NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTT4275 - Estimation, Detection and Classification

Classification of Iris flowers and hand written numbers

Authors:
August André Lukkassen, Knut Tovshus Ødegård

Date 30.04.2025

Summary

In this paper we will present a classification project given in the course TTT4275 - Estimation, Detection and Classification at NTNU. The project consists of training, implementation and testing of two classifiers: one linear discriminant classifier to classify different Iris flowers, and one template based classifier using a nearest neighbour algorithm to classify handwritten digits.

Our implementation of the classifier for the Iris task yielded satisfactory performance, with an error rate of 3.3%. Implementations using less features to classify the flowers also gave satisfying results. During testing a small bias in the dataset used was discovered, with the first data-samples seemingly being easier to classify than the last.

The project also examines the effect of k-means clustering as a preprocessing step for template-based classification of handwritten digits from the MNIST dataset. Classifiers based on nearest neighbor (NN) and k-nearest neighbor (KNN) methods are evaluated with and without clustering. The work compares classification performance and runtime, and discusses the importance of a suitable classification distance metric. K-nearest neighbour greatly improved runtime at the expense of classification accuracy.

Overall, we deemed the project successful due to the satisfactory results and the valuable learning experience it provided.

Table of Contents

1 Introduction

The goal of our project is to get relevant practical experience with implementation and use of classifiers. This helps put the theory learned in the course into a real life perspective, while also showing how easy it is to use this theory into creating something actually useful. By using similar methods as the ones used in this project, one may hopefully be able to create classifiers for real life problems - whether in the food industry, medicine, writing tools, etc.

This report will first go through relevant theory needed to understand the classifiers being used, then introduce the tasks, before going through our implementation, showing of results and discussing these.

2 Theory

This chapter will give an introduction to the necessary theory needed to understand our project, including why the implementation works and how to interpret the results. First there will be some general theory related to classification, then going more into specifics about certain classifiers, before showing how one can measure their performance.

2.1 Classification

First and foremost what is a classification? Classification is just what it sounds like, deciding what class/label something belongs to based on a given set of information. We as humans are constantly classifying different people or things. Say for example you are choosing which avocados to pick at the store, you are classifying the avocados into either being good or bad. To do this you will look at the color and test their stiffness. This is what we call features; what we use to decide what class something belongs to.

A mathematical classifier is in principle just a function which, with knowledge of features corresponding to different labels (classes), is able to classify data most accurately. Just as we humans look and test the stiffness of the avocado to find the good ones, a classifier can use the information about these features to do the same job. Sometimes less or other features may also be used, as for example the ones used in this avocado scanner (CoopMega 2025), which is a prime example of practical classification.

2.2 Training of classifiers

There are different ways to train classifiers. Supervised learning, which is the way of training used in this project, is a way of using labeled data, data we know belongs to a certain class, to learn/train the classifier to be better at classifying data similar to this data. This is verry similar to us humans as we often learn by relating previous events we know corresponded to given outcomes, aka. features relating to labels.

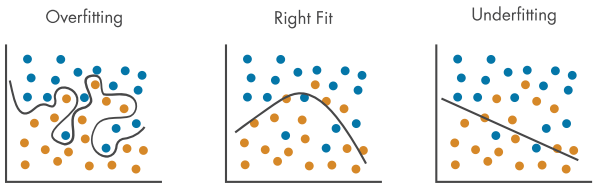


Figure 1: Visualization of under-/overfitting of a classifier. (MathWorks 2025)

When training a classifier, two well known problems can occur: overfitting and underfitting. Overfitting is when the classifier models the training data too closely, including specific variations for this data, and is therefore unable to make generalizations. This results in the classifier potentially having bad performance on data it is not trained on. Underfitting is the opposite problem, where the classifier generalizes too much and is unable to correctly classify data. These concepts can be seen in figure 1.

To test if the classifier was correctly trained, especially to look for overfitting, it is common to divide the data set into two parts: one training set and one test set. The training set is used to train the classifier while the test set is used to validate the classifier and detect potential problems. An overfitted classifier will tend to perform well on the training set, but substantially worse on the test set, while an underfitted will have bad performance in both cases.

2.3 Separability of data

Data to be classified can generally be divided into three groups: linear separable, nonlinear separable and non separable. For data to be linearly separable it means that there exists a (set of) linear line(s), plane(s) or hyperplane(s) (depending on the dimension/amount of features used) which is able to divide the dataset into groups corresponding to the different labels. Further there exist data that still is separable, but no longer linearly separable. This means that there exist some sort of non-linear border between the different data corresponding to different labels. For both linear and non-linear separable problems, it is possible to create a perfect classifier without any misclassifications, with linear problems being the easiest of the two.

Unfortunately almost all practical problems are part of the last group, non separable. This means that there does not exist any way to divide the data set into groups only containing all the samples corresponding to the different labels. Therefore almost all

practical classifiers will have some errors as there is not possible to create a perfect classifier in this case.

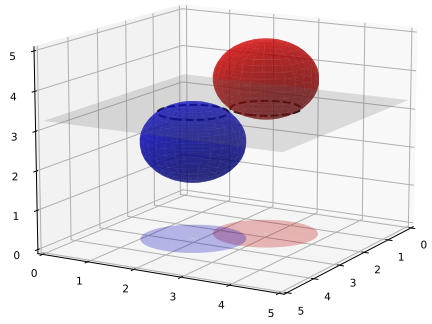


Figure 2: Example of data being seperable in 3d (using 3 features), but not in 2d (using only 2 features).

Further there may be the possibility of a dataset being non separable using only some of its features, but being separable using all of its features. An example of this is shown in figure 2. This can also be the case of how "separable" data can be, meaning that there is less overlap between the data samples using more features then less. Therefore designing a good classifier is generally easier using the most amount of features possible.

2.4 Linear discriminant classifier

A linear discriminant classifier (LDC) is a type of classifier working best with linearly separated problems, but due to its simplicity is often used for real life non separable problems. The following LDC described, and the training of it, is based on the chapters 2.4 and 3.2 of the classification compendium on blackboard (Johnsen 2017).

The LDC is a discriminant classifier which means that the decision rule for which class, w_i, some data, x, belongs to is given by

$$x \in w_j \Leftrightarrow g_j(x) = \max_i g_i(x) \tag{1}$$

The LDC classifies based on which of the discriminant functions, $g_i(x)$, have the highest value. For each class to be classified the discriminant function is given by

$$g_i(x) = w_i^T x + w_{i0} \tag{2}$$

which of course being a linear function, is the reason why it is called a linear discriminant classifier. When you are classifying more than two different classes the discriminant functions can be written on a matrix form as

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

resulting in a "discriminant" vector, g, where the LDC classifies based on the highest value in this vector. Further we can alter the W matrix and the input vector x to $\begin{bmatrix} W & w_0 \end{bmatrix}$ and $\begin{bmatrix} x^T 1 \end{bmatrix}^T$ respectively, which lets us reduce our discriminant function to

$$g = Wx \tag{4}$$

Here it is really important to note that all data vectors, x, must increase their dimension by 1, by having a 1 added to them because of this alteration!

