TTT4275 - Classification of Iris Flowers and Handwritten Numbers

Christian Danh Nguyen[a], Vegard Stornes Iversen[a]

[a]*Department of Electronic Systems, Norwegian University of Science and Technology, 7491 Trondheim.*

Summary

This paper is for a classification project in the course TTT4275 at NTNU. The project is split in two parts, and each part will test and evaluate a type of classifier. The first part is a classic classification task, classifying the three species of an Iris flower. To do this we use a Linear Discriminant Classifier, LDC for short. With this we manage to classify the Iris species with an error rate of 3.3%. In the second part of the project, we will design a Template Based Classifier using the Nearest Neighbour algorithm, NN for short, on the MNIST handwritten numbers dataset. We also implemented clustering and observed that the performance rate with clustering, compared to without, went from 3.7% to 4.7%, while the processing time went from approximately 2 hours to 2 minutes. Further we compared the NN algorithm to the K Nearest Neighbour algorithm with clustering, KNN for short, revealing that the higher value of K in the KNN algorithm results in worse performance. The most optimal Template Based Classifier would then be the 1NN with clustering with an error rate of 4.7%.

Contents

1. Introduction

The goal of the project is to get a bigger understanding of how a classifier works. Classification have been in large growth lately and has many important use cases. It could be used in medicine, sorting (for example trash), security and much more. With increasing computing power and better algorithms the classifications has become more and more effective. Also now in smaller formats, with light weight accurate algorithms on small and powerful computing chips, which increases the use cases. First will we introduce the necessary theory to understand the task at hand. Then we will go into the implementation and results separately for the two task/classifiers mentioned above. In the end we will conclude and describe our findings.

2. Theory

In this section we will explain the necessary theory to understand and complete this project.

Classification is a method of separating and distinguishing things form each other. An example could be identifying if a person is a man or a female. Naturally humans will look for features known to distinguish a man for a female. This could be hair length, height, voice etc. The way we humans have learned to distinguish is by observing and learning from experience. The method used in this report is not very different, and is called *supervised learning* [1](page 695-697). This means that the machine sees maps an input with an output based on example input-output pairs.

2.1. Linear discriminant classifier

Linear discriminant classifier, LDC for short, is a type of classifier used for linear separable problems. The method is to find linear combinations of features that makes it possible to distinguish the classes from each other. Each class in the discriminant classifier is described by a function $g_i(x)$ and the decision rule:

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

The discriminant function $g_i(x)$ is defined by the function

$$g_i(x) = w_i^T x + w_{io},^{1} \quad i = 1, ..., C \tag{2}$$

Here are the w_{io} the offset for the class w_i, and C is the number of classes. For $C > 2$ we write expression in a compact matrix form

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

where g and w_o are vectors with dimension C and W is a $C \times D$ matrix, where D is the number of features. For simplifications we alter equation (3), to include the offset in a new W matrix.

$$g(x) = Wx \tag{4}$$

Here are $W = [W \ w_o]$ and $x = [x^T \ 1]^T$. Notice that a column of ones is added to the dataset features, x. This is for getting correct dimensions, and will also work as a bias.

To train the classifier we need a cost function, so the algorithm knows when its wrong, and can optimize by minimizing the cost function. There are many different types of cost functions and some can be found here [2]. In this project we will be taking a look at the *Minimum Square Error- MSE* function. The MSE function in vectorial form is given as

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

Here is t_k the target class vector, which is a vector that shows what the desired class is. For example if the target class is the first class, the target class vector is $t_k = [1\ 0\ 0]^T$. To minimize the MSE we need to take the derivative in relation to the weights, W, and set it equal to zero. The goal is therefore to find the weights, W, that produces the lowest cost. Then we rewrite the cost function in respect to the weights, W:

$$\nabla_W MSE = \sum_{k=1}^{N} \nabla_{g_k} MSE \ \nabla_{z_k} g_k \ \nabla_W z_k, \tag{6}$$

and from this it is easily shown that

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

By inserting this in equation (6) we get

$$\nabla_W MSE = sum_{k=1}^{N}[(g_k - t_k) \circ g_k \circ (1 - g_k)]x_k^T \tag{7}$$

The goal is to calculate and update the weights, W, to minimize the MSE we move W to the opposite direction of the gradient:

$$W(m) = W(m-1) - \alpha \nabla_W MSE, \tag{8}$$

where m is the iteration number and α is the step factor, and is a carefully chosen constant. Since the target class vector t_k is either zero or one, will we have the output of equation (4), $g(x)$, to be a possibility distribution. To get the output to be between 0 and 1, we use an activation function[2]. There are many different activation function you could use, but here we use the sigmoid function.

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{9}$$

The reader is encourage to read more about this in [3].

[1] This means transposed.

[2] Activation function is a node that defines the output by mapping [3].

2.2. Fitting

An optimal model will be able to generalize, such that it can classify unseen data of the classes it is trained on. This is not always the case, and there are two cases of mistakes when it comes to fitting a model. This is visualized in Figure 2.1. Underfitting, is when a model has not been able to learn and this is usually the case when it is not trained on enough data. Overfitting is the second case. This is when the model is not learning and generalizing but only remembering the last data points. Thus it would perform very good on the data that it is trained on, but bad on unseen data. Fixing overfitting can vary from case to case, and is dependent on the type of machine learning algorithms used. In deep neural networks can dropping some of the nodes be a good way of minimizing chance of overfitting. It can also be good in classification problems to remove features that does not contribute and will only be noise for the algorithm. Further reading can be done in [1] (page 705-706).

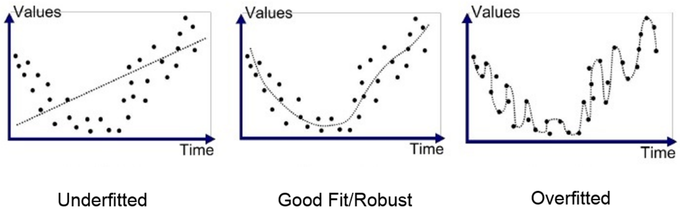

Figure 2.1: Underfitting and overfitting visualized. Taken from [4].

2.3. Template based classifier

The template based classifier has an input x, which is matched towards a set of references (templates) which have the same form as x. The decision rule is as simple as finding the reference which is closest to x and assume that x belongs to the same class as this reference. This method is known as the *Nearest Neighbour - NN* algorithm.

The K Nearest Number - KNN algorithm. KNN is a more advanced decision rule where you find the $K > 1$ nearest references. After finding these references, the class that occurs most often among those is the one x belongs to. However, if the class majority within the references are equal, one of them will be picked randomly. Figure 2.2 illustrates how the decision ruling works for KNN. K is the black solid circle, the green circle is x, and the blue squares and red triangles are the references. As we can observe from the illustration, the $K = 3$ nearest references to x are two red triangles and a blue square, which means x is classified to be a red triangle. However, for the $K = 5$ nearest references, x is classified as a blue square.

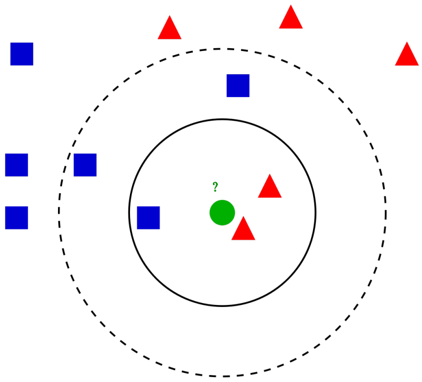

Figure 2.2: Illustration of KNN where the solid line is $K = 3$ and the dotted line is $K = 5$. Taken from [3].

There exists several ways of measuring the distance between the reference and x, but for the NN and KNN, Euclidean distance will be used. Further reading on other ways of measuring can be done in [5].

2.4. Euclidean Distance

Euclidean distance between two points in Euclidean space is the length of a line segment between two points and can be calculated from the Cartesian coordinates of the points using the Pythagorean theorem [6]:

$$d(x, ref_{ik}) = \sqrt{\sum_{k=1}^{N}(x - \mu_{ik})^2}, \tag{10}$$

where the parameters $ref_{ik} = \mu_{ik}$ can be chosen directly from the training data set. Euclidean distance is one of the more popular used methods for measuring distance between references in NN algorithms.

2.5. Clustering

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points, and dissimilar to data points in other groups [7].

The Partitioning Method. The partitioning method is a clustering method where data objects are divided into K clusters that are not overlapping. This means that each object only belongs to one cluster. There are algorithms that decides which objects should belong to which cluster, and one of them is the K-means clustering algorithm.

K-Means clustering. K-means clustering is an algorithm where the best centroid[3] of each cluster is computed using an iterative method. The user needs to specify the number of K clusters to assign, and then randomly initialize K centroids. Afterwards, the steps are as follows:

- Assign each point to its closest centroid.
- Compute the new centroid (mean) of each cluster.

[3] The centroid is the centre point of the object.

These two steps are repeated until the centroid positions do not change. Thus leaving us with K clusters based on similarity. Figure 2.3 illustrates a flowchart of the K-Means cluster algorithm.

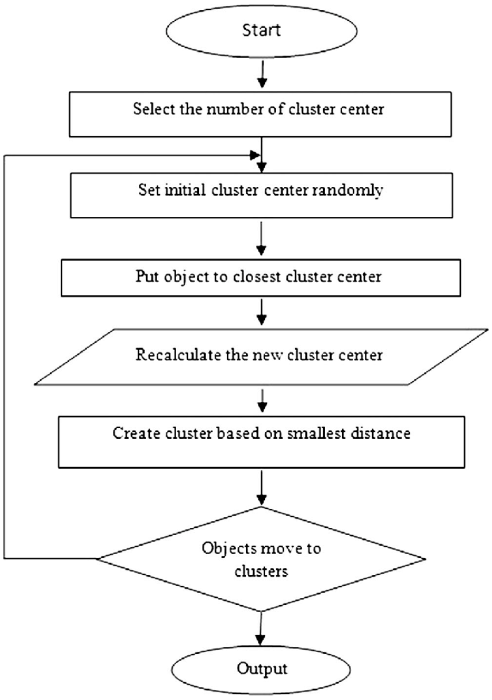

Figure 2.3: Flowchart of K-means clustering algorithm. Taken from [8].

2.6. Confusion matrix

A confusion matrix is calculated by comparing the predicted label with the true label. It shows the value in each cell with row i and column j, which tells how many of the predicted label j was in the true label i. So a perfect classifier will produce a confusion matrix which is a diagonal matrix. And from this it easy to deduct the total error rate by looking at the wrong predictions. Total error rate, ERR_T in percentage is given by

$$ERR_T = \frac{\text{total wrong predictions}}{\text{total number of predictions (samples)}} \cdot 100 \tag{11}$$

The reader is encourage to read more about error evaluation in [1] (page 708-713).

3. The task

The task at hand is a two part classification problem. First we look at classification of the Iris flower by using its features, *sepal length, sepal width, petal length and petal width*. We will use a linear discriminant classifier, explained in Section 2.1, to classify if the Iris flower is a *Setosa, Versicolor* or *Virginica*. Then we will remove overlapping features and see if this gives better results.

In the second part we take a look at the MNIST handwritten numbers dataset. The goal is to correctly classify each handwritten number image by using supervised learning methods. The methods used in this task is the **Nearest Neighbour** algorithm, and the **K Nearest Neighbour** algorithm, explained in Section 2.3. We will observe how the algorithms works with and without clustering, their individual processing time and performance.

4. Implementation & Results

First will we go through the implementation and results of the Iris flower classification task. Then we will go through the MNIST classification of handwritten number task. The code for both of them are written in python, because its a easy programming language, and has a lot of documentation backing it.

4.1. Implementation - Iris flower task

In this section we will be implementing a LDC with a MSE as cost function. Then will we look at the differences depending on the input given as train and test data, and see if the result will depend on this. After this, we will take a look at the features at hand. We will see if we can achieve better results by removing features that show overlapping. This will also show how many features we actually need to have a good classifier of Iris flowers. We shall also take a look at the effect of different α, and what this does with the result.

So this task is split into two parts. Part one, is the train and test size of the data, and their order. Part two, the affects of which features that gives the best results and why. The full code for the implementation Iris classification can be found in Appendix C, with comments attached.

The dataset at hand consist of 150 samples of Iris flower. There are 3 different species of Iris flower that we are looking at, *Setosa, Versicolor* and *Virginica*. They are represented equally in the dataset, 50 samples per type of Iris flower. Each sample have 4 features, *sepal length, sepal width, petal length* and *petal width*. Number of classes will be denoted as C and number of samples per class will be denoted as N. The 4 features will be denoted as $x_i, \quad i = 1, 2, 3, 4$.

Training the classifier. Before training we split the dataset in a training set and a testing set[4]. A normal split ratio is 80-70 % training set and then 20-30% test set. To get a whole number we use a 60/40 split, first with the 30 first samples from each class as training, and the 20 last from each class as testing. This training set runs for 2000 iterations. We want the MSE to converge, and later we see that 2000 iterations is enough for this. After this we initialize a matrix full of zeros to be the weights, W. After a testing we can see in Figure 4.1 that $\alpha = 0.01$ gave the best results.

[4]This means that we only train the classifier on the training set, and test how good the classifier is with the test set.

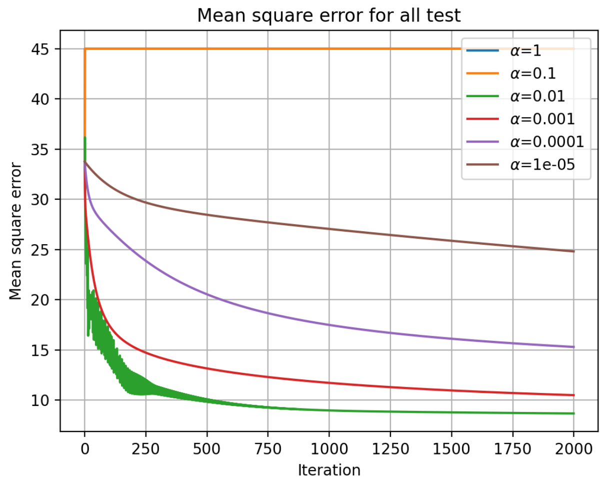

Figure 4.1: Plotting the MSE for 2000 iteration with different α.

As we see in Figure 4.1 to big α will not be able to learn, and to small will not converge. We therefore choose $\alpha = 0.01$, since this gives the smallest MSE. There is a possibility to have a dynamic α but this will be discussed in Section 5.1. After training and updating the weights, W every iteration we get these weights.

$$W = \begin{pmatrix} 0.428 & 1.680 & -2.500 & -1.158 & 0.304 \\ 1.472 & -2.917 & -0.194 & -1.166 & 1.593 \\ -2.954 & -2.493 & 4.314 & 3.762 & -1.888 \end{pmatrix} \tag{12}$$

Weights after training with the 30 first samples of each class, rounded numbers.

