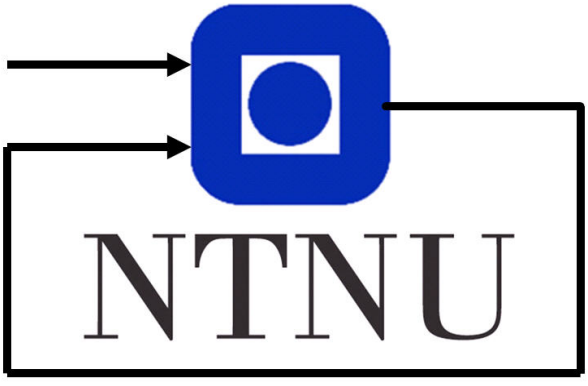
TTT4275 Estimation, detection and classification
Classification project

Adrian L. Pavlak, 544951
Haakon K. Garfjell, 544992

April, 2023



Department of Engineering Cybernetics

Summary

In this report, discussion and results regarding two classification tasks from the course TTT4275 at the Norwegian University of Science and Technology will be presented.

In the first task we trained a Linear Discriminant Classifier using minimum square error as a cost function. The training was done on features from 30 iris flowers with the goal of creating a model that can distinguish between three types of irises. The learning rate α and the number of iterations was chosen experimentally. Depending on what data the model was trained on we accomplished an error rate of 1.67% to 5.0% when applying the LDC on unseen data. The error rate was highly dependent on the choice of training data and which features were used in the model.

The second task is related to template based classification. Here we classified handwritten digits from the MNIST database. By using the Nearest Neighbor algorithm we were able to achieve an error rate of 3.0%, but at the cost of a runtime of 01:14:05. By clustering the data, effectively reducing the dataset from 60,000 to 640, the runtime decreased to only 00:02:19, but the error increased to 4.61% for the same nearest neighbor algorithm. We also applied the K-nearest neighbor algorithm, resulting in an increased error of 5.71% and 6.49% for $K = 3$ and $K = 7$ respectively.

Contents

1 Introduction

The purpose of this project is to gain a deeper understanding of classifiers because of their increasing significance in various fields. For instance, in the field of autonomy, a self-driving car needs to be able to identify a red stop sign from other objects. Similarly, in medicine, CT scans must be able to classify harmful from non-harmful tumors. With the advancement of computing power and better algorithms, classifiers have become more efficient and effective.

To achieve our goal, we will begin by introducing the necessary theory required to comprehend the task at hand. We will then explore the implementation and results separately for each of the problems that were studied. Finally, we will draw conclusions and describe our findings based on the analysis conducted throughout the project.

2 Theory

This chapter will explain the underlying theory used in the project. All derivations in this chapter are taken from [1]. Classification is the process of grouping similar data into classes based on a given set of attributes or features. When someone is out foraging mushrooms they have to separate the ones that are edible from the ones that are poisonous. This is in it self a challenging classification problem for humans to solve, but when we want to apply classification to a large scale like for example in email spam filtering, computer algorithms are much more applicable. The classifiers used in this project are the *linear discriminant classifier*, and the *template based classifier*.

2.1 Linear discriminant classifier

The linear discriminant classifier (LDC) is a method suitable for linearly separable problems. These are problems where one can divide the feature space by a straight line or hyperplane in order to separate the data points belonging to each class. In a general discriminant classifier each class is described by a discriminant function $g_i(x)$ and the decision rule:

$$x \in \omega_j \quad \iff \quad g_j(x) = \max_i g_i(x) \tag{1}$$

for class ω_j. For a linear discriminant classifier, the discriminant function can be written as:

$$g_i(x) = \omega_i^T x + \omega_{io}, \quad i = 1, \ldots, C \tag{2}$$

Here ω_{io} is the offset for class ω_i and C is the number of classes. As 2 only consists of linear terms we can rewrite it for $C > 2$ classes as:

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

where q and ω_o are vectors of length C, and W is a matrix of dimension CxD, D being the number of features. By changing the weights in W and adjusting the offset ω_o we can create a line (or hyperplane) that hopefully separates each class. To make the output compatible with one hot labeling explained in 2.4, we can apply the sigmoid function to map the output to be between 0 and 1.

$$g_{ik} = sigmoid(z_{ik}) = \frac{1}{1 + e^{-z_{ik}}}, \quad i = 1, \ldots, C \tag{4}$$

Here we use the $z_k = Wx_k$.

