Classification of Iris flowers and handwritten digits

Edvard Vedeler
Eirik Runde Barlaug

April 29, 2022

NTNU
Department of Electronic Systems

Abstract

In this course we are to learn and apply theory on the topics of estimation, detection and classification. This specific project concerns the topic of classification and considers different aspects of and methods within classifying. Classification is highly relevant in popular technologies within artificial intelligence and machine learning, making this paper interesting for the general public. The task at hand asks us to train and test well suited classifiers in order to solve two different classification problems. First we classified Iris flowers with a linear discriminant classifier, then we classified handwritten numbers with a Nearest-Neighbour (NN) based classifier.

The linear classifier turns out to be a suitable classifier for the Iris flower classification problem. With a well tuned α and a sufficient number of iterations the training converges, and when classifying we get error rates no higher than 13.33%. Training the classifier with the full data set and applying 30 samples for training and 20 for testing from each class it classifies with an error rate between no lower than 3.333%. When removing features that overlap across the classes the trend is that the classification gets worse and we conclude that this can be explained by the fact that the overall separability is higher with more features.

The NN classifier performed well with error rates below 7% for both $k = 3$ and $k = 7$. Clustering the training data proved effective on run times, at the cost of accuracy. When plotting error rates as function of k, we concluded that $k = 3$ in fact was the optimal k for this specific data set.

Contents

1 Introduction

This report documents a project in the subject "TTT4275 Estimation, detection and classification" at NTNU. The project is divided into two subtasks, one task concerning the well known classification problem of classifying the Iris flowers Iris Setosa, Iris Versicolor and Iris Virginica, and the other task concerning classifying handwritten numbers from the MNIST database. The goal of the project is to apply relevant classification methods in order to classify both the Iris flowers and handwritten numbers with an acceptable error rate. Moreover, the project forces us to reflect over different relevant issues on optimizing the methods through the tasks we are to solve. An important part of the project concerns applying the theory in practice and implementing the functions and algorithms in order to plot and discuss the results.

The report is organized in the following way: In section 2, the relevant theory for the project is presented. It address theory on classification as a whole, and dives deeper into the more task specific theory on the topics of linear classification and theory concerning the K-nearest neighbours classifier. Theory needed to understand both the problems given and the implementation of the classifiers is covered in this section. In section 3 the Iris task is presented and in section 4 the classification problem concerning the handwritten numbers is presented. Both sections describes their respective tasks, explains the implementation of the classifiers and discusses the results thoroughly. Different figures and plots display and visualize the results and these are seen in context with the theory presented in section 2.

2 Theory

The following section will present the necessary theory for understanding the classification problems presented in section 3and 4, as well as the project as a whole. The section contains some theory regarding general classification, as well as in-depth explanations of some techniques that have been used extensively throughout the project.

2.1 General classification

Classification is a method for dividing data into *classes* based on given decision rules. The data may describe features of some entity that has been measured in some way, and the goal is to distinguish the entities from one another with just the given data.

An example may be to identify a dog from a giraffe. Intuitively, a human will classify the animal by comparing *features* that separates dogs from giraffes, e.g. size, shape, color, length of the fur, length of the neck and so on. For a human, a decision rule may be: "if the animal has a very long neck and does not say "woof", then it is a giraffe". As we will see, this way of classifying something is not too far away from what classifiers in computers do. The main goal in classification is to make the *error rate* (i.e. how often something is misclassified) for the classifier as small as possible. This error depends on the type of classifier and decision rule that is used, what features the data describes and the amount of data that is accessible.

Classification problems are divided into the three main categories *linearly separable*, *non-linearly separable* and *non-separable* problems. For the first two it is possible to completely separate the data, either linearly or non-linearly, into distinct classes as the overlap is non existent. This makes it theoretically possible to have an error rate of zero if the decision rule is chosen correctly. However, this is seldom possible when the problems grow in complexity and size and becomes non-separable with overlapping data points. Figure 1 shows what the three main categories may look like in two dimensions.

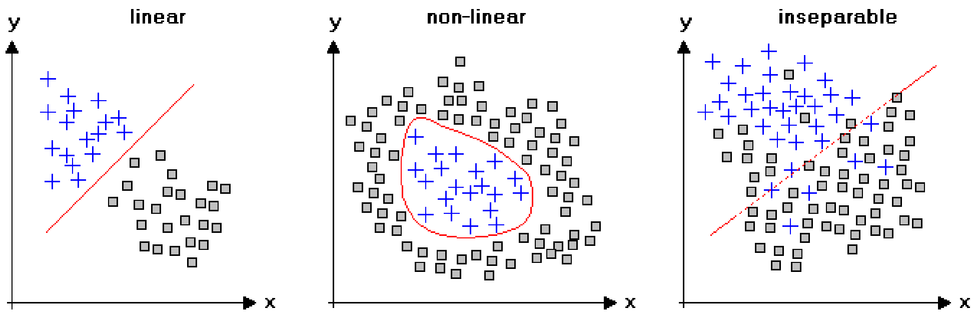


Figure 1: An example of what separable and non-separable problems may look like in two dimensions. Source: [1]

2.2 The linear discriminant classifier

The linear classifier is a simple form of classifier for linearly separable problems. In practice these occur very seldom. In some cases however, a linear classifier may be applied in order to classify problems who are *close to* linear separable. This is because the errors can be considered as negligible.

The decision rule for a discriminant classifier is given by

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

The linear discriminant classifier on matrix form is given by equation 2. [2]

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

w_0 is here the offset and C is the number of classes. g and w_0 are vectors with C elements, x is a vector with D elements (one for every feature) and W is a CxD matrix. By defining $W = [W \;\; w_0]$ and $x = [x^T \;\; 1]$, one may write 2 on the form

$$g(x) = Wx, \tag{3}$$

and the training of the linear classifier simplifies to finding the right W matrix. In order to train the matrix W we want to minimize the error between the output of the classifier and the target values. A popular approach to optimizing the discrimination of the classes is to minimize the Minimum Square Error between the target values and output vectors for all N samples. The MSE function is shown in equation 4.

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

However, there exists no explicit solution to equation 4. Therefore, one can apply a gradient based technique to minimize the MSE. An expression for minimizing the MSE is given by equation 5.

$$\begin{aligned} \nabla_W MSE &= \sum_{k=1}^{N} \nabla_{g_k} MSE \nabla_{z_k} g_k \nabla_W z_k \\ &= \sum_{k=1}^{N} [(g_k - t_k) \circ g_k \circ (1 - g_k)] x_k^T \end{aligned} \tag{5}$$

Where t_k is the true label and ($\circ$) means element-wise multiplication. x_k is the feature vector for the k'th sample and $g_k = Wx_k$ is the output vector corresponding to a single input x_k. The *sigmoid* function, given in equation 6, is applied as a continuous approximation of the Heavyside function and is important for mapping the continuous outputs of the linear discriminant function to binary values in order to be able to compare them with the target values.

$$g_{ik} = \frac{1}{1 + e^{-Wx_{ik}}} \quad i = 1, ..., C \tag{6}$$

To minimize the MSE in practice, the weight matrix W is trained iteratively and updated in the opposite direction of the gradient of the MSE as follows

$$W(m+1) = W(m) - \alpha \nabla_W MSE \;\; . \tag{7}$$

Where m is the iteration number and α the size of the step we take. Upcoming sections will call attention to the importance of a well tuned α; it should not be too large and not too small. Also, the number of iterations plays an important role when it comes to the convergence and in turn the performance of the classifier.