A way to train this type of classifier is to minimize the mean square error (MSE) between the "discriminant" vector g mapped to be in the range $[0, 1]$, g_m, and the target vector t which is a unit vector corresponding to the correct label for the data. A good choice for target vectors is to have unit vectors with a 1 on different axis for all the different classes. The MSE is given by

$$MSE = \frac{1}{2}\sum_{k=1}^{N}(g_{m,k} - t_k)^T(g_{m,k} - t_k) \tag{5}$$

with the mapping of the "discriminant" vector g_k being done by the sigmoid function

$$g_{m,ik} = \frac{1}{1 + e^{-g_{ik}}} \tag{6}$$

on every element g_{ik}, which in turn are the elements in the g_k vector given by eq. 4. The sigmoid function is chosen due to it being differentiable, which is necessary for differentiating the MSE which in turn is needed to optimize the MSE in an efficient way. We can differentiate the MSE by using the chain rule

$$\nabla_W MSE = \sum_{k=1}^{N} \nabla_{g_{m,k}} MSE \nabla_{z_k} g_{m,k} \nabla_W z_k \tag{7}$$

which results in

$$\nabla_W MSE = \sum_{k=1}^{N} [(g_{m,k} - t_k) \circ g_{m,k} \circ (1 - g_{m,k})] x_k^T \tag{8}$$

Here the Hadamard product, $\circ$, is the element wise multiplication of vectors. To find the best W we can use a kind of steepest decent algorithm given by

$$W_{n+1} = W_n - \alpha \nabla_W MSE \tag{9}$$

which lets us train our classifier by modifying the value of W until the training converges (the new choices of W do not change the MSE in any significant way). The variable α is called the step length. Further details about training can be found in the compendium (Johnsen 2017).

It should be noted that there are other possible ways to train the LDC, but this is an easy and sufficient way of doing it.

2.5 Template-based classification

Template-based classification is a non-statistical method in which patterns are classified by comparing with a set of references. The pattern is then assigned to the class of the the reference that is the most similar based on a chosen metric (Duda et al. 2000). The most basic implementation of this classifier is the nearest neighbour classifier (NN-classifier), where the entire training dataset is used as templates. Elements are then classified as the class-label of the "nearest" reference. Since the NN procedure matches the pattern with the singular most similar reference, it is prone to misclassification caused by reference patterns that are not representative for their class. K-nearest neighbour is a generalization of NN and a more advanced decision rule that handles this weakness. The KNN classifier finds the $K > 1$ nearest reference patterns, and assigns the most frequent class label in this selection to the input pattern.

A metric function $D(\cdot, \cdot)$ is what provides a generalized distance between two patterns (Duda et al. 2000). The Minkowski metric is a general class of metrics for use on d-dimensional patterns (Duda et al. 2000).

$$L_k(\mathbf{a}, \mathbf{b}) = \left(\sum_{i=1}^{d} |a_i - b_i|^k \right)^{1/k}, \tag{10}$$

For image classification the L_2 norm (eq. 10 with k=2) is a commonly used metric. An image with resolution $n \times m$ is typically flattened into a feature vector of length $n \cdot m$, where each element represents a pixel. Each pixel holds a corresponding intensity value (e.g., in 8-bit grayscale, from 0 to 255), and the resulting feature vector lies in the space $\mathbb{R}^{n \cdot m}$. The Euclidean distance(L2-norm) between the test and training pattern is then used as the decision rule when classifying. Euclidean distance can however be a suboptimal metric since the geometric spreading of the data points is not taken into consideration. The Mahalanobis distance metric captures this by utilizing the covariance matrix Σ:

$$D_M(\mathbf{x}, \boldsymbol{\mu}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})} \tag{11}$$

This will encapsulate the correlation and variance of features which often rules this metric preferred for template matching when a reliable covariance estimate is available.

A shared limitation of both Euclidean and Mahalanobis metrics is not accounting for invariances such as translation, rotation, scaling etc. of patterns. Pattern classification (Duda et al. 2000) exemplifies this using the MNIST dataset. A handwritten 5 is shifted s pixels to the right and the Euclidean distance to the original pattern is computed. Even a small shift of $s = 1$ is shown to yield an Euclidean distance comparable to that of a centred handwritten eight. For $s > 1$ the distance is much larger than between the sample and the eight. These finds are presented in figure 3.

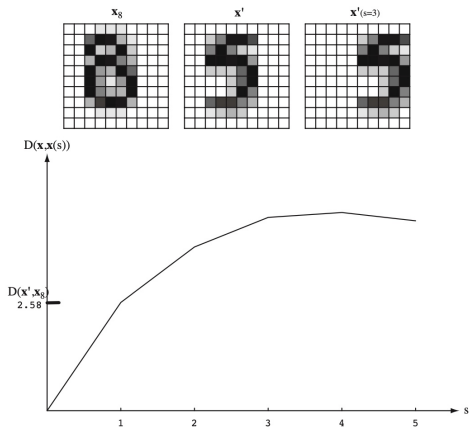


Figure 3: Euclidean distance between a handwritten 5 x' and the same figure translated s pixels to the right.

Duda et al. note that preprocessing to transform images to a "standard" form is both theoretically challenging and computationally intensive. A preferred alternative is the tangential distance metric. Expected transformations in the data set are linearly approximated and incorporated into the distance computation.

2.6 Clustering

Although template classifiers like KNN can yield good performance, applying them directly with the entire training set as references is computationally expensive. The KNN runtime is directly correlated with the M templates size of the feature vector. With no preprocessing of the training data, the selection of distance metrics is also limited to Euclidean distance. This motivates preprocessing of data with clustering to reduce the number of references and estimate the

covariance matrix. Clustering is a type of unsupervised learning that forms groupings of unlabelled input patterns based on their similarities (Duda et al. 2000). Unsupervised learning is when a dataset is trained without the use of labels. These groupings can then be used as templates in the classification algorithm, hence reducing the computation time.

Clustering can be implemented in multiple ways, but this report will focus on clustering algorithms intended for use in template-based classification. K-means is an elementary clustering method often used to generate representative templates for template classifiers (Duda et al. 2000). It's a special case of the more general algorithm presented in the classification compendium (Johnsen 2017). The algorithm from (Johnsen 2017) has the benefit of generating a covariance matrix for each class but is in turn more complex in its implementation. The k-means algorithm is illustrated in figure 4. The symbols μ_i represent the cluster centroids, each defined as the mean of all patterns assigned to cluster **i**. These centroids are subsequently used as templates for the classifier.

Algorithm 1 (K-means clustering)

1 **begin initialize** $n, c, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_c$
2 **do** classify n samples according to nearest $\boldsymbol{\mu}_i$
3 recompute $\boldsymbol{\mu}_i$
4 **until** no change in $\boldsymbol{\mu}_i$
5 **return** $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_c$
6 **end**

Figure 4: K-means algorithm from (Duda et al. 2000).

2.7 Confusion matrix and error rate

A confusion matrix is a way of visualizing the performance of a classifier. The columns of the matrix corresponds to the classified/predicted label while the rows are the true label of the data. For a perfect classifier the confusion matrix will be a diagonal matrix. A less diagonal matrix symbolizes a worse classifier. The main advantage of a confusion matrix is that it is easy to see what the classifier is good at classifying and what it is bad at.

Together with a confusion matrix the error rate of the classifier should also be used to give an idea of the general performance of the classifier. Since we only have a given constrained dataset and not an infinitely large one, we can only give an estimate for the error rate of the classifier by

$$EER_T = \frac{E_T}{N_T} \tag{12}$$

where E_T and N_T are the number of errors and the data set size for the test set. It is also possible to estimate an error rate using the training set, but the estimated error rate of the classifier is computed using the training set.

3 The tasks

Our classification project consist of two task: Iris flower classification and classification of handwritten numbers. These tasks were chosen due to their relevance in classification, containing relevant topics like linear and template based classifiers as well as clustering.