A block-diagram that shows the overall process can be seen in Figure 4.2

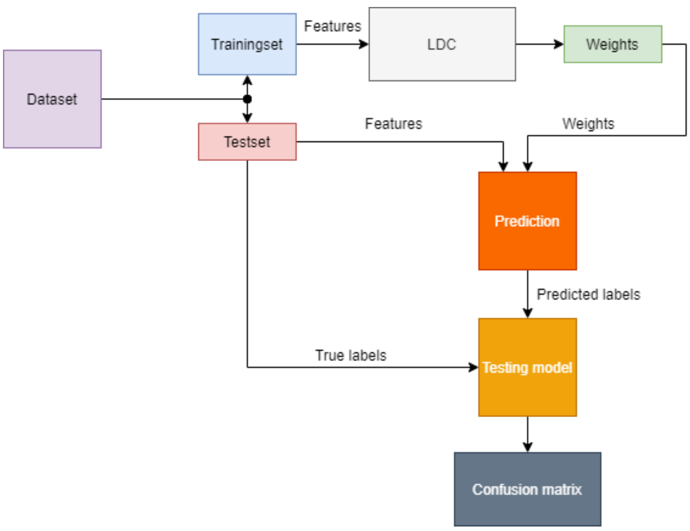

Figure 4.2: Basic block-diagram that shows the overall process.

4.2. Results- Iris flower task

In this section will we take a look at the results by comparing the confusion matrices and error rates.

Testing the model. We test how well the model with weights (12) works, by using them first with the test set and then the training set. From this we can generate the confusion matrices, seen in Figure 4.3 and 4.4. We can see that the model misses on 2 flowers in test set, and 3 flowers from the training set. All misses are between the two species *Versicolor* and *Virginica*. This will be discussed later.

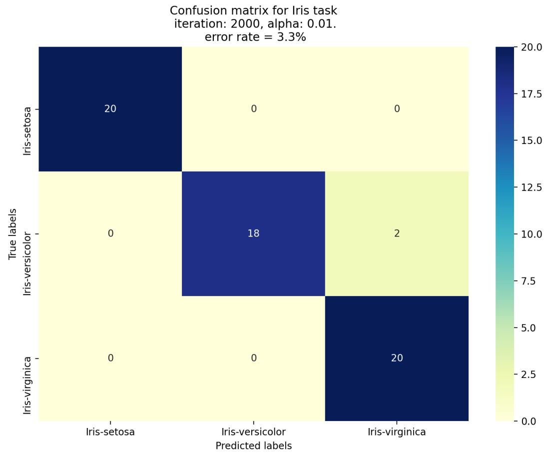

Figure 4.3: Confusion matrix from the test set and the weights (12), with $\alpha = 0.01$ and an error rate of 3.3%.

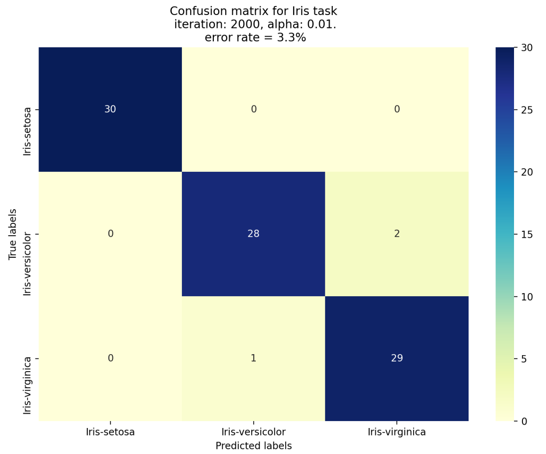

Figure 4.4: Confusion matrix from the train set and the weights (12), with $\alpha = 0.01$ and an error rate of 3.3%.

The combined error rate for both test and training set are with these weights given by equation (11)

$$ERR_T = \frac{5}{150} \cdot 100 = 3.3\% \tag{13}$$

which is also can be seen in the title of the two figures.

Now we do the same but now the training set consist of the 30 last samples, and the test set consist of the 20 first. New training data gives new weights for the model, rounded with 3 decimals.

$$W = \begin{pmatrix} 0.494 & 1.680 & -2.642 & -1.234 & 0.310 \\ 0.293 & -2.317 & -1.578 & -3.603 & 3.004 \\ -2.163 & -2.921 & 3.560 & 4.263 & -2.453 \end{pmatrix} \quad (14)$$

It is not a big change in the weights and the main characteristics is similar too the weights from before in matrix (12).

The results can be seen in Figure 4.5 and 4.6.

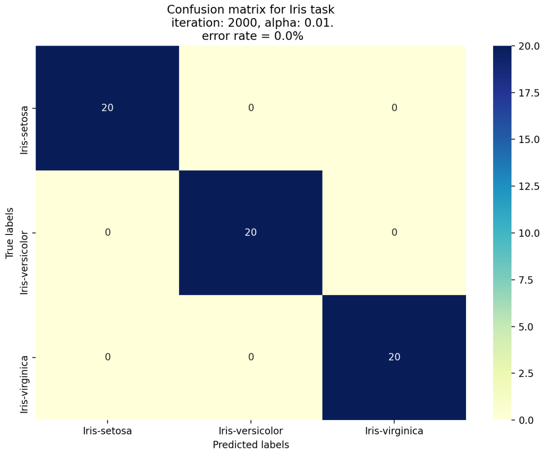

Figure 4.5: Confusion matrix from the test set and the weights (14), with $\alpha = 0.01$ and an error rate of 0%.

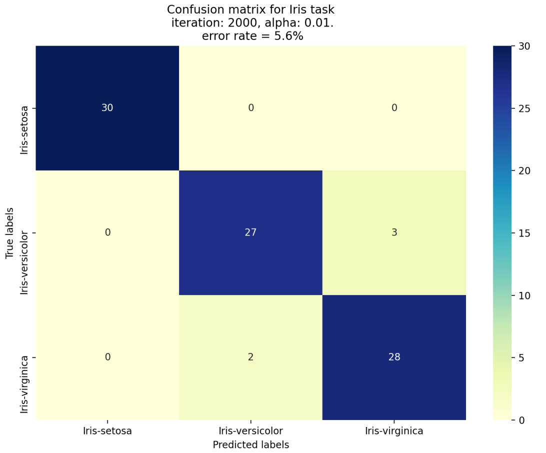

Figure 4.6: Confusion matrix from the train set and the weights (14), with $\alpha = 0.01$ and an error rate of 5.6%.

As we can see the result changed a bit, we now have a 0 % error rate when using the testing set, and a 5.6% with the training set. The total error of the two combined will still be 3.3%.

$$ERR_{T_{combined}} = \frac{5}{150} \cdot 100 = 3.3\% \quad (15)$$

Features and linear separability. To understand the results and the data, and see if there is possibilities to improve the result one must take a look at the features. In this section we will analyze the data and its features. This will be used to look for improvements and understand the relation between the input and the output of the model.

Histograms. Plotting the histogram for the different classes and features, makes it possible to see if different features av an overlap over the classes. Features with big overlap, can make it more difficult to distinguish between the different classes. The histogram can be seen in Figure 4.7.

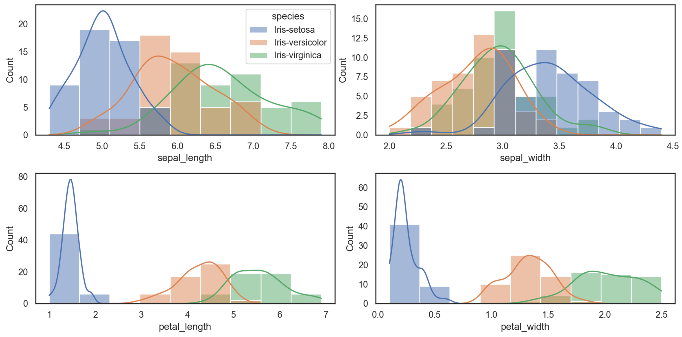

Figure 4.7: Histogram for each feature and the classes representation there. With a kernel density estimation line[5] that shows the overlapping.

Removing the most overlapping features one by one, and train the model again every time. By examining Figure 4.7 we see that the feature *Sepal width* have the most overlap. We then remove this from the dataset, and split it into training- and testing set. Here with the 30 first samples from each class for the training set, and the remaining 20 samples for the testing set. The new weights are now:

$$W = \begin{pmatrix} 1.638 & -3.081 & -1.472 & 0.644 \\ -0.677 & 1.617 & -2.824 & 0.251 \\ -3.787 & 4.284 & 3.092 & -2.469 \end{pmatrix} \quad (16)$$

and the result can be seen in Figure 4.8 and 4.9. Notice that the dimensions have decreased since we removed one feature.

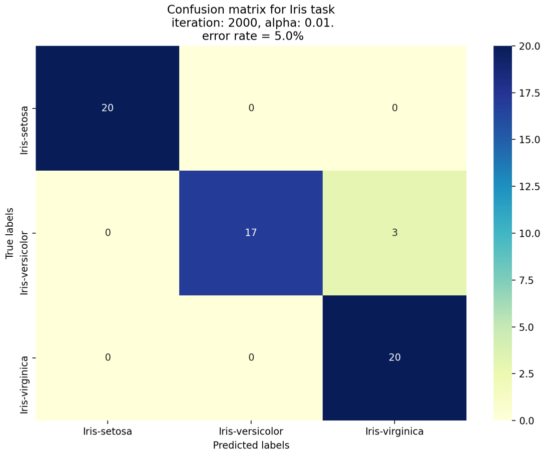

Figure 4.8: Confusion matrix from the test set and the weights (16), with $\alpha = 0.01$ and an error rate of 5%. Removed the feature *Sepal width*.

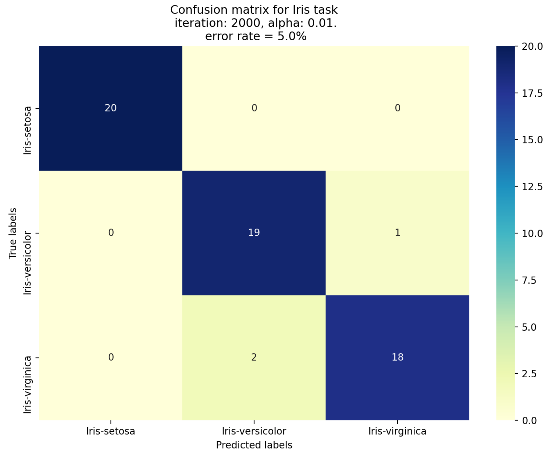

Figure 4.10: Confusion matrix from the test set and the weights (17), with $\alpha = 0.01$ and an error rate of 5%. Removed the features *Sepal width, Sepal length*.

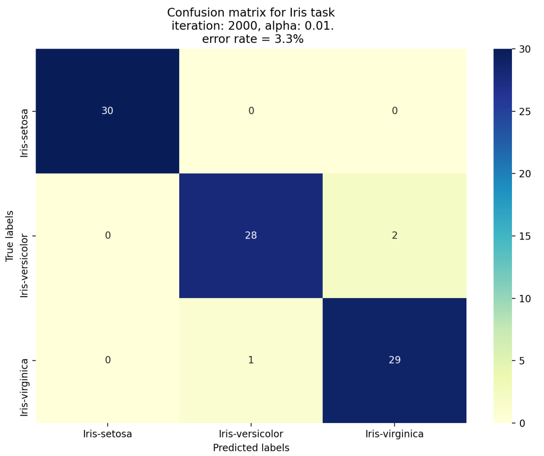

Figure 4.9: Confusion matrix from the train set and the weights (16), with $\alpha = 0.01$ and an error rate of 3.3%. Removed the feature *Sepal width*

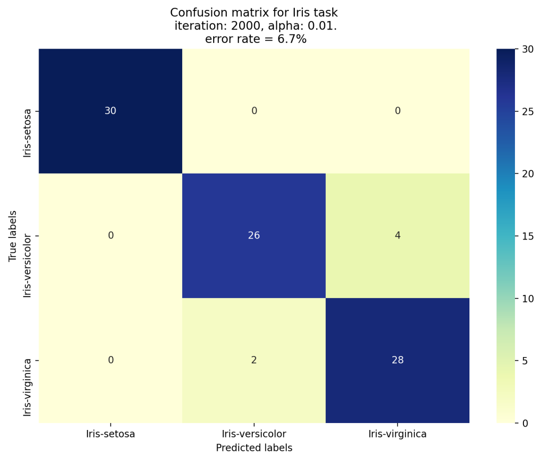

Figure 4.11: Confusion matrix from the train set and the weights (17), with $\alpha = 0.01$ and an error rate of 6.7%. Removed the features *Sepal width, Sepal length*.

As we can see in Figure 4.8 and 4.9 the result gets a bit worse, but not much.

Removing more features. Now we remove features until we only have the one left that shows the least amount of overlapping. From the histograms in Figure 4.7, we remove *Sepal length* and after that *Petal width*. The different weights can be seen in Appendix A

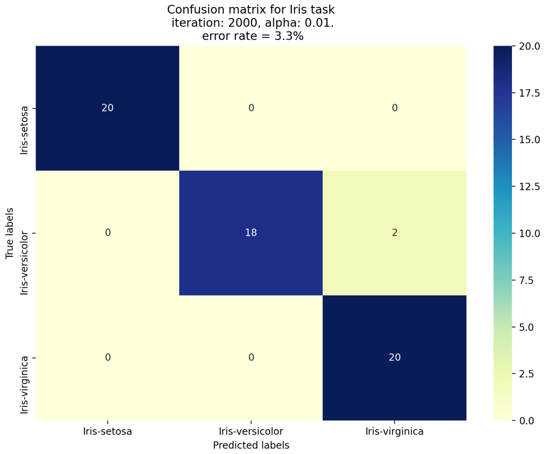

Figure 4.12: Confusion matrix from the test set and the weights (18), with $\alpha = 0.01$ and an error rate of 3.3%. Removed the features *Sepal width, Sepal length, Petal width.*

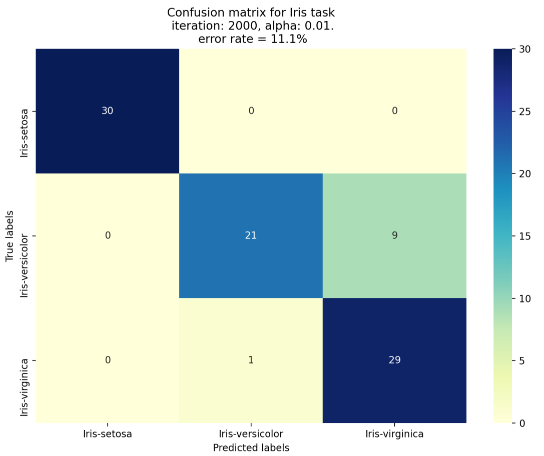

Figure 4.13: Confusion matrix from the train set and the weights (18), with $\alpha = 0.01$ and an error rate of 11.1%. Removed the features *Sepal width, Sepal length, Petal width.*

Since there can be difficult to decide which is the most overlapping of *Petal width* or *Petal length* are we checking both options.