2.3 The K-nearest neighbours classifier

The K-Nearest Neighbour algorithm (kNN for short) is a template based classifier. It takes a d-dimensional real input vector $\boldsymbol{x}$, compares it to a set of references/templates $\boldsymbol{r}$ on the same form as $\boldsymbol{x}$, and outputs what class $\boldsymbol{x}$ belongs to or is "most similar" to. The decision rule is quite simple and says that $\boldsymbol{x}$ must belong to the same class as *the majority of* the k nearest references (or "neighbours"). The distance to the individual references is measured by finding the Euclidean distance between x and the given reference, $\boldsymbol{r_i}$, as

$$d(\boldsymbol{x}, \boldsymbol{r_i}) = \sqrt{\sum_{j=1}^{N} (x_j - r_{ij})^2}, \tag{8}$$

where N is the dimension of both $\boldsymbol{x}$ and the reference vector $\boldsymbol{r_i}$. $\boldsymbol{r_i}$ can be chosen directly from the training data set. Figure 2 illustrates what the process might look like in two dimensions.

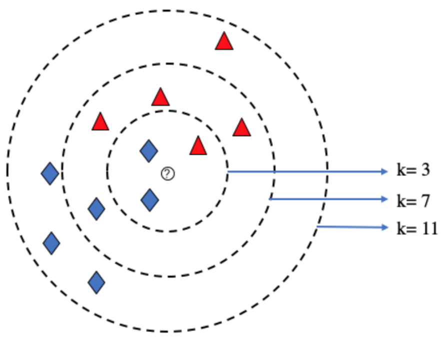


Figure 2: A simple illustration of a data point's nearest neighbours for $k = 3, 7, 11$. Source: [3]

.

In figure 2, $\boldsymbol{x}$ is the circle with a question mark and is to be classified as either blue diamond (BD) or red triangle (RT). For $k = 3$, the three closest neighbours are two BD and one RT. BD has the majority, so kNN outputs that $\boldsymbol{x}$ is BD. For $k = 7$, $\boldsymbol{x}$ is RT for the same reason, and for $k = 11$, $\boldsymbol{x}$ is BD. This also shows that the choice of k may affect the algorithm. Lastly, it should be noted that when $k > 1$ the algorithm is called "K-Nearest Neighbours", but if $k = 1$ it is simply called "Nearest Neighbour" (NN) and $\boldsymbol{x}$ is only matched to the closest reference.

2.4 K-means clustering

Clustering is a way of reducing a set of N observations (data points) into M clusters. Each cluster is a subset of observations that are similar (enough) to one another in some way, such that one can group them. Because of these similarities, each cluster centroid (the mean of the cluster) may be considered as a sufficiently good exemplar for the points inside that specific cluster. In this manner, it is possible to reduce a data set from N points to $M << N$ points. This can make it simpler to work with and possibly reduce run times and computational complexity. *K-means clustering* is simply an iterative algorithm for doing so.

More specifically, K-means aims to partition a set of n observations $(\boldsymbol{x_1}, ..., \boldsymbol{x_n})$, where each observation $\boldsymbol{x_i}$ is a d-dimensional real vector, into $k < n$ subsets $\boldsymbol{S} = \{S_1, ..., S_k\}$ by minimizing the within cluster variance. That is,

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu_i}\|^2 = \arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} |S_i| Var(S_i), \tag{9}$$

where $\boldsymbol{\mu_i}$ is the mean of the cluster points in S_i. The standard (sometimes called "naive") K-means algorithm starts off by selecting a random seed of k clusters means. It then repeats the following two steps until convergence:

I. **Assignment step**: Assign each observation to the cluster with the nearest mean. Distance is calculated using Euclidean distance (8).

II. **Update step**: Recalculate the means (centroids) for the observations that have been assigned to each cluster.

The iteration stops when the cluster means no longer move (or moves very little), i.e. when the assignments does not change from one iteration to the next. The algorithm is not guaranteed to find the global optimum when converged.

2.5 Error rate and the confusion matrix

One way to measure the performance/accuracy of the classifier is by using the *error rate*. The total error rate ER_T is the proportion of misclassified instances over the whole set of instances. That is,

$$ER_T = \frac{FP + FN}{TP + TN + FP + FN}, \tag{10}$$

where FP, FN, TP and TN is the number of false positives, false negatives, true positives and true negatives, respectively. The lower the error rate, the more accurate is the classifier.

A more detailed way of visualizing the results attained by the classifier is though a *confusion matrix*. The confusion matrix is a CxC-matrix, where C is the number of classes. It holds true labels along the y-axis, and predicted labels along the x-axis. An example is given in table 1.

		Predicted			
		A	B	C	D
Actual	A	96	1	44	13
	B	0	97	3	3
	C	38	0	70	1
	D	14	5	0	99

Table 1: Example of a confusion matrix for four classes A,B,C and D.

In this way, it is easy to see if the classifier often mislabels some of the classes as another. If it does, then those classes are "close" in the input space, so the classifier has a hard time telling them apart. E.g., in table 1, it seems as if the thought classifier struggles to tell class A apart from class D. However, it does not struggle with class B. A theoretically optimal classifier would have an error rate of zero and a diagonal confusion matrix.

3 The Iris task

This section presents the task concerning classification of the Iris flowers. Firstly, the task is described, then the implementation of the classifier in Python is explained and seen in context with the methods presented in the theory. Finally, the results are presented and discussed in light of the theory.

3.1 Task description

The objective of the task is to implement a linear classifier in order to handle a well known problem within classification, namely the Iris flower classification. Our task is to train and test a classifier for the database called "Fisher Iris data", which contains measurements of three types of Iris flowers. Using measurements of the flowers' Petal and Sepal leaves one should be able to discriminate the three flowers. Because the problem is close to linearly separable, one can design a linear classifier to classify the flowers.

The database contains 150 samples, 50 for each of the three classes (flowers). Every sample has four features: width and length of both the Sepal and Petal leaves.

The task is divided into two main parts. In the first part, we are to design and evaluate a linear classifier. Training a linear classifier concerns implementing the discriminant function and training the weighting matrix. Using the gradient based optimization algorithm we are able to train the classifier until it converges, as long as we apply a suitable number of iterations and a good enough step length, α, as described in section 2.2.

In the second part we remove some of the features and observe how the performance of the classifier changes. When removing features with the most overlap across the different classes, one should think that the performance improves, regarding the linear separability. The results will be visualized and discussed in section 3.3.

3.2 Implementation

The Iris task was in its entirety implemented in Python 3.8.3. The classifier was implemented with the assistance of the libraries "NumPy", "Matplotlib" and "sklearn". Numpy and Matplotlib were helpful for linear algebra routines and plotting, respectively. Sklearn made the loading of the database easier and made the visualization of the results from the classification neat and easy to interpret. Furthermore, these libraries made the program more readable. The code is split into the files `iris.py` and `plot.py` in order to make the code somewhat more tidier.