3.1 Iris flower classification

The Iris flower task is about classification of different Iris flowers (Setosa, Versicolor and Virginica) using a linear discriminant classifier. Classification can here be done by looking at 4 features of the flowers: Sepal width, sepal length, petal width and petal length. The task is split into two parts; first designing, training and evaluating the classifier, then looking at and removing different features of the Iris flowers to see how this relates to linear separability and impacts performance of the classifier.

3.2 MNIST handwritten digits

The second task aims to classify the handwritten numbers in the Modified National Institute of Standards and Technology (MNIST) database using template-based classifier NN and KNN. The classifiers will be tested with and without supervised preprocessing of the training patterns. In the preprocessing step, clustering is performed separately within each class. Performance and runtime differences will be compared between the two approaches. Further analysis of the importance of initial cluster centroids and the number of clusters per class will be carried out to expand upon the performance analysis.

4 Implementation, results and discussion

This chapter is about the implementation and results of the different tasks, as well as discussing the results. The chapter is structured in such a way that the whole implementation, results and discussion of the Iris task is done before doing the same for the MNIST task. The programming language Python was used for the implementation, due to both its ease of use and multiple useful libraries for classification.

4.1 Iris task - Implementation

To classify the iris flowers a linear discriminant classifier was used. A flowchart showing the overall struc-

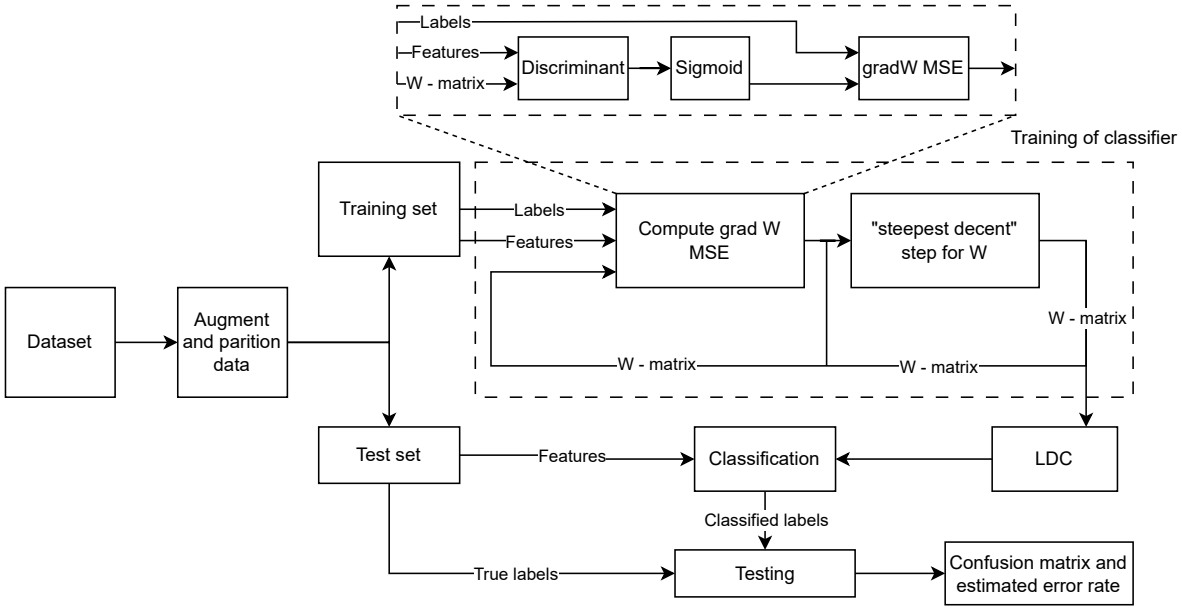


Figure 5: Flow chart showing the overall structure for our implementation of the LDC.

ture for the implementation of the classifier is shown in figure 5.

The dataset we used, often referred to as "Fisher Iris data" is a collection of 150 samples, 50 for each Iris flower. This dataset was partitioned into a training set and a test set in two different ways, depending on the task, to look at if the dataset was evenly distributed or not. The different partitioning can be seen in table 1.

	Training set	Test set
Partition 1	30 first	20 last
Partition 2	30 last	20 first

Table 1: The different partitioning of the dataset.

Further the dataset was augmented to work with the LDC. To match the data with the format of the LDC used, each data sample had an entry with a 1 added to them. If wanted, the data was also augmented to only include a wanted subset of the features of the flowers.

The reason for removing certain features was to look at the separability of the different features for the Iris flowers. To determine which features to look at, a histogram of all the features in the dataset was used and the most overlapping features were removed one after another. For each instance a classifier was trained and tested.

Training of the LDC was done by using the "steepest descent" algorithm given by eq. 9 over a fixed set of iterations = 4000, with the W matrix initially set to 0. The python implementation is shown in appendix A. Different step lengths α were tested and are shown in figure 6.

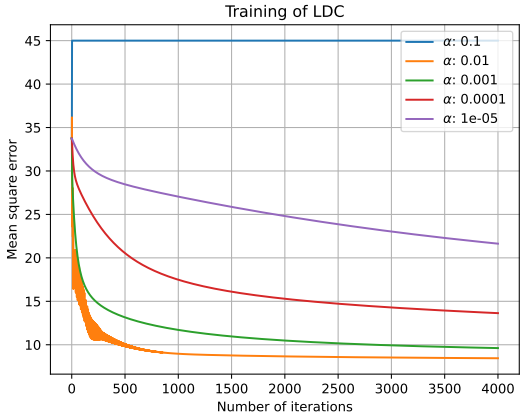


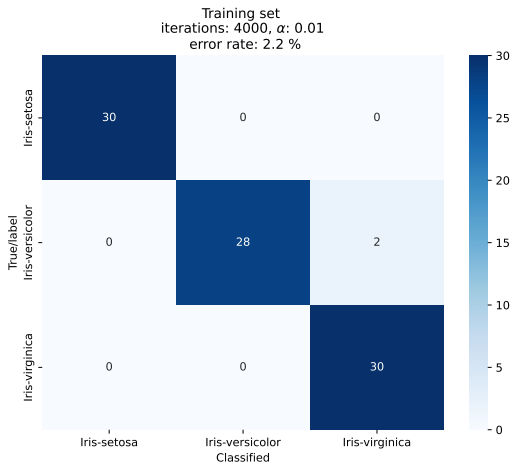
Figure 6: Mean square error for each iteration of training of LDC using different step lengths α.

From figure 6 we see that using $\alpha = 0.01$ is the best choice as this is the only one resulting in convergence of the training. This was therefore chosen as the α for the training of the classifiers.

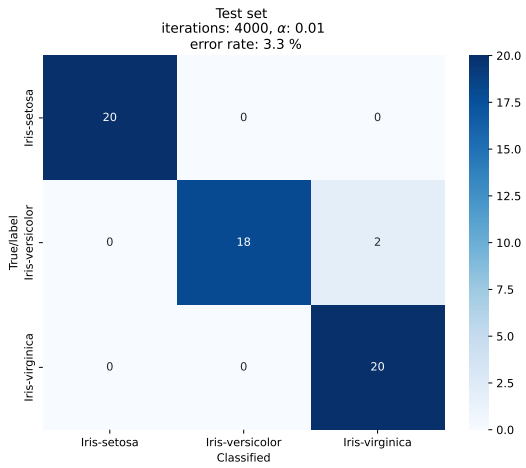
To test the classifiers primarily the test set, but also the training set was classified using the trained LDC. The data was then compared with the true labels and confusion matrices as well as the estimate for the error rate (eq. 12) was computed.