2.1.1 MSE based training

The expression in equation 3 can be simplified further by merging the input and matrix to $[x^T\, 1]^T \to x$ and $[W\, \omega_o]^T \to W$, the expression becomes $g = Wx$. To find good weights for W an optimization criteria is needed. In this project we used the Minimum square error (MSE) as our cost function.

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

Here t_k are one hot encoded 2.4 target vectors that shows what class a given input x_k is supposed to be. By optimizing the cost function, or training the model, for a given number of inputs and target vectors we find weights for W that gives the best performance for a given criteria. Note the MSE is one of many cost functions that could be used, and should be chosen based on a given problem. Using the chain rule on 5 we get:

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

where it can be shown that

$$\begin{aligned} \nabla_{g_k} MSE &= g_k - t_k \\ \nabla_{z_k} g_k &= g_k \circ (1 - g_k) \\ \nabla_W z_k &= x_k^T \end{aligned} \tag{7}$$

In order to reduce the MSE, W has to be moved iteratively in the opposite direction of the gradient which gives:

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

for iteration number m. The learning rate α is a tuning variable that must be chosen with care. A too large value will lead to fluctuations in MSE while a too small value will require an unnecessary amount of iterations to minimize the MSE.

2.2 Template based classifier

Template-based classifiers are a type of classification algorithm that uses pre-defined templates to classify new instances. These templates are created based on prior knowledge of the domain and can take many forms, such as rules, decision trees, or even complete feature sets. The basic idea behind template-based classification is to match the input instance to one of the pre-defined templates and use the corresponding label as the classification result. Three template-based classification methods used in this project are Nearest neighbor and K Nearest neighbor.

2.2.1 Nearest neighbor - NN

The nearest neighbor classifier is a type of classification algorithm that assigns the class label of the nearest data point in the training set to a new data point. It is a simple yet effective method for classification, but its performance heavily relies on the distance metric used to determine the nearest neighbors. The most common distance metric used is Euclidean distance, which measures the straight-line distance between two data points in the feature space. Mathematically, the Euclidean distance between two data points x and y can be represented as:

$$d_{Euclidean}(x, y) = (x - y)^T (x - y) \tag{9}$$

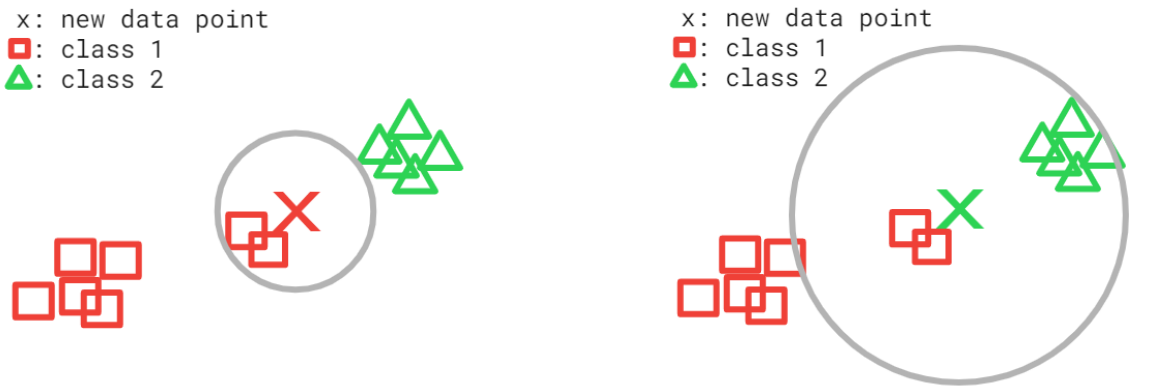
In nearest neighbor classification, the class label of a new data point is assigned to the class label of the nearest data point in the training set based on the Euclidean distance. This can be represented mathematically as:

$$y = \underset{j \in 1,\ldots,k}{\operatorname{argmin}} \; d_{Euclidean}(x, x_j) \tag{10}$$

where y is the predicted class label for the new data point, x_i is the j th data point in the training set, and k is the number of data points in the training set.

