NTNU
Kunnskap for en bedre verden

Department of Electronic Systems

TTT4275 - Estimation, Detection and Classification

Project: Classification_Image Iris & Digit Recognition

Authors:
Oddbjørn Mandelid Horn
Martin Fæste Slettom

28th April 2023

Summary

The goal of this project is to design a linear classifier for classifying iris-flowers and a Nearest Neighbor classifier for recognizing digits from the MNIST-database.

In this project, a linear classifier has been designed to classify which among three types of irises a flower is. First this is done for four different features, then for fewer and fewer features to review the property of separability for the features.

There has also been designed a k-Nearest Neighbor classifier to recognize digits based on a 28x28-pixel image. The classifier performs the best when using all the templates, but has a considerable amount of processing time at around 30 minutes. When clustering the templates using a 64 k-means algorithm, the error-rate increases. The processing time however was reduced to around 5 minutes.

Table of Contents

1 Introduction

Human brains are complex and as such are able to perform complex classification of objects. This is done based on qualities that we recognize. We can for example recognize and classify that the animal is a dog based on visual features such as color, length, width or features such as the sounds it makes. In the same way, there is often a need to use a computer for classifying objects when the amount of data to be classified is large. Similar to how we use unique details to classify, a computer can use a set of spesific measurements, called features.

Classification becomes more and more used in modern technology. Everything from automated speech recognition, fingerprint and DNA sequence identification and much more uses classification. The amount of solutions using artificial intelligence and machine learning are increasing at a rapid rate. The way that modern AI and machine learning is built up, often using deep neural networks, requires knowledge regarding classification.

This report describes the design and implementation of a classifier for three different types of iris flower and a classifier that recognizes what a number is based on an image.

Section 2 covers the background theory required to understand classification and the spesific goal that this project is based on. A spesific description of the project is given in section 3. Section 4 covers the implementation, testing and the results of the tests of the classifiers and section 5 concludes the project.

2 Theory

This section will give the required theoretical knowledge to understand classification as a whole, the tasks described in section 3 and the implementation described in section 4. First there will be a description of what classification is, then there will be some theory on how to choose features, model and the dimensionality of the feature space. At last there will be a description of linear and Nearest Neighbour (NN) classifiers.

As mentioned in section 1, classification is the task which separates different objects into unique classes based on measurements, called features. When choosing what features to measure and use, we need to know how the features are related between the classes. Ideally, there should be no overlap of a feature across classes. The features of the different classes can be linearly separable, non-linearly separable or not separable at all. These cases are shown in figure 1 for the x_1x_2-feature space.

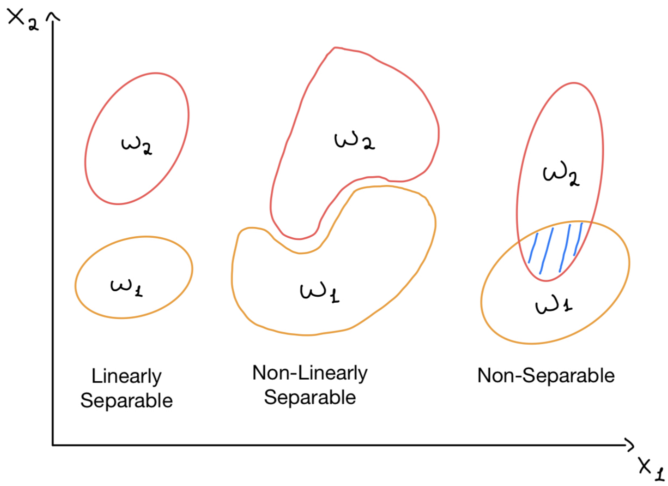


Figure 1: Different kinds of separability shown in the feature-space consisting of the features x_1 and x_2.

Given a linearly or non-linearly separable feature space, one can design an error-free classifier. Most of the actual classification cases revolve around a non-separable feature space. Thus there is a need to choose the most optimal model for classification. One can still use a linear or non-linear classifier and although they are not ideal, they might be satisfactory. Choosing one over the other is most often a trade-off between performance and complexity [2, p. 6].

Linear models is a simple classification model in terms of complexity, but can have poor performance on non-linearly or non-separable features.

Examples of linear classifiers are perceptrons, logistic regression and Support Vector Machines (SVM). This classifier uses linear hyperplanes to separate the various classes in the feature-space into decision regions. A sample is then classified based on the decision region it falls into.

There are also various models useful for non-linear classification. An example classifier is the k-Nearest Neighbour (kNN). Instead of using a hyperplane, this model compares the distance between the sample and the various training samples, known as templates. This model then picks the most common class among the k templates with the smallest distance.

Linear models using multiclass perceptrons are described more thorougly in subsection 2.1 and the kNN-model in subsection 2.2.

There are also challenges surrounding the dimensionality of the feature space, often referred to as the Curse of Dimensionality, which states that "the number of necessary training patterns for acceptable performance grows exponentially with dimensionality" [4]. This is due to a reduction of the density in the feature-space between samples, also within one class. To counter this, there is a requirement of more data samples for training. Increasing the amount of data required also increases the computational cost of the algorithm, and thus the processing times and resource requirements become increased.

There is also a downside to choosing a too small dimensionality. If the classes are non-separable and we have too few features to look at, the task of separating the classes becomes increasingly more difficult.

2.1 Linear Classification

This subsection will discuss more the approach to using a linear classifier model, more specifically a perceptron model.

The discriminant function for class ω_i, $i = 1, 2, .., C$ where C is the amount of classes is denoted with $g_i(\mathbf{x})$, where $\mathbf{x}$ is the vector[1] containing the values of all features for a sample. The discriminant functions are not correlated across classes.

The boundary surface between for example ω_1 and ω_2 is given by $g_1(\mathbf{x}) = g_2(\mathbf{x})$. We can then divide the feature space into decision regions.

This means that a decision region indicates the region where samples of a specific class belong to, i.e.

$$\mathbf{x} \in w_j \Rightarrow g_j(\mathbf{x}) = \underset{i}{max}[g_i(\mathbf{x})] \tag{1}$$

These discriminant functions can be put in a vector that contains the discriminant functions for all classes, $\mathbf{g}(\mathbf{x})$. The linear discriminant functions then become

$$\mathbf{g}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{w_0} \tag{2}$$

where $\mathbf{W}$ is the matrix containing all the weights for all classes and $\mathbf{w_0}$ the biases for all classes. $\mathbf{g}(\mathbf{x})$ is used as perceptron outputs, which will later be used for classifying the sample.

[1] Bold letters means a vector. A matrix is represented as a bold and capital letter.

The approach when using perceptrons is to adjust the weights to fit the problem using a training set. One way of adjusting is by using gradient descent. This algorithm aims to find a local, or ideally global, minimum of a differentiable function. One can use the mean-square error (MSE) as a differentiable function. The MSE is then the error between the calculated perceptron outputs, or the discriminant functions, and the targeted, or true, outputs for the sample in the training set. The MSE is then effectively a cost function.

The true value is saved in a target vector $\mathbf{t}$. There are many ways to represent this target vector, but hot-one encoding is commonly used when the cost of misclassification is equal across all classes. The target vector for ω_1 with three classes is then

$$\mathbf{t} = [1, 0, 0]^T \tag{3}$$

To accurately calculate the MSE between $\mathbf{g}(\mathbf{x})$ and $\mathbf{t}$, we want the perceptron outputs, $\mathbf{g}(\mathbf{x})$, to be on the same scale as the target vectors. This means that $\mathbf{g}(\mathbf{x})$ is on a scale from 0 to 1, where 0 is no similarity and 1 is the most amount of similarity possible. For this we can use an activation function, e.g. a sigmoid fucntion $\phi(\mathbf{x})$. The sigmoid function used on $\mathbf{g}(\mathbf{x})$ is given as

$$\boldsymbol{\phi}(\mathbf{g}(\mathbf{x})) = \frac{1}{1 + \exp(-\mathbf{g}(\mathbf{x}))} = \frac{1}{1 + \exp(-(\mathbf{W}\mathbf{x} + \mathbf{w_0}))} \tag{4}$$

Where the vector $\boldsymbol{\phi}(\mathbf{g}(\mathbf{x})) = [\phi(\mathbf{g}_1(\mathbf{x}), \phi(\mathbf{g}_2(\mathbf{x}), .., \phi(\mathbf{g}_C(\mathbf{x})]$. For ease of notation, we redefine $\boldsymbol{\phi}(\mathbf{g}(\mathbf{x})) \rightarrow \mathbf{g}$.

The expression for the MSE then becomes

$$MSE = \frac{1}{2}(\mathbf{g} - \mathbf{t})^T(\mathbf{g} - \mathbf{t}) = \frac{1}{2}(\mathbf{g}^T\mathbf{g} - 2 \cdot \mathbf{g}^T\mathbf{t} + \mathbf{t}^T\mathbf{t}) \tag{5}$$

Since we change the weights and the bias to reduce the MSE, we find the gradient with respect to $\mathbf{W}$ and $\mathbf{w_0}$. Since $\mathbf{g}$ is dependent on $\mathbf{W}$ and $\mathbf{w_0}$, we then get using the chain rule

$$\nabla_{\mathbf{W}}\mathbf{MSE} = \nabla_{\mathbf{g}}\mathbf{MSE} \cdot \nabla_{\mathbf{z}}\mathbf{g} \cdot \nabla_{\mathbf{W}}(\mathbf{W}\mathbf{x} + \mathbf{w_0}) \tag{6}$$

$$\nabla_{\mathbf{w_0}}\mathbf{MSE} = \nabla_{\mathbf{g}}\mathbf{MSE} \cdot \nabla_{\mathbf{z}}\mathbf{g} \cdot \nabla_{\mathbf{w_0}}(\mathbf{W}\mathbf{x} + \mathbf{w_0}) \tag{7}$$

Where we have defined $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{w_0}$. We call the first product $\mathbf{u}$, which will equal

$$\mathbf{u} = (\mathbf{g} - \mathbf{t}) \circ \mathbf{g} \circ (\mathbf{1} - \mathbf{g}) \tag{8}$$

where $\circ$ is element-wise multiplication. We then get the gradient wrt. the weights

$$\nabla_{\mathbf{W}}\mathbf{MSE} = \mathbf{u}\mathbf{x}^T \tag{9}$$

which is the outer product between $\mathbf{u}$ and $\mathbf{x}$. In the same way we get the gradient wrt. the biases

$$\nabla_{\mathbf{w_0}}\mathbf{MSE} = \mathbf{u} \tag{10}$$

The weights are then updated using this matrix and a learning rate α which indicates how much to adjust.

$$\mathbf{W}(k+1) = \mathbf{W}(k) - \alpha \cdot \nabla_{\mathbf{W}}\mathbf{MSE} \tag{11}$$

The biases are updated in the same way.

$$\mathbf{w_0}(k+1) = \mathbf{w_0}(k) - \alpha \cdot \nabla_{\mathbf{w_0}}\mathbf{MSE} \tag{12}$$

Where k is the iteration variable. The algorithm iterates through the training set multiple times. The reason for iterating through the training set for multiple iterations is to get the best posssibility of convergence to a local or global minimum for the MSE.

The approach where the weights are adjusted for each training sample, is called online learning. The advantage of using online learning is that it requires less memory and computational power.

When classifying a sample $\mathbf{x}$, we classify it to the class with the largest corresponding sigmoid function. I.e.

$$\mathbf{x} \in w_j \Rightarrow \phi(g_j(\mathbf{x})) = \underset{i}{max}[\phi(g_i(\mathbf{x}))] \tag{13}$$

2.2 k-Nearest-Neighbour Classification

This subsection will discuss the approach of classifying data using a k-Nearest-Neighbour classifier.

The kNN-classifier is a relatively simple classifier, but can become computationally demanding for large dimensions with many templates. The approach with this classifier is to look at the distances between the sample we want to classify to the template samples.

If $k = 1$, then we classify the sample as the class of the nearest template. However, if $k > 1$, we then assign the class which is the most frequent among the k closest templates to the sample. This is shown in figure 12 in appendix A. The figure is borrowed from [1, p. 183].