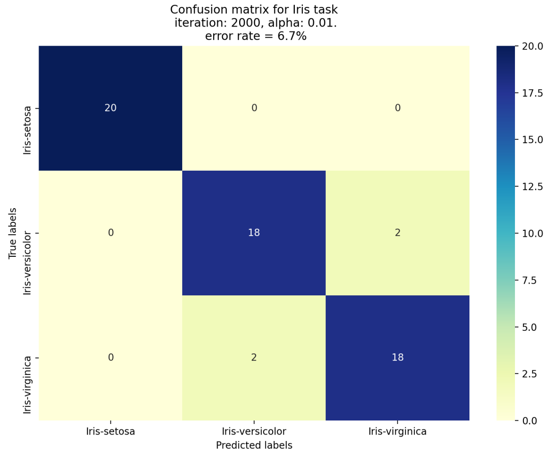

Figure 4.14: Confusion matrix from the test set and the weights (19), with $\alpha = 0.01$ and an error rate of 6.7%. Removed the features *Sepal width, Sepal length, Petal length.*

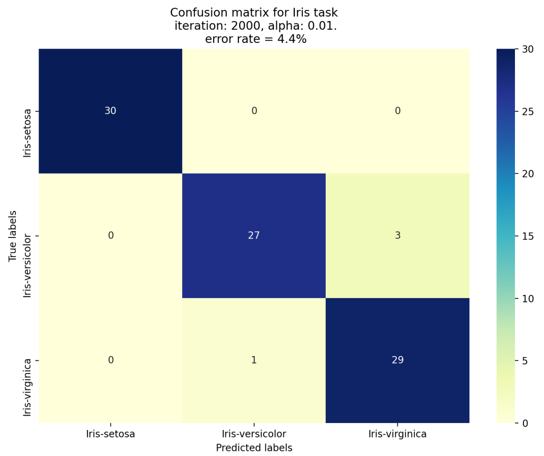

Figure 4.15: Confusion matrix from the train set and the weights (19), with $\alpha = 0.01$ and an error rate of 4.4%. Removed the features *Sepal width, Sepal length, Petal length.*

Here we can see that with one feature, only having *Petal width* gives a smaller error rate than *Petal length*, and therefore the least overlapping feature, which also can be seen from the kernel density estimation line in Figure 4.7.

We compare all the results in Table 1.

Table 1: Results from each model, with different features, the check-mark, ✓ means it used in the model. Sw= *Sepal width*, Sl=*Sepal length*, Pw= *Petal width*, Pl=*Petal length*.

First 30 samples	Sw	Sl	Pw	Pl	Total error
✓	✓	✓	✓	✓	3.3%
	✓	✓	✓	✓	3.3%
✓		✓	✓	✓	4%
✓			✓	✓	6%
✓				✓	8%
✓			✓		5.3%

4.3. Implementation - MNIST handwritten numbers

We will here take a look at the MNIST handwritten numbers dataset. The section is divided into two parts. First we try to classify the handwritten numbers by using the NN algorithm, and afterwards we will attempt the same but with clustering and the KNN algorithm. All the code has been implemented in Appendix D.

The Dataset. The dataset at hand consists of a training set of 60,000 samples and a test set of 10,000 samples. The images have a resolution of 28x28 pixels and each pixel have a value that ranges from 0 to 255. The MNIST dataset has gone through lots of preprocessing such that all the numbers have been size-normalized and centered in a fixed-size image. The dataset is imported from the *Keras* library and stored in four arrays: *training images, training labels, test images and test labels.* The labels represents the true value of the handwritten numbers while the images represents the image of size 28x28. Plots of some images from the dataset are shown in Figure 4.16.

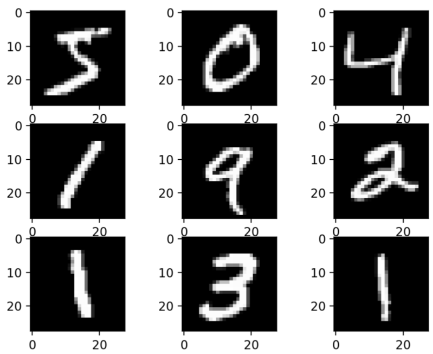

Figure 4.16: Plot of 9 images from the MNIST dataset.

Designing the NN-based classifier. In order to design the NN-based classifier we need a way to measure the distance in the images of the test and training samples. To do this we use the Euclidean distance as described in Section 2.4. With a way of measuring the distances between images, we can now compare each test image with the whole training set and find the training image that has the smallest distance to each test image. Thus, the label linked to the training image with the smallest distance is the predicted number. The code is implemented in line 128 - 142, and line 269 - 280.

Implementing the clustering. We will now try to implement clustering. The training set will be split into 64 clusters for each class, where one class is a unique number. Thus resulting in 10 classes with a total of 640 clusters. To easily cluster the training set we do some preprocessing. The training set is first sorted based on labels in an ascending order and then sent for clustering. The clustering is done by importing the *KMeans* function from the *sklearn* library. The code is implemented in line 33 - 69. Thus, the clusters of images is the new training set.

Designing the NN-based classifier with clustering. Just like for a NN-based classifier without clustering. Each test set is now getting compared to the whole cluster where the cluster image with smallest distance is the predicted number. The code is implemented in line 152 - 177

Designing the KNN-based classifier with clustering. The implementation of a KNN-based classifier with clustering is very similar to the NN-based classifier with clustering. Instead of finding the cluster with the smallest distance, we find the K = 7 clusters with the smallest distance. Thus, the majority class among these 7 clusters will be the predicted number. The code is implemented in line 92 - 119.

Plotting numbers and confusion matrices. There may be some test images that has been predicted uncorrectly, and we wish to plot those numbers as well as the confusion matrix. The latter has been described in detail in Section 2.6. The code is implemented in line 184 - 256.

4.4. Results - Classification of MNIST numbers

We will here take a look at the results of the classification of MNIST handwritten numbers with regards to different types of classification methods. The individual processing time and performance will also be commented. All the results are attached to Appendix B.

NN-based classifier. The NN-based classifier had an incredibly high processing time which forced us to test the classifier for a smaller training set size before using the whole set. Figure B.1 shows the confusion matrix of the NN-based classifier with a training set size of 10 000. This resulted in an error rate of 14.4%, and a processing time of approximately 20 minutes, which is satisfactory for the small amount of training size we used. Further testing with the whole training set size resulted in the confusion matrix in Figure B.2 with an error rate of 3.7% and a processing time of approximately 2 hours.

Figure B.3 shows three different plots of test images, their predicted image, and the differences in those pictures. As seen from the plots and the confusion matrix it is understandable that these misclassifications happen. Other common misclassifications occurs between 1 and 7, 5 and 3, and 3 and 8. Further, in Figure B.4, we plot some of the correctly classified numbers. This plot shows satisfactory results as the differences are minimal.

NN-based classifier with clustering. The NN-based classifier with clustering resulted in the confusion matrix in Figure B.5. The error rate increased by 1%, giving an error rate of 4.7%, but the processing time decreased to a total of 2 minutes and 50 seconds. The processing time on clustering was 1 minute and 36 seconds. The sacrifice in performance for faster processing time is in this case satisfactory and justifiable. Figure B.6 and Figure B.7 shows the plots of misclassified and correctly classified images and their differences.

KNN-based classifier with clustering. The KNN-based classifier with clustering resulted in the confusion matrix in Figure B.8. The error rate increased to 5.6% while the processing time remained the same as for NN-based classifier with clustering. The results of all implementations are shown in Table 2.

Table 2: Results from NN-based classifier with and without clustering, and KNN-based classifier with clustering. The runtime total is in hours : minutes : seconds.

Clustering	KNN	Error (%)	Runtime total
NO	1NN	3.7	2:04:25
YES	1NN	4.7	0:02:50
YES	3NN	4.8	0:02:37
YES	5NN	5.4	0:02:37
YES	7NN	5.6	0:02:50
YES	9NN	6.1	0:02:40

5. Discussion

In this section we will discuss the results presented in Section 4. First will we discuss the results for the Iris flower classification, after we discuss the results of the MNIST handwritten numbers classification.

5.1. Discussion- Iris flower task

As we can see in Table 1 the model with the lowest error rate, and therefore the most accurate model is the model that uses all features in the training process. But all the other model, even the ones with just one feature had an acceptable low error rate. The advantage with fewer feature, is that it is much less computations that needs to be done. Also if the person that collected the data knew that one feature was enough, a lot of time could have been saved. Another point is that all the misses are between *Versicolor* and *Virginica*, this is because they have the most overlapping features as seen in Figure 4.7.

As we see in Table 1 there is no different in the total error rate if we use the 30 first or the 30 last samples as training data. This means that the features are evenly distributed. It is often normal to shuffle the dataset, and its a way to improve the model by minimizing overfitting. An interesting point is that if we look at the confusion matrices in Section 4 we see that generally the test set does better than the training set. Often in other cases this is the opposite. This can mean that our model is not overfitting.

There could be many ways to improve our model. One is of course to increase the size of the dataset, but there is other methods too. In Section 4.1 we found α often called, learning rate by just testing many different α and training the model before plotting the MSE. As we see in Figure 4.1 α has a huge impact on the model. Therefore we could have made a dynamic that starts with high, but decays over time. This is smart when working with gradient decent, but again choosing the decay factor could be as crucial as just choosing α as a constant.

To shuffle the data could also help optimizing the model, but as we mentioned earlier the model seems to not be overfitting much. We could have tried other algorithms and cost function that might have lower the error rate. Our model delivered good results, but it is possible to get even lower with other methods, like this research paper [10].

It is also important to remember that in other cases its not always the best to minimize error rate as we have done here. If a model that shall classify if you need to check out a mole or not. Having an error rate of 1% but almost all the errors was classifying that a mole needed to be checked is much better than having a model with error rate of 0.5% if almost all the errors was not classified as no more checks, but actually needed to be checked out [1](page 710). In our case minimizing the error is sufficient.

5.2. Discussion - MNIST handwritten numbers

Table 2 shows the results of the different methods. The best performance came from the 1NN-based classifier without clustering with an error rate of 3.7%, while the worst came from the 7NN-based classifier with an error rate of 5.6%. Although the performance was worse for the classifiers with clustering, the substantial decrease in processing time was satisfactory. The long processing time is due to the incredibly large training set, resulting in even bigger distance matrices. By clustering the whole training set into 640 clusters, we managed to reduce the distance matrices by almost hundred times, ultimately reducing the processing time by almost sixty times. The substantial decrease in processing time far outweighs the minimal decrease in performance when comparing the 1NN with and without clustering.

The plots of the misclassified numbers were understandable for some, such as the ones who are usually hard to classify for humans as well, while very off for others. Common classification mistakes done by humans include the numbers 4 and 9, 8 and 3, and 1 and 7. These were also the numbers most commonly misclassified by the template-based classifiers as seen from the confusion matrices.

The higher error rate for the 7NN classifier was surprising as it was expected to outperform the 1NN classifier. There doesn't seem to be a reasonable explanation to why, hence we tried to see if different values for K would help. As seen from Table 2 it seems that it doesn't. The most optimal value for K when classifying the MNIST dataset would then be K = 1.

6. Conclusion

In conclusion, we have solved two different classification task, and in both cases gotten an acceptable result. In the Iris classification we made a Linear Discriminant Classifier, and managed to get the error rate down to 3.3%. We also managed to get an error rate of 5.3% with just one feature, which is most often very acceptable. In the MNIST handwritten number classification we made a Template Based Classifier with the KNN algorithm for K = 1 and K = 7. We also did this with and without clustering which resulted in huge processing time improvements (60 times better) when clustering, with of course a minimal sacrifice in performance (1% worse). The higher value of K leads to worse performance and processing time as seen from Table 2.

Its important to point out that our algorithms are not the best. Both classification of Iris flower and classification of handwritten numbers are often used tasks, and there are many different solutions on them online. There is probably much better algorithms than those we wrote online too. Libraries like *Tensorflow, PyTorch, scikit-learn* etc. have algorithms developed by big teams of expert, and are optimized to many different classification problems. So in a setting were the goal is not to learn about classification algorithms, but to get the best and fastest result, these libraries are recommended. But the process of writing it yourself is also rewarding. This project have learned us the importance of understanding the input data, and the inner working of a classification algorithm.

7. Github

The entire project can be found on github https://github.com/VegardIversen/TTT4275_project.

8. Thanks

Thanks the the teaching assistants in TTT4275 (2021) for good follow-up on our project. We would also like to thank our Professor Pierluigi Salvorossi for assistance when asked.

References

[1] Russell, Stuart J. and Peter Norvig: *Artificial intelligence: a modern approach.* Pearson, 2016.

[2] 48saily: *Cost function: Types of cost function machine learning*, Mar 2021. https://www.analyticsvidhya.com/blog/2021/02/cost-function-is-no-rocket-science/.

[3] *Activation function*, Apr 2021. https://en.wikipedia.org/wiki/Activation_function.

[4] Hoffman, Ken: *Machine learning: How to prevent overfitting*, Feb 2021. https://medium.com/swlh/machine-learning-how-to-prevent-overfitting-fdf759cc00a9.

[5] ROSA, Taca, Rifkie Primartha, and Adi Wijaya: *Comparison of distance measurement methods on k-nearest neighbor algorithm for classification.* January 2020.

[6] Wang, Liwei, Yan Zhang, and Jufu Feng: *On the euclidean distance of images.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 27(8):1334–1339, 2005.

[7] *Cluster analysis*, Mar 2021. https://en.wikipedia.org/wiki/Cluster_analysis.

[8] Younus, Zeyad, Dzulkifli Mohamad, Tanzila Saba, Mohammed Alkawaz, Amjad Rehman, Mznah Al-Rodhaan, and Abdullah Al-Dhelaan: *Content-based image retrieval using pso and k-means clustering algorithm.* Arabian Journal of Geosciences, 8, August 2014.