2.2.2 K Nearest neighbor - KNN

KNN is a classification algorithm similar to the NN classifier but instead of only looking at the actual nearest neighbor it is based on finding the k-nearest neighbors to a new data point. The algorithm then assigns the data point to the class label that is most frequent among its k-nearest neighbors illustrated in figure 1. This way of classifying is more robust to outliers but requires careful tuning of k to avoid ignoring local clusters.



(a) X classified as class 1 for NN or KNN, K=2

(b) X classified as class 2 for KNN, K=6

Figure 1: Comparison between NN and KNN

2.2.3 K-means clustering

K-means clustering is an algorithm used to divide the data set into clusters containing similar features. The algorithm begins by randomly selecting K initial centroids from the training dataset. Each data point is then assigned to the nearest centroid based on its Euclidean distance. The centroid is updated by computing the mean of all the data points assigned to it. This process is repeated until convergence, meaning that the assignment of data points to centroids no longer changes. Once the centroids are fixed, the classification of new data points is determined by assigning them to the nearest centroid. K-means is a simple and efficient algorithm, but it can be sensitive to the initial centroid selection and may converge to a suboptimal solution. The figure 2 from [2] illustrates the algorithm on a 2D dataset.

2.3 Confusion matrix and error rate

A confusion matrix lists the predicted labels and the true labels in a table. By counting how many of the predicted labels were put into a given class and comparing them to the true label we get an overview of how well we are able to separate between the classes. To investigate how well the classifier performed overall we can use the error rate, which measures how many predictions were wrong compared to the total number of predictions and can be read from a confusion matrix M as:

$$ERR_T = 1 - \frac{correct\,predictions}{total\,predictions} = 1 - \frac{Tr(M)}{\sum_{i=1}^{N}\sum_{i=1}^{N} M_{ij}} \tag{11}$$

2.4 One hot label encoding

In classification tasks, categorical variables often need to be represented as numerical values for use in machine learning algorithms such as LDC. One hot label encoding is a common technique for encoding categorical variables as binary vectors. For example, suppose we have a dataset containing the colors of fruits: red, green, and blue. We can assign a unique numerical code to each category, such as red = 1, green = 2, and blue = 3. Then, we can represent each category as a binary vector using one hot label encoding. Thus, red is represented as [1 0 0], green as [0 1 0], and blue as [0 0 1]. The norm of each vector is equal since the classes are nominal rather than ordinal (e.g., green is not "bigger" or "more" than blue or red). By using one hot label encoding, classification can effectively handle categorical variables and achieve accurate results.