4.2 Iris task - Results

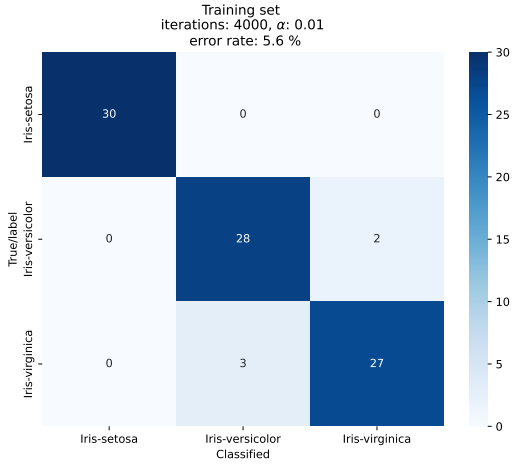
First the dataset was partitioned using the 30 first samples for training and the 20 last for testing, aka. partition 1 in table 1. Training of the classifier on this test set using $\alpha = 0.01$ and 4000 iterations gave



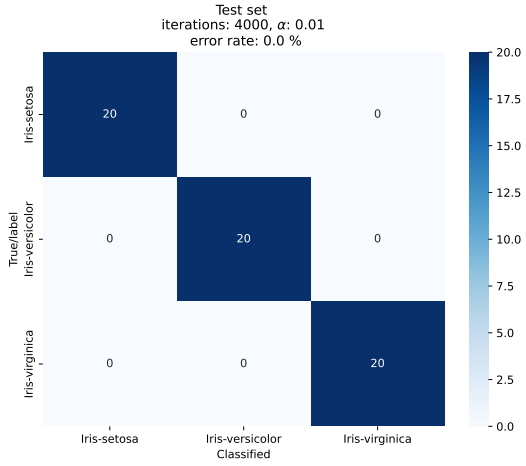
(a) Confusion matrix for LDC on training set using partition 1.



(b) Confusion matrix for LDC on test set using partition 1 .



(c) Confusion matrix for LDC on training set using partition 2.



(d) Confusion matrix for LDC on test set using partition 2.

Figure 7: Confusion matrices for classifiers using different partitioning of dataset.

the following W matrix, rounded to 3 decimals

$$W = \begin{bmatrix} 0.466 & 1.855 & -2.752 & -1.280 & 0.332 \\ 1.463 & -3.170 & -0.307 & -0.964 & 2.583 \\ -3.491 & -3.049 & 5.142 & 4.779 & -2.705 \end{bmatrix} \tag{13}$$

The performance of an LDC using this W matrix is shown in the form of confusion matrices with the performance on the training set shown in figure 7a and the test set in figure 7b. The error rates were 2.2% and 3.3% respectively, with the latter also being the estimated error rate for this classifier. Interestingly the errors in both the confusion matrices are the same, with it miss-classifying some Iris Versicolor flowers as Virginica.

Training of a classifier on the test data using partition 2 in table 1 was also done, resulting in the W matrix

$$W = \begin{bmatrix} 0.546 & 1.860 & -2.931 & -1.384 & 0.341 \\ -0.160 & -2.419 & 2.090 & -4.325 & 4.867 \\ -2.361 & -3.588 & 4.124 & 5.154 & -3.564 \end{bmatrix} \tag{14}$$

The performance of the classifier using this W matrix is shown in figure 7c and figure 7d. The error rate on the training set is 5.6%, but here the error rate of the test set, also being the estimated error rate of this classifier, is 0.0%. While there are no errors in the test set there are more in the training set. We also see that here, once again, the classifier wrongly classifies some Versicolor flowers as Virginica flowers, now with the addition of the opposite problem: classifying Virginica flowers as Versicolor.

A histogram showing the distribution of the features for the different flowers in the dataset can be seen in figure 8. In addition to the bars showing the amount of each feature in an interval, a kernel density estimation line is plotted showing the estimated density

function for that feature for each flower. From this we can see that there is most overlap between the Iris flowers for the feature "sepal width", followed by "sepal length". The features "petal length" and "petal width" seems to have quite similar amount of overlap, with "petal width" seeming to have the least amount. We can therefore rank the features in terms of linear separability, from most to least: "petal width", "petal length", "sepal length" and "sepal width"

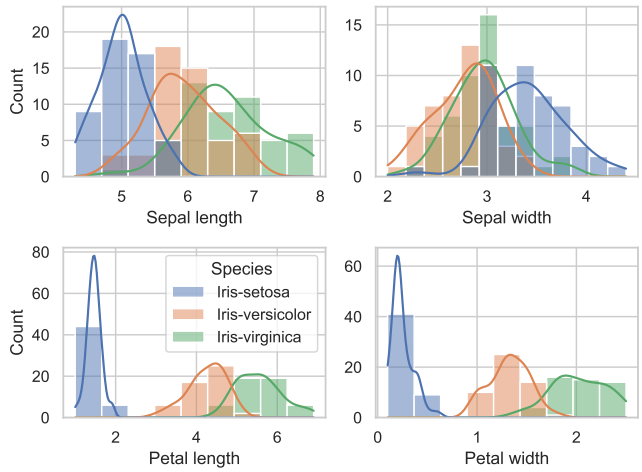


Figure 8: Histogram for the features of the different Iris flowers, including a kernel density estimation line.

Following this, the performance of classifiers when removing these features was tested. All classifiers were trained using 4000 iterations with $\alpha = 0.01$ and used the same training set corresponding to partition 1 from table 1. First a classifier with the feature "sepal width" removed was trained and tested, then the same was done with the features "sepal width" and "sepal length" removed, before lastly a classifier using only the feature "petal width" was tested. The confusion matrices for the tests are shown in figures 9b, 9d and 9f with the estimated error rates being 3.3%, 5.0% and 6.7% respectively. Confusion matrices for the training set are shown in figures 9a, 9c and 9e. Here the error rates were 3.3%, 5.6% and 4.4%. Once again we see that errors only occurred for Versicolor and Virginica classification. A comparison between the classifiers using the different features are shown in table 2

SW	SL	PL	PW	EER
✓	✓	✓	✓	3.3%
	✓	✓	✓	3.3%
		✓	✓	5.0%
			✓	6.7%

Table 2: Estimated error rate (EER) for classifiers using different features (SW = sepal width, SL = sepal length, PL = petal length, PW = petal width).

We see that the estimated error rates of the classifiers increase somewhat by removing more and more features, except when removing the most overlapping feature "sepal width" which results in similar performance as the original classifier.

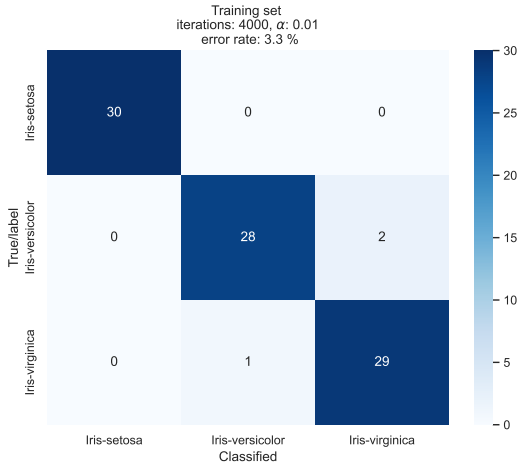
4.3 Iris task - Discussion

Looking at the results corresponding to training using either partition 1 or 2 (figures 7a, 7b, 7c and 7d) we see that the error rate corresponding to the training and test sets are quite similar. This shows us that overfitting did not occur and thus the training of the classifier can be deemed successful wrt. this. The same trend can be seen in the classifiers using a subset of the features (figures 9a, 9b, 9c, 9d, 9e and 9f). It is only the classifier using only the "petal width" to classify that starts showing some signs of overfitting, with it being the only one having a higher error rate on the test set than the training set. This does not seem to be a substantial problem, since an error rate of 6.7% is still OK, but it is worth nothing.