[9] *Kernel density estimation*, Mar 2021. https://en.wikipedia.org/wiki/Kernel_density_estimation.

[10] Sarab, Sura: *Iris flowers classification using neural network*, October 2019.

Appendices

A. Appendix A: Weights from different tests .

Weights after removing *Sepal width.*

$$W = \begin{pmatrix} -1.516 & -2.334 & 5.607 \\ 1.087 & -2.164 & -2.168 \\ 0.315 & 4.023 & -8.198 \end{pmatrix} \tag{17}$$

Weights after removing *Sepal width, Sepal length.*

$$W = \begin{pmatrix} -2.332 & 6.146 \\ 0.183 & -1.372 \\ 1.768 & -8.673 \end{pmatrix} \tag{18}$$

Weights after removing *Sepal width, Sepal length, Petal width.*

$$W = \begin{pmatrix} -5.747 & 4.125 \\ 0.283 & -1.024 \\ 4.676 & -7.725 \end{pmatrix} \tag{19}$$

Weights after removing *Sepal width, Sepal length, Petal length.*

B. Appendix B: Results for MNIST handwritten digits.

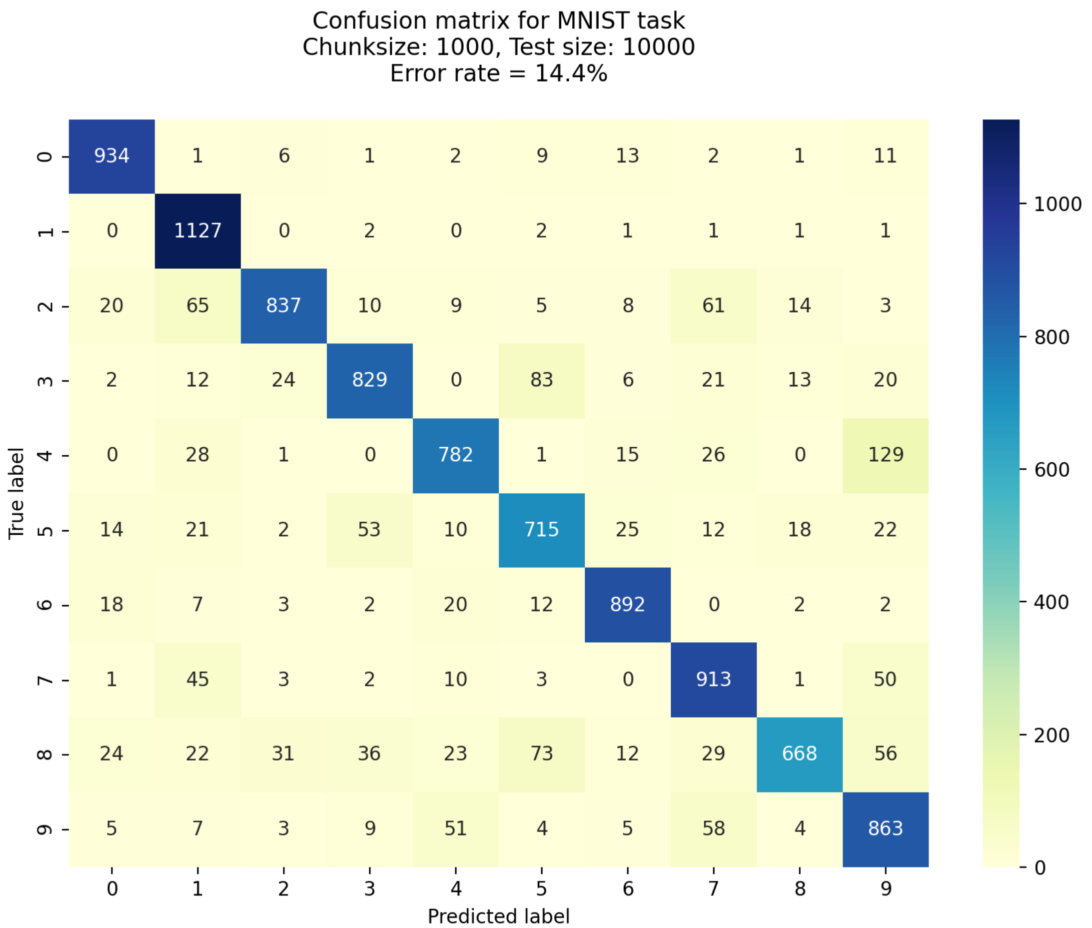

Figure B.1: Confusion matrix of a NN algorithm with training set size of 10 000. Error rate 14.4%.

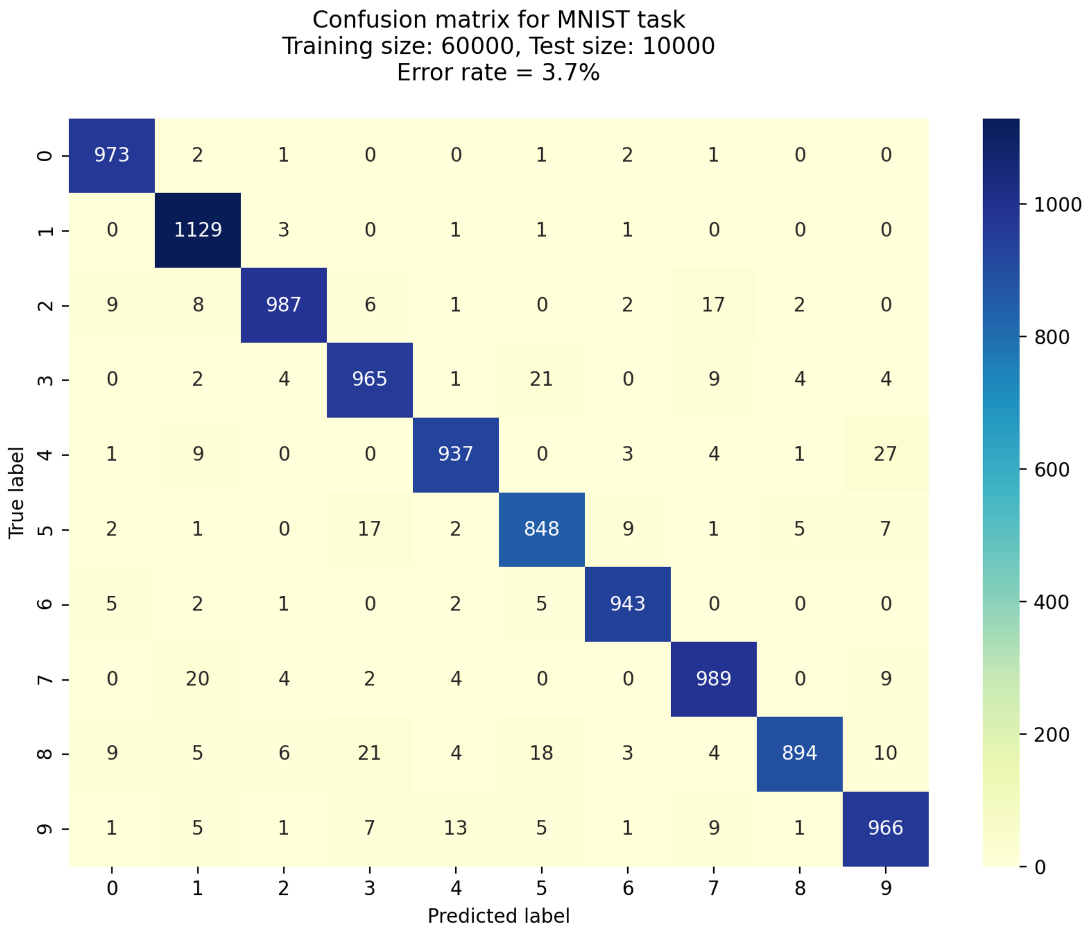

Figure B.2: Confusion matrix of a NN algorithm with training set size of 60 000. Error rate 3.7%.

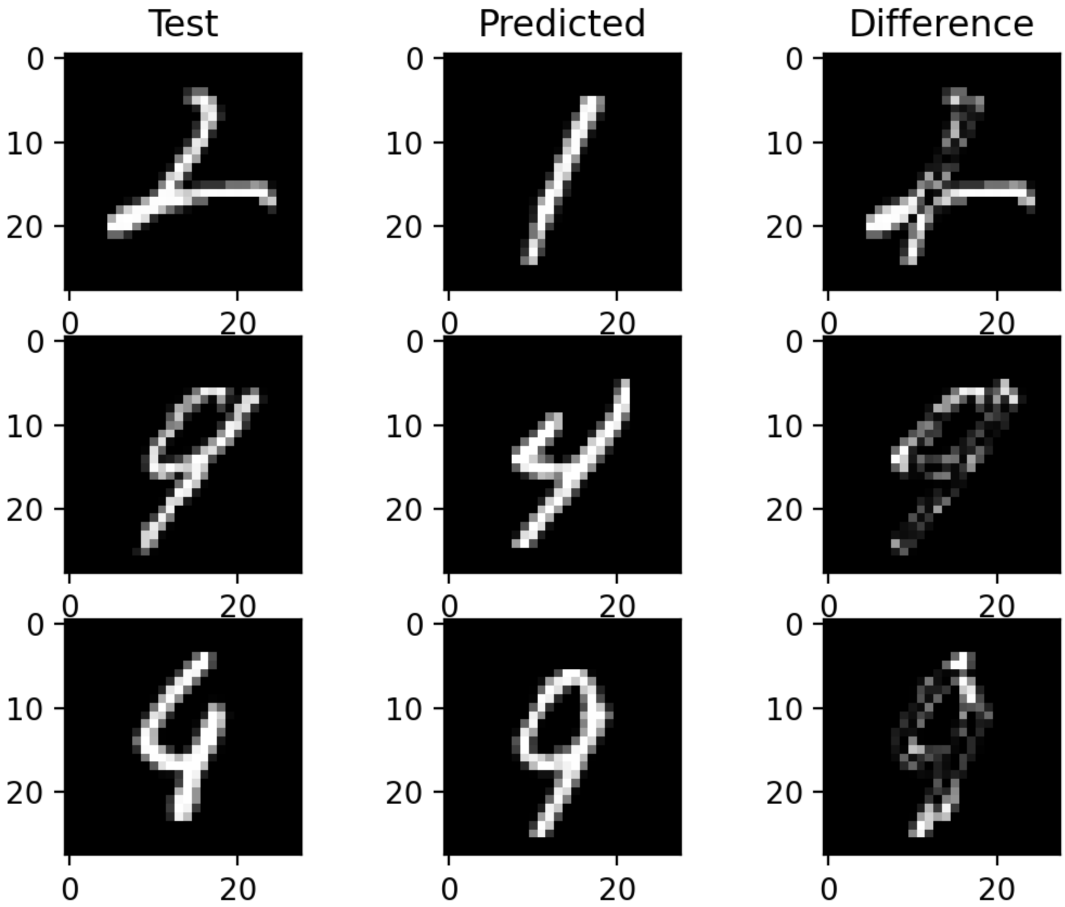

Figure B.3: Plot of digits which was classified incorrectly for the NN-classifier.

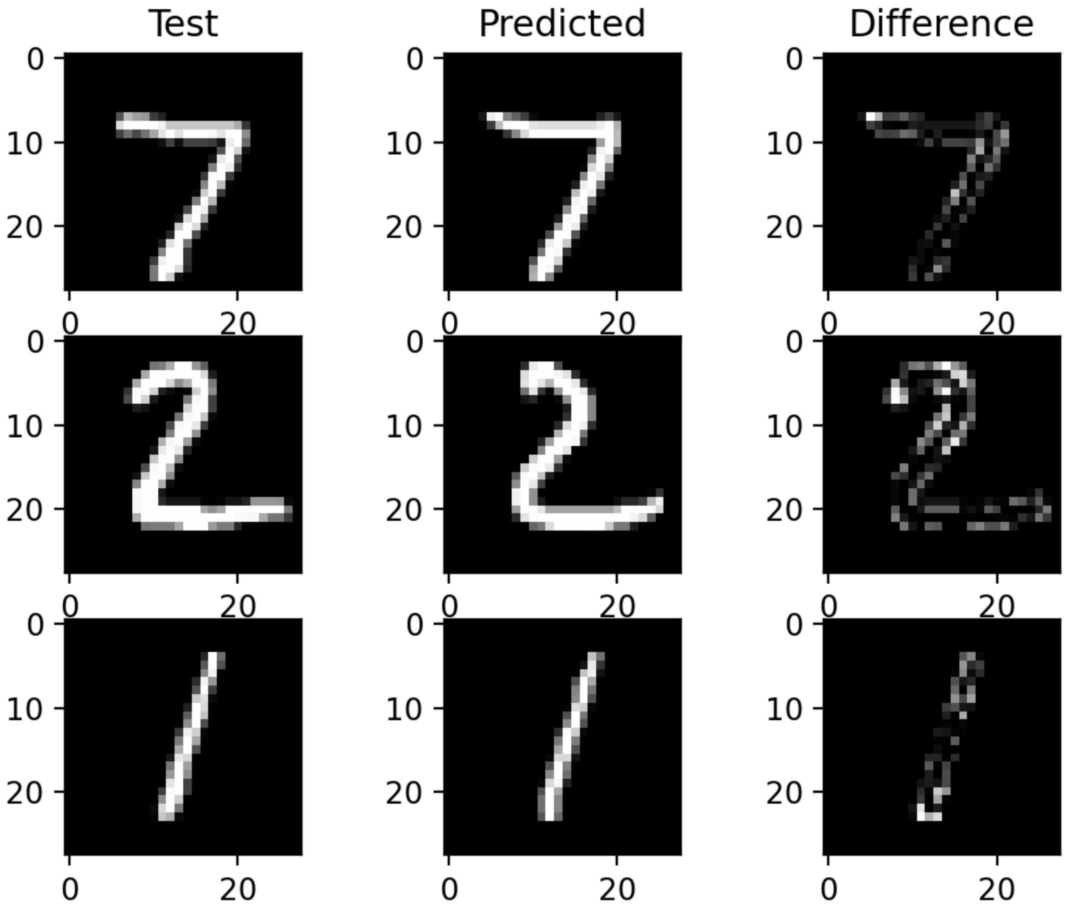

Figure B.4: Plot of digits which was classified correctly for the NN-classifier.