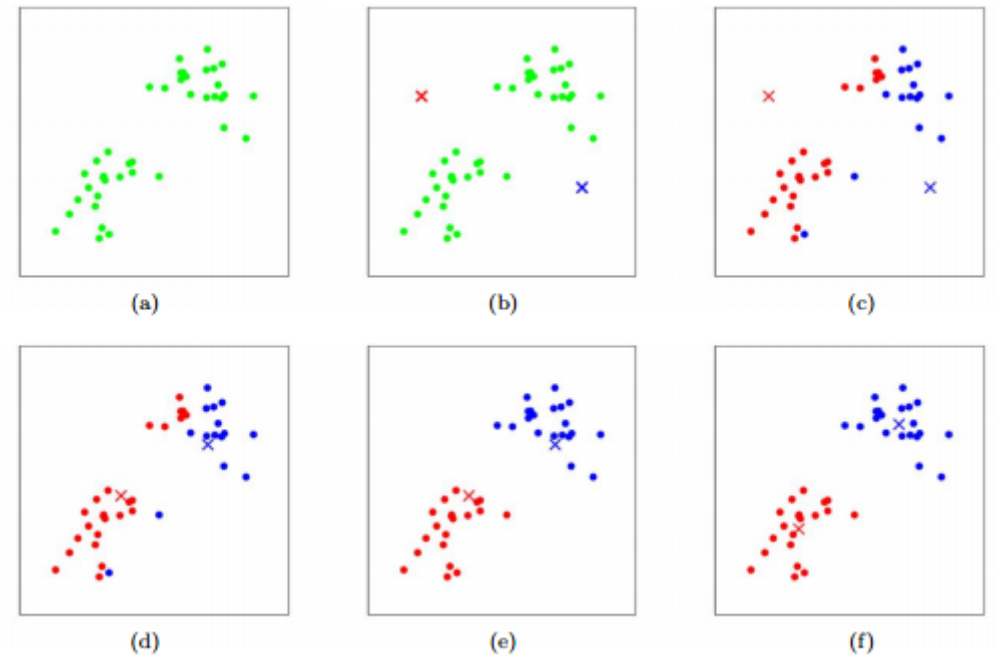


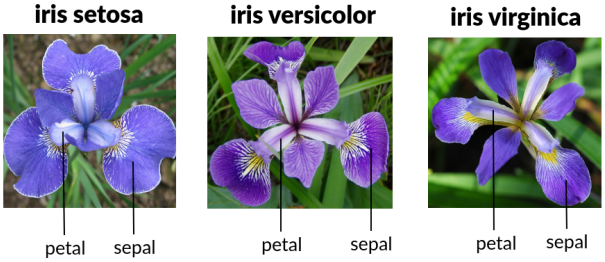
Figure 2: a) Initial dataset b) Select random centroids. c) Cluster dataset based on distance to selected centroids. d) Assign selcted centroid to the centre of nearest cluster. e) Repeat from c. f) Terminate when change in assigned cluster centres is small.

2.5 Normalizing data

Normalizing data is a critical step in classification as it ensures that features with different scales contribute equally to the decision-making process. It can be done in several ways, but a typical way is to divide the data by the highest value. Without normalization, features with larger scales may dominate the decision process, leading to inaccuracies in classification.

3 The task

The project is divided in to two separate parts. In part one, we use a linear discriminant classifier described in chapter 2.1 to classify Iris flowers based on the four features: *sepal length, sepal width, petal length and petal width.* By training the model on 30 labeled samples, the goal is to correctly classify an arbitrary new sample into one of three classes: *Setosa, Versicolour* and *Virginica.* We will also test how the model performed after removing different features.



SL	SW	PL	PW
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2

Figure 3: Raw Iris data

Figure 4: Pictures of the three Irises taken from [3] and the corresponding table of the features SL: Spetal Length, SW: Spetal Width, PL: Petal Length, PW: Petal Width.

The second part is to classify handwritten numbers by using a template based classifier described in 2.2. Using the MNIST dataset we investigate how a NN classifier performs when using the entire training set as labels, and compare it to when clustering is applied. We will also implement a KNN classifier to compare the performance with the NN classifier.

The MNIST database consists of 60000 training examples written by 250 different persons and 10000 test examples written by 250 other persons. The data is a picture of size 28x28 pixels and is represented by a gray scale from 0-255.

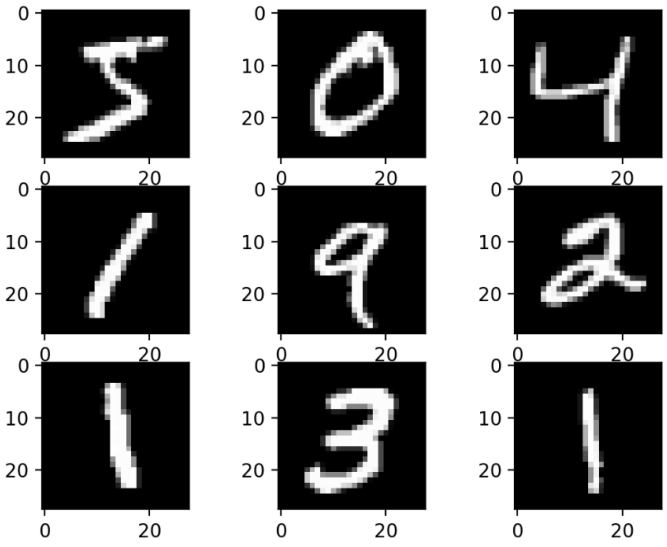


Figure 5: Examples of handwritten digits from MNIST database. 28x28 pixels, grayscale 0-255.