Two common ways to calculate distances are euclidean distance and mahalanobis distance. Euclidean distance is the straight-line distance between two points in the feature-space. Thus if two points are equally far away from a point, even in different directions, the distances will be the same. Mahalanobis distance is useful when we know there is a correlation between the features or if we have a preferred way of orientation in the feature-space. This means that two points with equal euclidean distance to another point, may not have the same mahalanobis distance. This difference is illustrated in figure 13 which is borrowed from [3].

For the digit recognition-task, we assume that all directions are equal, i.e. the features are independent, and therefore we use euclidean distance.

The distance between two vectors $\mathbf{a}$ and $\mathbf{b}$ with a covariance matrix $\mathbf{\Sigma}$ is $\sqrt{(\mathbf{a}-\mathbf{b})^T\mathbf{\Sigma}^{-1}(\mathbf{a}-\mathbf{b})}$, but $\mathbf{\Sigma} = \mathbf{1}$ since we assume euclidean distance. Thus the euclidean distance D between vectors $\mathbf{a} = [a_1, a_2, ..., a_L]^T$ and $\mathbf{b} = [b_1, b_2, ..., b_L]^T$ becomes

$$D = \sqrt{(\mathbf{a}-\mathbf{b})^T(\mathbf{a}-\mathbf{b})} = \sqrt{\sum_{l=1}^{L}(a_l - b_l)^2} \tag{14}$$

Thus, the more the two vectors resemble each other, i.e. a_l is close to b_l for $l = 1, 2, ..., L$, the smaller the distance D gets.

Once we have the distances between the sample and all the templates, we classify the sample using the method explained earlier.

3 Task Description

This section will give a more thorough description of the tasks that are to be solved.

3.1 Iris-task

An iris is a flower consisting of two types of leaves, where the largest one is called the sepal and the smallest one is called the petal. A common classification problem is to classify which among three types of irises a flower is based on the length and width of the sepal and petal. The three irises are setosa, versicolor and virginica.

The classification task is based upon 50 samples of each flower. The goal is to design a linear classifier as described in subsection 2.1.

The first part of the task is to evaluate the performance when the first 30 samples are used for training and the rest for testing and compare it to the performance when the last 30 samples are used for training and the rest for testing.

The second part is to look at the separability of features across classes, to then remove some of these and evaluate the performance. The features with the most overlap is removed first, and then we remove the features until we only have one feature left.

The performance is measured in terms of confusion matrices and the error-rates.

3.2 Digit Recognition-task

For the digit-recognition task, the task is to recognize what digit is written based on digitalized handwritten digits from the MNIST-database. This database consists of 60 000 training samples and 10 000 testing samples. Each image is digitalized as a 28x28-pixel array. The data for each image is then saved as a 1x784-array. All the images have been processed, which means they have been scaled correctly and centered.

The task is to design a kNN-classifier as described in subsection 2.2 using euclidean distance.

The first part of the task is to use all training samples as templates. The second part revolves around reducing the training set to 64 samples from each class by clustering and using these as templates instead. Both a 1NN- and a 7NN-classifier is to be designed.

The performance shall be compared at the end. The performance consists of confusion matrices, error-rates and also processing times. The processing time is most interesting across the systems using the entire training set and the system using the clustered training set.

4 Implementation and Results

This section will contain an explanation of how the systems are implemented. Results of the tests will be shown and discussed. First for the iris-task, then for the digit recognition-task.

4.1 Iris-task

The iris-task is implemented using Python and the code is shown in appendix F. This subsection will describe the main parts of the implementation, as well as the tests and results. At the end there is some discussion regarding the performance.

Flow-chart

To easier see the structure of the implementation, figure 2 shows a flow-chart of the training and testing process. This flow is the same for both the first and the second part of the iris-task. The only difference is the data which is read.

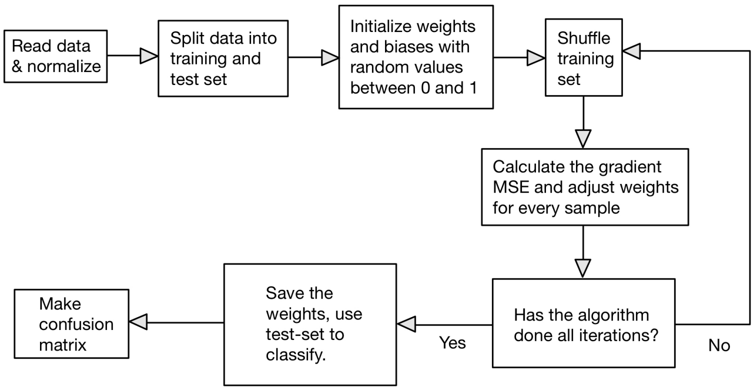


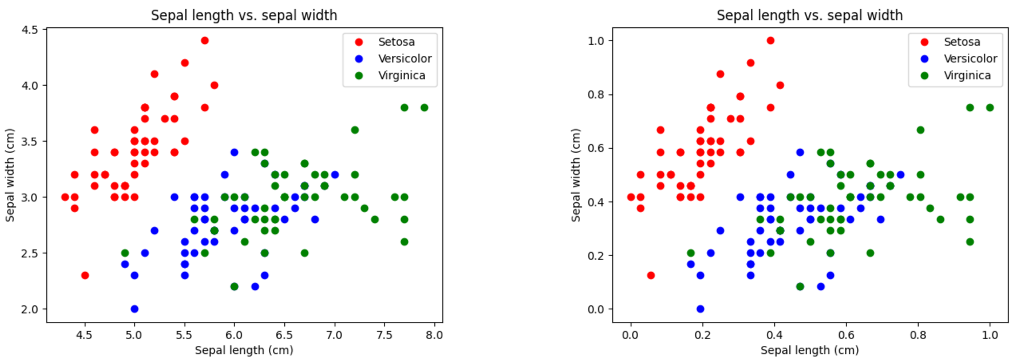
Figure 2: General flow-chart for the Iris-task.

The implementation of the splitting of features is done in the functions `three_features()`, `two_features()` and `one_feature()` which is shown in appendix F.

Normalization

After the data has been read and sorted into arrays for each class, the data is then normalized. This is done using min-max normalization in the function `normalization()` shown in appendix F.

The reason for normalizing the data is because the absolute size of the data is irrelevant, and we are only interested in the size relative to the global maximum and minimum across classes. Figure 3a and 3b shows a plot of the sepal length vs. sepal width for the different classes for the unormalized and the normalized data respectively. As can be seen, normalizing the data has no effect on the relative distance between the samples, but removes the relevance of the absolute size of the features.



(a) Plot of sepal length vs. sepal width for the unormalized data.

(b) Plot of sepal length vs. sepal width for the normalized data.

Figure 3: Plots of the sepal length vs. sepal width features.

Training

The training of the weights is implemented directly as described in subsection 2.1. The function `training()` shown in appendix F is executing the training algorithm.

The weights are initialized as a CxD-matrix with random variables and the bias as a Cx1-vector, where C is the number of classes and D is the number of features. The weight and bias values are assigned random variables between 0 and 1 at the start to remove any symmetry and systematics. A negative effect of this implementation, is that the results may differ for each test-run of the classifier.

The algorithm iterates through the training-set a preset number of times, each time the training-set is shuffled to remove systematics in the training. Since online learning is used, the weights are adjusted for every sample in the training-set.

The number of iterations through the training set and the learning rate α are two measures that are important to give an appropriate value. These can be found by plotting the total MSE for the training set with respect to iterations for multiple learning rates. This is shown in figure 4.

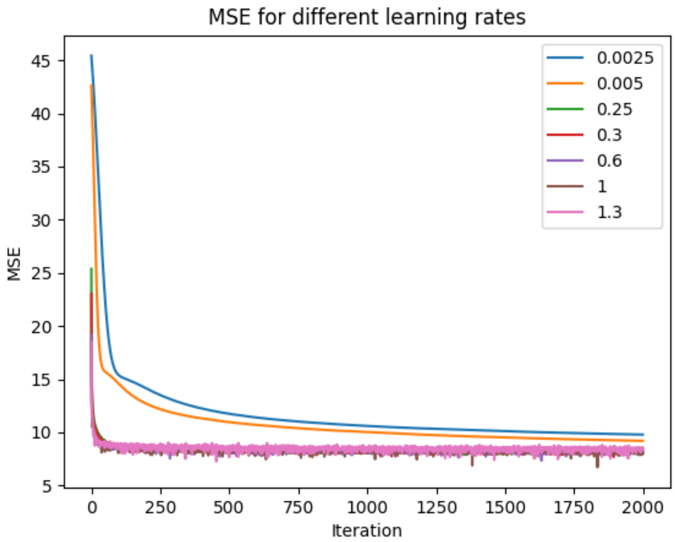


Figure 4: Total MSE for the training set, plotted for different learning rates with respect to the number of iterations.

For larger learning rates, the MSE converges more quickly. The learning rate is chosen to be $\alpha = 0.6$. It can be chosen to be larger, but this is not done to avoid the risk of overshooting and diverging [1, p. 225]. For this learning rate, 250 iterations should be enough. There should however be more iterations to account for the randomly initalized weights. The system was found to be the most consistent when the algorithm used 1000 iterations.

One can alternatively implement the training such that it stops iterating over the training set once the total MSE has converged. This was not implemented because it isn't a requirement of the system.

Each sample in the training-set consists of two elements. The first element is the values for the features, $data[0]$, and the second element is the class number, zero indexed. This means that $\mathbf{T}[data[1]]$ gets the correct target vector since $data[1]$ indicates what class the sample is (0 - iris, 1 - versicolor, 2 - virginica), given that $\mathbf{T}$ is an identity matrix.

For every sample we calculate the discriminant functions $\mathbf{g}(\mathbf{x})$ and the value $\mathbf{u}$. We use the value $\mathbf{u}$ differently to find the MSE for the weights and the biases as explained earlier. The gradient MSE wrt. the weights is given as the outer product between $\mathbf{u}$ and $\mathbf{x}$, the gradient MSE wrt. the biases are given as $\mathbf{u}$. This is seen from equations (9) and (10). Since we use online learning, the weights and biases are adjusted for each sample in the training-set as shown in equations (11) and (12).

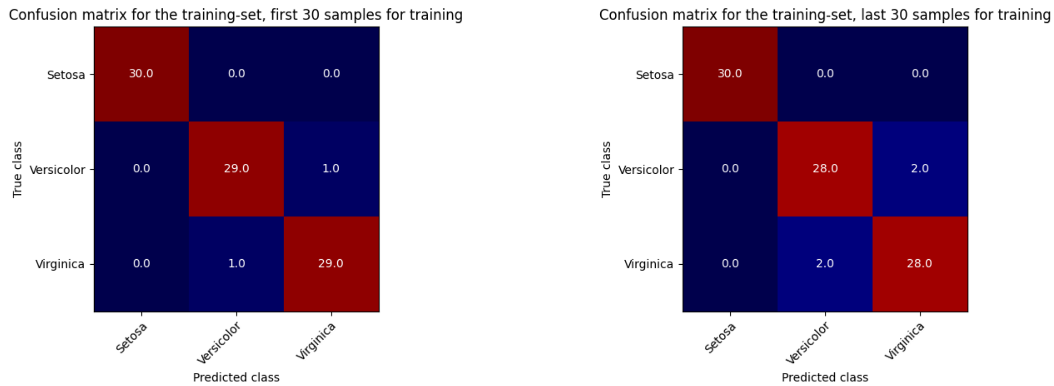
Testing

Appendix F shows how the thirty first and thirty last samples are used for training in the functions `training_30_first_samples()` and `training_30_last_samples()` respectively. The testing is done as shown in function `testing()` and the plotting of the confusion matrices is done in `plotting_confusion_matrix()`.

The samples in the testing-set consists of two elements the same way that each sample in the training-set does. This is to keep track of the true label for comparison when classifying.

When the sample from the testings-set is in calculating $\mathbf{g}$, the sample is classified as the class with the largest value in $\mathbf{g}$. This is shown in equation (13).

Figure 5a and 5b shows the confusion matrices for the training-set when the thirty first and thirty last samples are used for training respectively.