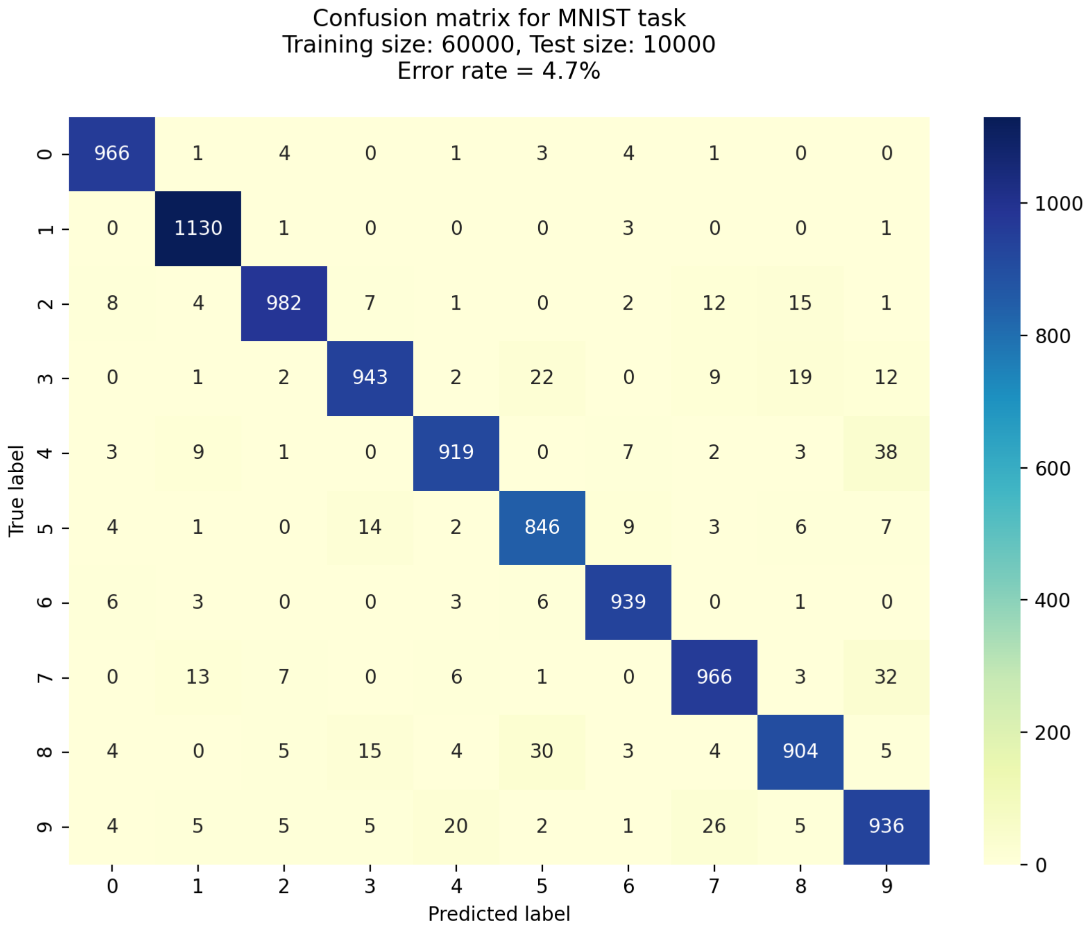

Figure B.5: Confusion matrix of a NN algorithm with clustering. Training set size of 60 000. Error rate 4.7%.

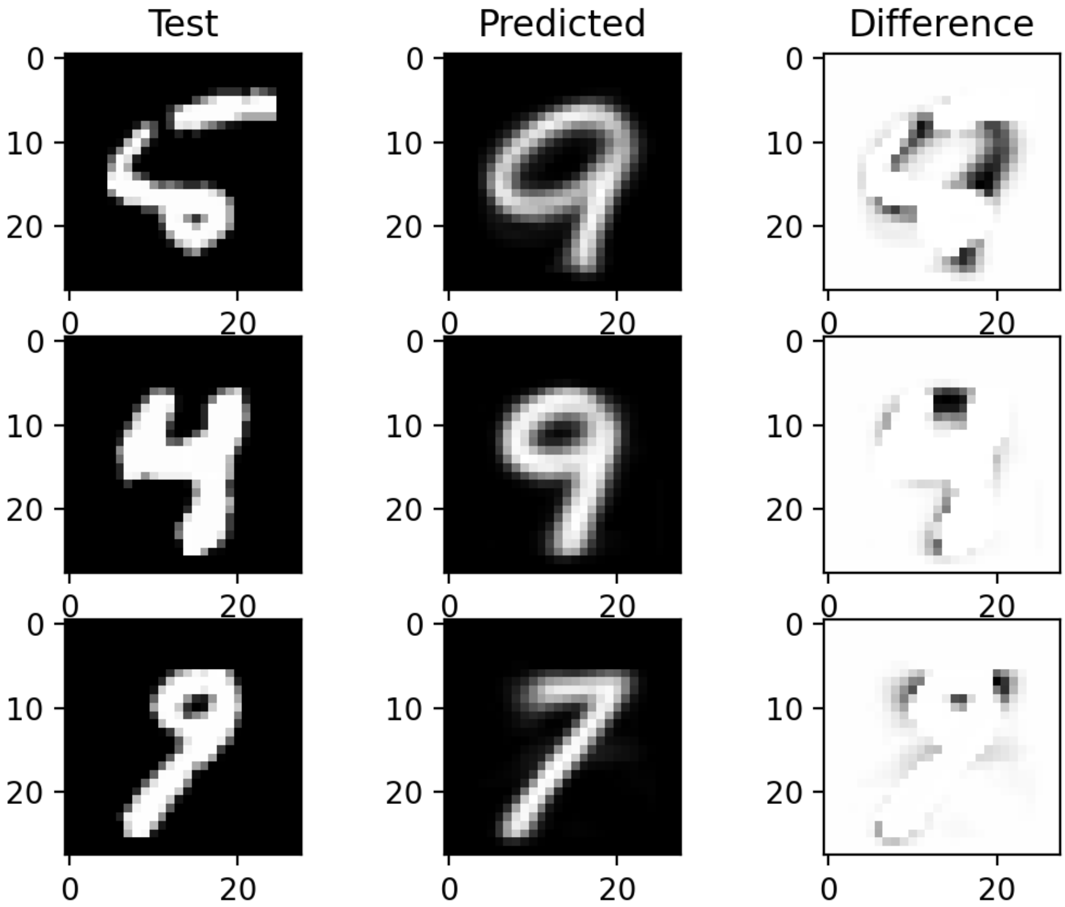

Figure B.6: Plot of digits which was classified for the NN-classifier with clustering.

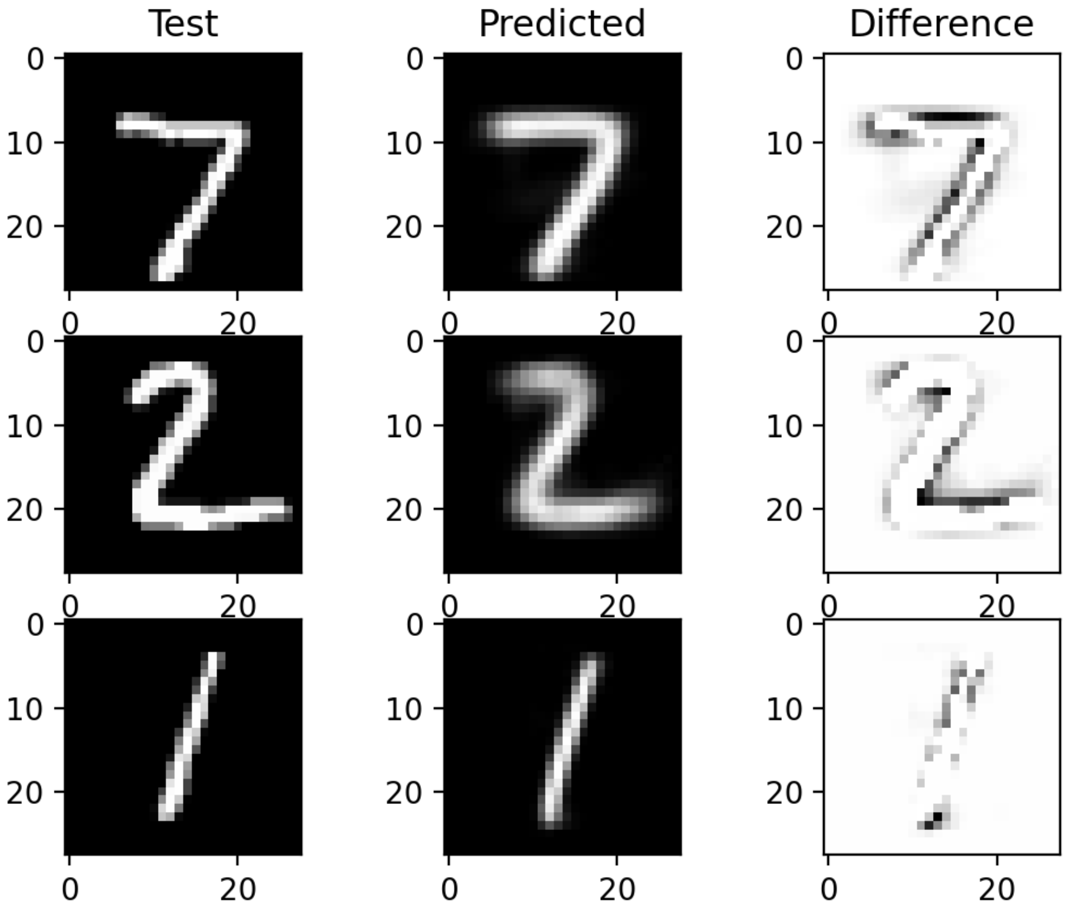

Figure B.7: Plot of digits which was classified correctly for the NN-classifier with clustering.

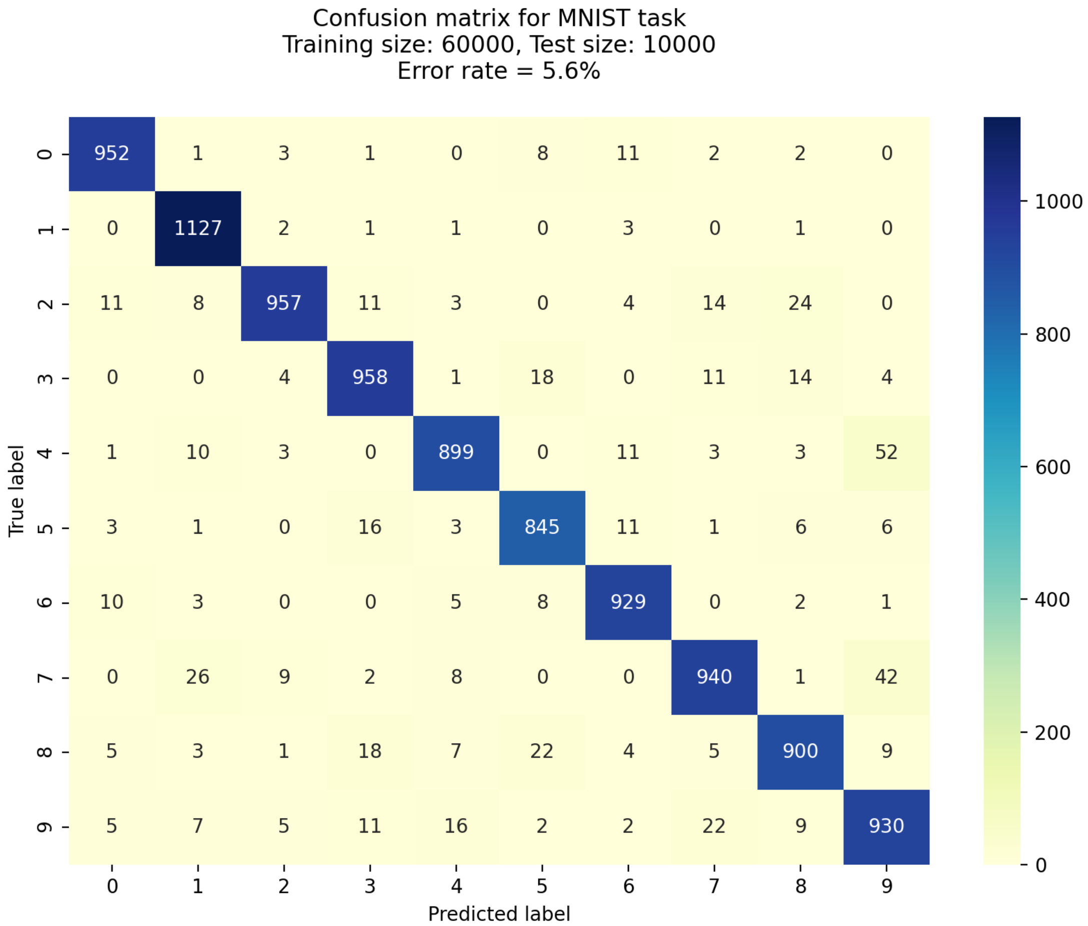

Figure B.8: Confusion matrix of a KNN algorithm with clustering. Training set size of 60 000. Error rate 5.6%.

C. Appendix C: Iris LDC code.

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

#to make a LDC, we take in a training, test an r_k list, and number of iteraterions, alpha and list of
     features
#make the class, d = LDC(train,test,t_l,iterations, alpha, list_of_features)
#then use w = d.train()
#could use @dataclass to not use self
class LDC:

    #initilizing the variables for the class.
    def __init__(self, train, test, t_k, iterations, alpha, list_of_features):
        '''
        function init: initilize the variables used in the class.
        param self: necessarry to object. Makes it so you can access all the variables in __init__ in
    the other functions
        param train: training data.
        param test: test data.
        param t_k: list with the true labels.
        param iterations: choice of number of iterations.
        param alpha: list of chosen alphas.
        param list_of_features: list of the used features.