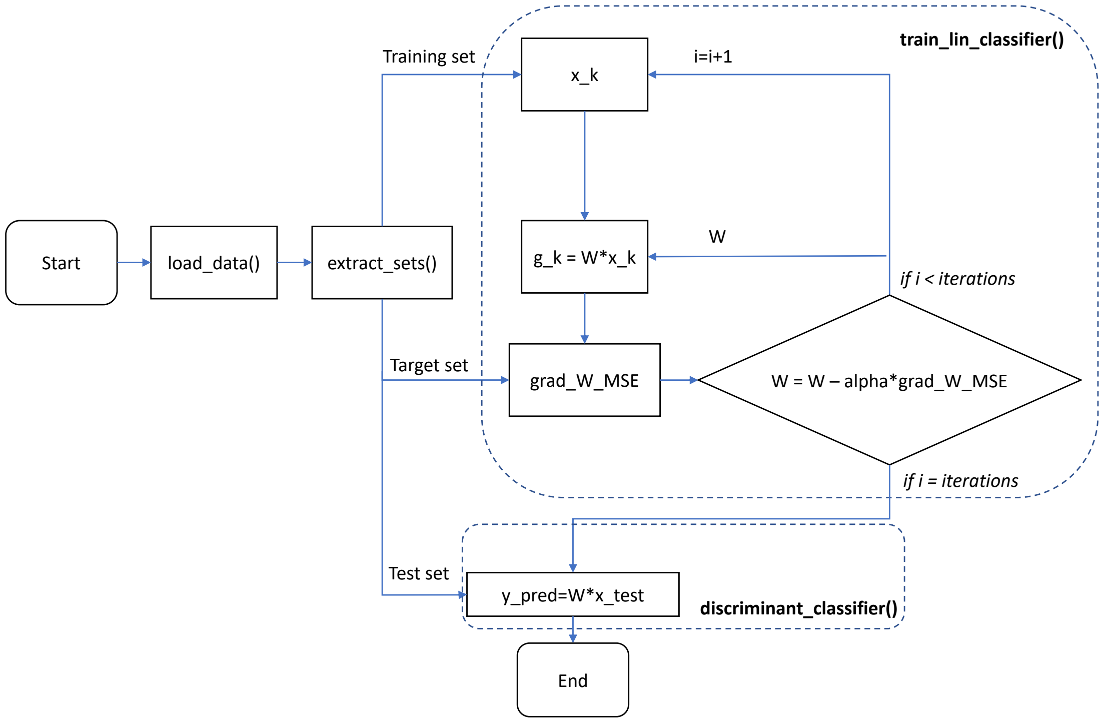


Figure 3: Flowchart describing the extracting of data sets, the training and the classification for the implemented classifier.

Figure 3 shows how the data is applied and how the training and classification procedures are implemented specifically for our system. Initially, the data is loaded and split it into the sets of data we want. The function `extract_sets()`, splits the data into training- and test sets when given a `np.ndarray` of the raw iris data set along with a target vector, number of classes and the desired training set size. Then, in order to implement the discriminant function and train the linear classifier, we calculate the matrix W in our linear classifier. In order to do this by implementing the iterative algorithm described in equation 7 in section 2.2, a satisfying step size, α, and a good number of total iterations must be found. Some experimenting on different values of α and how fast the model converges for different numbers of total iterations is therefore done. The results of this will be displayed and discussed in section 3.3. However, implementation-wise the experimenting is conducted with the function `display_ER_MSE_iterations()` in `plot.py`. Here , the classifier is trained for four different values of α and the error rate and MSE value are sampled for every 20th iteration. In that way we can obtain plots of the convergence of the training.

With a satisfying α and sufficient number of iterations we can start solving the explicit tasks given. The model is trained in the function `train_lin_model()`, which returns the trained 3x5 matrix W to be used in the linear classifier. The function `discriminant_classifier()` takes the entire test set as input, and returns its prediction.

In the second part we made histograms for every feature, and were to remove the features with the most overlap across the classes. Then, we removed one feature at a time until we were left with only one feature. The performance of the classifier with the different features removed and some discussion around the results can be seen in section 3.3.

3.3 Results and discussion

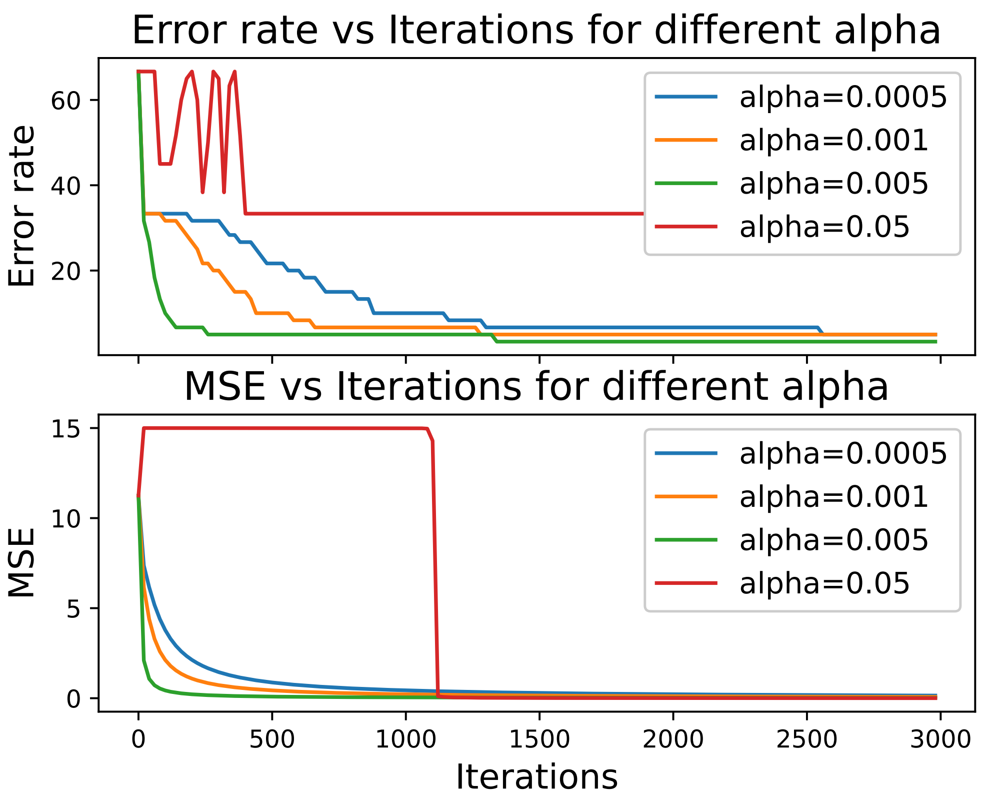


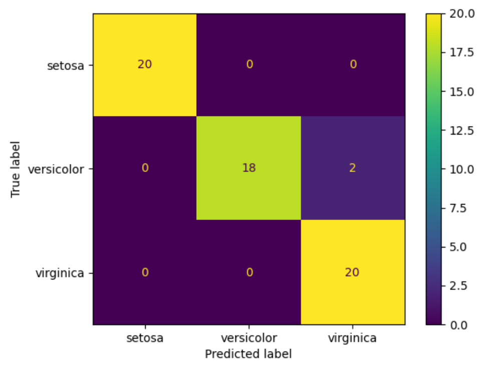
Figure 4: Error rate and MSE values plotted against iterations for different values of alpha.