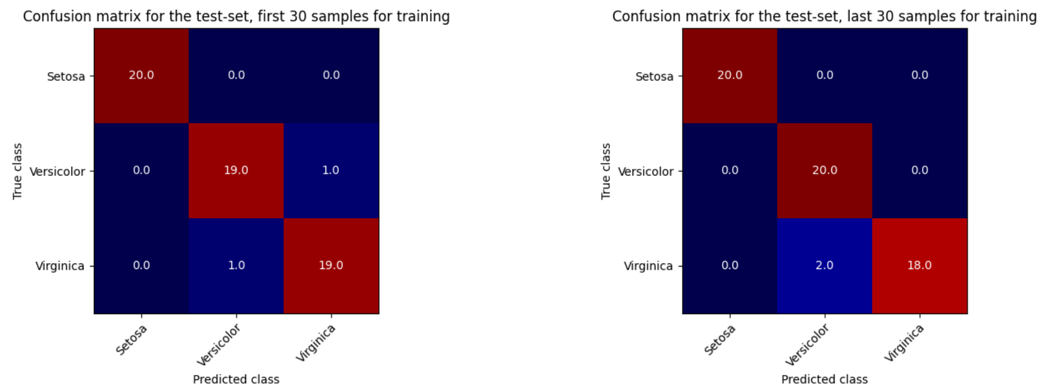
(a) Confusion matrix for the training-set, thirty first samples used for training.

(b) Confusion matrix for the training-set, thirty last samples used for training.

Figure 5: Confusion matrix for the training-set.

The error-rates are 2.22% and 4.44% respectively.

Figure 6a and 6b shows the confusion matrices, but for the testing-set, which consists of the remaining twenty samples.



(a) Confusion matrix for the testing-set, 30 first samples used for training.

(b) Confusion matrix for the testing-set, 30 last samples used for training.

Figure 6: Confusion matrix for the testing-set.

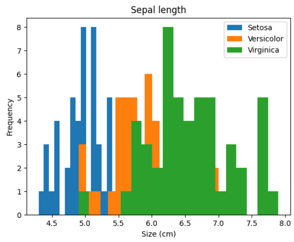
The error-rates are 3.33% for both tests.

The error-rate for the training-set is the lowest when using the first thirty samples of the training-set. The reason for this is that the linear classifier has been trained using this set, thus performs better when classifying the samples. The last thirty samples of the training-set performs worse than both cases of the testing-set. The reason for this can be that the last thirty samples of the training-set have features that are more difficult to distinguish than the samples in the testing-set. Thus even if the classifier has been trained using this set, it still struggles to re-classify the samples.

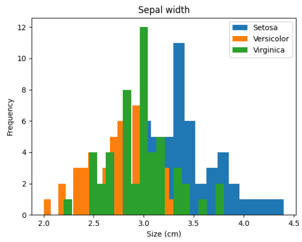
For the remaining tests, the first thirty samples are used for training and the rest for testing.

Removing features

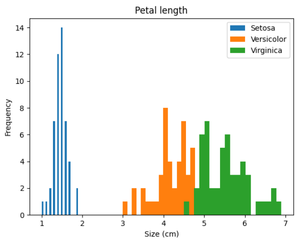
The histograms for the features are shown in figures 7a, 7b, 7c and 7d.



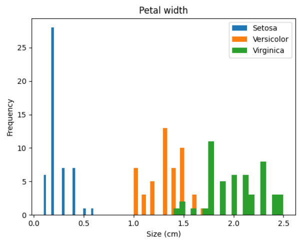
(a) Histogram of the sepal-length for all three classes.



(b) Histogram of the sepal-width for all three classes.



(c) Histogram of the petal-length for all three classes.



(d) Histogram of the petal-width for all three classes.

Figure 7: Histogram of the features.

As the histograms show, there is a lot more overlap for the sepal length and width than for the petal length and width.

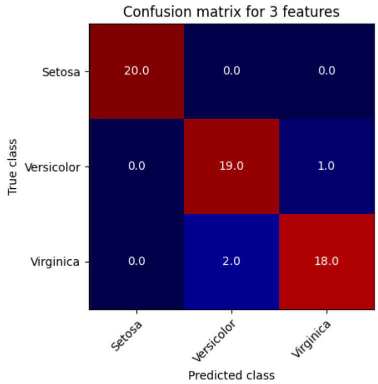
Since we start by removing one feature, we remove the sepal width since it has the most overlap. Thus, the three features that we use are the sepal length, petal length and the petal width.

When we remove one more feature, we remove the sepal length. This means that when we are using two features, we use the petal length and width.

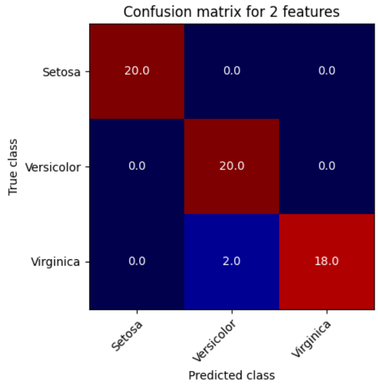
The last feature that we remove is the petal width since it seems to have slightly more overlap between the versicolor and virginica flowers than the petal length. Thus we use only the petal length.

The complete implementation for separating the features, plotting the histograms and testing using specified features is shown in appendix F. The functions are `separating_and_plotting()` and `three_features()`, `two_features()` and `one_feature()` respectively.

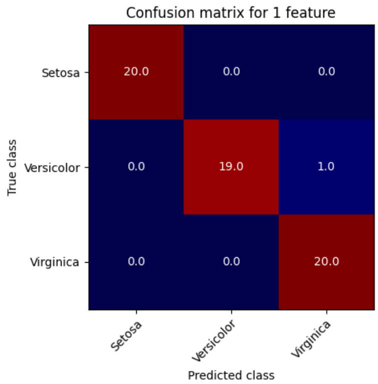
The confusion matrix when 3 features are used is shown in 8a and it has an error-rate of 5%. The confusion matrix when 2 features are used is shown in 8b and it has an error-rate of 3.33%. The confusion matrix when 1 feature is used is shown in 8c and it has an error-rate of 1.67%.



(a) Confusion matrix using 3 features.



(b) Confusion matrix using 2 features.



(c) Confusion matrix using 1 feature.

Figure 8: Confusion matrices using reduced amount of features.

The error-rate goes up when only three features are used, but goes down again when more features are removed. The reason for this is that there is still some overlap between features when only three features are used, but lower dimensionality. Thus classification becomes increasingly more difficult. As more features are removed, the density increases with little to no overlap and the logical result is an increased error-rate. The error-rate when only using the petal length is the lowest across all tests. The reason for this is the insignificant overlap between classes as explained.

As mentioned in section 2, linear separability means that the classes in the feature space can be divided by a straight line or hyperplanes. The Curse of Dimensionality [4] that was mentioned before is also related to the problem of linear separability. For higher dimensions of the feature space, the boundaries between classes can become more complex. Thus it can become more difficult to separate the classes linearly. This, as explained before, might be the reason why the classifier performs better for fewer features.

One can see from figure 14 that the setosa class is, or at least close, to linearly separable from the other classes for all features. Thus, the more separable the versicolor and virginica class becomes, the lower the error-rate becomes.

The classifier is still not error-free when using one or two features. This is due to the small overlap between the versicolor and virginica petal length and width. Hence, all classes might not be linearly separable when looking at one or both of these two features.

4.2 Digit Recognition-task

This subsection will describe the main parts of the implementation of the digit recognition-task. This task is implemented using MATLAB. The code is shown in appendix G.

Flow-chart

A general flow-chart for the implementation of the digit recognition-task is shown in figure ..

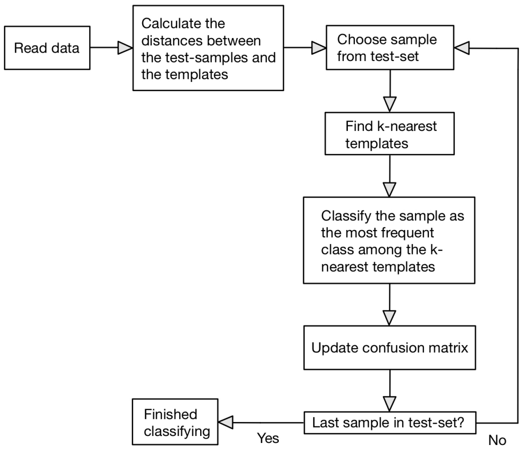


Figure 9: General flow-chart for the Digit Recognition-task.

Calculating distances

The templates are saved in a 60000x784-matrix and the test data is saved in a 10000x784-matrix. We wish to compute the distance between all 10000 samples to all 60000 templates. This should be saved in a 60000x10000-matrix.

We calculate the distance as shown in equation (14). The calculation of distance between two matrices is implemented in the function `calculate_distance()` shown in appendix G

A straight-forward implementation of equation (14) is by using the MATLAB-function `dist()`. This function calculates all the distances in one line and returns a matrix containing the distances.

Sorting and clustering

To sort the templates into new arrays based on the class the sample belongs to, the function `sorting()` is used. This function returns a 1x10-cell, where each cell contains an array of templates of only one class.

Clustering is done to create a smaller, new set of templates that adequately represents the class. One algorithm that can be used for clustering is the k-means clustering algorithm. This algorithm divides the data-set into M clusters in the feature space. It then iterates and sorts the data-points into the clusters until it has achieved the minimum within-cluster variance. The data-set is all samples of only one class.

To use k-means clustering in MATLAB, the function `kmeans()` is used. The `kmeans()` function gets the cluster centroid locations of the M clusters. This function is then used on the sorted classes with $M = 64$ to get the 64 new templates based on the clutser centroid locations of each class. These 64 templates of each class will then be samples that together best represents the class, since it can also cover outliers.

The templates selected using k-means clustering are added to a new matrix called *new_training_set* for all classes. This will then be the new set of templates used for classifying.

Appendix C shows the 64 cluster centroid samples for all classes. One can see that some of the numbers within a class look relatively similar, but it also contains the different typical ways to write a digit.

Classifying

This paragraph describes the process of classifying a single sample using a kNN-classifier.

Firstly, the sorted indices from smallest distance to the largest is found using the `sort()` function on the distance-set. Then the labels of the k-smallest distances is added to the vector min_labels. The sample is classified as the most frequent label in min_labels, which is found using the `mode()` function.

If two labels are equally frequent, we should pick the class which is the closest to the sample. However, the `mode()` function picks the class with the smallest label, which is not necessarily the best option. Thus the classifier can be further improved.

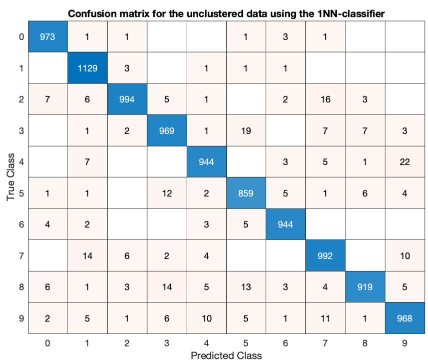
The confusion matrix is updated by incrementing the value at (true label, classified label), i.e. $cm(tl + 1, pl + 1) = cm(tl + 1, pl + 1) + 1;$. Here cm is the confusion matrix, tl and pl are the true and predicted (classified) label respectively. The +1 is needed because MATLAB is not zero-indexed.

Since it is desirable to plot correctly and wrongly classified samples, they are saved to the arrays wd, wl, cd and cl, which are data and labels for the wrongly and correctly classified samples respectively. The amount of wrongly and correctly classified samples are saved to w and c and are used to compute the error-rate.

The implementation of the NN-classifier is done as general as possible with the possibility of an arbitrary k-value. Thus, if it is desirable to only use a 1NN-classifier, the only adjustment needed is to use $k = 1$ when calling the function. In the same way, $k = 7$ gives a 7NN-classifier.

Tests and results

Figure 10a and 10b shows the confusion matrix for the 1NN-classifier for the unclustered and clustered templates respectively.



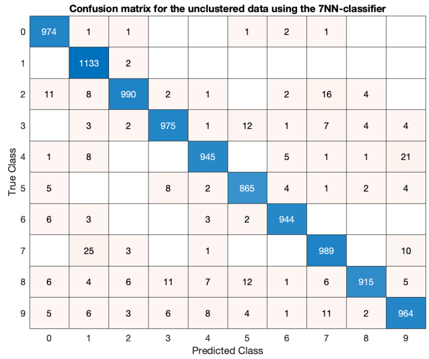
(a) Confusion matrix, 1NN using unclustered templates.

(b) Confusion matrix, 1NN using clustered templates.