        '''
```

```
        #attributes under
        self.train = train #np.array
        self.test = test #np.array
        self.t_k = t_k #list or np.array doesnt matter
        self.iterations = iterations #int
        self.alpha = alpha #list or np.array doesnt matter
        self.list_of_features = list_of_features #list or np.array doesnt matter
        self.class_names = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'] #could have this as an input to generlize class
        self.features = len(self.list_of_features) +1 #int
        self.classes = 3 #could have this as an input but didnt bother, shouldnt change
        self.weigths = np.zeros((self.classes,self.features)) #setting up weigths matrix
        self.g_k = np.zeros(self.classes) #setting up g_k array
        self.mses = np.zeros(self.iterations) #setting up array to be filled with mses
        self.confusion_matrix = np.zeros((self.classes,self.classes))#setting up confusion matrix basic

    #useful function to get and set variable, after class has been initilized.
    # ----------------------------------------#
    def set_iterations(self, iterations):
        self.iterations = iterations

    def set_alpha(self, alpha):
        self.alpha = alpha

    def set_train(self, train):
        self.train = train

    def set_test(self, test):
        self.test = test

    def set_train_test(self,train,test):
        self.train = train
        self.test = test

    def set_tk(self, tk):
        self.t_k = tk

    def set_list_of_features(self, list_of_features):
        self.list_of_features = list_of_features

    def set_num_of_classes(self,classes):
        self.classes = classes

    def get_iterations(self):
        return self.iterations

    def get_alpha(self):
        return self.alpha

    def get_train(self):
        return self.train

    def get_test(self):
        return self.test

    def get_train_test(self):
        return self.train, self.test

    def get_weigths(self):
        print(self.weigths)
        return self.weigths

    def get_tk(self):
        return self.t_k

    def get_list_of_features(self):
        return self.list_of_features

    def get_num_of_classes(self):
        return self.classes
```

```
     # ----------------------------------------#
    #just needed for resetting the cm before i test the trainset
    def reset_confusion_matrix(self):
        print('Resetting confusion matrix...')
        self.confusion_matrix = np.zeros((self.classes,self.classes))


    #sigmoid function activation function, eq 3.20 in compendium
    def sigmoid(self, x):
        '''
        function sigmoid: Implementation of sigmoid function.
        param self: makes it so you can access all the variables in __init__.
        param x: np.array of the instance.
        '''

        return np.array(1/(1+ np.exp(-x)))
    #another sigmoid function, not used for now
    def sigmoid2(self, x, w):
        return 1/(1+np.exp(-np.matmul(w,x)))
    #calculation the gradient_gk MSE, part of eq:3.21 compendium
    def grad_gk_mse_f(self, g_k, t_k):
        '''
        function grad_gk_mse_f: implementation of eq:3.21 compendium.
        param g_k: discriminant array.
        param t_k: true label, ex: [0,1,0] for class 2.
        '''
        grad = np.multiply((g_k-t_k),g_k)
        return grad
    #calculation the gradient_w z_k, part of eq:3.21 compendium
    def grad_W_zk_f(self, x):
        '''
        function grad_W_zk_f: #calculation the gradient_w z_k, part of eq:3.21 compendium. Same as transposing one dim array
        param x: features.
        '''
        grad = x.reshape(1,self.features)
        return grad
    #calculation the gradient_W mse, eq:3.22 compendium
    def grad_W_MSE_f(self, g_k, grad_gk_mse, grad_W_zk):
        '''
        function grad_W_MSE_f: calculation the gradient_W mse, eq:3.22 compendium.
        param g_k: discriminant array.
        param grad_gk_mse: gradient of g_kMSE.
        param grad_W_zk: gradient for Wz_k.
        '''
        return np.matmul(np.multiply(grad_gk_mse,(1-g_k)),grad_W_zk)
    #calculation MSE, eq:3.19
    def MSE_f(self, g_k,t_k):
        '''
        function MSE_f: calculation of the MSE eq:3.19
        param g_k: discriminant array.
        param t_k: true label, ex: [0,1,0] for class 2.
        '''
        return 0.5*np.matmul((g_k-t_k).T,(g_k-t_k))


    #training the model
    def train_model(self):
        print(f'Training model with {self.iterations} iterations and alpha={self.alpha}.')
        #setting some init variables.
        self.g_k[0] = 1
        #looping through every iterations
        for i in range(self.iterations):
            #setting start values, and resetting these every iteration
            grad_W_MSE = 0
            MSE = 0
            k = 0 #this is just to know whats the target class is.

            for j, x in enumerate(self.train): #isnt really necessary to use enumerate, see if i should change
```

```
                if j%30==0 and j!=0:
                    k += 1
                #calculating g_k, eq:3.20 also figure 3.8/3
                self.g_k = self.sigmoid(np.matmul(self.weigths,x.reshape(self.features,1)))
                #addiing to the MSE, see function
                MSE += self.MSE_f(self.g_k,self.t_k[k])
                #calcultation this iteration of grad gk mse
                grad_gk_mse = self.grad_gk_mse_f(self.g_k,self.t_k[k])
                #calcultation this iteration of grad W zk
                grad_W_zk = self.grad_W_zk_f(x)
                #calcultation this iteration of grad W MSE
                grad_W_MSE += self.grad_W_MSE_f(self.g_k, grad_gk_mse, grad_W_zk)
            #adding the MSE to the list of mses to see the model converge
            self.mses[i] = MSE[0]
            #updating the weigths
            self.weigths = self.weigths-self.alpha*grad_W_MSE

            #printing the progress
            if(100*i /self.iterations) % 10 == 0:

                print(f"\rProgress passed {100 * i / self.iterations}%", end='\n')


        print(f"\rProgress passed {(i+1)/self.iterations *100}%", end='\n')
        print('Done')
        #returning the weigths, this is not necesarry as the self does it automatically
        return self.weigths

    #function for testing the model
    def test_model(self): #or call this def fit(), to be simular as other lib.
        #just checking for some wrong inputs
        if(np.all((self.weigths==0 ))):
            print('You need to train the model first')
            return False
        if(np.all((self.confusion_matrix != 0))):
            print('You have to reset the confusion matrix first')
            print('Resetting confusion matrix')
            self.reset_confusion_matrix()

        # if test is None: #used another fix for this
        #     test = self.test
        # else:
        #     print(test)
        #     print('Testing model with training set')
        #     print('Resetting confusion matrix')

            #self.confusion_matrix = np.zeros((self.classes,self.classes))
        print(f'Testing model with {self.iterations} iterations and alpha={self.alpha}.')
        #making predictons by matmul weigths and rows in the test set, then adding the prediction and true label too confusion matrix
        for clas, test_set in enumerate(self.test):
            for row in test_set:
                prediction = np.argmax(np.matmul(self.weigths,row))
                self.confusion_matrix[clas,prediction] += 1

        return self.confusion_matrix
    #just a function that prints the cm, could have done a nice print. also calculating error rate
    def print_confusion_matrix(self):
        print(self.confusion_matrix)
        dia_sum = 0
        for i in range(len(self.confusion_matrix)):
            dia_sum += self.confusion_matrix[i, i]
        error = 1 - dia_sum / np.sum(self.confusion_matrix)
        print(f'error rate = {100 * error:.1f}%')

    #plotting the confusion matrix, not much to see here
    def plot_confusion_matrix(self, name='ok', save=False):
        dia_sum = 0
        for i in range(len(self.confusion_matrix)):
            dia_sum += self.confusion_matrix[i, i]
        error = 1 - dia_sum / np.sum(self.confusion_matrix)
```

```python

        df_cm = pd.DataFrame(self.confusion_matrix, index = [i for i in self.class_names],
                  columns = [i for i in self.class_names])
        plt.figure(figsize = (10,7))
        sn.heatmap(df_cm, annot=True, cmap="YlGnBu")
        plt.title(f'Confusion matrix for Iris task\n iteration: {self.iterations}, alpha: {self.alpha}.\n error rate = {100 * error:.1f}%')
        if save:
            plt.savefig(f'./figurer/confusionmatrixIris_{name}_it{self.iterations}_alpha{self.alpha}.png',dpi=200)
        else:
            plt.show()
        plt.clf()
        plt.close()

    #plotting the MSE, not used on less i just have one alpha
    def plot_MSE(self, save=False, log=False):
        plt.plot(self.mses)
        plt.title(f'MSE for Iris task\n iteration: {self.iterations}, alpha: {self.alpha}.')
        plt.xlabel('Iteration')
        plt.ylabel('Mean square error')
        plt.grid('on')
        if log:
            plt.xscale('log')
        if save:
            plt.savefig(f'mse_it{self.iterations}_alpha{self.alpha}.png',dpi=200)
        else:
            plt.show()
#plotting many alphas and their mse, can see which works best
def plot_mses_array(arr, alphas, name='ok', save=False):
    a = 0
    alpha = r'$ \alpha $'
    for i in arr:
        plt.plot(i,label=f'{alpha}={alphas[a]}')
        a += 1

    plt.title('Mean square error for all test')
    plt.grid('on')
    plt.xlabel('Iteration')
    plt.ylabel('Mean square error')
    plt.legend(loc=1)
    if save:
        plt.savefig(f'./figurer/MSE_all_{name}.png', dpi=200)
    else:
        plt.show()
    plt.clf()
    plt.close()








#loading the data to a pandas dataframe. Using pandas as it has a nice visulaztion and is easy to manipulate

def load_data(path, one=True, maxVal=None, normalize=False, d=','): #change normalize to true to normalize the feature data
    data = pd.read_csv(path, sep=d) #reading csv file, and splitting with ","
    #data.columns = ['sepal_length','sepal_width','petal_length','petal_width','species']#making columnnames, for easier understanding
    #data.describe()#this gives all the information you need: count, mean, std, min, 25%, 50%,75%,max
    if one: #dont wont a column of ones when plotting
        lenght = len(data)
        #adding ones
        if lenght>60:

            data.insert(4,'Ones',np.ones(lenght),True)

```

```
        else:
            data['Ones'] = np.ones(lenght)
    #normalize
    if normalize:
        data = data.divide(maxVal)

    return data

#removing the feature from dataset, this can be a list
def remove_feature_dataset(data, features):
    data = data.drop(columns=features)
    print(data.head())
    return data

#this will filter out the dataframe, not used now but nice to have
def filter_dataset(data,features):
    data = data.filter(items=features)
    return data

#-------------global variables---------------#
classes = 3
iris_names = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
#features = ['sepal_length','sepal_width','petal_length','petal_width']
path = 'Iris_TTT4275/iris.csv'
path_setosa = 'Iris_TTT4275/class_1.csv'
path_versicolour = 'Iris_TTT4275/class_2.csv'
path_virginica = 'Iris_TTT4275/class_3.csv'
#-------------global variables---------------#




#----------------get data--------------------#

tot_data = load_data(path, normalize=False)
max_val = tot_data.max(numeric_only=True).max() #first max, gets max of every feature, second max gets max of the features
setosa = load_data(path_setosa,max_val)
versicolor = load_data(path_versicolour, max_val)
virginica = load_data(path_virginica, max_val)

#---------------^get data^-------------------#

#alphas, this could be a chosen by the user.
#alphas = [1,0.1,0.01,0.001,0.0001,0.00001]
alphas = [0.01]

#the tasks, hacky setup but ez copy paste every task. there is some waste with the loading of dataset, but it works
def task1a(s=False):
    train_size = 30
    arr= []
    features = ['sepal_length','sepal_width','petal_length','petal_width']


    #----------------prepros data--------------------#
    #split_data_array = [setosa,versicolor,virginica] #not necessary

    #splitting up in test and train sets
    train = pd.concat([setosa[0:train_size],versicolor[0:train_size],virginica[0:train_size]])
    train_for_test = np.array([setosa[0:train_size],versicolor[0:train_size],virginica[0:train_size]])
    test = np.array([setosa[train_size:],versicolor[train_size:],virginica[train_size:]]) #could mb have done this for train to,
    t_k = np.array([[[1],[0],[0]],[[0],[1],[0]],[[0],[0],[1]]]) #making array to check whats the true class is
    #just making dataframe to numpy array
    train = train.to_numpy()
    #---------------^prepros data^-------------------#

    for i in range(len(alphas)):
        print(f'Making model with 2000 iteration and an alpha of {alphas[i]} ')
```

```python
        model = f'w{i}'
        model = LDC(train,test,t_k,2000,alphas[i], features)
        model.train_model()
        model.get_weigths()
        arr.append(model.mses)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='test', save=s)
        print('Testing the model with the training set')
        model.reset_confusion_matrix()
        model.set_test(train_for_test)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='train_1a', save=s)


    plot_mses_array(arr, alphas, name='test_1a', save=s)

def task1d(s=False):
    train_size = 20 #still a training size of length 30, but to get the 30 last i use 20 here
    arr = [] #
    features = ['sepal_length','sepal_width','petal_length','petal_width']

    #----------------prepros data--------------------#
    #split_data_array = [setosa,versicolor,virginica] #not necessary

    #splitting up in test and train sets
    train = pd.concat([setosa[train_size:],versicolor[train_size:],virginica[train_size:]])
    train_for_test = np.array([setosa[train_size:],versicolor[train_size:],virginica[train_size:]])
    test = np.array([setosa[0:train_size],versicolor[0:train_size],virginica[0:train_size]]) #could mb have done this for train to,
    t_k = np.array([[[1],[0],[0]],[[0],[1],[0]],[[0],[0],[1]]]) #making array to check whats the true class is
    #just making dataframe to numpy array
    train = train.to_numpy()
    #---------------^prepros data^-------------------#

    for i in range(len(alphas)):
        print(f'Making model with 2000 iteration and an alpha of {alphas[i]} ')
        model = f'wl{i}'
        model = LDC(train,test,t_k,2000,alphas[i], features)
        model.train_model()
        model.get_weigths()
        arr.append(model.mses)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='test', save=s)
        print('Testing the model with the training set')
        model.reset_confusion_matrix()
        model.set_test(train_for_test)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='train_1d', save=s)


    plot_mses_array(arr, alphas, name='test_1d', save=s)

#changing the name of the train test was necesarry to make it work when running all script at once, since the global variable i had changed the set for the others. this is still a bad solution.
def task2a(s=False):
    #global setosa,versicolor,virginica
    train_size = 30
    arr = []
    features = ['sepal_length','petal_length','petal_width']
    #removing the sepal width feature because it shows most overlap
    re_feature = ['sepal_width']
    setosa1 = remove_feature_dataset(setosa,re_feature)
    versicolor1 = remove_feature_dataset(versicolor,re_feature)
    virginica1 = remove_feature_dataset(virginica,re_feature)

    #----------------prepros data--------------------#
```

```
    #split_data_array = [setosa,versicolor,virginica] #not necessary

    #splitting up in test and train sets
    train = pd.concat([setosa1[0:train_size],versicolor1[0:train_size],virginica1[0:train_size]])
    train_for_test = np.array([setosa1[0:train_size],versicolor1[0:train_size],virginica1[0:train_size]])
    test = np.array([setosa1[train_size:],versicolor1[train_size:],virginica1[train_size:]]) #could mb have done this for train to,
    t_k = np.array([[[1],[0],[0]],[[0],[1],[0]],[[0],[0],[1]]]) #making array to check whats the true class is
    #just making dataframe to numpy array
    train = train.to_numpy()
    #---------------^prepros data^-------------------#
    for i in range(len(alphas)):
        print(f'Making model with 2000 iteration and an alpha of {alphas[i]} ')
        model = f'w2{i}'
        model = LDC(train,test,t_k,2000,alphas[i], features)
        model.train_model()
        model.get_weigths()
        arr.append(model.mses)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='test_2a', save=s)
        print('Testing the model with the training set')
        model.reset_confusion_matrix()
        model.set_test(train_for_test)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='train_2a', save=s)


    plot_mses_array(arr, alphas, name='test_2a', save=s)


def task2b_1(s=False):
    #also removing sepal length since it also showed alot of overlap
    #global setosa,versicolor,virginica
    train_size = 30
    arr = []
    features = ['petal_length','petal_width']
    #removing the sepal width feature because it shows most overlap
    re_feature = ['sepal_length','sepal_width']
    setosa2 = remove_feature_dataset(setosa,re_feature)
    versicolor2 = remove_feature_dataset(versicolor,re_feature)
    virginica2 = remove_feature_dataset(virginica,re_feature)