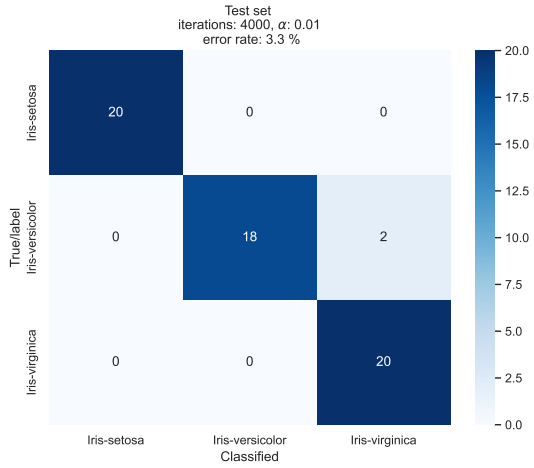
To train the classifiers one may also have used other methods than the implemented "steepest descent" algorithm. It is also possible to just make small changes by having a dynamic step size, but the LDC yielded quite satisfactory results and should therefore be sufficient for the task.

Further looking at the differences between the two ways of partitioning one might at first glance think that they result in classifiers with similar performance from looking at the plots. Looking at their W matrices (13 and 14) one starts to see some differences between the matrices, but once again this is as expected. Only when looking even closer at the confusion matrices we start to see a difference: the first samples of the data set seems to result in less errors then the last. Looking at the errors on the training set for partition 1 (figure 7a) and the test set for partition 2 (figure 7d), both using the first part of the data set, we see that both of them have a lower error rate than their corresponding test and training sets. This is probably due to the first data samples having features being "more" separable than the last data samples, and should be considered when choosing how to partition the data set. As we saw there were no big difference between the performance of the classifiers, but this small trend in the data set did have some impact and can lead to problems similar to overfitting. Therefore it may be an idea to shuffle the data set before partitioning, to prevent this imbalance of the dataset to have an impact on the training.

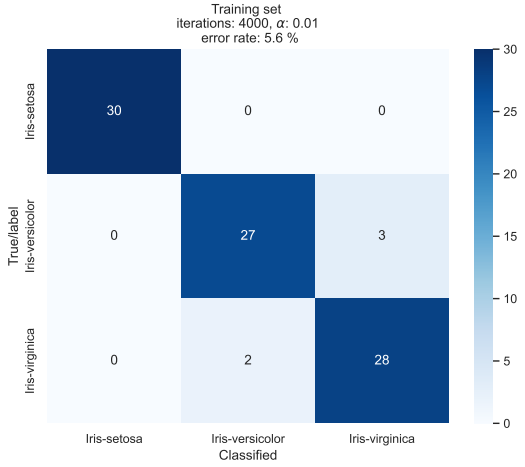
Continuing on the topic of separability of features, it is quite clear that some of the features have a much bigger impact on the classifier than others. From table 2 we see that removing the feature "sepal width" results in no change in the estimated error



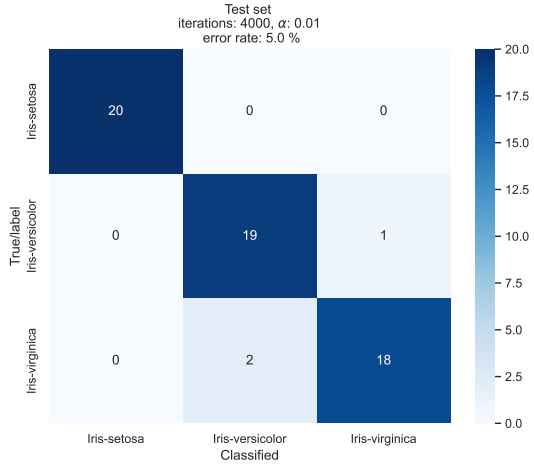
(a) Confusion matrix for LDC on training set using partition 1 with the feature "sepal width" removed



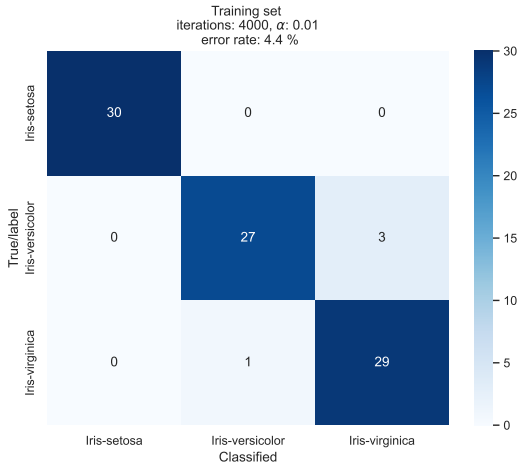
(b) Confusion matrix for LDC on test set using partition 1 with the feature "sepal width" removed



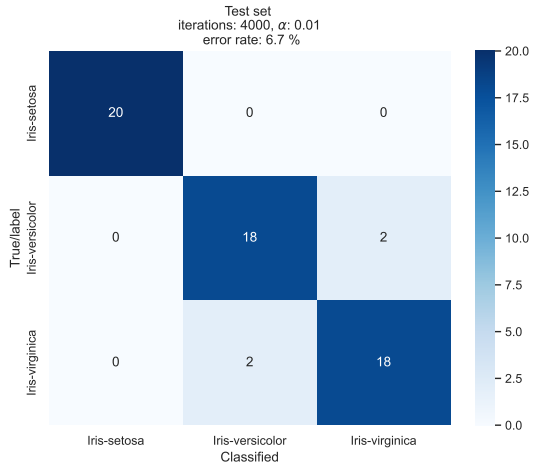
(c) Confusion matrix for LDC on training set using partition 1 with the features "sepal width" and "sepal length removed



(d) Confusion matrix for LDC on test set using partition 1 with the features "sepal width" and "sepal length removed



(e) Confusion matrix for LDC on training set using partition 1 with the features "sepal width", "sepal length" and "petal length" removed



(f) Confusion matrix for LDC on test set using partition 1 with the features "sepal width", "sepal length" and "petal length" removed

Figure 9: Confusion matrices for classifiers using different features of the Iris flowers

rate, while further removing features results in a slight loss of performance. To explain this we can look at the histogram of the features in figure 8. We see that for the feature "sepal width" almost all the flowers have an overlap of values for this feature and thus classifying based on this feature has almost non effect on performance. On the other hand we see that for the features "petal length" and "petal width" the Iris Setosa is completely linearly independent from the other flowers. This is most likely the reason why the different classifiers never miss-classifies this flower as for all the different cases of classifying this is linearly separable from the other flowers. The topic of linear separability may also be the reason why removing more features results in worse performance as the Iris Versicolor and Virginica may be "more" linearly separable using more features.

An advantage of making a classifier using less features is that it uses less data and therefore is faster, but as we see may lose some performance. Therefore when creating a classifier this computing/classification performance relationship should be considered.