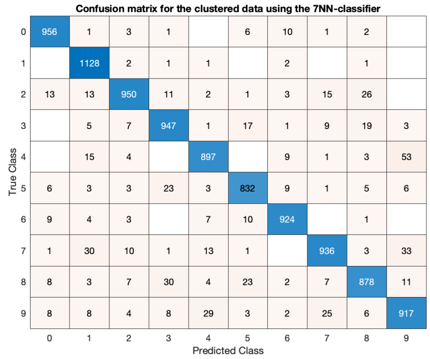
Figure 10: Confusion matrices using the 1NN-classifier.

The error-rates are 3.09% and 4.82% respectively.

Figure 11a and 11b shows the confusion matrix for the 7NN-classifier for the unclustered and clustered templates respectively.



(a) Confusion matrix, 7NN using unclustered templates.



(b) Confusion matrix, 7NN using clustered templates.

Figure 11: Confusion matrices using the 7NN-classifier.

The error-rates are 3.06% and 6.35% respectively.

The error-rates are lower when the unclustered templates are used, which is to be expected. This is because we have more templates for comparison with the sample we want to classify. There is however an advantage to use the clustered templates when looking at the processing times. When using all the templates, the total processing time, which means calculating the distances and classifying, was around 30 minutes. However, when using the clustered templates, the processing time was reduced down to about 5 minutes including sorting, clustering, calculating distances and classifying. When taking the processing time into account, the increased error-rate might be a satisfactory trade-off.

We can also see that the 7NN-classifier performs slighty better for the unclustered templates and considerably worse for the clustered templates in comparison with the 1NN-classifier. An explanation for this can be found by looking at the clustered templates in appendix C. A lot of these templates vary a lot in the way the digit oriented and in shaping within a class. Thus, even if the nearest template is of the same class as the sample and the others are not, it will misclassify. Another reason that the 7NN-classifier performs worse when clustering, is that the k might be too high such that the classifier is overfitting. A possible improvement is to reduce k.

It is interesting to view some of the correctly and the wrongly classified digits to see if they are reasonably classified.

Figure 25, 26, 27 and 28 from appendix D show respectively three random digits that were correctly classified for the 1NN-classifier using unclustered and clustered templates, and also unclustered and clustered for the 7NN-classifier.

Almost all of the correctly classified digits are reasonable. However, the third digit shown in figure 27 does not resemble the digit 6 in my opinion. In any case, it was classified correctly. This indicates that the classifier might perform well for even badly drawn digits.

Figure 29, 30, 31 and 32 from appendix E show three randomly selected digits that were wrongly classified for the 1NN-classifier using unclustered and clustered templates and for the 7NN-classifier respectively. For the wrongly classified digits, there is a more varying degree of reasonability.

For example the third digit in figure 30 might seem unreasonably misclassified. However, there are still some arguments as to why it was classified this way. If one compares the misclassified sample with the clustered templates in figure 21, one can see that there isn't a template that resembles this shape. When using the unclustered templates, misclassification most likely occurs due to the classified sample having an orientation, sizing or shape that is unusual for that class.

One of the main issues with this implementation of the kNN-classifier is that it does not take into account the shape of the digit. It only compares each single pixel with the respective pixel of the templates. One way to incorporate the shape of the digit in the classification, is to use a neural network instead of a kNN-classifier.

5 Conclusion

In this project there has been designed a linear classifier for classifying types of irises. There has also been designed a kNN-classifier for recognizing digits. The linear classifier has been designed in Python and uses weights that are adjusted using a training-set and gradient decent with the MSE as a cost function. The kNN-classifier has been designed in MATLAB and uses the distance between a sample and templates to classify the sample. This has been tested using all templates and k-means clustering of the templates.

The linear classifier performs the best on the training-set when the first thirty samples are used for training, but performs equally on the testing-set. The error-rate when using the thirty first samples for training are 2.22% and 3.33% for the training-set and testing-set respectively. When using the thirty last samples for training, the error-rates are 4.44% and 3.33% for the training-set and testing-set. When reduced to two, and then one feature, the classifier performs the best with the error-rates 3.33% and 1.67% respectively. Most likely due to linear separability.

The performance might be satisfactory for this task, but can be further improved. This is due to the features not being linearly separable for all classes, thus another classifier could be used.

The kNN-classifier performs the best when the unclustered templates are used. Clustering increases the error-rate from 3.09% and 3.06% for the 1NN and 7NN to 4.82% and 6.35% for the 1NN and 7NN respectively. However, the processing time was also reduced from 30 minutes to 5 minutes when using clustering.

An issue with the kNN-classifier is that it only compares each respective pixel between the sample and templates. To incorporate the shape of the digit aswell, one could use a neural network instead.

Bibliography

[1] R. O. Duda, P. E. Hart and D. G. Stork. *Pattern Classification*. John Wiley & Sons, Inc, 2001.

[2] Magne H. Johnsen. *Classification*. Dec. 2017.

[3] Adel Nasri and Xianfeng Huang. 'Images Enhancement of Ancient Mural Painting of Bey's Palace Constantine, Algeria and Lacuna Extraction Using Mahalanobis Distance Classification Approach'. In: *Sensors* 22 (Sept. 2022), p. 6643. DOI: 10.3390/s22176643.

[4] P. S. Rossi. *Classification and Classification Systems*. Unpublished lecture notes.

Appendix

A Figures

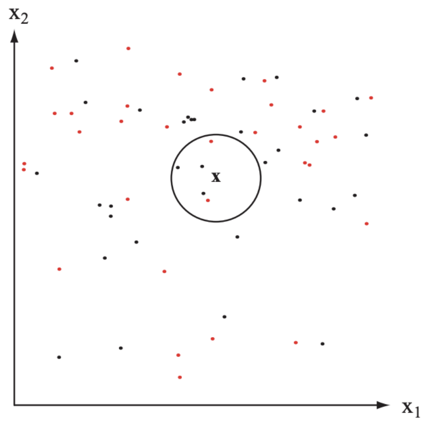


Figure 4.15: The k-nearest-neighbor query starts at the test point and grows a spherical region until it encloses k training samples, and labels the test point by a majority vote of these samples. In this $k = 5$ case, the test point $\mathbf{x}$ would be labelled the category of the black points.

Figure 12: Figure borrowed from [1, p. 183].

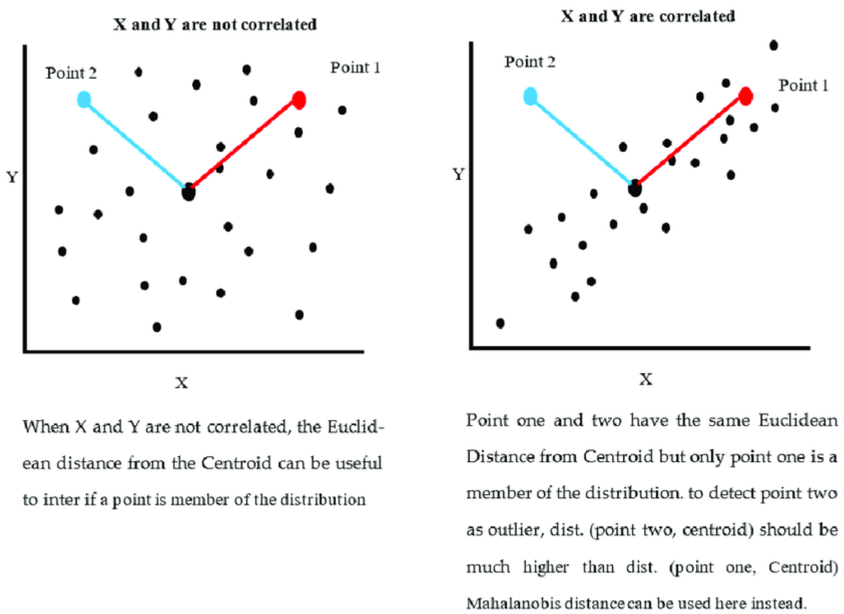
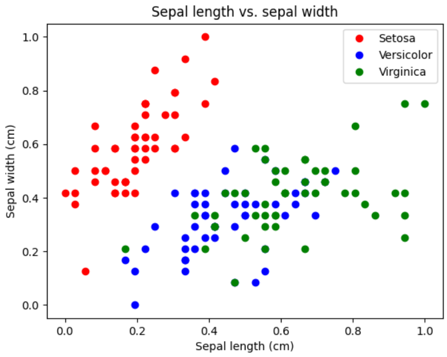
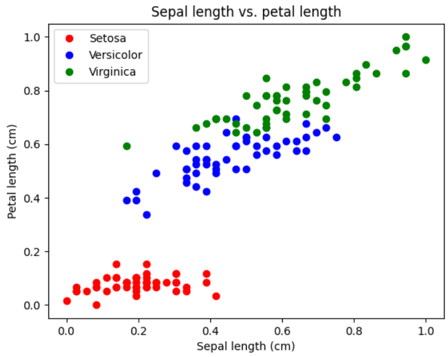


Figure 13: Figure showing the difference between euclidean and mahalanobis distance. Borrowed from [3]

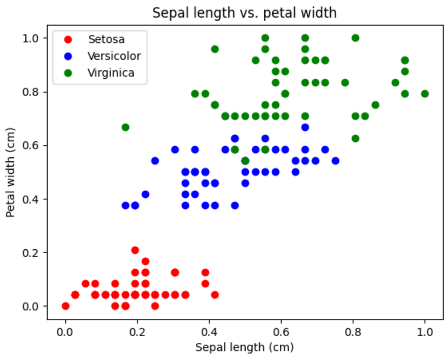
B All 2D-feature spaces - Iris-task



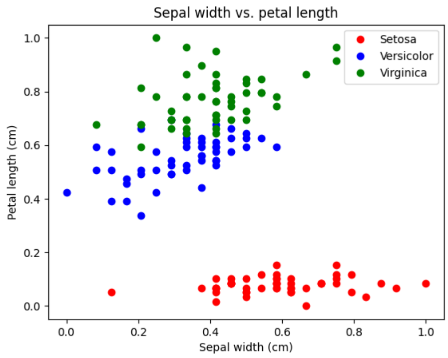
(a) 2D-feature space for the normalized sepal length and sepal width.



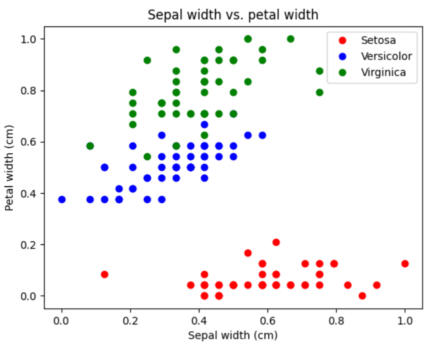
(b) 2D-feature space for the normalized sepal length and petal length.



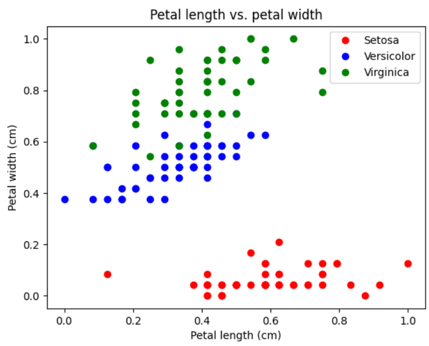
(c) 2D-feature space for the normalized sepal length and petal width.



(d) 2D-feature space for the normalized sepal width and petal length.



(e) 2D-feature space for the normalized sepal width and petal width.



(f) 2D-feature space for the normalized petal length and petal width.

Figure 14: Plots of all 2D-feature spaces for the normalized data.

C Clustered templates - Digit Recognition-task

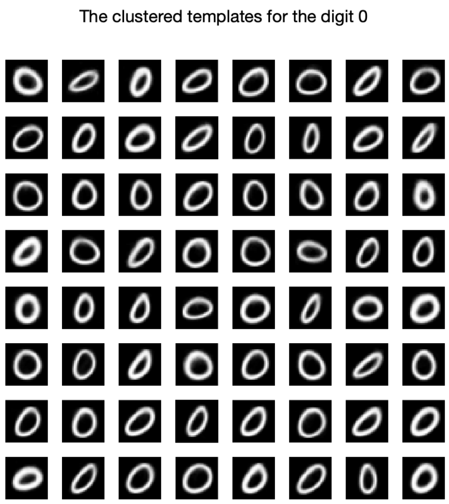


Figure 15: The 64 cluster centroid samples for the clustering of the digit 0.