    #----------------prepros data--------------------#
    #split_data_array = [setosa,versicolor,virginica] #not necessary

    #splitting up in test and train sets
    train = pd.concat([setosa2[0:train_size],versicolor2[0:train_size],virginica2[0:train_size]])
    train_for_test = np.array([setosa2[0:train_size],versicolor2[0:train_size],virginica2[0:train_size]])
    test = np.array([setosa2[train_size:],versicolor2[train_size:],virginica2[train_size:]]) #could mb have done this for train to,
    t_k = np.array([[[1],[0],[0]],[[0],[1],[0]],[[0],[0],[1]]]) #making array to check whats the true class is
    #just making dataframe to numpy array
    train = train.to_numpy()
    #---------------^prepros data^-------------------#
    for i in range(len(alphas)):
        print(f'Making model with 2000 iteration and an alpha of {alphas[i]} ')
        model = f'w2{i}'
        model = LDC(train,test,t_k,2000,alphas[i], features)
        model.train_model()
        arr.append(model.mses)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='test_2b1', save=s)
        print('Testing the model with the training set')
        model.reset_confusion_matrix()
```

```python
        model.set_test(train_for_test)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='train_2b1', save=s)


    plot_mses_array(arr, alphas, name='test_2b1', save=s)

def task2b_2(s=False):
    #also removing petal width
    #global setosa,versicolor,virginica
    train_size = 30
    arr = []
    features = ['petal_length']
    #removing the sepal width feature because it shows most overlap
    re_feature = ['sepal_length','sepal_width','petal_width']
    setosa3 = remove_feature_dataset(setosa,re_feature)
    versicolor3 = remove_feature_dataset(versicolor,re_feature)
    virginica3 = remove_feature_dataset(virginica,re_feature)

    #----------------prepros data--------------------#
    #split_data_array = [setosa,versicolor,virginica] #not necessary

    #splitting up in test and train sets
    train = pd.concat([setosa3[0:train_size],versicolor3[0:train_size],virginica3[0:train_size]])
    train_for_test = np.array([setosa3[0:train_size],versicolor3[0:train_size],virginica3[0:train_size
]])
    test = np.array([setosa3[train_size:],versicolor3[train_size:],virginica3[train_size:]]) #could mb
 have done this for train to,
    t_k = np.array([[[1],[0],[0]],[[0],[1],[0]],[[0],[0],[1]]]) #making array to check whats the true
class is
    #just making dataframe to numpy array
    train = train.to_numpy()
    #---------------^prepros data^-------------------#
    for i in range(len(alphas)):
        print(f'Making model with 2000 iteration and an alpha of {alphas[i]} ')
        model = f'w3{i}'
        model = LDC(train,test,t_k,2000,alphas[i], features)
        model.train_model()
        arr.append(model.mses)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='test_2b2', save=s)
        print('Testing the model with the training set')
        model.reset_confusion_matrix()
        model.set_test(train_for_test)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='train_2b2', save=s)


    plot_mses_array(arr, alphas, name='test_2b2', save=s)

def task2b_2_1(s=False):
    #Testing with removing petal length
    #global setosa,versicolor,virginica, not nice solution

    train_size = 30
    arr = []
    features = ['petal_width']
    #removing the sepal width feature because it shows most overlap
    re_feature = ['sepal_length','sepal_width','petal_length']
    setosa4 = remove_feature_dataset(setosa,re_feature)
    versicolor4 = remove_feature_dataset(versicolor,re_feature)
    virginica4 = remove_feature_dataset(virginica,re_feature)

    #----------------prepros data--------------------#
    #split_data_array = [setosa,versicolor,virginica] #not necessary

    #splitting up in test and train sets
    train = pd.concat([setosa4[0:train_size],versicolor4[0:train_size],virginica4[0:train_size]])
```

```
    train_for_test = np.array([setosa4[0:train_size],versicolor4[0:train_size],virginica4[0:train_size]])
    test = np.array([setosa4[train_size:],versicolor4[train_size:],virginica4[train_size:]]) #could mb have done this for train to,
    t_k = np.array([[[1],[0],[0]],[[0],[1],[0]],[[0],[0],[1]]]) #making array to check whats the true class is
    #just making dataframe to numpy array
    train = train.to_numpy()
    #---------------^prepros data^-------------------#
    for i in range(len(alphas)):
        print(f'Making model with 2000 iteration and an alpha of {alphas[i]} ')
        model = f'w4{i}'
        model = LDC(train,test,t_k,2000,alphas[i], features)
        model.train_model()
        arr.append(model.mses)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='test_2b2_1', save=s)
        print('Testing the model with the training set')
        model.reset_confusion_matrix()
        model.set_test(train_for_test)
        model.test_model()
        model.print_confusion_matrix()
        model.plot_confusion_matrix(name='train_2b2_1', save=s)


    plot_mses_array(arr, alphas, name='test_2b2_1', save=s)

#runs if this program is ran in the terminal, py/python iris_classes.py. ofc need to uncomment the task, can use argument s=True to save the images with good quality
if __name__ == '__main__':
    path = 'iris.csv'
    path_setosa = 'class_1.csv'
    path_versicolour = 'class_2.csv'
    path_virginica = 'class_3.csv'
    # task1a()
    # task1d()
    # task2a()
    # task2b_1()
    # task2b_2()
    # task2b_2_1()
    pass
```

D. Appendix D: MNIST code.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from keras.datasets import mnist
import operator
import seaborn
import time
from sklearn.cluster import KMeans
from scipy.spatial import distance
import datetime

#Loading the MNIST dataset from keras
(train_X, train_y), (test_X, test_y) = mnist.load_data()

# print('X_train: ' + str(train_X.shape))
# print('Y_train: ' + str(train_y.shape))
# print('X_test:  '  + str(test_X.shape))
# print('Y_test:  '  + str(test_y.shape))

#  -----------------------------------------------------
# |                                                     |
# |                     Clustering                      |
# |                                                     |
```

```
#  ----------------------------------------------------
"""
:function sortData: Sorting the MNIST images after each class from 0 to 9 and keeping track of how many of each.
:param train_X: Training images.
:param train_y: Labels of the training images.
:return sortedTrainX: The sorted array of images from 0 to 9.
:return numbCount: List of how many images of each class.
"""
def sortData(train_X, train_y):
    numbCount = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]      #Keeps track of how many of each label.

    for i in range(len(train_y)):                    #Iterates through whole length of train_y to find how many images of each class.
        numbCount[train_y[i]] += 1

    sortedTrainY = np.argsort(train_y)               #Sort after index  .
    sortedTrainX = np.empty_like(train_X)            #Empty array with same shape as train_X for sorted array of train_X.

    for i in range(len(train_y)):                    #Adds all of train_X in a sorted manner, based on label.
        sortedTrainX[i] = train_X[sortedTrainY[i]]
    return sortedTrainX, numbCount

"""
:function cluster:      Clustering the training images with a total of 640 clusters, 64 for each class.
:param train_X:         Training images.
:param train_y:         Labels of the training images.
:param M:               Number of clusters in each class.
:return clusters:       Array of clusters (images) in ascending order from 0 to 9.
"""
def cluster(train_X, train_y, M):
    clusterStart = time.time()
    sortedTrainX, numbCount = sortData(train_X, train_y)                    #Retrieving the sorted array of images and the list of how many images of each class.
    flattenedSortedTrainX = sortedTrainX.flatten().reshape(60000, 784)      #Reshaping sortedTrainX to desired format
    clusters = np.empty((len(numbCount), M, 784))                           #Making an empty array of desired size
    before = 0
    after = 0

    for count, i in enumerate(numbCount):                                   #Making 64 clusters, classwise.
        after += i                                                          #Splice tracking.
        clustered = KMeans(n_clusters=M, random_state=0).fit(flattenedSortedTrainX[before:after]).cluster_centers_  #Get the 64 clusters.
        before = after                                                      #Splice tracking.
        clusters[count] = clustered                                         #Add to cluster array.
        print(count)

    clusterEnd = time.time()
    return clusters.flatten().reshape(len(numbCount)*64, 784)               #Reshaped cluster for distance measuring.

#  ----------------------------------------------------
# |                                                    |
# |              NN and KNN implementation             |
# |                                                    |
#  ----------------------------------------------------
"""
:class NN: Nearest Neighbour class. Alle the NN and KNN implementations are done here.
"""
class NN():
    def __init__(self, K=7):
        self.K = K
    def fit(self, train_X, train_y):
        self.train_X = train_X
        self.train_y = train_y
```

```
    """
    :function predictCKNN:  Implementation of KNN algorithm with clustering.
    :param self:            Internal variables.
    :param test_X:          Test images.
    :param M:               Number of clusters in each class.
    :return predictions:    List of predicted/classified labels.
    """
    def predictCKNN(self, test_X, M):
        predictions = []
        index = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        start = time.time()
        clusters = cluster(self.train_X, self.train_y, M)
        clusterTimeEnd = time.time()
        print(f"Runtime of the clustering is: {str(datetime.timedelta(seconds = (time.time() - start)))}.")
        reshapedTestX = test_X.flatten().reshape((len(test_X), 784))
        start = time.time()

        for i in range(len(test_X)):            #Iterate through length of test_X and add the predicted class to predictions.
            dist = []

            for count in range(len(clusters)):  #Iterate through each class and find the K cluster images with the least distance from test image.
                dist.append(distance.euclidean(reshapedTestX[i], clusters[count]))

            sortedDist = np.argsort(dist)[:self.K]
            classCount = {}

            for j in sortedDist:                #Iterate through the K cluster images and find the class with the majority.
                number = index[int(j//64)]
                if number in classCount:
                    classCount[number] += 1
                else: classCount[number] = 1

            predictions.append(max(classCount, key=classCount.get))
        print(f"Runtime of the KNN with clustering is: {str(datetime.timedelta(seconds = (time.time() - start)))}.")
        return predictions
    """
    :function predictNN:            Implementation of NN algorithm without clustering.
    :param self:                    Internal variables.
    :param test_X:                  Test images.
    :return predictions:            List of predicted/classified labels.
    :return success_predictions:    Array of images successfully classified.
    :return fail_predictions:       Array of images unsuccessfully classified.
    """
    def predictNN(self, test_X):
        predictions = []
        fail_predictions = []
        success_predictions = []
        for i in range(len(test_X)):                    #Iterate through length of test_X and add the predicted class to predictions.
            dist = []
            for j in range(len(self.train_X)):          #Iterate through length of training set and find training image with the least distance from test image.
                dist.append(eucledianDistance(test_X[i], self.train_X[j]))
            NN_index = np.argmin(dist)                  #Get the index of that training image.
            if test_y[i] != train_y[NN_index]:
                fail_predictions.append([test_X[i], train_X[NN_index]])
            else:
                success_predictions.append([test_X[i], train_X[NN_index]])
            predictions.append(self.train_y[NN_index])  #Add the training image label to predictions.
        return predictions, success_predictions, fail_predictions
    """
    :function predictCNN:           Implementation of NN algorithm clustering.
    :param self:                    Internal variables.
    :param test_X:                  Test images.
    :param M:                       Number of clusters in each class.
    :return predictions:            List of predicted/classified labels.
```

```python
    :return success_predictions:    Array of images successfully classified.
    :return fail_predictions:       Array of images unsuccessfully classified.
    """
    def predictCNN(self, test_X, M):
        predictions = []
        fail_predictions = []
        success_predictions = []
        index = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        start = time.time()
        clusters = cluster(self.train_X, self.train_y, M)
        clusterTimeEnd = time.time()
        print(f"Runtime of the clustering is: {str(datetime.timedelta(seconds = (time.time() - start)))}.")
        reshapedTestX = test_X.flatten().reshape((len(test_X), 784))
        start = time.time()

        for i in range(len(test_X)):                  #Iterate through length of test_X and add the predicted class to predictions.
            dist = []

            for count in range(len(clusters)):       #Iterate through each class and find the cluster image with the least distance from test image.
                dist.append(distance.euclidean(reshapedTestX[i], clusters[count]))

            NN_index = np.argmin(dist)               #Get the index of that training image.
            if test_y[i] != index[int(NN_index//64)]:
                fail_predictions.append([test_X[i], clusters[NN_index].flatten().reshape((28, 28))])
            else:
                success_predictions.append([test_X[i], clusters[NN_index].flatten().reshape((28, 28))])
            predictions.append(index[int(NN_index//64)])
        print(f"Runtime of the NN with clustering is: {str(datetime.timedelta(seconds = (time.time() - start)))}.")
        return predictions, success_predictions, fail_predictions

#  -----------------------------------------------------
# |                                                     |
# |                 Plot functions                      |
# |                                                     |
#  -----------------------------------------------------
def getConfusionMatrix(predictions):
    confusion_matrix = np.zeros((10,10))
    for i, x in enumerate(predictions):
            confusion_matrix[test_y[i], x] += 1
    return confusion_matrix
def getConfusionMatrixNormalized(predictions):
    confusion_matrix = np.zeros((10,10))
    for i, x in enumerate(predictions):
            confusion_matrix[test_y[i], x] += 1
    return confusion_matrix/np.amax(confusion_matrix)
def plotConfusionMatrix(confusion_matrix, testSize, trainingSize, text):
    dia_sum = 0
    for i in range(len(confusion_matrix)):
        dia_sum += confusion_matrix[i, i]
        error = 1 - dia_sum / np.sum(confusion_matrix)
    class_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    df_cm = pd.DataFrame(confusion_matrix, index = [i for i in class_names], columns = [i for i in class_names])
    plt.figure(figsize = (10,7))
    seaborn.heatmap(df_cm, annot=True, cmap="YlGnBu", fmt='g')
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title(f'Confusion matrix for MNIST task\n Training size: {trainingSize}, Test size: {testSize}\n Error rate = {100 * error:.1f}% \n ')
    #plt.savefig(f'./figures/Confusion_matrix_{text}_c{trainingSize}_t{testSize}_e{100*error:.0f}_raw.png', dpi=200)
    plt.show()
def plotFailedPredictions(fail_predictions, text):
    # for i in range(9):
    plt.subplot(330 + 1 )
    plt.title('Test')
```

```python
    plt.imshow(fail_predictions[0][0], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 1 )
    plt.title('Predicted')
    plt.imshow(fail_predictions[0][1], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 2 )
    plt.title('Difference')
    plt.imshow(differenceImage(fail_predictions[0][0], fail_predictions[0][1]), cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 3)
    plt.imshow(fail_predictions[1][0], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 4)
    plt.imshow(fail_predictions[1][1], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 5)
    plt.imshow(differenceImage(fail_predictions[1][0], fail_predictions[1][1]), cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 6)
    plt.imshow(fail_predictions[2][0], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 7)
    plt.imshow(fail_predictions[2][1], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 8)
    plt.imshow(differenceImage(fail_predictions[2][0], fail_predictions[2][1]), cmap=plt.get_cmap('gray'))
    #plt.savefig(f'./figures/{text}_failed_predictions.png', dpi=200)
    plt.show()
def plotSuccessPredictions(success_predictions, text):
    plt.subplot(330 + 1 )
    plt.title('Test')
    plt.imshow(success_predictions[0][0], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 1 )
    plt.title('Predicted')
    plt.imshow(success_predictions[0][1], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 2 )
    plt.title('Difference')
    plt.imshow(differenceImage(success_predictions[0][0], success_predictions[0][1]), cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 3)
    plt.imshow(success_predictions[1][0], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 4)
    plt.imshow(success_predictions[1][1], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 5)
    plt.imshow(differenceImage(success_predictions[1][0], success_predictions[1][1]), cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 6)
    plt.imshow(success_predictions[2][0], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 7)
    plt.imshow(success_predictions[2][1], cmap=plt.get_cmap('gray'))
    plt.subplot(330 + 1 + 8)
    plt.imshow(differenceImage(success_predictions[2][0], success_predictions[2][1]), cmap=plt.get_cmap('gray'))
    #plt.savefig(f'./figures/{text}_success_predictions.png', dpi=200)
    plt.show()

#  -----------------------------------------------------
# |                                                     |
# |                Distance functions                   |
# |                                                     |
#  -----------------------------------------------------
"""
    :function differenceImage:  Finding the difference between the two images.
    :param img1:                Test image.
    :param img2:                Training image.
    :return a*b:                The difference between the images.
"""
def differenceImage(img1, img2):
    a = img1-img2
    b = np.uint8(img1<img2) * 254 + 1
    return a * b
"""
    :function eudcledianDistance:  Implementation of KNN algorithm with clustering.
    :param img1:                   Test image.
    :param img2:                   Training image.
```

```
    :return ...:                    The Eucledian distance between the images.
"""
def eucledianDistance(img1, img2):
    return np.sum(differenceImage(img1, img2))

#  -----------------------------------------------------
# |                                                     |
# |                     Run code                        |
# |                                                     |
#  -----------------------------------------------------
def runNN(trainingSize, testSize, plotConfusionMat, plotFailedPred, plotSuccessPred):
    model = NN()
    model.fit(train_X[:trainingSize], train_y[:trainingSize])
    predictions, success_predictions, fail_predictions = model.predictNN(test_X[:testSize])
    if plotConfusionMat:
        plotConfusionMatrix(getConfusionMatrix(predictions), testSize, trainingSize, 'NN')
    if plotFailedPred:
        plotFailedPredictions(fail_predictions, 'NN')
    if plotSuccessPred:
        plotSuccessPredictions(success_predictions, 'NN')

def runCNN(trainingSize, testSize, M, plotConfusionMat, plotFailedPred, plotSuccessPred):
    model = NN()
    model.fit(train_X[:trainingSize], train_y[:trainingSize])
    predictions, success_predictions, fail_predictions = model.predictCNN(test_X[:testSize], M)
    if plotConfusionMat:
        plotConfusionMatrix(getConfusionMatrix(predictions), testSize, trainingSize, 'CNN')
    if plotFailedPred:
        plotFailedPredictions(fail_predictions, 'CNN')
    if plotSuccessPred:
        plotSuccessPredictions(success_predictions, 'CNN')

def runCKNN(trainingSize, testSize, M, plotConfusionMat, plotFailedPred, plotSuccessPred):
    model = NN()
    model.fit(train_X[:trainingSize], train_y[:trainingSize])
    predictions = model.predictCKNN(test_X[:testSize], M)
    if plotConfusionMat:
        plotConfusionMatrix(getConfusionMatrix(predictions), testSize, trainingSize, 'CKNN')
    if plotFailedPred:
        plotFailedPredictions(fail_predictions, 'CKNN')
    if plotSuccessPred:
        plotSuccessPredictions(success_predictions, 'CKNN')

    # Load the data
    # Initialize the value of k
    # To getting the predicted class, iterate from 1 to the total number of training data points
    # Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric.
    # Sort the calculated distances in ascending order based on distance values
    # Get top k rows from the sorted array
    # Get the most frequent class of these rows
    # Return the predicted class

#  -----------------------------------------------------
# |                                                     |
# |                       Main                          |
# |                                                     |
#  -----------------------------------------------------
def main():
    # runNN(60000, 10000, True, False, False)          #Takes 2 hours, best performance
    # runCNN(60000, 10000, 64, True, True, True)       #Takes 2-3 minutes, next best performance
    runCKNN(60000, 10000, 64, True, False, False)     #Takes 2-3 minutes, worst performance
    return
if __name__=='__main__':
    main()
```