In figure 4, error rates and MSE's are both plotted against the number of iterations for four different values of α. The iterations run from 0 to 3000 and in the error rate plot we see how three out of the four α results in error rates that converge to an acceptably low number within an acceptable number of iterations. However, when $\alpha = 0.05$ the training does not converge within the number of iterations. Furthermore, an α value too large will result in the fluctuating MSE, as seen from the MSE plot. This might result in a non-converging error rate. With an α value too small, i.e. $\alpha = 0.0005$, the model converges slower as expected.

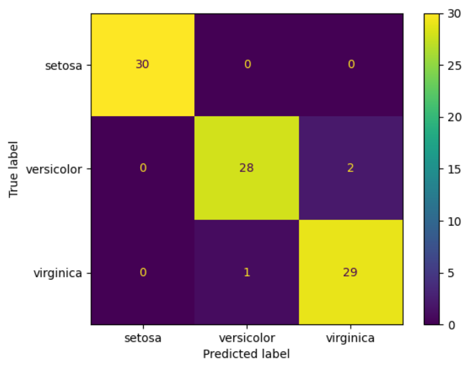
The plot of MSE plot tells about relatively fast convergence towards 0 for all values of α except for $\alpha = 0.05$. For $\alpha = 0.05$ the MSE moves in the wrong direction and stays there for some time before it converges first at about 1200 iterations. The optimal α seems to be somewhere close to $\alpha = 0.005$.For the iterations, it seems as if everything from about 1500 to 3000 iterations will work well. The optimal value for both of these attributes are, however, of course dependent on the training set and other factors.

We chose $\alpha = 0.006$ and a total number of iterations equal to 2000. This led to convergence and made the classifier perform well. Therefore, these values are used for the rest of the task. It is worth mentioning that these are close to optimal for the case where we apply all four features, and the first 30 and last 20 samples from each class are used for training and testing, respectively. The error rate is dependent on α, but also on the test and training set. For different sets one may end up with different results. For the tasks where we apply the 20 first for testing and the last 30 for training, and for the cases where we remove features, our chosen values for α and number of iterations may not lead to a that good performance. However the plot in figure 4 takes many factors into account and covers quite a few cases, so we find it rather descriptive.

Starting with the case where we apply the first 30 samples and the last 20 samples from each class for training and testing respectively, the confusion matrices for both the training set and test set for the trained classifier is given in figure 5.



(a) Test set. ER = 3.333%

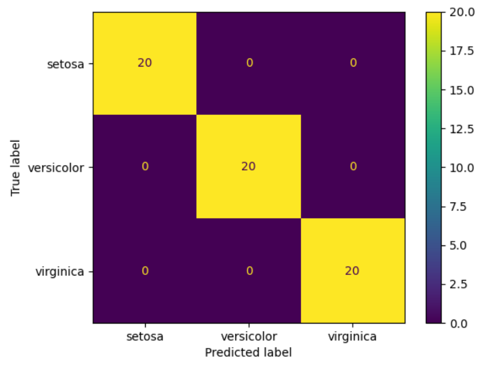


(b) Train set. ER = 3.333%

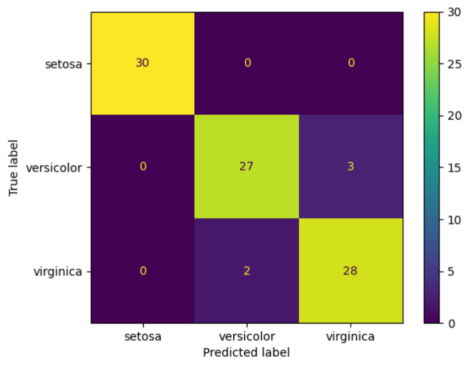
Figure 5: Confusion matrix. Applying the 30 first samples for training and 20 last samples for testing

The error rate for both of the cases in which the confusion matrices are displayed above are $ER = 3,33$. This, we consider sufficiently low. Observe how the Iris Setosa is correctly classified for both cases above, as well as for the cases to come. The non zero values on the off diagonal elements of the confusion matrices (i.e. a non zero error rate) is explained by the fact that this is *not* a completely linear separable problem after all.

When applying the 30 *last* samples for training and the 20 *first* samples for testing from each of the classes, the performance improves when classifying the test set, but gets worse when classifying the training set. The only reason we can see for this is that there are differences in the samples as well. This can be a result of some parts of the set being more linearly separable than others. In this case specifically, the results suggests that the first part of the samples are more linearly separable than the last. In the histogram in figure 7 one can see how some parts of the measurements overlap more with other classes. The confusion matrices for these cases are shown in figure 6.



(a) Test set. ER = 0.0%



(b) Train set. ER = 5.555%

Figure 6: Confusion matrix. Applying the 30 *last* samples for training and 20 *first* samples for testing

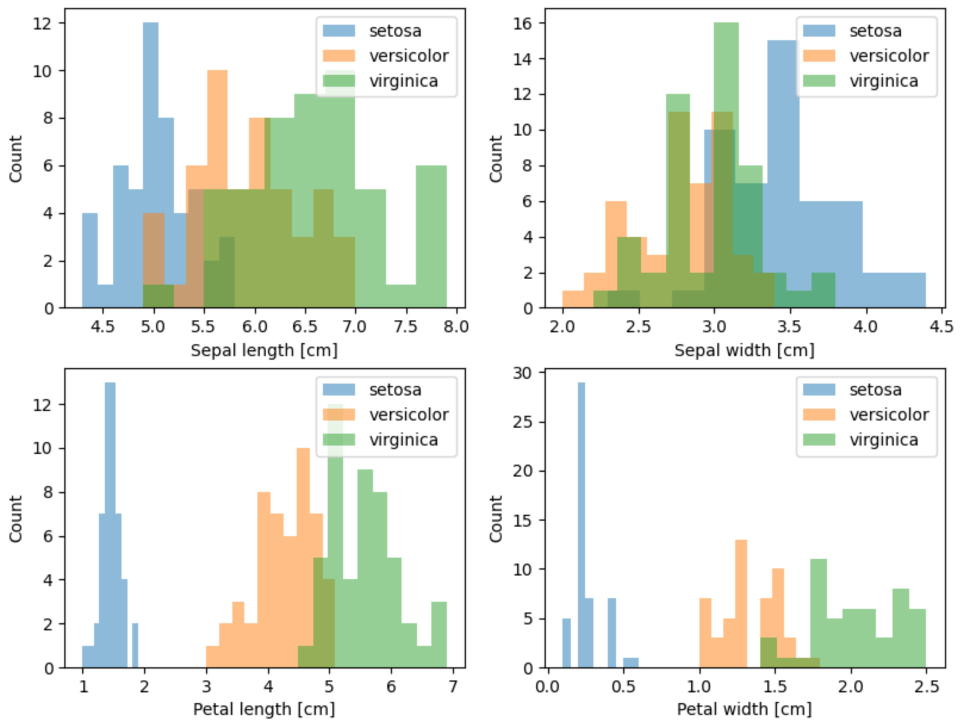
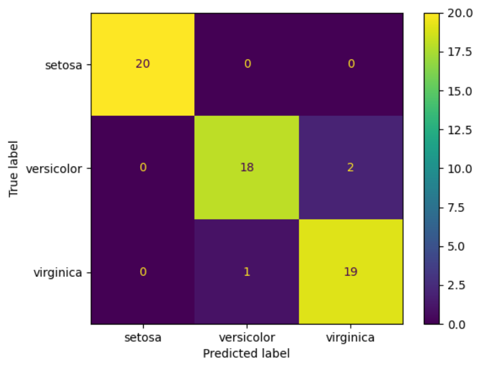


Figure 7: Histogram showing every feature for the different classes. All 150 samples are included.