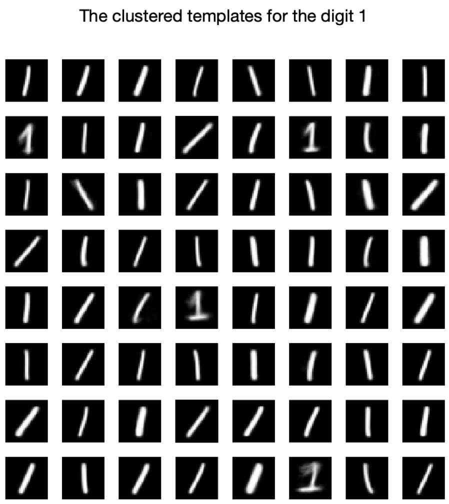


Figure 16: The 64 cluster centroid samples for the clustering of the digit 1.

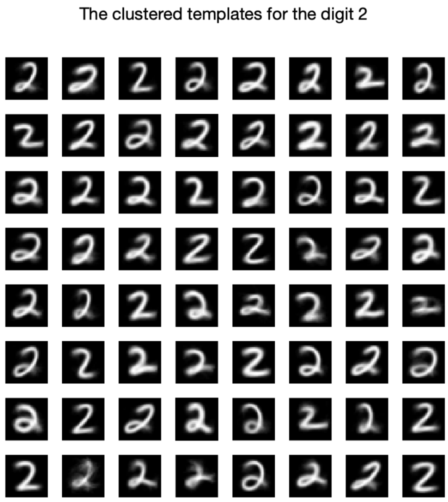


Figure 17: The 64 cluster centroid samples for the clustering of the digit 2.

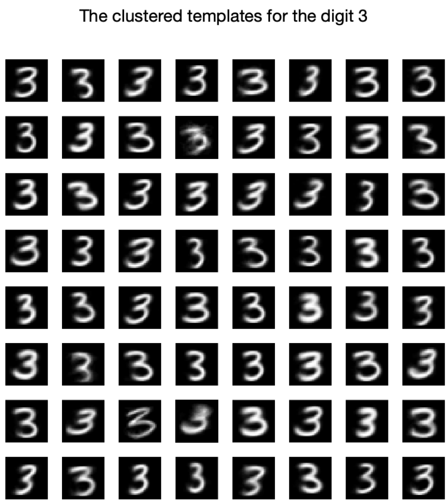


Figure 18: The 64 cluster centroid samples for the clustering of the digit 3.

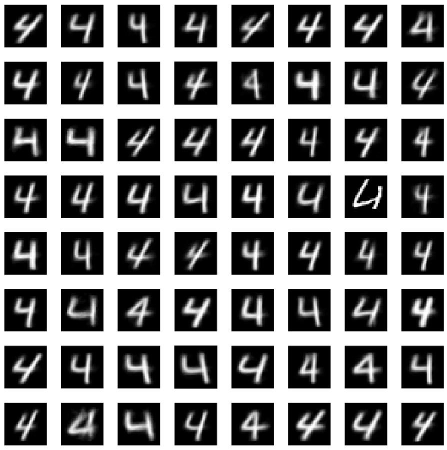


Figure 19: The 64 clutser centroid samples for the clustering of the digit 4.

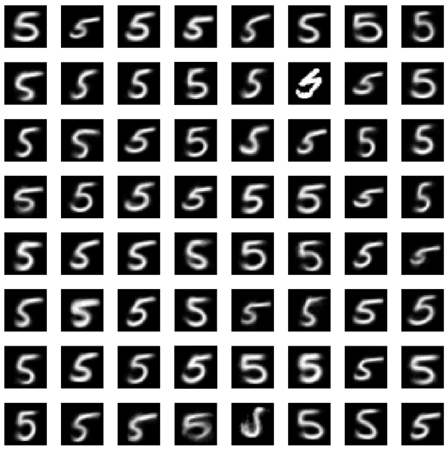


Figure 20: The 64 cluster centroid samples for the clustering of the digit 5.

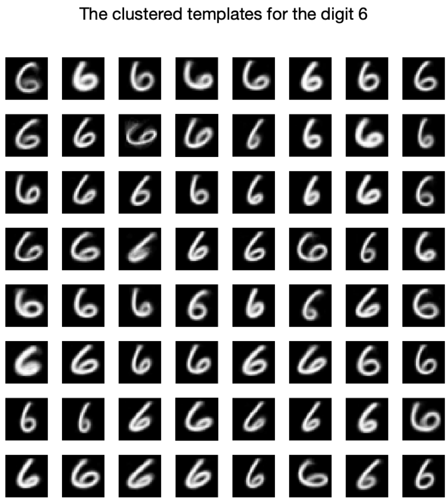


Figure 21: The 64 cluster centroid samples for the clustering of the digit 6.

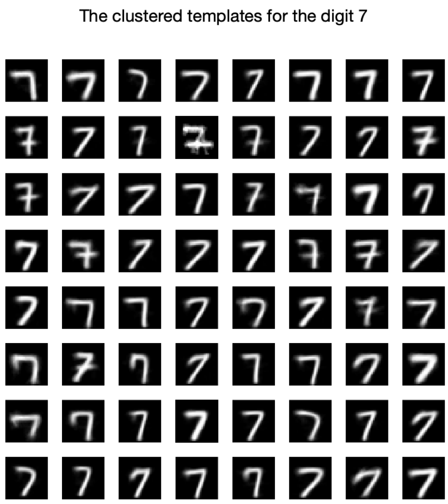


Figure 22: The 64 cluster centroid samples for the clustering of the digit 7.

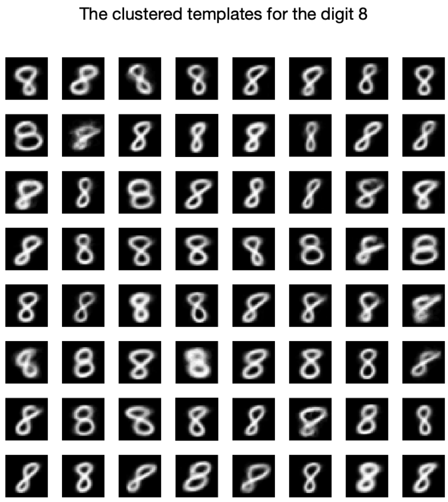


Figure 23: The 64 cluster centroid samples for the clustering of the digit 8.

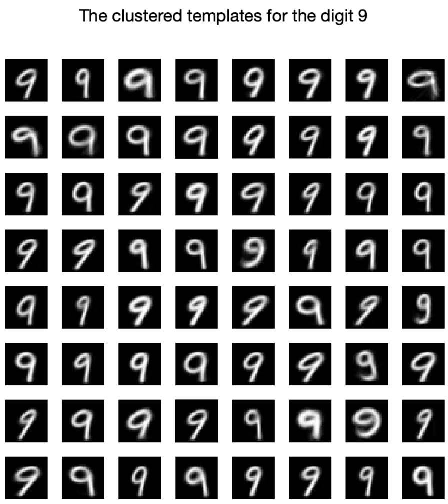


Figure 24: The 64 cluster centroid samples for the clustering of the digit 9.

D Correctly Classified Digits - Digit Recognition-task

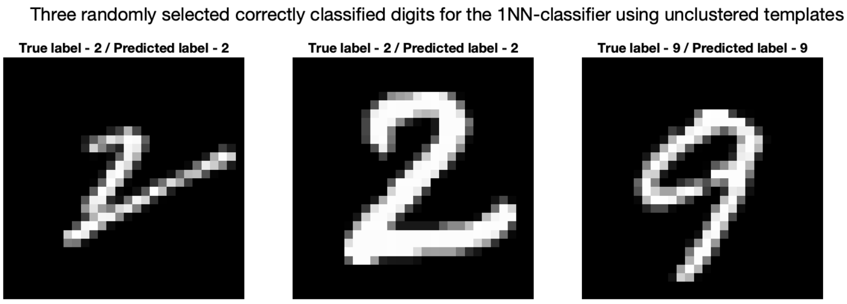


Figure 25: Three randomly selected digits that were classified correct using the 1NN-classifier with unclustered templates.

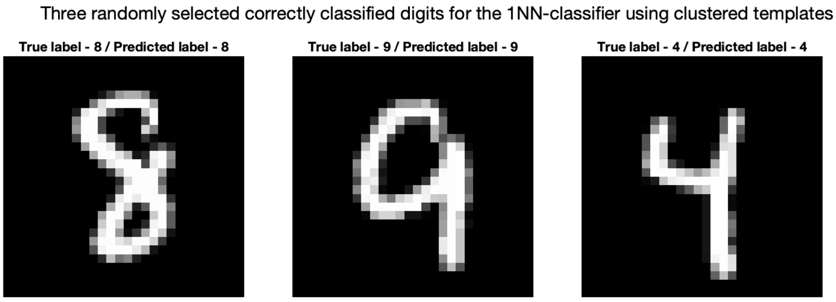


Figure 26: Three randomly selected digits that were classified correct using the 1NN-classifier with clustered templates.

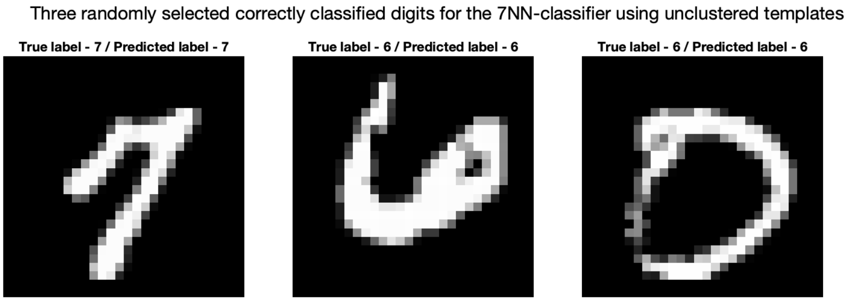


Figure 27: Three randomly selected digits that were classified correct using the 7NN-classifier with unclustered templates.

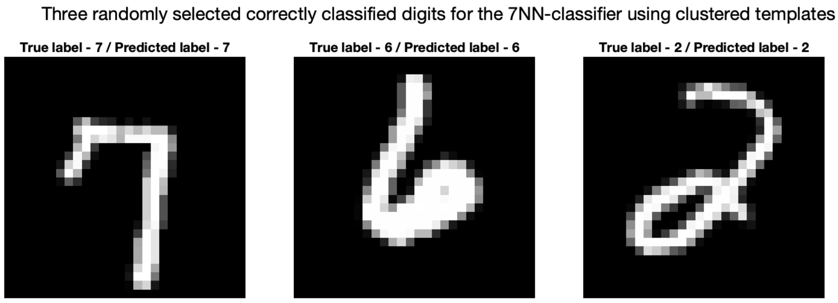


Figure 28: Three randomly selected digits that were classified correct using the 7NN-classifier with clustered templates.

E Wrongly Classified Digits - Digit Recognition-task

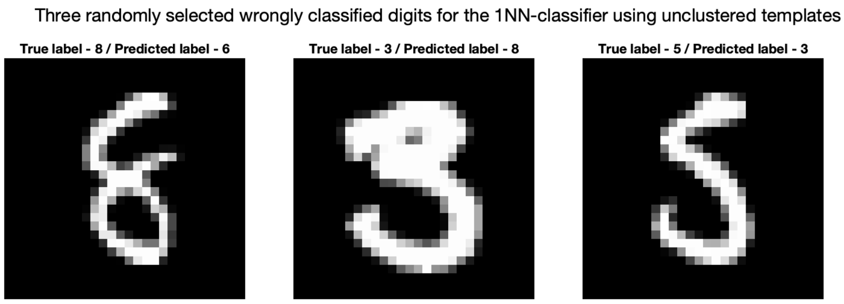


Figure 29: Three randomly selected digits that were classified wrong using the 1NN-classifier with unclustered templates.