We decided to take on this task because it presents an intriguing and challenging opportunity, and it also shows how theory can be applied in practice. In addition, we anticipate that the experience and knowledge gained from this project will be useful for future undertakings.

4 Implementation and results

In this part of the report we will first describe the implementation of the Iris task and then handwritten digit task. Both tasks are implemented in Python due to many helpful libraries for the kind of data processing needed in these tasks.

4.1 Iris project

4.1.1 Splitting data into training and testing sets

To implement a LDC for the Iris task, we needed to select the number of training and test data. The results will vary depending on the number of test and train data one chooses. The given task specified a specific training and test division, where 30 data points were used for training and 20 for testing.

4.1.2 Normalizing the data

Once we had split the data into training and testing sets, we normalized it for the reasons explained in section 2.5. To accomplish this, we utilized the Numpy function .max() to identify the highest value for each of the four features in the training set, as displayed in table 1. Subsequently, we divided each feature value by its corresponding maximum value, effectively scaling the data. This normalization procedure enabled us to ensure that each feature contributed equally to the decision-making process and improved the performance of our models.

Septal Length	Septal Width	Petal Length	Petal Width
7.7	4.4	6.9	2.5

Table 1: Maximum features from training data

4.1.3 One hot encode target vectors

As discussed in section 2.4, we encoded the target vectors by creating three separate Numpy arrays. Each iris was assigned a unique encoding, such that Setosa was represented by the array $\begin{bmatrix}1 & 0 & 0\end{bmatrix}$, Versicolor by $\begin{bmatrix}0 & 1 & 0\end{bmatrix}$, and Virginica by $\begin{bmatrix}0 & 0 & 1\end{bmatrix}$.

4.1.4 Training the LDC

Once the data was properly processed, we began training the LDC by implementing the algorithm described in 2.1. Choosing the right learning rate and number of iterations was crucial to achieving optimal results. After some experimentation, we set the learning rate to $\alpha = 0.005$ and the number of iterations to 1000, at which point the Mean Squared Error (MSE) function converged, as depicted in Figure 7. In retrospect we could have used the convergence of MSE as a termination criteria by stopping the algorithm when the change in MSE is small instead of deciding on 1000 iterations. This would be useful if the training dataset was larger, but at the scale of this project it had little to no impact on runtime.

To initialize the weight matrix W, we created a 3x5 matrix, with 3 rows for the one-hot encoded labels and 5 columns for each of the features, plus an extra column for the bias. Then, we ran a loop to train the algorithm. Once training was complete, the weight matrix 12 was obtained, providing a basis for classification. The whole process is visualized in figure 6.

$$W = \begin{bmatrix} 0.35 & 1.34 & -2.00 & -0.92 & 0.24 \\ 1.02 & -1.99 & 0.02 & -1.01 & 0.51 \\ -1.95 & -1.53 & 2.80 & 2.24 & -0.94 \end{bmatrix} \tag{12}$$