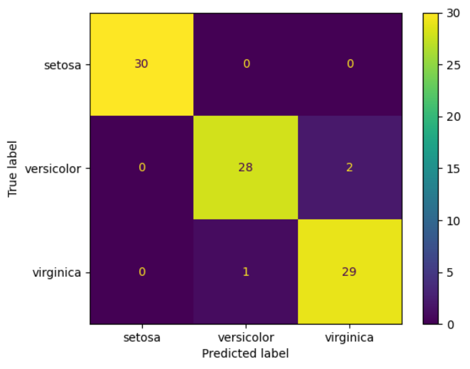
The histogram in figure 7 shows the features from each class and how they overlap. It occurs that the length and width of the Sepal leaves is the feature that is the most difficult to separate among the classes. The overlapping of features across the classes is problematic for the performance of the linear classifier.

Notice also the alone, distinct pillars of the measurements of Iris Setosas Petal leave lengths and widths. This feature makes the Iris Setosa class more linearly separable compared to the other classes, and it is easy to see how the classifier very seldom misclassifies the Iris Setosa. In fact, we will see how the tuned classifier does not get Iris Setosa wrong in any of the cases presented in this report.

When observing the histograms, one may wonder if the classifier would improve if we remove the feature with the most overlap. In the following, confusion matrices we show the performance of the classifier when we first remove the Sepal width feature, then the Sepal length feature and lastly the Petal width. We once again apply the first 30 samples from each class for training and the last 20 samples for testing.

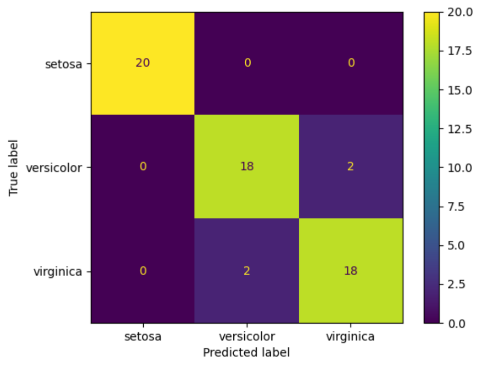


(a) Test set. ER = 5.000%

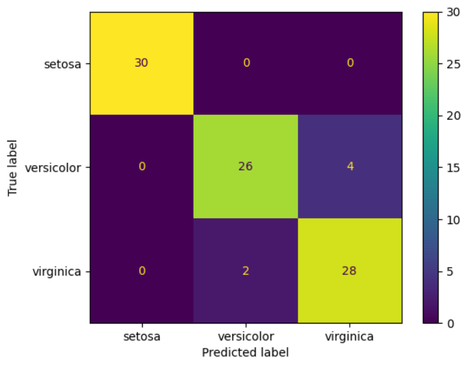


(b) Train set. ER = 3.333%

Figure 8: Confusion matrix. Removed Sepal width

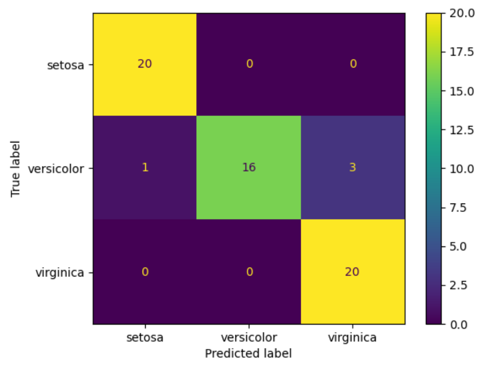


(a) Test set. ER = 6.666%

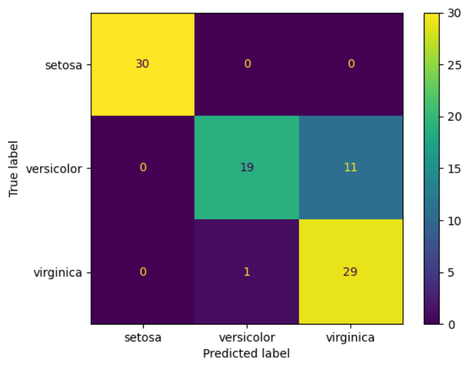


(b) Train set. ER = 6.666%

Figure 9: Confusion matrix. Removed Sepal width and Sepal length



(a) Test set. ER = 6.666%



(b) Train set. ER = 13.333%

Figure 10: Confusion matrix. Removed Sepal width, Sepal length and Petal width

In the figures 8, 9 and 10 the results of removing one feature after the other are displayed and visualized with the confusion matrices for both the test set and training set. For comparison, see figure 5 where we observe the confusion matrices with all features included when training and testing the linear model. Table 2 shows the error rates for the four cases.

	4 features	**3 features**	**2 features**	**1 feature**
Error rate test set	3.333%	5.000%	6.667%	6.667%
Error rate train set	3.333%	3.333%	6.667%	13.333%

Table 2: Error rates when removing one more feature at the time for the test and training set.

We see from both the figures and the table how the error rates tend to increase as we remove features. This might seem counter intuitive and is not necessarily the case in general when removing overlapping features. A classifier might, when training with overlapping samples for some features, become more "confused" and perform worse. However, in our case it seems like the classifier gains more on being able to train on more features than training on less but somewhat more distinguishable features. In other words, a higher number of features does increase the overall separability for our classifier. Worth mentioning is that in one way the training sets are different sets when removing features and one might be able to get a better performance with an α and number of total iterations tuned specifically for each case. However, we believed this was outside the scope of the task and that the purpose of this experiment was to compare the different cases on the same basis.

Finally, we observe from table 2 the differences between the error rates for the test set and training set within the different cases. Sometimes the training set gets the higher error rate and sometimes the test set gets the higher error rate. This is for the same reason as mentioned in the discussion around figure 6. Removing features, and in that way changing the data set used for training and testing, results in different performance. The fact that it varies whether the training set or the test set gets the highest error rate is in this case a result of some parts of the data set becoming more linearly separable than others depending on what feature is being removed.

4 The digits task

In this section the handwritten numbers classification problem is presented. Firstly, the task is described and then the implementation of the methods needed to train and test the classifier is explained. Finally, the results from the task are presented as plots and confusion matrices and are discussed.