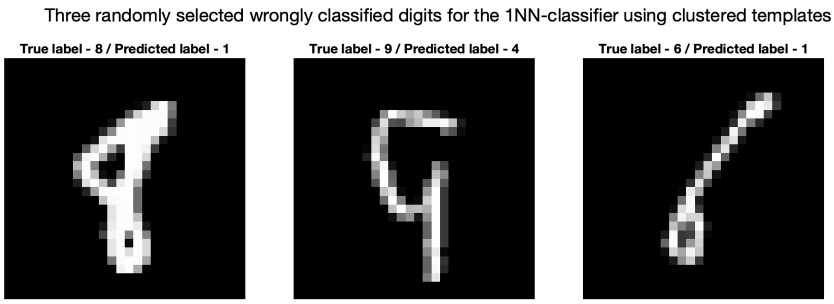


Figure 30: Three randomly selected digits that were classified wrong using the 1NN-classifier with clustered templates.

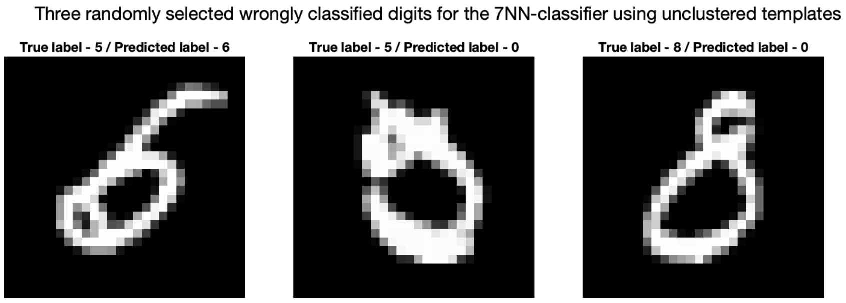


Figure 31: Three randomly selected digits that were classified wrong using the 7NN-classifier with unclustered templates.

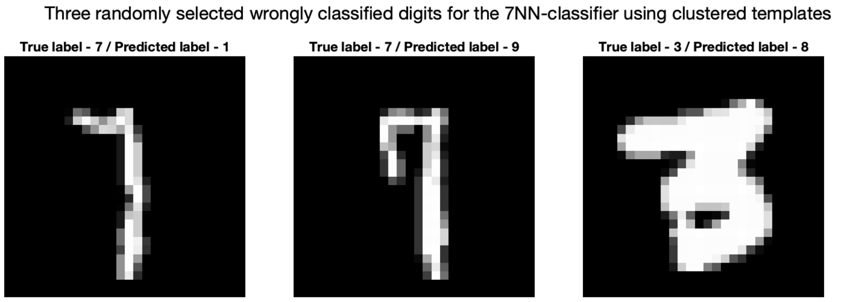


Figure 32: Three randomly selected digits that were classified wrong using the 7NN-classifier with clustered templates.

F Code - Iris-task

```

import numpy as np
import matplotlib.pyplot as plt
import copy

# Defining constants

N_CLASSES = 3 # Number of classes
N = 50 # Number of samples in each class

# Load the data for the Iris classes
# The data lines are stores in the order: sepal length, sepal width, petal length, petal width - All in cm
setosa = np.genfromtxt("Iris_TTT4275/class_1", delimiter=",")
versicolor = np.genfromtxt("Iris_TTT4275/class_2", delimiter=",")
virginica = np.genfromtxt("Iris_TTT4275/class_3", delimiter=",")

all_samples = np.vstack((setosa, versicolor, virginica))


'''
Ex. of finding the max and min of a specific feature
max_sepal_length = max([x[0] for x in all_samples])
```

```
min_sepal_length = min([x[0] for x in all_samples])
'''

# Want to keep the original data for plotting later
setosa_unormalized, versicolor_unormalized, virginica_unormalized = copy.deepcopy(setosa), copy.deepcopy(versicolor), copy.deepcopy(virginica)

def normalization(flower_set, samples):
    for flower in flower_set:
        flower[0] = (flower[0] - min([x[0] for x in all_samples]))/(max([x[0] for x in all_samples]) - min([x[0] for x in all_samples]))
        flower[1] = (flower[1] - min([x[1] for x in all_samples]))/(max([x[1] for x in all_samples]) - min([x[1] for x in all_samples]))
        flower[2] = (flower[2] - min([x[2] for x in all_samples]))/(max([x[2] for x in all_samples]) - min([x[2] for x in all_samples]))
        flower[3] = (flower[3] - min([x[3] for x in all_samples]))/(max([x[3] for x in all_samples]) - min([x[3] for x in all_samples]))

normalization(setosa, all_samples)
normalization(versicolor, all_samples)
normalization(virginica, all_samples)

def plotting(setosa_set, versicolor_set, virginica_set, title1, title2, name1, name2):
    # Plotting the sepal width vs. petal width for the three classes
    plt.figure(1)
    plt.plot([x[1] for x in setosa_set], [x[3] for x in setosa_set], 'ro', label='Setosa')
    plt.plot([x[1] for x in versicolor_set], [x[3] for x in versicolor_set], 'bo', label='Versicolor')
    plt.plot([x[1] for x in virginica_set], [x[3] for x in virginica_set], 'go', label='Virginica')
    plt.xlabel('Petal length (cm)')
    plt.ylabel('Petal width (cm)')
    plt.title('Petal length vs. petal width')
    plt.title(title1)
    plt.legend(['Setosa', 'Versicolor', 'Virginica'])
    # name = 'petal_l_vs_petal_w_normalized'
    # plt.savefig('Plots/Iris_Features/' + name + ".png")
    plt.show()

# plotting(setosa, versicolor, virginica, 'Sepal length vs. sepal width', 'Petal length vs. petal width', 'sepal_l_vs_sepal_w_normalized', 'petal_l_vs_petal_w_normalized')
# plotting(setosa_unormalized, versicolor_unormalized, virginica_unormalized, 'Sepal length vs. sepal width', 'Petal length vs. petal width', 'sepal_l_vs_sepal_w_unormalized', 'petal_l_vs_petal_w_unormalized')

T = [[1, 0, 0],
     [0, 1, 0],
     [0, 0, 1]] # Target vectors

# Training the network but updating the weights and bias after each training input
def training(set_for_training, M = 5000, alpha = 0.3):
    # Creating a weighting matrix and a bias vector, starting with random values between 0 and 1
```

```python
    w_matrix = np.random.random((N_CLASSES, len(set_for_training[0][0]))) # Weights
    w0 = np.random.random(N_CLASSES) # Bias
    for m in range(M):
        np.random.shuffle(set_for_training) # Randomize the training set for each iteration
        # Training the network for all training inputs, shuffled
        for data in set_for_training:
            t = T[data[1]]
            x = data[0]
            g = sigmoid(np.matmul(w_matrix, np.transpose(x)) + w0)
            u = np.multiply(np.multiply((g-t), g), (1-g))

            w_matrix -= alpha*np.outer(u, x) # Updating the weights with the error of the weights
            w0 -= alpha*u # Updating the bias with the error of the bias

    return [w_matrix, w0]

def sigmoid(x):
    return 1/(1+np.exp(-x))

# Training the network for all training inputs for M iterations
iterations = 1000
learning_rate = 0.6

def testing(testing_set, weights):
    confusion_matrix = np.zeros((N_CLASSES, N_CLASSES))
    wrong = 0

    for test_sample in testing_set:
        true_class = test_sample[1]
        sample = [np.transpose(test_sample[0]), 1]
        g = 1/(1+np.exp(-np.matmul(weights[0], sample[0]) - weights[1]*sample[1]))
        predicted_class = np.argmax(g)
        confusion_matrix[true_class][predicted_class] += 1
        if predicted_class != true_class:
            wrong += 1
    return confusion_matrix, wrong

def plotting_confusion_matrix(confusion_matrix, title, name):
    # Plotting the confusion matrix
    fig, ax = plt.subplots()
    im = ax.imshow(confusion_matrix, cmap="seismic")
    ax.set_xticks(np.arange(N_CLASSES))
    ax.set_yticks(np.arange(N_CLASSES))
    ax.set_xticklabels(["Setosa", "Versicolor", "Virginica"])
    ax.set_yticklabels(["Setosa", "Versicolor", "Virginica"])
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")
    for i in range(N_CLASSES):
        for j in range(N_CLASSES):
            text = ax.text(j, i, confusion_matrix[i, j],
                           ha="center", va="center", color="w")
    ax.set_title(title)
    ax.set_xlabel("Predicted class")
    ax.set_ylabel("True class")
```

```python
    fig.tight_layout()
    # plt.savefig(name)
    plt.show()

#---------------------------
# Task 1
#---------------------------

print("Starting task 1")

# Using the first 30 samples for training and the last 20 for testing
def training_30_first_samples():
    N_TRAINING = 30
    # Creating a set for training
    training_set = []
    for setosa_data in setosa[:N_TRAINING]: training_set.append([setosa_data, 0])
    for versicolor_data in versicolor[:N_TRAINING]: training_set.append([versicolor_data, 1])
    for virginica_data in virginica[:N_TRAINING]: training_set.append([virginica_data, 2])
    # Creating a set for testing
    testing_set = []
    for setosa_data in setosa[N_TRAINING:]: testing_set.append([setosa_data, 0])
    for versicolor_data in versicolor[N_TRAINING:]: testing_set.append([versicolor_data, 1])
    for virginica_data in virginica[N_TRAINING:]: testing_set.append([virginica_data, 2])

    weights = training(training_set, iterations, learning_rate)

    confusion_matrix_testing, wrong_testing = testing(testing_set, weights)
    confusion_matrix_training, wrong_training = testing(training_set, weights)

    print("Using first 30 samples for training, 20 last samples for testing")

    print(f"Confusion matrix for test-set: \n{confusion_matrix_testing}")
    print(f"Confusion matrix for train-set: \n{confusion_matrix_training}")

    error_rate_testing = wrong_testing/len(testing_set)
    error_rate_training = wrong_training/len(training_set)
    print(f"Error rate for test-set: {error_rate_testing}")
    print(f"Error rate for training-set: {error_rate_training}\n")

    plotting_confusion_matrix(confusion_matrix_testing, "Confusion matrix for the test-set, first 30 samples for training", "Plots/Iris_Foerste_Utkast/Confusion_matrix_30_first_testing.png")
    plotting_confusion_matrix(confusion_matrix_training, "Confusion matrix for the training-set, first 30 samples for training", "Plots/Iris_Foerste_Utkast/Confusion_matrix_30_first_training.png")

# Using the last 30 samples for training and the first 20 for testing
def training_30_last_samples():
    N_TESTING = 20
    # Creating a set for training
    training_set = []
    for setosa_data in setosa[N_TESTING:]: training_set.append([setosa_data, 0])
    for versicolor_data in versicolor[N_TESTING:]: training_set.append([versicolor_data, 1])
```

```
    for virginica_data in virginica[N_TESTING:]: training_set.append([virginica_data, 2])
    # Creating a set for testing
    testing_set = []
    for setosa_data in setosa[:N_TESTING]: testing_set.append([setosa_data, 0])
    for versicolor_data in versicolor[:N_TESTING]: testing_set.append([versicolor_data, 1])
    for virginica_data in virginica[:N_TESTING]: testing_set.append([virginica_data, 2])

    weights = training(training_set, iterations, learning_rate)

    confusion_matrix_testing, wrong_testing = testing(testing_set, weights)
    confusion_matrix_training, wrong_training = testing(training_set, weights)

    print("Using last 30 samples for training, 20 first samples for testing")

    # print(f"Wrong: {wrong}, Total: {len(testing_set)}")
    print(f"Confusion matrix for test-set: \n{confusion_matrix_testing}")
    print(f"Confusion matrix for train-set: \n{confusion_matrix_training}")

    error_rate_testing = wrong_testing/len(testing_set)
    error_rate_training = wrong_training/len(training_set)
    print(f"Error rate for test-set: {error_rate_testing}")
    print(f"Error rate for training-set: {error_rate_training}\n")

    plotting_confusion_matrix(confusion_matrix_testing, "Confusion matrix for the test-set, last 30 samples for training", "Plots/Iris_Foerste_Utkast/Confusion_matrix_30_last_testing.png")
    plotting_confusion_matrix(confusion_matrix_training, "Confusion matrix for the training-set, last 30 samples for training", "Plots/Iris_Foerste_Utkast/Confusion_matrix_30_last_training.png")

training_30_first_samples()
training_30_last_samples()

print("Finished task 1\n")

#---------------------------
# Task 2
#---------------------------

print("Starting task 2")

def separating_and_plotting():
    # Getting all features into one array
    setosa_features = np.zeros((4, N)) # [[all sepal length], [all sepal width], [all petal length], [all petal width]]
    versicolor_features = np.zeros((4, N))
    virginica_features = np.zeros((4, N))

    for i in range(N):
        setosa_features[0][i], setosa_features[1][i] = setosa_unormalized[i][0], setosa_unormalized[i][1]
        setosa_features[2][i], setosa_features[3][i] = setosa_unormalized[i][2], setosa_unormalized[i][3]