4.4 MNIST task - Implementation

A python function for the NN-algorithm was implemented as displayed in line 16 of appendix B. The cdist function from the SciPy library was used to generate the Euclidean distance between patterns. Error rate is computed within the function according to eq. 12. The entire MNIST training set was used as templates. Classification performance of the MNIST test patterns was evaluated quantitively from confusion matrices and error rates. To reduce stress on computer memory the function divides the test set into chunks of 1000 elements that were classified separately.

A clustering function utilizing the k-means algorithm from the machine learning library scikit-learn was implemented as shown in line 54 of B. The k-means function from scikit-learn differs slightly from 4. To speed up convergence and improve the possibility of finding global minima, a smarter initialization method $\mathbf{k-means++}$ finds more optimal starting centroids (Pedregosa et al. 2024). K-means is then ran at a default of 10 times with different initial centroids. The clusters with the lowest inertia of these are then chosen (Pedregosa et al. 2024). This leads to an algorithm that is less prone to getting stuck at local minima.

The clustering function in line 54 isolates the feature vectors from each class and performs k-means clustering, set to compute 64 clusters per class. As class labels are utilized during data preprocessing, the approach incorporates supervised learning. The function returns two NumPy arrays containing computed templates for all classes and the corresponding labels. The clustering function is used for template generation in the NN and KNN algorithms in the python functions in lines 73 and 93, respectively. KNN is implemented as described in 2.5 and presented in line 93 of B. Although NN is a special case of KNN, it's included as a separate function for clarity and to follow the order of tasks.

For the NN classifier, a selection of the correctly and incorrectly classified images was created to perform a qualitative analysis.

4.5 MNIST task - Results

The classification performance is presented in graphical plots of the confusion matrices with the error rates stated in the titles. Figure 11a presents the confusion matrix for the NN classifier. The error rate is found to be 3.1%. The runtime for this computation is 8:45. The KNN classifier without clustering achieved an error rate of 3.0%, as shown in the confusion matrix 11b. The corresponding runtime was 9 minutes and 33 seconds.

When the NN algorithm is ran with the clustering output as templates, the error rate is found as 4.7%. Confusion matrix is presented in 11c. Runtime was 10 seconds.

The KNN classifier with preprocessing (clustering) yielded the highest error rate among all methods tested, at 5.6% (see 11d). The runtime matched that of NN with clustering, at 10 seconds.

A selection of three misclassified and correctly classified images are presented in 10. Runtimes and error rates for all classifier implementations are presented in table 3.

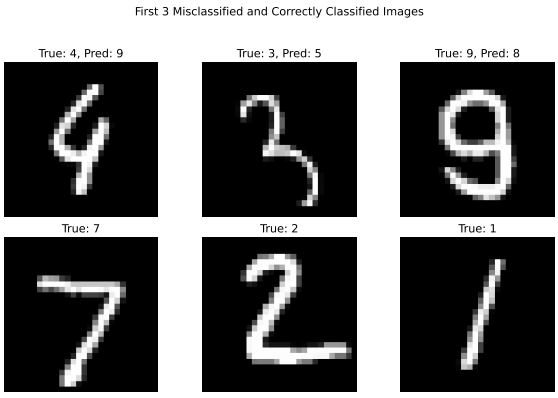
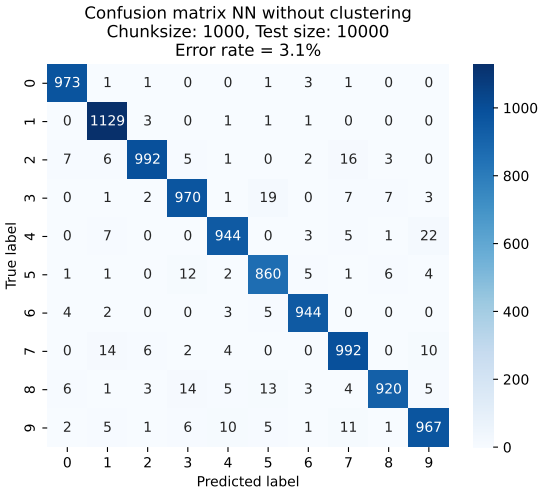
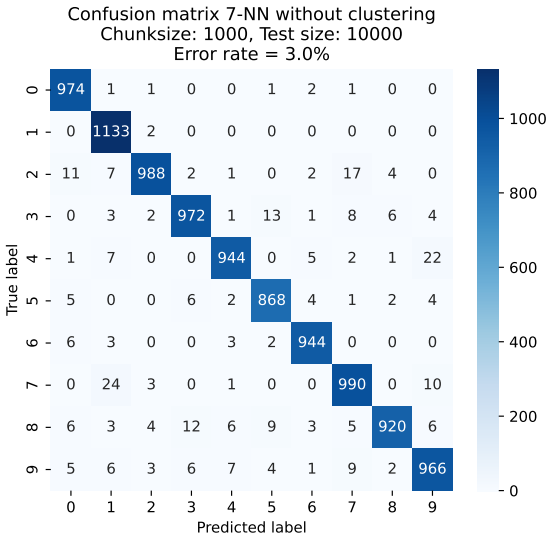


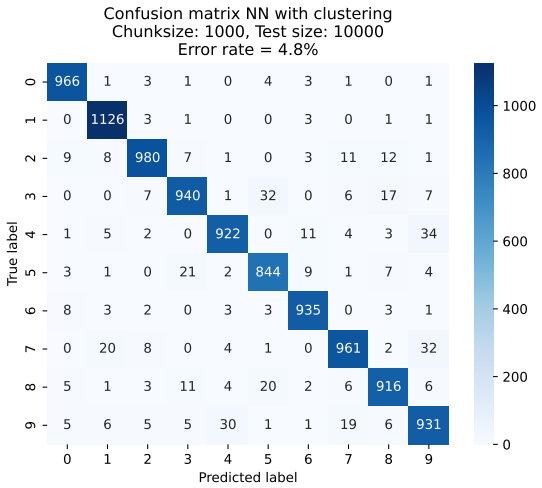
Figure 10: Examples of misclassified (top row) and correctly classified (bottom row) test images.



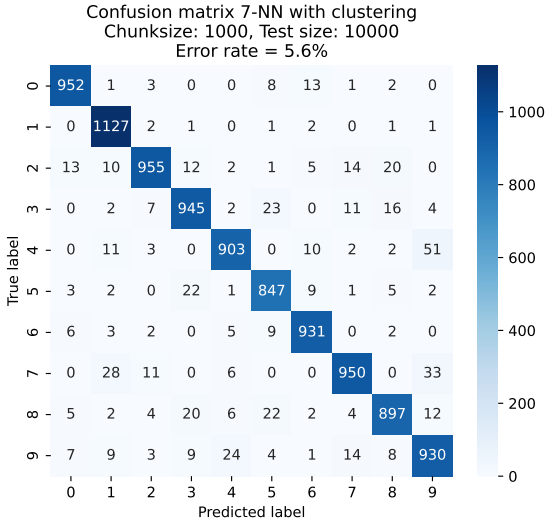
(a) Confusion matrix for NN classifier without clustering.



(b) Confusion matrix for 7-NN classifier without clustering. .



(c) Confusion matrix for NN classifier with clustering.



(d) Confusion matrix for 7-NN classifier with clustering.

Figure 11: Confusion matrices for NN and 7-NN classifiers with and without the use of clustering.