4.1 Task description

The goal of the task is to implement a classifier that can predict the correct label for images of handwritten digits. The data set we train and test on is acquired from the MNIST-database, which contains 70 000 28x28 pixel images of handwritten digits. 60 000 for training and 10 000 for testing. Examples from the data set are given in figure 11. **For all tasks, the 28x28 pixel images are flattened to 1x784 vectors.**

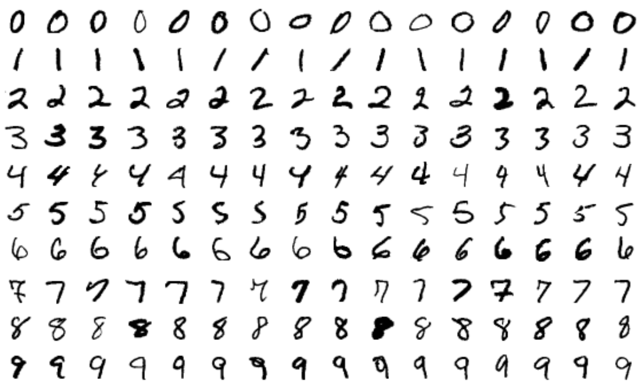


Figure 11: Example of 160 images from the MNIST data set. Source: [4]

The task is divided into two parts. In the first part, a NN-based classifier is designed using Euclidean distance evaluation. The data set is divided into n chunks, and error rates and confusion matrix for the test set is obtained and plotted. Furthermore, some correctly classified, and some misclassified images are plotted. In the second part, K-means clustering is used to cluster the entire training set into templates for each class. Then the NN classifier is applied once again with different K on the clustered data, and error rates and confusion matrices are obtained and plotted. Run times are compared to the first system, i.e. the one with no clustering.

4.2 Implementation

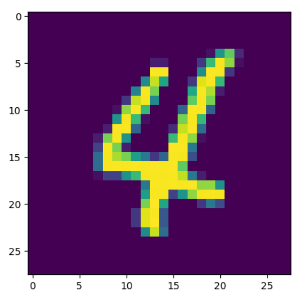
The classifier was implemented in Python 3.9 with the assistance of the libraries "NumPy", "sklearn" and "Keras". NumPy was used for linear algebra routines, sklearn was used for the kNN-algorithm, K-means clustering and other helpful utilities, and Keras was used for importing MNIST. The imports made the program more readable, and minimized the amount of unnecessary boiler plate coding. As the imported algorithms are quite simple, there is no need for us to write them ourselves.

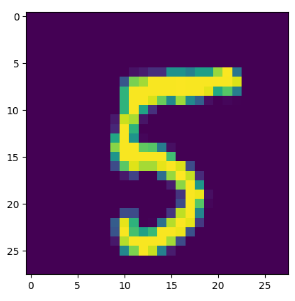
For the first part, a kNN-classifier with $k = 3$ was implemented with Euclidean distance evaluation. For the choice of k, we generally want $k > 1$ to make noise in the data less influential, i.e. we want to avoid over-fitting of the boundaries in the classifier - small values for k leads to high model complexity. Also, we do not want a too large k as this is more computationally expensive, and may lead to under-fitting. Also, the second part of the task required $k = 7$, so it seemed as if $k = 3$ was a good initial choice. Additionally, as we will see, it performed better than many other values of k. Further, choosing an odd k rules out ties in the cases where the classifier tries to distinguish two classes from one another. There exist ways of finding the optimal k, but that is beyond this task, and may be a natural extension of the program some time in the future.

In the second part, we clustered *each class* into 64 clusters with K-means and used the cluster centroids as new template vectors for every class, we then implemented a new kNN classifier with the clustered training data. To cluster the training vectors *class-wise*, we first had to sort the images by class. This is done in the `sort_data()` function, which is called in the `cluster()` function. With clustering, we effectively reduced the training data set from $n = 60000$ to $n = 640$ observations, which is a 98,93% decrease.

4.3 Results and discussion

The classifier worked properly with error rates below 7% for all experiments. It is interesting to see if we humans agree with the choices of the classifier. Some examples for both cases are therefore given in figure 12 and 13, respectively.


(a) True: 4, Predicted: 4


(b) True: 5, Predicted: 5

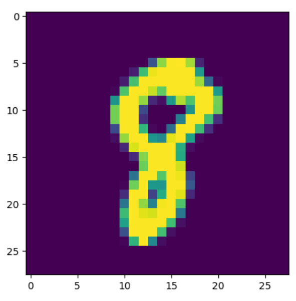
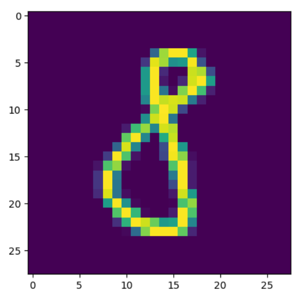
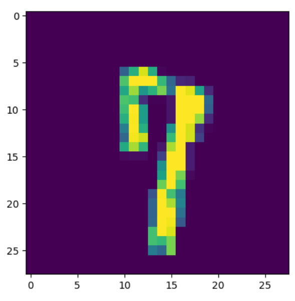

(c) True: 8, Predicted: 8

Figure 12: Example of three correctly classified digits.


(a) True: 8, Predicted: 3


(b) True: 7, Predicted: 9

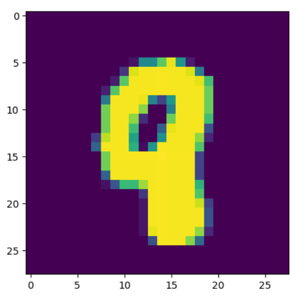
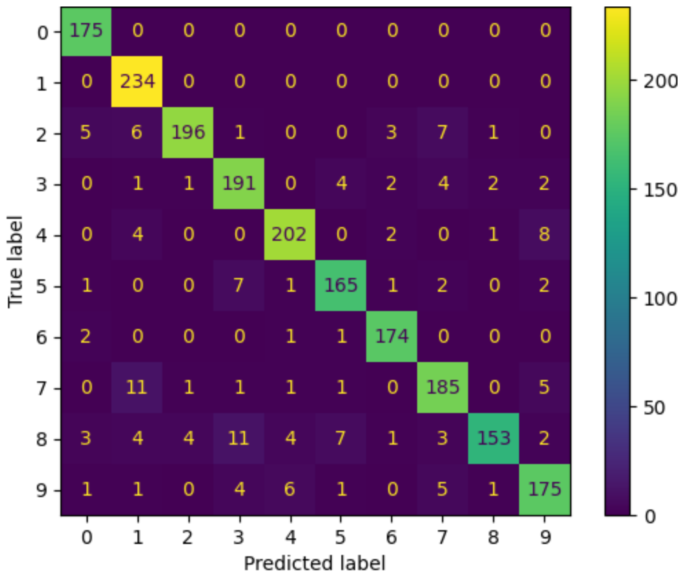

(c) True: 8, Predicted: 9

Figure 13: Example of three misclassified digits.

