needed);
    x3_test = x3all(N_training + 1:end, features_needed);


    for i = 1:N_test
        % Extract samples
        x1 = x1_test(i, :)';
        x2 = x2_test(i, :)';
        x3 = x3_test(i, :)';

        % Classify samples
        x1_class = classifier(x1, W);
        x2_class = classifier(x2, W);
        x3_class = classifier(x3, W);

        % Update confusion matrix
        confusion_matrix_testing(1, x1_class) = confusion_matrix_testing(1,
        ↪ x1_class) + 1;
        confusion_matrix_testing(2, x2_class) = confusion_matrix_testing(2,
        ↪ x2_class) + 1;
        confusion_matrix_testing(3, x3_class) = confusion_matrix_testing(3,
        ↪ x3_class) + 1;
    end

    for i = 1:N_training
        % Extract samples
        x1 = x1_training(i, :)';
        x2 = x2_training(i, :)';
        x3 = x3_training(i, :)';

        % Classify samples
        x1_class = classifier(x1, W);
        x2_class = classifier(x2, W);
        x3_class = classifier(x3, W);

        % Update confusion matrix
        confusion_matrix_training(1, x1_class) = confusion_matrix_training(1,
        ↪ x1_class) + 1;
        confusion_matrix_training(2, x2_class) = confusion_matrix_training(2,
        ↪ x2_class) + 1;
        confusion_matrix_training(3, x3_class) = confusion_matrix_training(3,
        ↪ x3_class) + 1;
    end

end


function class = classifier(x, W_tilde)
%Returns class of Fisher vector x based on the linear discriminant
%classifier defined by the weight matrix W_tilde on homogenous form

    x_tilde = [x' 1]';    % Homogenous form
    g = W_tilde*x_tilde;

    [~, class] = max(g);
end
```

## B Matlab Code - Numbers

All scripts can be downloaded from https://github.com/hjordheimen/EDC-project.

### A Formatting of Data

```
% This program will take some time ( approx 15 minutes?) but should be used only
↪  once. Use the Matlab file data_all after this
fid =  fopen('train_images.bin','r');
magic_num = fread(fid,1,'int32','ieee-be');
num_train=fread(fid,1,'int32','ieee-be');
row_size=fread(fid,1,'int32','ieee-be');
col_size=fread(fid,1,'int32','ieee-be');

vec_size = row_size*col_size;
 trainv=zeros(num_train,vec_size);

for k = 1:num_train
        for n = 1:vec_size
                 trainv(k,n) =fread(fid,1,'uchar','ieee-be');
        end
end

fclose(fid);

fid =  fopen('test_images.bin','r');
magic_num = fread(fid,1,'int32','ieee-be');
num_test=fread(fid,1,'int32','ieee-be');
row_size=fread(fid,1,'int32','ieee-be');
col_size=fread(fid,1,'int32','ieee-be');

vec_size = row_size*col_size;
 testv=zeros(num_test,vec_size);

for k = 1:num_test
        for n = 1:vec_size
                testv(k,n) =fread(fid,1,'uchar','ieee-be');
        end
end

fclose(fid);

disp('labels')
fid =  fopen('train_labels.bin','r');
magic_num = fread(fid,1,'int32','ieee-be');
num_train=fread(fid,1,'int32','ieee-be');

trainlab=zeros(num_train,1);

for k = 1:num_train
        trainlab(k) =fread(fid,1,'uchar','ieee-be');
end

fclose(fid);

fid =  fopen('test_labels.bin','r');
```

```
magic_num = fread(fid,1,'int32','ieee-be');
num_test=fread(fid,1,'int32','ieee-be');

testlab = zeros(num_test,1);

for k = 1:num_test
        testlab(k) =fread(fid,1,'uchar','ieee-be');
end

fclose(fid);

save data_all num_train num_test row_size col_size vec_size trainv  trainlab
↪  testv  testlab
```

## B Task 1 Implementation & Plotting

```
clc;
clear;
%% Initializing and set up
load('data_all.mat');
C = 10;                  % Number of classes, 0-9

chunk_size = 1000;
N = num_test/chunk_size;

% Store results in order to display them later
correctly_classified_numbers = NaN(1, num_test);
misclassified_numbers = NaN(1, num_test);

confusion_matrix = zeros(C, C);