Classifier	Error Rate (%)	Runtime (min:sec)
NN (no clustering)	3.1	8:45
NN (with clustering)	4.8	0:11
7-NN (no clustering)	3.0	9:33
7-NN (with clustering)	5.6	0:10

Table 3: Comparison of error rates and runtimes for NN and 7-NN with and without clustering. Runtimes are total, including that of clustering.

4.6 MNIST task - Discussion

The discussion will focus on comparing how clustering impacts the performance and runtime of KNN classification, additionally comparing 7-NN performance to NN in both cases. The choice of metric will be discussed in regard to the overall performance obtained from this project's implementation. Sample images from the correctly and incorrectly classified sets will be discussed qualitatively.

As hypothesized from theory, preprocessing using k-means clustering with 64 clusters per class drastically reduces the computational cost. While this behaviour is desirable, it comes at the expense of performance as shown in table 3. For NN and 7-NN classification, the error rate increases 55% and 87% respectively, when clustering is applied. Degradation in performance might be caused by the clustering smoothening out important variances, resulting in centroids that are difficult to separate between classes. The issue is particularly problematic in datasets with high variability within each class. These observations indicate underfitting, as the training set is overly generalized. This is directly tied to the number of clusters per class. For datasets that are noisy, increasing the number of clusters might be beneficial

in smoothing out noise. Too few clusters or using the training set at templates directly, might in these cases lead to overfitting. The results do not indicate this for MNIST.

To quantify the optimal number of clusters for the 7-NN classifier, the error rate and silhouette score was plotted at cluster counts ranging from 2 to 100 clusters in intervals of 2. The silhouette score is a method for validating clusters by determining how similar patterns are to their own cluster compared to other clusters on average. This might indicate how easily separable the clusters are for the template classifier.

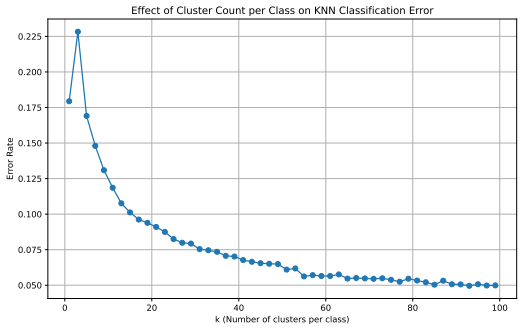


Figure 12: KNN error rate as a function of clusters per class in preprocessing.

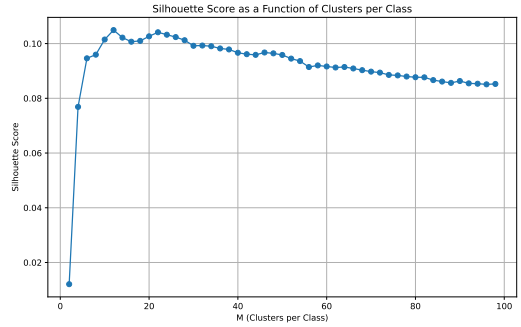


Figure 13: Silhouette score of k-means as a function of clusters per class.

Figure 12 indicates an inverse relation between cluster count and error rate. The error rate decreases rapidly until approx. 20-25 clusters and then slows down. This area does to some degree match with the silhouette score peak found in figure 13

The optimal cluster count is relative to the choice of **K** in the KNN classifier it is applied to. Because clustering smoothens out the patterns in the training data, it reduces the amount of variation available for the classifier to utilize. The optimal combination of **K** and clusters **C** could in theory be found by applying the approach in 12 across a range of K values, however this will not be discovered in this report.

Following this relationship, the comparison between 7-NN and NN classifier performance is influenced by the number of clusters per class. Without any preprocessing, 7-NN performs slightly better than NN, with a 0.1 percentage points lower error rate. However, when clustering is applied with 64 clusters per class, NN outperforms 7-NN by 0.8 percentage points. This change might be explained by the performance degradation of KNN when clustering is introduced, as discussed in the previous paragraph.

Although unlikely due to the optimalizations of k-means from scikit-learn, poor performance with clustering can be caused k-means algorithms getting stuck at local minima. The scatter plots in figure 14 display 4 randomly generated clusters and how the centroids converge for two initializations in k-means. In the leftmost plot, purposely bad starting centroids are used to display convergence towards clearly suboptimal local minima. On the right, optimal starting points show how the centroids are globally optimal.

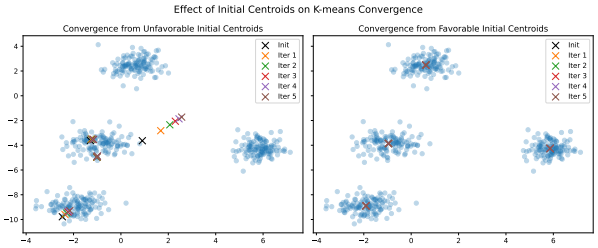


Figure 14: Comparison of k-means convergence from unfavourable (left) and kmeans++ favourable (right) initial centroids. Poor initialization leads to suboptimal local minima, while kmeans++ promotes faster and more accurate convergence.

Regarding the overall performance across classifiers, it could be argued that Euclidean distance is suboptimal for this dataset. Qualitatively, this is supported by the selection of misclassified digits in figure 10 as all three can easily be classified by a human being. Across all classifiers, the confusion matrices indicate that classes 1 and 7, and 8, 3 and 5 are the most difficult to separate. As presented in section 2.5 both Mahalanobis and tangent distance could prove more robust, potentially improving classification accuracy. Mahalanobis distance would require a clustering algorithm that generates covariance matrix estimates. A possibility is the one in Patter Classification (Duda et al. 2000). The task description states: “For practical purpose one should note that the images have been ”preprocessed”; i.e. centred and scaled to prepare them for classification”. The specific preprocessing steps are not explained, and as stated in (Duda et al. 2000), doing this correctly and sufficiently is challenging. Tangential distance metric might therefore still be the best option for this dataset.

5 Conclusion

Concluding we can say that our project was a success. We managed to learn and use theory to make two working classifiers with satisfactory performance. The implementation also went well, and the use of python made it easy to implement.

For the Iris task we managed to classify the different flowers with an estimated error rate as low as 3.3% (excluding testing using partition 2). The training was generally successful, with no problems relating to overfitting, but we learned that the Iris dataset given was somewhat biased. This was evident by observing that for both the ways of partitioning (table 1) the training/test set containing the first data samples performed better than their counterpart. Therefore it may be an idea to shuffle the dataset before partitioning. Using less features for classification also yielded good results, with classification using only the "petal width" of the flowers having an estimated error rate of 6.7%. A classifier not using the "sepal width" feature actually had equal performance to the original classifier, 3.3%, and should therefore seriously be considered if this was to implemented on a larger scale, as this would need less data and therefore be slightly more efficient.

Clustering as a preprocessing step for template-based classification was shown to significantly reduce runtime, but at the cost of increased error rates due to underfitting. This trade-off highlights the importance of selecting a suitable number of clusters per class. The choice of distance metric is thought to influence classification performance, with Euclidean distance possibly being suboptimal for this task. Future work could explore implementation of alternative metrics such as Mahalanobis or tangent distance and look into the joint optimization of clustering and choice of K in KNN.

References

CoopMega (2025). *Coop Mega tester avokadoskanner*. URL: https://www.coop.no/coop-mega/om-oss/baerekraft/coop-tester-avokadoskanner (visited on 29th Apr. 2025).

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

Johnsen, Magne H (18th Dec. 2017). *Classification*. URL: https://ntnu.blackboard.com/bbcswebdav/pid-2674007-dt-content-rid-78707810_1/xid-78707810_1 (visited on 29th Apr. 2025).