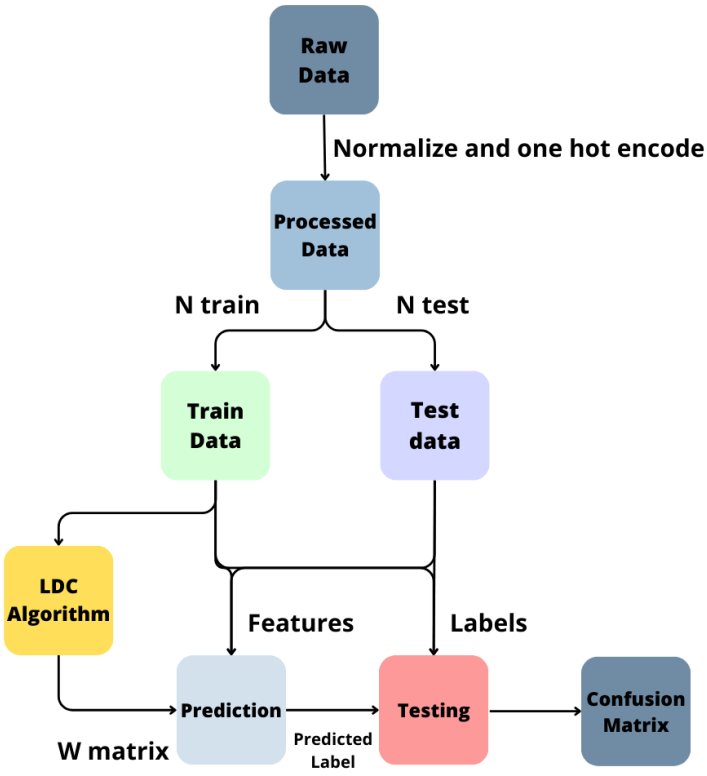


Figure 6: Iris classification process

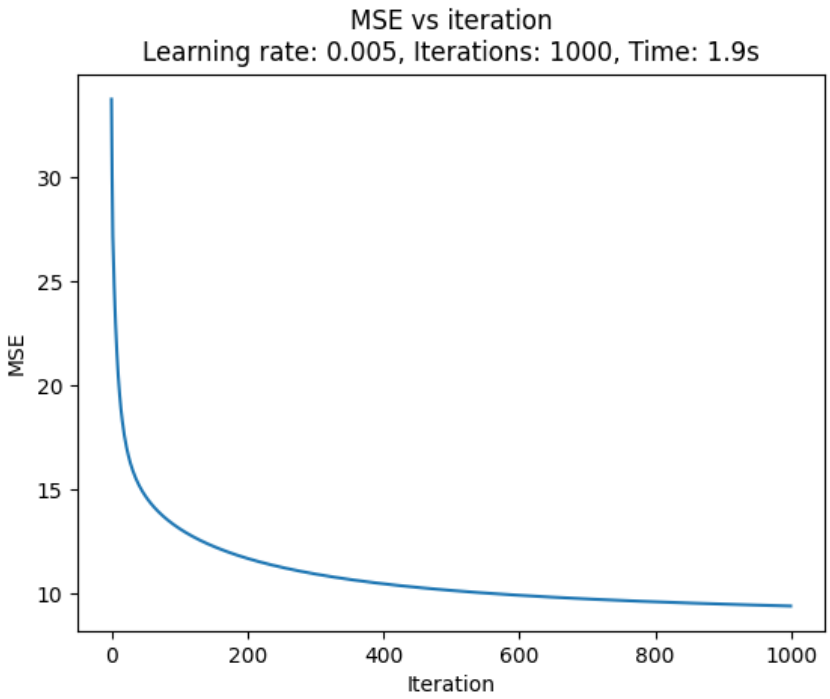
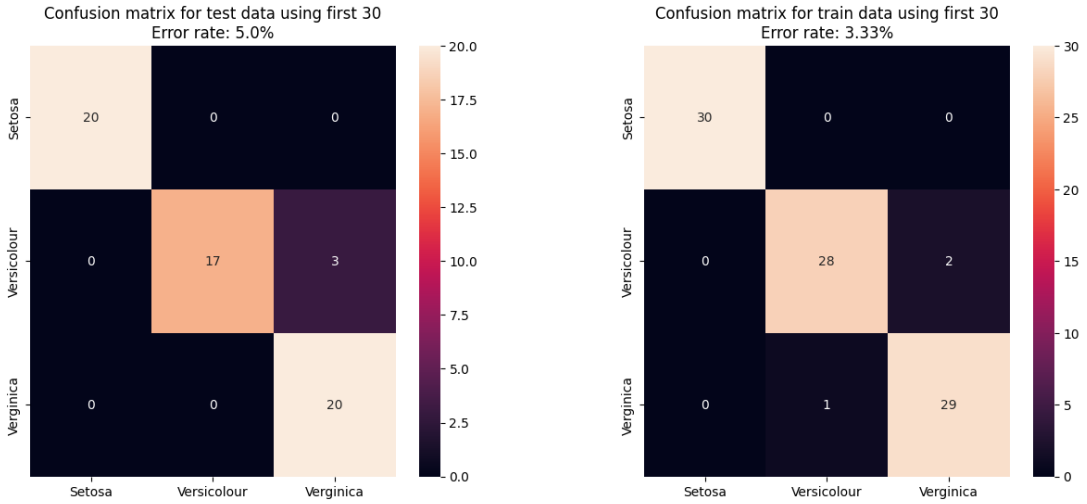


Figure 7: The convergence of the Minimum Square Error.