%% Run NN classifier on testv with trainv as template (takes ~30min to run)
tic
for k = 1:N
    disp(k);

    chunk_base_index = (k - 1)*chunk_size;
    % Use entire training set as templates
    templates = trainv;
    test_chunk = testv(chunk_base_index + 1:k*chunk_size, :);

    % Each column of Z holds the distance to all templates for one test sample
    % min extracts the index of the NN for each sample
    Z = dist(templates, test_chunk');
    [~, I] = min(Z);

    for sample_chunk_index = 1:chunk_size
        sample_index = chunk_base_index + sample_chunk_index;
        class = trainlab(I(sample_chunk_index));
        label = testlab(sample_index);

        if class == label
            correctly_classified_numbers(sample_index) = class;
        else
            misclassified_numbers(sample_index) = class;
        end
        confusion_matrix(label + 1, class + 1) = confusion_matrix(label + 1,
        ↪  class + 1) + 1;
    end
end
toc

error_rate = 1-(trace(confusion_matrix)/num_test);

disp("Confusion matrix:")
disp(confusion_matrix)
disp("Error rate:")
disp(error_rate)


clc;
clear;
%% Initializing and set up
load('data_all.mat');
C = 10;                  % Number of classes, 0-9

chunk_size = 1000;
N = num_test/chunk_size;

% Store results in order to display them later
correctly_classified_numbers = NaN(1, num_test);
misclassified_numbers = NaN(1, num_test);

confusion_matrix = zeros(C, C);

%% Run NN classifier on testv with trainv as template (takes ~30min to run)
tic
for k = 1:N
    disp(k);

    chunk_base_index = (k - 1)*chunk_size;
    % Use entire training set as templates
    templates = trainv;
    test_chunk = testv(chunk_base_index + 1:k*chunk_size, :);

    % Each column of Z holds the distance to all templates for one test sample
    % min extracts the index of the NN for each sample
    Z = dist(templates, test_chunk');
    [~, I] = min(Z);

    for sample_chunk_index = 1:chunk_size
        sample_index = chunk_base_index + sample_chunk_index;
        class = trainlab(I(sample_chunk_index));
        label = testlab(sample_index);

        if class == label
            correctly_classified_numbers(sample_index) = class;
        else
            misclassified_numbers(sample_index) = class;
        end

        confusion_matrix(label + 1, class + 1) = confusion_matrix(label + 1,
        ↪  class + 1) + 1;
    end
end
```

```
toc

error_rate = 1-(trace(confusion_matrix)/num_test);

disp("Confusion matrix:")
disp(confusion_matrix)
disp("Error rate:")
disp(error_rate)
```

## C   Task 2 Implementation & Plotting

```
clc
clear

%% Initializing and set up
load('data_all.mat');

C = 10; % # of classes
M = 64; % # of clusters

% Match training samples and labels in a cell array
training_data = cell(num_train, 2);
for sample_index = 1:num_train
   number_i = trainv(sample_index, :);
   label_i = trainlab(sample_index);
   training_data(sample_index, :) = {number_i, label_i};
end

% Sort the training data by label
training_data = sortrows(training_data, 2);
sorted_training_data = cell2mat(training_data(:, 1));

% Count the number of samples from each class
num_examples = zeros(1, C);
for sample_index = 1:num_train
    num_examples(trainlab(sample_index) + 1) =
    ↪  num_examples(trainlab(sample_index) + 1) + 1;
end

% Generate start index of each class
% Zero indicates the start of the first class
class_indices = cumsum(num_examples);
class_indices = [0 class_indices];


% Generate clusters and insert them into the new training set
new_trainv = NaN(M*C, vec_size);

for sample_index = 1:C
    class_i = sorted_training_data(class_indices(sample_index) +
    ↪  1:class_indices(sample_index + 1), :);
    [~, new_trainv((sample_index - 1)*M + 1:sample_index*M, :)] = kmeans(class_i,
    ↪  M);
end


% Creating new labels for training set
```

```
new_trainlab = [ zeros(M, 1);
                  ones(M, 1);
                2*ones(M, 1);
                3*ones(M, 1);
                4*ones(M, 1);
                5*ones(M, 1);
                6*ones(M, 1);
                7*ones(M, 1);
                8*ones(M, 1);
                9*ones(M, 1); ];

%% Run NN classifier using the clusters as templates
tic
confusion_matrix_NN = zeros(C, C);