MathWorks (2025). *Overfitting*. URL: https://se.mathworks.com/discovery/overfitting.html (visited on 29th Apr. 2025).

Pedregosa, F. et al. (2024). *scikit-learn 1.4.2 documentation*. Accessed: 2025-04-30. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

Appendix

A Python code for training and testing of LDC

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
from sklearn import metrics

## computing functions
def sigmoid(zk: np.ndarray) -> np.ndarray:
    return 1 / (1 + np.exp(-zk))

def MSE(data: np.ndarray, targets, W):
    assert(len(data) == len(targets))

    Z = data @ W.T
    G = sigmoid(Z)
    return 1/2 * np.sum((G - targets)**2)

def gradWMSE(data, targets, W):
    assert(len(data) == len(targets))

    Z = data @ W.T
    G = sigmoid(Z)
    return ((G - targets) * G * (1 - G)).T @ data

def trainingStep(data, targets, W, alpha):
    return W - alpha * gradWMSE(data, targets, W)


##General training and test functions

def training(X_D, T_D, alpha, iterations):
    C = len(T_D[0])
    D = len(X_D[0])
    W0 = np.zeros((C, D))
    W = [W0]

    MSEList = [MSE(X_D, T_D, W0)]
    for i in range(iterations):
        W.append(trainingStep(X_D, T_D, W[i], alpha))
        MSEList.append(MSE(X_D, T_D, W[i]))

    return MSEList, W[-1]

def test(data, targets, W):
    assert(len(data) == len(targets))

    predicted = []
    actual = []
    errors = 0

    N = len(data)

    Z = data @ W.T
    G = sigmoid(Z)

    predicted = np.argmax(G, axis=1)
```

```
    actual = np.argmax(targets, axis=1)
    errors = np.sum(predicted != actual)

    cm = metrics.confusion_matrix(actual, predicted)
    errRate = errors/N

    return cm, errRate

def trainAndTest(data, features, partitioning, alpha=0.01, iterations=4000):
    # Partition and retrive correct data
    X_D, T_D, X_T, T_T = getData(data, features, classes, partitioning)

    _, W = training(X_D, T_D, alpha, iterations)

     # Test for training set
    cm, errRate = test(X_D, T_D, W)
    trainFig = plotConfusionMatrix(cm, classes.keys(), f'Training set \n iterations:
    ↪  {iterations}, ' + r'$\alpha$' + f': {alpha} \n error rate: {(errRate * 100):.2} %')

    # Test for test set
    cm, errRate = test(X_T, T_T, W)
    testFig = plotConfusionMatrix(cm, classes.keys(), f'Test set \n iterations:
    ↪  {iterations}, ' + r'$\alpha$' + f': {alpha} \n error rate: {(errRate * 100):.2} %')
    return trainFig, testFig
```

B Python code for template based classifiers

```
import numpy as np
from scipy.io import loadmat
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
import time
from sklearn.cluster import KMeans
from sklearn.cluster import MiniBatchKMeans
from sklearn.metrics import silhouette_score
from sklearn import metrics
from sklearn.datasets import make_blobs
import seaborn as sn
from collections import Counter



def NN_classifier(chunk_size):
    start_time = time.time()
    total_prediction = np.empty(num_test)

    for i in range(0, num_test, chunk_size):
        chunk_end = min(i + chunk_size,num_test)
        test_vector_chunk = test_vector[i:chunk_end]

        distance_matrix = cdist(test_vector_chunk, training_vector, metric='euclidean')
        prediction_vector = np.argmin(distance_matrix, axis=1)
        predicted_numbers = training_labels[prediction_vector]
        total_prediction[i:chunk_end] = predicted_numbers

    correct_predictions = (total_prediction == test_labels)

    error_rate = 1 - np.sum(correct_predictions)/num_test
    print("Runtime NN_classifier:", time.time() - start_time)
    return total_prediction, error_rate, correct_predictions

def KNN_classifier(chunk_size, K):
    start_time = time.time()
    total_prediction = np.empty(num_test)
    for i in range(0, num_test, chunk_size):
        chunk_end = min(i + chunk_size,num_test)
        test_vector_chunk = test_vector[i:chunk_end]

        distance_matrix = cdist(test_vector_chunk, training_vector, metric='euclidean')
        k_nearest = np.argsort(distance_matrix)[:,:K]
        k_nearest_labels = training_labels[k_nearest]
        predictions = np.array([Counter(row).most_common(1)[0][0] for row in
        ↪  k_nearest_labels])

        total_prediction[i:chunk_end] = predictions

    correct_predictions = (total_prediction == test_labels)
    error_rate = 1 - np.sum(correct_predictions)/num_test
    print("Runtime KNN_classifier:", time.time() - start_time)
    return total_prediction, error_rate, correct_predictions

def clustering(M):
    cluster_start_time = time.time()

    class_centres = np.zeros((num_classes, M, training_vector.shape[1]),
    ↪  dtype=training_vector.dtype)
```

```
58
59      for cl in range(num_classes):
60          class_i = training_vector[training_labels == cl]
61          kmeans = KMeans(n_clusters=M,random_state=42)
62          id_xi = kmeans.fit_predict(class_i)
63          class_centres[cl] = kmeans.cluster_centers_
64
65      templates = class_centres.reshape(-1, training_vector.shape[1])
66      template_labels = np.repeat(np.arange(num_classes), M)
67
68      cluster_end_time = time.time()
69      print("Clustering duration:", cluster_end_time - cluster_start_time)
70      return templates.reshape(-1, 784), template_labels
71
72
73  def cluster_NN_classifier(M, chunk_size):
74      start_time = time.time()
75      total_prediction = np.empty(num_test)
76
77      templates, template_labels = clustering(M)
78      for i in range(0, num_test, chunk_size):
79          chunk_end = min(i + chunk_size,num_test)
80          test_vector_chunk = test_vector[i:chunk_end]
81
82          distance_matrix = cdist(test_vector_chunk, templates, metric='euclidean')
83          prediction_vector = np.argmin(distance_matrix, axis=1)
84          predicted_numbers = template_labels[prediction_vector]
85          total_prediction[i:chunk_end] = predicted_numbers
86
87      correct_predictions = (total_prediction == test_labels)
88      error_rate = 1 - np.sum(correct_predictions)/num_test
89      print("Runtime cluster_NN_classifier:", time.time() - start_time)
90      return total_prediction, error_rate, correct_predictions
91
92
93  def cluster_KNN_classifier(M,chunk_size, K):
94      start_time = time.time()
95      templates, template_labels = clustering(M)
96
97      total_prediction = np.empty(num_test)
98      for i in range(0, num_test, chunk_size):
99          chunk_end = min(i + chunk_size,num_test)
100         test_vector_chunk = test_vector[i:chunk_end]
101
102         distance_matrix = cdist(test_vector_chunk, templates, metric='euclidean')
103         k_nearest = np.argsort(distance_matrix)[:,:K]
104         k_nearest_labels = template_labels[k_nearest]
105         predictions = np.array([Counter(row).most_common(1)[0][0] for row in
            ↪  k_nearest_labels])
106
107         total_prediction[i:chunk_end] = predictions
108
109     correct_predictions = (total_prediction == test_labels)
110     error_rate = 1 - np.sum(correct_predictions)/num_test
111     print("Runtime cluster_KNN_classifier:", time.time() - start_time)
112     return total_prediction, error_rate, correct_predictions
```