E. Appendix E: Running the two codes together.

```
from MNIST_TTT4275 import mnist as mn
```

```python
from Iris_TTT4275 import iris_class as ic




def main():
    #could add options to print different informations and/or make it possible to save the images.
    #If you want to change it to save images, just add the option of taking taskx(s=True)
    #with the new python is it possible to make this a switch case or as it is called in python match case, didnt have the version that supported this
    #could make it so that you dont need to type iris or mnist every time, but thats a possible improvement for later.
    run = True
    #just while loop that lets the user decide which task he/her want to run.
    while run:
        action = str(input('What task do you want to check out?\n For the Iris task type <<Iris>> and for the MNIST task type <<MNIST>> or quit by typing <<quit>> or <<q>> at anytime (its is not case sensitive).\n your choice: ')).lower()
        if action == 'iris':
            task = str(input('Which task do you want to see? We have task 1ac, task 1d, task 2a, task 2b or task 2b1. \n just type the number and letter, ex: <<1ac>> or 2a. Or just run all with <<all>>.\n Your choice: ')).lower()
            if task == 'q' or task == 'quit':
                print('Quitting...')
                print('Goodbye!')
                run = False
                #action = 'q'
            elif task == '1ac':
                print('Running task 1ac...')
                ic.task1a()
            elif task == '1d':
                print('Running task 1d...')
                ic.task1d()
            elif task == '2a':
                print('Running task 2a...')
                ic.task2a()
            elif task == '2b':
                print('Running task 2b...')
                ic.task2b_1()
            elif task == '2b1':
                option = str(input('There are her two options, both models are only using 1 feature.\n Option 1 is only using Petal length and option 2 is only using Petal width. \n Type 1 or 2: ')).lower()
                if option == 'q' or option == 'quit':
                    print('Quitting...')
                    print('Goodbye!')
                    run = False

                elif option == '1':
                    print('Running task 2b1_1...')
                    ic.task2b_2()
                elif option == '2':
                    print('Running task 2b1_2...')
                    ic.task2b_2_1()
                else:
                    print('Wrong input')
                    action = 'iris'
            elif task == 'all':
                print('Running all task...')
                ic.task1a()
                ic.task1d()
                ic.task2a()
                ic.task2b_1()
                ic.task2b_2()
                ic.task2b_2_1()
            else:
                print('Wrong input')
                action = 'iris'

        elif action == 'mnist':
```

```
            task = str(input('Which task do you want to see? We have task Nearest Neighbor (NN), clustering Nearest Neighbor (CNN) and clustering K nearest neighbor (CKNN). \n just type <<NN>>, <<CNN>> or <<CKNN>>, not case sensitive.\n Your choice: ')).lower()
            if task == 'q' or task == 'quit':
                print('Quitting...')
                print('Goodbye!')
                run = False
                #action = 'q'
            elif task == 'cnn':
                print('Running task CNN...')
                mn.runCNN(60000, 10000, 64, True, True, True)

            elif task == 'nn':
                safety = str(input('Are you sure you want to run this? This takes 2 hours to run. (y/n): ')).lower()
                if safety == 'y':
                    print('Running task NN...')
                    mn.runNN(60000, 10000, True, True, True)
                else:
                    print('Smart! Going back to choosing task.')


            elif task == 'cknn':
                print('Running task CKNN...')
                mn.runCKNN(60000, 10000, 64, True, False, False)
            else:
                print('Wrong input')
                action = 'mnist'



        elif action == 'quit' or action == 'q':
            print('Quitting...')
            print('Goodbye!')
            run = False

        else:
            print('Wrong input, please try again.')


if __name__ == '__main__':
    main()
```

F. Appendix F: Plotting for Iris task.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.lib.function_base import gradient
import pandas as pd
from sklearn.model_selection import train_test_split
import seaborn as sns
import matplotlib.mlab as mlab
#from scipy.stats import norm
import scipy.stats
from scipy.stats import norm
#maybe just import the dataset from sklearn, makes it much easier to work with.
#could have used pandas.plot function but didnt do it now.
#todo, optimize code, and make it easier to run. now i gets the data many times.it is slow.
#------------Variables--------------#
classes = 3
iris_names = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
features = ['sepal_length','sepal_width','petal_length','petal_width']
path = 'iris.csv'
path_setosa = 'class_1.csv'
path_versicolour = 'class_2.csv'
path_virginica = 'class_3.csv'
#-----------^Variables^-------------#

#---------Importing data------------#
```

```python
def load_data(path, one=False, maxVal=None, normalize=False, d=','): #change normalize to true to normalize the feature data
    data = pd.read_csv(path, sep=d) #reading csv file, and splitting with ","
    #data.columns = ['sepal_length','sepal_width','petal_length','petal_width','species']#making columnnames, for easier understanding
    #data.describe()#this gives all the information you need: count, mean, std, min, 25%, 50%,75%,max
    # if one: #dont wont a column of ones when plotting
    #     lenght = len(data)
    #     #adding ones
    #     if lenght>60:

    #         data.insert(4,'Ones',np.ones(lenght),True)

    #     else:
    #         data['Ones'] = np.ones(lenght)
    #normalize
    t = one
    if normalize:
        data = data.divide(maxVal)

    return data



#--------^Importing data^-----------#

#-----------plotting----------------#
def plot_petal(data):
    color = ['red', 'blue', 'green'] #get different colors to the plot
    #petal_length = np.array(data['petal_length'])
    #petal_width = np.array(data['petal_width'])
    for i in range(len(data)):# iterate through the three datasets
       name = iris_names[i] #get the name for the classes from the global array, iris_names
       plt.scatter(np.array(data[i]['petal_width']),np.array(data[i]['petal_length']), label=name, color=color[i]) #plot a scatter plot,with length as x-axis and width as y-axis
    #add som useful information to plot under.
    plt.legend()
    plt.xlabel('Petal width in cm')
    plt.ylabel('Petal length in cm')
    plt.title('Petal data')
    plt.grid('On')
    plt.savefig('petal_scatterplot_gridon_width-length.png')
    plt.show()

def plot_sepal(data):
    color = ['red', 'blue', 'green'] #get different colors to the plot
    #petal_length = np.array(data['petal_length'])
    #petal_width = np.array(data['petal_width'])
    for i in range(len(data)):# iterate through the three datasets
       name = iris_names[i] #get the name for the classes from the global array, iris_names
       plt.scatter(np.array(data[i]['sepal_width']),np.array(data[i]['sepal_length']), label=name, color=color[i]) #plot a scatter plot,with length as x-axis and width as y-axis
    #add som useful information to plot under.
    plt.legend()
    plt.xlabel('Sepal width in cm')
    plt.ylabel('Sepal length in cm')
    plt.title('Sepal data')
    #plt.grid('On')
    plt.show()

def plot_histogram(data): #change step size, to change the dimension on the histogram bars, org:0.03, used 0.003 when normalized
    sns.set()
    sns.set_style("white")

    # make the 'species' column categorical to fix the order
    data['species'] = pd.Categorical(data['species'])

    fig, axs = plt.subplots(2, 2, figsize=(12, 6))
    for col, ax in zip(data.columns[:4], axs.flat):
```

```
        sns.histplot(data=data, x=col, kde=True, hue='species', common_norm=False, legend=ax==axs
    [0,0], ax=ax)
    plt.tight_layout()
    plt.savefig('newhist_withbestfit.png',dpi=200)
    plt.show()

def oldhist(data,step=0.03):
    #--------making histogram basis-------#
    fig, axes = plt.subplots(nrows= 2, ncols=2, sharex='col', sharey='row')#basis for subplots
    colors= ['blue', 'red', 'green', 'black'] #colors for histogram
    max_val = np.amax(data)# Finds maxvalue in samples



    for i, ax in enumerate(axes.flat):#loop through every feature
        for label, color in zip(range(len(iris_names)), colors): #loop through every class
            #plot histogram from class[feature]
            ax.hist(data[label][features[i]], label=iris_names[label], color=color, stacked=True,alpha
    =0.5)
            ax.set_xlabel(features[i]+'( cm)') #add axis name
            ax.legend(loc='upper right')

        ax.set(xlabel='Measured [cm]', ylabel='Number of samples') #sets label name
        ax.label_outer() #makes the label only be on the outer part of the plots
        ax.legend(prop={'size': 7}) #change size of legend
        ax.set_title(f'Feature {i+1}: {features[i]}') #set title for each plot
        #ax.grid('on') #grid on or off

        #plt.savefig('histogram_rap.png',dpi=200)

        plt.show()






#----------^plotting^---------------#
if __name__ == '__main__':

    #----------------get data--------------------#
    tot_data = load_data(path, normalize=False)

    max_val = tot_data.max(numeric_only=True).max() #first max, gets max of every feature, second max
    gets max of the features
    setosa = load_data(path_setosa,max_val)
    print(setosa.head())

    versicolor = load_data(path_versicolour, max_val)
    virginica = load_data(path_virginica, max_val)
    split_data_array = [setosa,versicolor,virginica]

    #-----------------plot------------------------#
    #plot_histogram(split_data_array)
    plot_histogram(tot_data)
    #plot_petal(split_data_array)
```