% Use the clustered samples as templates
% Each column of Z holds the distance to all templates for one test sample
% min extracts the index of the NN for each sample
Z = dist(new_trainv, testv');
[~, I] = min(Z);

for sample_index = 1:num_test
    class = new_trainlab(I(sample_index));
    label = testlab(sample_index);

    % +1 is due to the indexing starting at 1.
    confusion_matrix_NN(label + 1, class + 1) = confusion_matrix_NN(label + 1,
    ↪  class + 1) + 1;
end


error_rate_NN = 1 - (trace(confusion_matrix_NN)/num_test);

toc

disp("Confusion matrix, clustering:");
disp(confusion_matrix_NN);
disp("Error rate, clustering:");
disp(error_rate_NN);


%% KNN - calculate distances and extract NN indices

K = 7;

% Each column of Z holds the distance to all templates for one test sample
% mink extracts the indices of the KNNs for each sample
Z = dist(new_trainv, testv');
[~, I] = mink(Z, K);


%% KNN Method 1
% If multiple classes are most frequent, choose most
% frequent lowest class

confusion_matrix_KNN_1 = zeros(C, C);

% For each sample
```

```
for sample_index = 1:num_test

    NN_classes = NaN(1, K);
    for nn = 1:K
       NN_classes(nn) = new_trainlab(I(nn, sample_index));
    end

    % choose most frequent lowest class
    predicted_class = mode(NN_classes);
    label = testlab(sample_index);

    confusion_matrix_KNN_1(label + 1, predicted_class + 1) =
     ↪  confusion_matrix_KNN_1(label + 1, predicted_class + 1) + 1;
end

error_rate_1 = 1 - (trace(confusion_matrix_KNN_1)/num_test);

disp("Confusion matrix 1:");
disp(confusion_matrix_KNN_1);
disp("Error rate 1:");
disp(error_rate_1);


%% KNN Method 2
% If multiple classes are most frequent, choose most
% frequent class with lowest total distance to sample

confusion_matrix_KNN_2 = zeros(C, C);

% For each sample
for sample_index = 1:num_test

    NN_class_and_dist_count = zeros(2,C);

    % Count the number of NNs from each class and the total distance
    % from the sample to each class among the NNs.
    for nn = 1:K
       NN_class = new_trainlab(I(nn, sample_index));
       NN_dist = Z(I(nn,sample_index));
       NN_class_and_dist_count(1,NN_class+1) =
        ↪  NN_class_and_dist_count(1,NN_class+1) + 1;
       NN_class_and_dist_count(2,NN_class+1) =
        ↪  NN_class_and_dist_count(2,NN_class+1) + NN_dist;
    end

    % Find the most frequent class. If several, choose the nearest one
    predicted_class = NaN;
    max_count = 0;
    min_dist  = inf;
    for class = 0:C-1
        class_count = NN_class_and_dist_count(1,class+1);
        class_dist  = NN_class_and_dist_count(2,class+1);
        if class_count > max_count
            max_count = class_count;
            min_dist  = class_dist;
            predicted_class = class;
        elseif (class_count == max_count) && (class_dist < min_dist)
            max_count = class_count;
            min_dist  = class_dist;
            predicted_class = class;
        end
    end

    label = testlab(sample_index);

    confusion_matrix_KNN_2(label + 1, predicted_class + 1) =
    ↪  confusion_matrix_KNN_2(label + 1, predicted_class + 1) + 1;
end

error_rate_2 = 1 - (trace(confusion_matrix_KNN_2)/num_test);

disp("Confusion matrix 2:");
disp(confusion_matrix_KNN_2);
disp("Error rate 2:");
disp(error_rate_2);

%% KNN Method 3
% If multiple classes are most frequent, choose most
% frequent class with lowest single distance to sample

confusion_matrix_KNN_3 = zeros(C, C);

% For each sample
for sample_index = 1:num_test

    NN_class_count = zeros(1,C);
    NN_min_dist = inf*ones(1,C);
    NN_class_count_and_min_dist = [NN_class_count; NN_min_dist];

    % Count the number of NNs from each class. Store the lowest single
    % distance from the sample to each class among the NNs.
    for nn = 1:K
       NN_class = new_trainlab(I(nn, sample_index));
       NN_dist = Z(I(nn,sample_index));
       NN_class_count_and_min_dist(1,NN_class+1) =
       ↪  NN_class_count_and_min_dist(1,NN_class+1) + 1;
       if NN_dist <  NN_class_count_and_min_dist(2,NN_class+1)
           NN_class_count_and_min_dist(2,NN_class+1) = NN_dist;
       end
    end