For the part where the data set was split up into n chunks, an error rate of $ER_T = 0.07$ was obtained. For this specific error rate, the training set contained only 12 000 images, and the testing set was of size 2000. That is, $n = 5$. Naturally, the splitting of the data lowered the computational cost and run times, and it made it easy for us to test out the classifier early in the project without having to wait too long for the model to train. The way we interpret the task, that was also the intention. The confusion matrix for the kNN on the small data set is shown in figure 14. Clearly, the numbers inside the confusion matrix only add up to 2000, and some entries are therefore smaller than what we will see in later confusion matrices when the whole data set is applied.



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	175	0	0	0	0	0	0	0	0	0
1	0	234	0	0	0	0	0	0	0	0
2	5	6	196	1	0	0	3	7	1	0
3	0	1	1	191	0	4	2	4	2	2
4	0	4	0	0	202	0	2	0	1	8
5	1	0	0	7	1	165	1	2	0	2
6	2	0	0	0	1	1	174	0	0	0
7	0	11	1	1	1	1	0	185	0	5
8	3	4	4	11	4	7	1	3	153	2
9	1	1	0	4	6	1	0	5	1	175

Figure 14: Confusiom matrix for the first kNN classifier, trained on 12000 images, and tested on 2000.

Notice for instance the intersection of the predicted 3 and correct 8, which is somewhat higher than other entries. An example of one out of those eleven misclassified (3,8)-pairs was shown in figure 13a. The high number indicates that some 3's intersect with 8's in the higher dimensional input space. As a result, some digits are falsely predicted. The same holds a few other digits, like (1,7) and (9,4). In the cases where the classifier misses, the image's k nearest neighbours in the d-dimensional input space belongs to the predicted digit, and not the true digit. Thus, the image is falsely predicted, as theory suggests.

For the remainder of this section, the whole data set is used for both training and testing. K-means clustering was introduced on the whole training data set as described in subsection 4.2. A few examples of cluster centroids for the labels "7" and "8" are given in figure 15 and 16, respectively. These plots may be considered as "average" sevens and eights for three of the 64 clusters in each label. Notice that they are quite blurry compared to the digits from figure 12 and 13, which is to expect as these in fact are the means of approximately $\frac{6000}{64} \approx 93$ images.

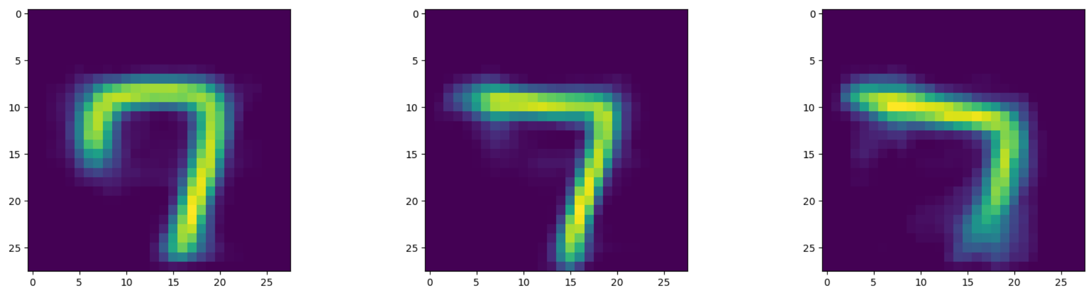


Figure 15: Three cluster centroids for the label "7".

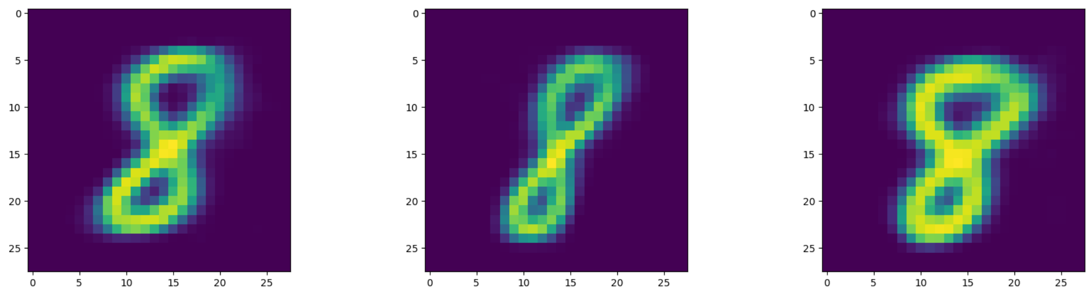


Figure 16: Three cluster centroids for the label "8".

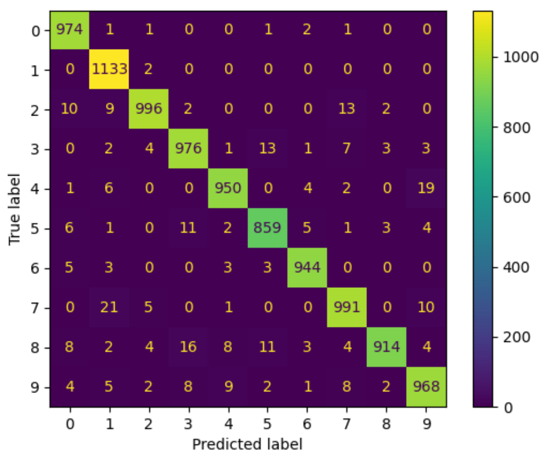
When class wise K-means clustering was implemented, the run times for both fitting the classifier and predicting the test images improved. However, the error rates increased. In table 3, run times and error rates for $k = 3$ and $k = 7$, both with and without clustering is shown.

Method used	**Accuracy**	**Error rate**	**Fit time [s]**	**Prediction time [s]**
k = 3, no cluster	0.97	0.03	0.1071	24.4330
k = 3, cluster	0.94	0.06	0.0040	0.5279
k = 7, no cluster	0.97	0.03	0.0958	23.9090
k = 7, cluster	0.94	0.06	0.0020	0.6124

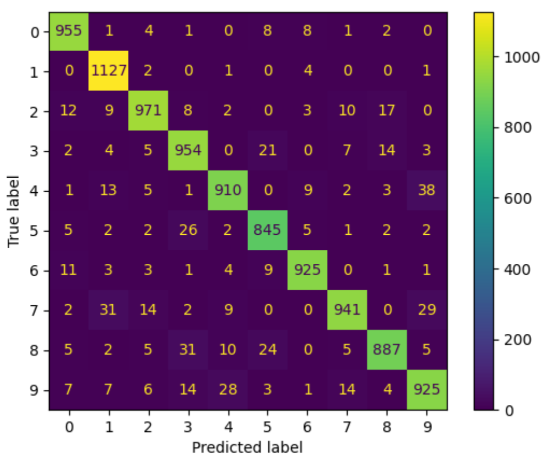
Table 3: Results after implementing kNN with $k = 3$ and $k = 7$, both with and without K-means clustering.

When inspecting table 3 it should be noted that **the process of clustering took a total time of 71.537s**. It is evident that clustering greatly improves both fit- and prediction times. That is, fit times are reduced by factors of approximately 27 (k=3) and 48 (k=7), and prediction times are reduced by factors of 46 (k=3) and 38 (k=7). If the data is re-clustered every time the program runs, this difference is not significant as the clustering takes a lot of time. However, if the clustered data is saved (to a file) once obtained it may be used several times. If so, the clustering method greatly improves run times.