```

```python
        versicolor_features[0][i], versicolor_features[1][i] = versicolor_unormalized[i][0], versicolor_unormalized[i][1]
        versicolor_features[2][i], versicolor_features[3][i] = versicolor_unormalized[i][2], versicolor_unormalized[i][3]

        virginica_features[0][i], virginica_features[1][i] = virginica_unormalized[i][0], virginica_unormalized[i][1]
        virginica_features[2][i], virginica_features[3][i] = virginica_unormalized[i][2], virginica_unormalized[i][3]

    plot_histograms([setosa_features[0], versicolor_features[0], virginica_features[0]], 'Sepal length', 'Plots/Iris_Foerste_Utkast/sepallength.png')
    plot_histograms([setosa_features[1], versicolor_features[1], virginica_features[1]], 'Sepal width', 'Plots/Iris_Foerste_Utkast/sepalwidth.png')
    plot_histograms([setosa_features[2], versicolor_features[2], virginica_features[2]], 'Petal length', 'Plots/Iris_Foerste_Utkast/petallength.png')
    plot_histograms([setosa_features[3], versicolor_features[3], virginica_features[3]], 'Petal width', 'Plots/Iris_Foerste_Utkast/petalwidth.png')


# Plot the histogram for the features of the three classes

def plot_histograms(features, title, name):
    plt.figure()
    plt.hist(features[0], bins=19, label='Setosa')
    plt.hist(features[1], bins=19, label='Versicolor')
    plt.hist(features[2], bins=19, label='Virginica')
    plt.xlabel('Size (cm)')
    plt.ylabel('Frequency')
    plt.title(title)
    plt.legend()
    # plt.savefig(name)
    plt.show()

N_TRAINING = 30

def three_features():
    # Creating the training and testing sets
    # Looks to be a good idea to use the petal length and petal width as features
    # Since we only remove one, sepal length is better than sepal width
    # Thus we use the elements 0, 2 and 3 of the arrays
    training_set_3_features = [[setosa_sample, 0] for setosa_sample in setosa[:N_TRAINING, [0, 2, 3]]]
    training_set_3_features += [[versicolor_sample, 1] for versicolor_sample in versicolor[:N_TRAINING, [0, 2, 3]]]
    training_set_3_features += [[virginica_sample, 2] for virginica_sample in virginica[:N_TRAINING, [0, 2, 3]]]

    testing_set_3_features = [[setosa_sample, 0] for setosa_sample in setosa[N_TRAINING:, [0, 2, 3]]]
    testing_set_3_features += [[versicolor_sample, 1] for versicolor_sample in versicolor[N_TRAINING:, [0, 2, 3]]]
```

```python
    testing_set_3_features += [[virginica_sample, 2] for virginica_sample in virginica[N_TRAINING:, [0, 2, 3]]]

    # weight_3_features = training(training_set_3_features, iterations, learning_rate)
    weight_3_features = training(training_set_3_features, iterations, learning_rate)

    confusion_matrix_3_features, wrong_3_features = testing(testing_set_3_features, weight_3_features)
    print(f"Confusion matrix for 3 features:\n{confusion_matrix_3_features}")
    print(f"Wrong predictions for 3 features: {wrong_3_features}")
    print(f"Error rate: {wrong_3_features/len(testing_set_3_features)}\n")

    plotting_confusion_matrix(confusion_matrix_3_features, "Confusion matrix for 3 features", "Plots/Iris_Foerste_Utkast/confusion_matrix_3_features.png")

def two_features():
    # Now removing two features (sepal length and sepal width)
    # Thus we use the elements 2 and 3 of the arrays
    training_set_2_features = [[setosa_sample, 0] for setosa_sample in setosa[:N_TRAINING, [2, 3]]]
    training_set_2_features += [[versicolor_sample, 1] for versicolor_sample in versicolor[:N_TRAINING, [2, 3]]]
    training_set_2_features += [[virginica_sample, 2] for virginica_sample in virginica[:N_TRAINING, [2, 3]]]

    testing_set_2_features = [[setosa_sample, 0] for setosa_sample in setosa[N_TRAINING:, [2, 3]]]
    testing_set_2_features += [[versicolor_sample, 1] for versicolor_sample in versicolor[N_TRAINING:, [2, 3]]]
    testing_set_2_features += [[virginica_sample, 2] for virginica_sample in virginica[N_TRAINING:, [2, 3]]]

    # weight_2_features = training(training_set_2_features, iterations, learning_rate)
    weight_2_features = training(training_set_2_features, iterations, learning_rate)

    confusion_matrix_2_features, wrong_2_features = testing(testing_set_2_features, weight_2_features)
    print(f"Confusion matrix for 2 features:\n{confusion_matrix_2_features}")
    print(f"Wrong predictions for 2 features: {wrong_2_features}")
    print(f"Error rate: {wrong_2_features/len(testing_set_2_features)}\n")

    plotting_confusion_matrix(confusion_matrix_2_features, "Confusion matrix for 2 features", "Plots/Iris_Foerste_Utkast/confusion_matrix_2_features.png")

def one_feature():
    # Now only using one feature (petal length)
    # Thus we use element 2 of the arrays
    training_set_1_features = [[setosa_sample, 0] for setosa_sample in setosa[:N_TRAINING, [2]]]
    training_set_1_features += [[versicolor_sample, 1] for versicolor_sample in versicolor[:N_TRAINING, [2]]]
    training_set_1_features += [[virginica_sample, 2] for virginica_sample in virginica[:N_TRAINING, [2]]]
```

```
    testing_set_1_features = [[setosa_sample, 0] for setosa_sample in
    ↪  setosa[N_TRAINING:, [2]]]
    testing_set_1_features += [[versicolor_sample, 1] for versicolor_sample in
    ↪  versicolor[N_TRAINING:, [2]]]
    testing_set_1_features += [[virginica_sample, 2] for virginica_sample in
    ↪  virginica[N_TRAINING:, [2]]]

    # weight_1_features = training(training_set_1_features, iterations,
    ↪  learning_rate)
    weight_1_features = training(training_set_1_features, iterations,
    ↪  learning_rate)

    confusion_matrix_1_features, wrong_1_features =
    ↪  testing(testing_set_1_features, weight_1_features)
    print(f"Confusion matrix for 1 feature:\n{confusion_matrix_1_features}")
    print(f"Wrong predictions for 1 feature: {wrong_1_features}")
    print(f"Error rate: {wrong_1_features/len(testing_set_1_features)}\n")

    plotting_confusion_matrix(confusion_matrix_1_features, "Confusion matrix for
    ↪  1 feature", "Plots/Iris_Foerste_Utkast/confusion_matrix_1_features.png")

three_features()
two_features()
one_feature()

print("Finished task 2\n")
```