    % Find the most frequent class. If several, choose the nearest one
    predicted_class = NaN;
    max_count = 0;
    min_dist  = inf;
    for class = 0:C-1
        class_count = NN_class_count_and_min_dist(1,class+1);
        min_class_dist  = NN_class_count_and_min_dist(2,class+1);
        if class_count > max_count
            max_count = class_count;
            min_dist  = min_class_dist;
            predicted_class = class;
        elseif (class_count == max_count) && (min_class_dist < min_dist)
            max_count = class_count;
            min_dist  = min_class_dist;
            predicted_class = class;
        end
    end

    label = testlab(sample_index);

    confusion_matrix_KNN_3(label + 1, predicted_class + 1) =
     ↪  confusion_matrix_KNN_3(label + 1, predicted_class + 1) + 1;
end

error_rate_3 = 1 - (trace(confusion_matrix_KNN_3)/num_test);

disp("Confusion matrix 3:");
disp(confusion_matrix_KNN_3);
disp("Error rate 3:");
disp(error_rate_3);


clc
clear

%% Initializing and set up
load('data_all.mat');

C = 10; % # of classes
M = 64; % # of clusters

% Match training samples and labels in a cell array
training_data = cell(num_train, 2);
for sample_index = 1:num_train
   number_i = trainv(sample_index, :);
   label_i = trainlab(sample_index);
   training_data(sample_index, :) = {number_i, label_i};
end

% Sort the training data by label
training_data = sortrows(training_data, 2);
sorted_training_data = cell2mat(training_data(:, 1));

% Count the number of samples from each class
num_examples = zeros(1, C);
for sample_index = 1:num_train
    num_examples(trainlab(sample_index) + 1) =
     ↪  num_examples(trainlab(sample_index) + 1) + 1;
end

% Generate start index of each class
% Zero indicates the start of the first class
class_indices = cumsum(num_examples);
class_indices = [0 class_indices];


% Generate clusters and insert them into the new training set
new_trainv = NaN(M*C, vec_size);

for sample_index = 1:C
    class_i = sorted_training_data(class_indices(sample_index) +
     ↪  1:class_indices(sample_index + 1), :);
    [~, new_trainv((sample_index - 1)*M + 1:sample_index*M, :)] = kmeans(class_i,
     ↪  M);
```

```
end


% Creating new labels for training set
new_trainlab = [ zeros(M, 1);
                   ones(M, 1);
                 2*ones(M, 1);
                 3*ones(M, 1);
                 4*ones(M, 1);
                 5*ones(M, 1);
                 6*ones(M, 1);
                 7*ones(M, 1);
                 8*ones(M, 1);
                 9*ones(M, 1); ];

%% Run NN classifier using the clusters as templates
tic
confusion_matrix_NN = zeros(C, C);

% Use the clustered samples as templates
% Each column of Z holds the distance to all templates for one test sample
% min extracts the index of the NN for each sample
Z = dist(new_trainv, testv');
[~, I] = min(Z);

for sample_index = 1:num_test
    class = new_trainlab(I(sample_index));
    label = testlab(sample_index);

    % +1 is due to the indexing starting at 1.
    confusion_matrix_NN(label + 1, class + 1) = confusion_matrix_NN(label + 1,
    ↪  class + 1) + 1;
end


error_rate_NN = 1 - (trace(confusion_matrix_NN)/num_test);

toc

disp("Confusion matrix, clustering:");
disp(confusion_matrix_NN);
disp("Error rate, clustering:");
disp(error_rate_NN);


%% KNN - calculate distances and extract NN indices

K = 7;

% Each column of Z holds the distance to all templates for one test sample
% mink extracts the indices of the KNNs for each sample
Z = dist(new_trainv, testv');
[~, I] = mink(Z, K);


%% KNN Method 1
% If multiple classes are most frequent, choose most
% frequent lowest class
```

```
confusion_matrix_KNN_1 = zeros(C, C);

% For each sample
for sample_index = 1:num_test

    NN_classes = NaN(1, K);
    for nn = 1:K
       NN_classes(nn) = new_trainlab(I(nn, sample_index));
    end