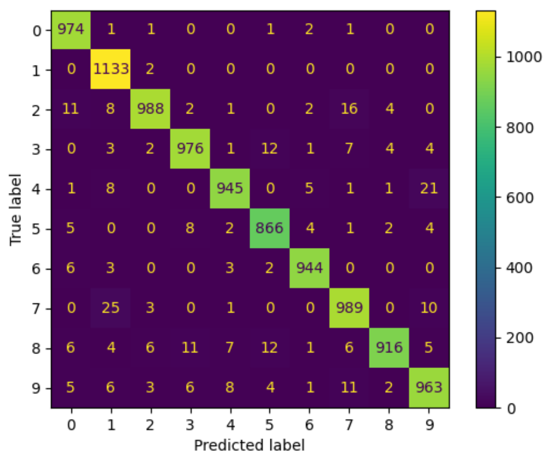
The improvement in fit times are because of the reduction of the size of the training set, making the fitting algorithm run through fewer vectors when fitting it to the model. The prediction times are also improved. This is because for each image to be predicted, the classifier has to calculate the distance to fewer template vectors, as the training set has been shrunk. As seen, the costs of lowered run times are increased error rates, as can be observed for both k. This may have been avoided if we instead of choosing k at "random" had found some optimal number of clusters for K-means. This may be done using cluster validation, and the *elbow method*[5] is a rather common way to choose an "optimal" k. This is perhaps a reasonable way to extend the program in the future. The increase in error rate is quite small, and one can argue that the large decrease of run times are worth the small drop in accuracy. The confusion matrices for the four aforementioned cases is given in figure 17.



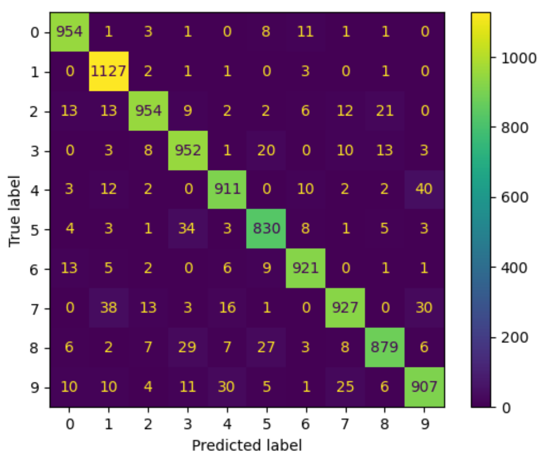
(a) $k = 3$, without clustering.



(b) $k = 3$, with clustering.



(c) $k = 7$, without clustering.

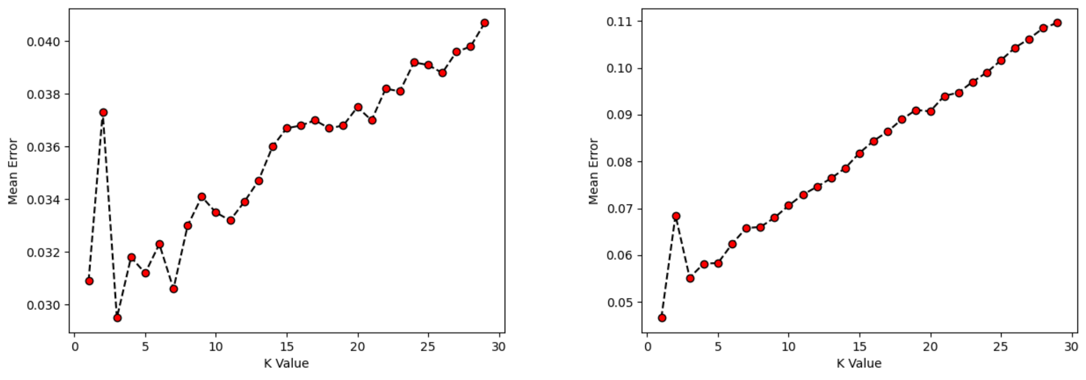


(d) $k = 7$, with clustering.

Figure 17: Confusion matrices with and without clustering for $k = 3$ and $k = 7$.

As seen, the off diagonal elements of the two matrices *with* clustering are particularly higher than for the two other matrices. This is equivalent to the error rate being higher. It also seems as if the classifier misclassifies the same digits as when the data set was split into n chunks in the first part of the task.

Lastly, one may notice from table 3 that the error rate remains (approximately) the same for both $k = 3$ and $k = 7$. This may seem contrary to the theory, as one would expect the classifier to perform better for increasing k, making the decision boundaries smoother and less susceptible to noise. However, this is not always the case, and for this particular data set $k = 3$ proved effective. Figure 18 displays the error rate for the classifier as a function of k. The error seems to increase linearly with k, and it seems as if the optimal k in fact was 3. That is, for all $k < 30$. Furthermore, notice the increased error for all 30 k in the clustered case 18b compared to the un-clustered case in 18a.



(a) Without clustering. (b) With clustering.

Figure 18: The classifiers mean prediction error as a function of k, with and without clustering of the training data.

5 Conclusion

The linear classifier proves itself suitable for the problem of classifying the Iris flowers. We trained the classifier with a learning rate of $\alpha = 0.006$ and 2000 iterations. When applying 30 samples from each class as training set the classifier is able to classify 20 samples with an error rate between 0.0% and 3.333%, depending on which parts of the samples are used for training and which parts are used for testing. This variation when applying different parts of the samples for training and testing showed how some parts of the data set were more linearly separable than others.

We then moved focus to the features of the classes and how they play an important part in the training of the classifier as well as for the linear separability of the data set. The confusion matrices and error rates presented in the task proves how the Iris flower problem only is *close* to a linear separable problem. Some features overlapped less across classes than others and were more important for the performance of the classifier. However, removing the ones that overlapped more did not improve the performance of the classifier. In fact the opposite happened to be the result. We conclude that this is a result of the features contributing to a higher overall separability for the data set given.

The kNN approach for classifying the handwritten digits from the MNIST data base proved effective. We managed to predict the test set to an accuracy of 0.97 for $k = 3$ and $k = 7$. However, the run times were quite high. When we clustered the training data with class wise K-means clustering, the kNN run times for both fitting the data and classifying the test set was reduced by a factor of 30 to 40 for both k. However, the accuracy dropped by 0.3.

References

[1] Knn illustration, 2020. Last accessed 21 April 2022. https://www.researchgate.net/figure/k-nearest-neighbor-diagram.

[2] M. H. Johnsen. *Classification*. NTNU, 2017.

[3] Classification vs. calibration. Last accessed 21 April 2022. http://www.statistics4u.com/fundstat-eng/cc-classif-calib.html.

[4] Mnist dataset samples, 2022. Last accessed 26 April 2022. https://en.wikipedia.org/wiki/MNIST-database/media/File:MnistExamples.png.

[5] M. A. Syakur et al. Integration k-means clustering method and elbow method for identification of the best customer profile cluster. *IOP Conf.*, 2018. Ser.: Mater. Sci. Eng. 336 012017.