G Code - Digit Recognition-task

```
disp("Loading data");
load('data_all.mat')

%%%------------------------------------
%%%------------------------------------
%%%             TASK 1
%%%------------------------------------
%%%------------------------------------

disp("----------------------");
disp("Beginning task 1");

tic

% Initializing task 1 variables

% Used in 1NN-classification
% w_1 & c_1 : Number of wrong and correct classifications
% cm_1 : Confusion-matrix
% wd_1 & wl_1 : Array containing data and labels respectively for wrongly
↪  classified images
% cd_1 & cl_1 : Same as the above, only with correctly classified images
% tp_lab_1 : An array that holds the true and predicted labels, used for
↪  plotting
% The label matrices contain [True label, Predicted label]
% Using deal to get all intialization on one line
```

```matlab
[w_1, c_1, cm_1, wd_1, wl_1, cd_1, cl_1, tp_lab_1] = deal(0, 0, zeros(10, 10), zeros(1, vec_size), zeros(1, 2), zeros(1, vec_size), zeros(1, 2), zeros(0, 2));

% Same as the above, only that they are used in the 7NN-classification
[w_1_7, c_1_7, cm_1_7, wd_1_7, wl_1_7, cd_1_7, cl_1_7, tp_lab_1_7] = deal(0, 0, zeros(10, 10), zeros(1, vec_size), zeros(1, 2), zeros(1, vec_size), zeros(1, 2), zeros(0, 2));

disp('Calculating distances and classifying');
size_bulk = 999; % 1000 - 1 (needs to be removed to account for indexing)
for i = 1:(num_test/size_bulk)
    % Splitting the testing-sets into 'bulks' of 1000 elements
    testing_data = testv(((i-1)*size_bulk+1):((i*size_bulk)+1), :);
    testing_labels = testlab(((i-1)*size_bulk+1):((i*size_bulk)+1), :);

    [number_of_tests, ~] = size(testing_data);
    % Calculating distances for the bulk of testing data
    % distances_set = calculate_distance(testing_data, trainv);
    distances_set = distances(:, ((i-1)*size_bulk+1):(i*size_bulk)+1);

    % The nn variable is an array holding index for the nearest neighbor
    % Only used for the 1NN-classifier
    % 1-NN classifier:
    [cm_1, wd_1, wl_1, w_1, cd_1, cl_1, c_1, tp_lab_1] = classify_kNN(distances_set, number_of_tests, testing_data, testing_labels, trainlab, cm_1, wd_1, wl_1, w_1, cd_1, cl_1, c_1, 1, tp_lab_1);
    % 7-NN classifier:
    [cm_1_7, wd_1_7, wl_1_7, w_1_7, cd_1_7, cl_1_7, c_1_7, tp_lab_1_7] = classify_kNN(distances_set, number_of_tests, testing_data, testing_labels, trainlab, cm_1_7, wd_1_7, wl_1_7, w_1_7, cd_1_7, cl_1_7, c_1_7, 7, tp_lab_1_7);
end

error_rate_1 = w_1/num_test;
disp("Error-rate for the 1NN-classifier for the unclustered data: " + error_rate_1);
error_rate_1_7 = w_1_7/num_test;
disp("Error-rate for the 7NN-classifier for the unclustered data: " + error_rate_1_7);

Plotting the confusion matrices
plot_confusion_matrix(tp_lab_1, "Confusion matrix for the unclustered data using the 1NN-classifier");
pause(5);
plot_confusion_matrix(tp_lab_1_7, "Confusion matrix for the unclustered data using the 7NN-classifier");
pause(5);

% 1NN
% Picking three random correctly classified digits to plot
[ri_1_c1, ri_1_c2, ri_1_c3] = deal(randi(length(cl_1), 1), randi(length(cl_1), 1), randi(length(cl_1), 1));
labels_c_1 = [cl_1(ri_1_c1, :); cl_1(ri_1_c2, :); cl_1(ri_1_c3, :)];
images_c_1 = [cd_1(ri_1_c1, :); cd_1(ri_1_c2, :); cd_1(ri_1_c3, :)];
plotting_3_images(images_c_1, labels_c_1, col_size, row_size, 'Three randomly selected correctly classified digits for the 1NN-classifier using unclustered templates');
```

```
% Picking three random wrongly classified digits to plot
[ri_1_w1, ri_1_w2, ri_1_w3] = deal(randi(length(wl_1), 1), randi(length(wl_1), 1), randi(length(wl_1), 1));
labels_w_1 = [wl_1(ri_1_w1, :); wl_1(ri_1_w2, :); wl_1(ri_1_w3, :)];
images_w_1 = [wd_1(ri_1_w1, :); wd_1(ri_1_w2, :); wd_1(ri_1_w3, :)];
plotting_3_images(images_w_1, labels_w_1, col_size, row_size, 'Three randomly selected wrongly classified digits for the 1NN-classifier using unclustered templates');

% 7NN
% Picking three random correctly classified digits to plot
[ri_1_c1_7, ri_1_c2_7, ri_1_c3_7] = deal(randi(length(cl_1_7), 1), randi(length(cl_1_7), 1), randi(length(cl_1_7), 1));
labels_c_1_7 = [cl_1_7(ri_1_c1_7, :); cl_1_7(ri_1_c2_7, :); cl_1_7(ri_1_c3_7, :)];
images_c_1_7 = [cd_1_7(ri_1_c1_7, :); cd_1_7(ri_1_c2_7, :); cd_1_7(ri_1_c3_7+1, :)];
plotting_3_images(images_c_1_7, labels_c_1_7, col_size, row_size, 'Three randomly selected correctly classified digits for the 7NN-classifier using unclustered templates');
% Picking three random wrongly classified digits to plot
[ri_1_w1_7, ri_1_w2_7, ri_1_w3_7] = deal(randi(length(wl_1_7), 1), randi(length(wl_1_7), 1), randi(length(wl_1_7), 1));
labels_w_1_7 = [wl_1_7(ri_1_w1_7, :); wl_1_7(ri_1_w2_7-5, :); wl_1_7(ri_1_w3_7, :)];
images_w_1_7 = [wd_1_7(ri_1_w1_7, :); wd_1_7(ri_1_w2_7-5, :); wd_1_7(ri_1_w3_7, :)];
plotting_3_images(images_w_1_7, labels_w_1_7, col_size, row_size, 'Three randomly selected wrongly classified digits for the 7NN-classifier using unclustered templates');

toc

disp("Ending task 1");
disp("-----------------------");

%%%------------------------------------
%%%------------------------------------
%%%         END OF TASK 1
%%%------------------------------------
%%%------------------------------------

%%%------------------------------------
%%%------------------------------------
%%%             TASK 2
%%%------------------------------------
%%%------------------------------------

disp("-----------------------");
disp("Beginning task 2");

tic

disp("Sorting and Clustering");

% Returns a 1x10 cell
% Each element, f. ex. trainv_sorted{1} contains all the data with label 0 (digit 0)
```

```
trainv_sorted = sorting(trainv, trainlab, num_train, vec_size);

M = 64;
new_training_set = zeros(10*M, vec_size);

for i = 0:9
   [~, Ci] = kmeans(trainv_sorted{i+1}, M);
   new_training_set(i*M+1:(i+1)*M, :) = Ci;
end

% Task 2 variables

% Used in 1NN-classification
% w_2 & c_2 : Number of wrong and correct classifications
% cm_2 : Confusion-matrix
% wd_2 & wl_2 : Array containing data and labels respectively for wrongly classified images
% cd_2 & cl_2 : Same as the above, only with correctly classified images
% tp_2 : Vector that holds the true and predicted labels, used for plotting
% The label matrices contain [True label, Predicted label]
% Using deal to get all intialization on one line
[w_2, c_2, cm_2, wd_2, wl_2, cd_2, cl_2, tp_lab_2] = deal(0, 0, zeros(10, 10), zeros(1, vec_size), zeros(1, 2), zeros(1, vec_size), zeros(1, 2), zeros(0, 2));

% 7-NN classifier:
[w_2_7, c_2_7, cm_2_7, wd_2_7, wl_2_7, cd_2_7, cl_2_7, tp_lab_2_7] = deal(0, 0, zeros(10, 10), zeros(1, vec_size), zeros(1, 2), zeros(1, vec_size), zeros(1, 2), zeros(0, 2));

disp("Calculating distances and classifying");
size_bulk = 999; % 1000 - 1 (needs to be removed)
for i = 1:(num_test/(size_bulk+1))
    % Splitting the testing-sets into 'bulks' of 1000 elements, calculating
    % distances for each of these sets. Adding all togheter to a new matrix
    testing_data = testv(((i-1)*size_bulk+1):((i*size_bulk)+1), :);
    testing_labels = testlab(((i-1)*size_bulk+1):((i*size_bulk)+1), :);
    [number_of_tests, ~] = size(testing_data);

    training_labels = [0*ones(M, 1); 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)];

    distances_clustering = calculate_distance(testing_data, new_training_set);

    % 1-NN classifier
    [cm_2, wd_2, wl_2, w_2, cd_2, cl_2, c_2, tp_lab_2] = classify_kNN(distances_clustering, number_of_tests, testing_data, testing_labels, training_labels, cm_2, wd_2, wl_2, w_2, cd_2, cl_2, c_2, 1, tp_lab_2);
    % 7-NN classifier
    [cm_2_7, wd_2_7, wl_2_7, w_2_7, cd_2_7, cl_2_7, c_2_7, tp_lab_2_7] = classify_kNN(distances_clustering, number_of_tests, testing_data, testing_labels, training_labels, cm_2_7, wd_2_7, wl_2_7, w_2_7, cd_2_7, cl_2_7, c_2_7, 7, tp_lab_2_7);
end

error_rate_2 = w_2/num_test;
```

```
disp("Error-rate for the 1NN-classifier for the clustered data: " + error_rate_2);
error_rate_2_7 = w_2_7/num_test;
disp("Error-rate for the 7NN-classifier for the clustered data: " + error_rate_2_7);

Plotting the confusion matrices
plot_confusion_matrix(tp_lab_2, "Confusion matrix for the clustered data using the 1NN-classifier");
pause(5);
plot_confusion_matrix(tp_lab_2_7, "Confusion matrix for the clustered data using the 7NN-classifier");
pause(5);

% 1NN
% Picking three random correctly classified digits to plot
[ri_2_c1, ri_2_c2, ri_2_c3] = deal(randi(length(cl_2), 1), randi(length(cl_2), 1), randi(length(cl_2), 1));
labels_c_2 = [cl_2(ri_2_c1, :); cl_2(ri_2_c2, :); cl_2(ri_2_c3, :)];
images_c_2 = [cd_2(ri_2_c1, :); cd_2(ri_2_c2, :); cd_2(ri_2_c3, :)];
plotting_3_images(images_c_2, labels_c_2, col_size, row_size, 'Three randomly selected correctly classified digits for the 1NN-classifier using clustered templates');
% Picking three random wrongly classified digits to plot
[ri_2_w1, ri_2_w2, ri_2_w3] = deal(randi(length(wl_2), 1), randi(length(wl_2), 1), randi(length(wl_2), 1));
labels_w_2 = [wl_2(ri_2_w1, :); wl_2(ri_2_w2, :); wl_2(ri_2_w3, :)];
images_w_2 = [wd_2(ri_2_w1, :); wd_2(ri_2_w2, :); wd_2(ri_2_w3, :)];
plotting_3_images(images_w_2, labels_w_2, col_size, row_size, 'Three randomly selected wrongly classified digits for the 1NN-classifier using clustered templates');

% 7NN
% Picking three random correctly classified digits to plot
[ri_2_c1_7, ri_2_c2_7, ri_2_c3_7] = deal(randi(length(cl_2_7), 1), randi(length(cl_2_7), 1), randi(length(cl_2_7), 1));
labels_c_2_7 = [cl_2_7(ri_2_c1_7, :); cl_2_7(ri_2_c2_7, :); cl_2_7(ri_2_c3_7, :)];
images_c_2_7 = [cd_2_7(ri_2_c1_7, :); cd_2_7(ri_2_c2_7, :); cd_2_7(ri_2_c3_7, :)];
plotting_3_images(images_c_2_7, labels_c_2_7, col_size, row_size, 'Three randomly selected correctly classified digits for the 7NN-classifier using clustered templates');
% Picking three random wrongly classified digits to plot
[ri_2_w1_7, ri_2_w2_7, ri_2_w3_7] = deal(randi(length(wl_2_7), 1), randi(length(wl_2_7), 1), randi(length(wl_2_7), 1));
labels_w_2_7 = [wl_2_7(ri_2_w1_7+11, :); wl_2_7(ri_2_w2_7, :); wl_2_7(ri_2_w3_7+6, :)];
images_w_2_7 = [wd_2_7(ri_2_w1_7+11, :); wd_2_7(ri_2_w2_7, :); wd_2_7(ri_2_w3_7+6, :)];
plotting_3_images(images_w_2_7, labels_w_2_7, col_size, row_size, 'Three randomly selected wrongly classified digits for the 7NN-classifier using clustered templates');

toc

disp("Ending task 2");
disp("----------------------");
```

```matlab

%%%------------------------------------
%%%------------------------------------
%%%         END OF TASK 2
%%%------------------------------------
%%%------------------------------------

% Classifying the test-vectors given the distance matrix using 1NN.
% Returns confusion-matrix, data, labels and amount for wrong and correct classification.
function [cm, wd, wl, w, cd, cl, c, true_pred_lab] = classify_kNN(distances_set, num_test, test_data, test_labels, training_labels, cm, wd, wl, w, cd, cl, c, k, true_pred_lab)
    for l = 1:num_test
        [~, indices] = sort(distances_set(:, l));

        min_labels = zeros(k, 1);
        for y = 1:k
            min_labels(y) = training_labels(indices(y));
        end

        pl = mode(min_labels); % Finding the predicted label (Most frequent of the 7 labels)
        tl = test_labels(l); % Finding the true label

        true_pred_lab(end+1, :) = [tl, pl];

        cm(tl+1, pl+1) = cm(tl+1, pl+1) + 1; % Updating confusion matrix

        % Checks is the predicted label was correct or wrong
        % Respectively adds data and [true label, predicted label] to matrices
        if pl ~= tl
            w = w + 1; % Updates number of wrong classification
            wd(w, :) = test_data(l, :); % Adds data
            wl(w, :) = [tl, pl]; % Adds true / predicted labels
        elseif pl == tl
            c = c + 1; % Updates number of correct classification
            cd(c, :) = test_data(l, :); % Adds data
            cl(c, :) = [tl, pl]; % Adds true / predicted labels
        end
    end
end

% Classifying the test-vectors given the distance matrix using 1NN.
% Returns confusion-matrix, data, labels and amount for wrong and correct classification.
function [cm, wd, wl, w, cd, cl, c, true_pred_lab] = classify_1NN(distances_set, num_test, test_data, test_labels, training_labels, cm, wd, wl, w, cd, cl, c, true_pred_lab)
    for i = 1:num_test
        [~, index] = min(distances_set(:, i)); % Finds index in training set with min. dist.
        pl = training_labels(index); % Predicted/Classified label
        tl = test_labels(i); % True label
        true_pred_lab(end+1, :) = [tl, pl]; % Adding true and predicted labels to array

        cm(tl+1, pl+1) = cm(tl+1, pl+1) + 1; % Updating confusion matrix
```

```
        % Checks is the predicted label was correct or wrong
        % Respectively adds data and [true label, predicted label] to
        % matrices
        if pl ~= tl
            w = w + 1; % Updates number of wrong classification
            wd(w, :) = test_data(i, :); % Adds data
            wl(w, :) = [tl, pl]; % Adds true / predicted labels
        elseif pl == tl
            c = c + 1; % Updates number of correct classification
            cd(c, :) = test_data(i, :); % Adds data
            cl(c, :) = [tl, pl]; % Adds true / predicted labels
        end
    end
end

% Plotting randomly picked wrongly classified and correctly classified
function plotting_images(image_data, labels, col_size, row_size)
    image(transpose(reshape(image_data, col_size, row_size)));
    title("True label - " + labels(1, 1) + " / Predicted label - " + labels(1, 2));
end

function plotting_3_images(image_datas, labels, col_size, row_size, title_overall)
    figure('Position', [100 100 800 300]);
    subplot('Position', [0.05 0.1 0.25 0.8]);
    imshow(transpose(reshape(image_datas(1, :), col_size, row_size)), []);
    title("True label - " + labels(1, 1) + " / Predicted label - " + labels(1, 2));
    subplot('Position', [0.35 0.1 0.25 0.8]);
    imshow(transpose(reshape(image_datas(2, :), col_size, row_size)), []);
    title("True label - " + labels(2, 1) + " / Predicted label - " + labels(2, 2));
    subplot('Position', [0.65 0.1 0.25 0.8]);
    imshow(transpose(reshape(image_datas(3, :), col_size, row_size)), []);
    title("True label - " + labels(3, 1) + " / Predicted label - " + labels(3, 2));

    % Overall title
    sgtitle(title_overall);
end

% Calculates the distances given test-vector and templates
function distances_return = calculate_distance(test_set, templates)
    distances_return = dist(templates, transpose(test_set));
end

function s = sorting(training_data, training_label, num_train, vec_size)
    s = {zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size), zeros(0, vec_size)};
    for j = 1:num_train
        s{training_label(j)+1}(end+1, :) = training_data(j, :);
    end
end

function plot_confusion_matrix(true_pred_labels, title_cm)
```

```
    confusionchart(true_pred_labels(:, 1), true_pred_labels(:, 2));
    title(title_cm);
end
```