    % choose most frequent lowest class
    predicted_class = mode(NN_classes);
    label = testlab(sample_index);

    confusion_matrix_KNN_1(label + 1, predicted_class + 1) =
     ↪  confusion_matrix_KNN_1(label + 1, predicted_class + 1) + 1;
end

error_rate_1 = 1 - (trace(confusion_matrix_KNN_1)/num_test);

disp("Confusion matrix 1:");
disp(confusion_matrix_KNN_1);
disp("Error rate 1:");
disp(error_rate_1);


%% KNN Method 2
% If multiple classes are most frequent, choose most
% frequent class with lowest total distance to sample

confusion_matrix_KNN_2 = zeros(C, C);

% For each sample
for sample_index = 1:num_test

    NN_class_and_dist_count = zeros(2,C);

    % Count the number of NNs from each class and the total distance
    % from the sample to each class among the NNs.
    for nn = 1:K
       NN_class = new_trainlab(I(nn, sample_index));
       NN_dist = Z(I(nn,sample_index));
       NN_class_and_dist_count(1,NN_class+1) =
        ↪  NN_class_and_dist_count(1,NN_class+1) + 1;
       NN_class_and_dist_count(2,NN_class+1) =
        ↪  NN_class_and_dist_count(2,NN_class+1) + NN_dist;
    end

    % Find the most frequent class. If several, choose the nearest one
    predicted_class = NaN;
    max_count = 0;
    min_dist  = inf;
    for class = 0:C-1
        class_count = NN_class_and_dist_count(1,class+1);
        class_dist  = NN_class_and_dist_count(2,class+1);
        if class_count > max_count
            max_count = class_count;
            min_dist  = class_dist;
            predicted_class = class;
        elseif (class_count == max_count) && (class_dist < min_dist)
            max_count = class_count;
            min_dist  = class_dist;
            predicted_class = class;
        end
    end

    label = testlab(sample_index);

    confusion_matrix_KNN_2(label + 1, predicted_class + 1) =
     ↪  confusion_matrix_KNN_2(label + 1, predicted_class + 1) + 1;
end

error_rate_2 = 1 - (trace(confusion_matrix_KNN_2)/num_test);

disp("Confusion matrix 2:");
disp(confusion_matrix_KNN_2);
disp("Error rate 2:");
disp(error_rate_2);

%% KNN Method 3
% If multiple classes are most frequent, choose most
% frequent class with lowest single distance to sample

confusion_matrix_KNN_3 = zeros(C, C);

% For each sample
for sample_index = 1:num_test

    NN_class_count = zeros(1,C);
    NN_min_dist = inf*ones(1,C);
    NN_class_count_and_min_dist = [NN_class_count; NN_min_dist];

    % Count the number of NNs from each class. Store the lowest single
    % distance from the sample to each class among the NNs.
    for nn = 1:K
       NN_class = new_trainlab(I(nn, sample_index));
       NN_dist = Z(I(nn,sample_index));
       NN_class_count_and_min_dist(1,NN_class+1) =
        ↪  NN_class_count_and_min_dist(1,NN_class+1) + 1;
       if NN_dist <  NN_class_count_and_min_dist(2,NN_class+1)
           NN_class_count_and_min_dist(2,NN_class+1) = NN_dist;
       end
    end

    % Find the most frequent class. If several, choose the nearest one
    predicted_class = NaN;
    max_count = 0;
    min_dist  = inf;
    for class = 0:C-1
        class_count = NN_class_count_and_min_dist(1,class+1);
        min_class_dist  = NN_class_count_and_min_dist(2,class+1);
        if class_count > max_count
            max_count = class_count;
            min_dist  = min_class_dist;
            predicted_class = class;
        elseif (class_count == max_count) && (min_class_dist < min_dist)
            max_count = class_count;
            min_dist  = min_class_dist;
            predicted_class = class;
        end
    end

    label = testlab(sample_index);

    confusion_matrix_KNN_3(label + 1, predicted_class + 1) =
     ↪  confusion_matrix_KNN_3(label + 1, predicted_class + 1) + 1;
end

error_rate_3 = 1 - (trace(confusion_matrix_KNN_3)/num_test);

disp("Confusion matrix 3:");
disp(confusion_matrix_KNN_3);
disp("Error rate 3:");
disp(error_rate_3);
```