4.1.5 Results using first 30 for training and last 20 for testing

We assessed the performance of our classifier by applying the LDC output function (refer to 4) to both the training and testing datasets, and compared the output with the true one-hot encoded labels. The resulting confusion matrices and error rates were then calculated to evaluate the accuracy of the classifier, as shown in 8.

The algorithm performed slightly worse on the test data compared to the training data, although this could be coincidental since the opposite was true for the test explained in 4.1.6. It is worth noting that our dataset is relatively small, which makes it difficult to draw conclusions about the performance of the LDC. However, we can confirm that the model effectively classifies the irises, as evidenced by the general low error rate of around 5%.



(a) Test data.

(b) Train data.

Figure 8: Confusion matrix for training on first 30 data.

4.1.6 Results using last 30 for training and first 20 for testing

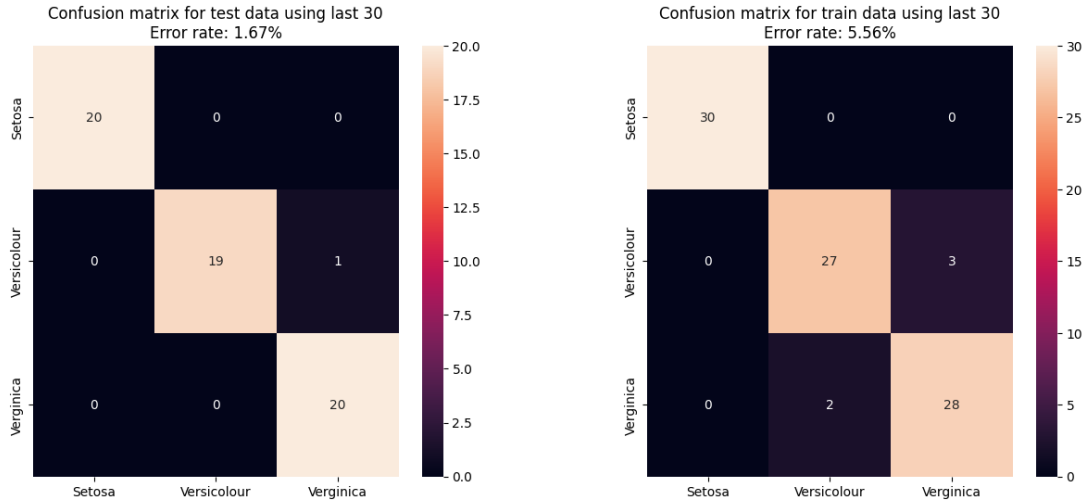
To investigate how the chosen training and test data affected the performance of our classifier we modified to dataset by using the last 30 data points as the training set, and the first 20 as the test set. As a result, the W matrix and resulting confusion matrices differed from those of the previous test, as shown in 13 and 9.

The model's performance varied from the previous test, which suggests that some data points in the dataset may be challenging to classify, particularly in the region where the dataset was split. This can create problems if these difficult data points end up in the training set as the LDC will not be able to classify them correctly. To handle these outliers, it might be better to use a non-linear classifier that better fits the data.

$$W = \begin{bmatrix} 0.40 & 1.33 & -2.09 & -0.96 & 0.25 \\ 0.45 & -1.61 & 0.71 & -1.75 & 1.02 \\ -1.59 & -1.70 & 2.41 & 2.61 & -1.17 \end{bmatrix} \tag{13}$$

4.1.7 Features analysis

In machine learning, selecting appropriate features is crucial for the success of the model. Certain features may be more effective than others in distinguishing between classes, while



(a) Test data.

(b) Train data.

Figure 9: Confusion matrix for training on last 30 data.

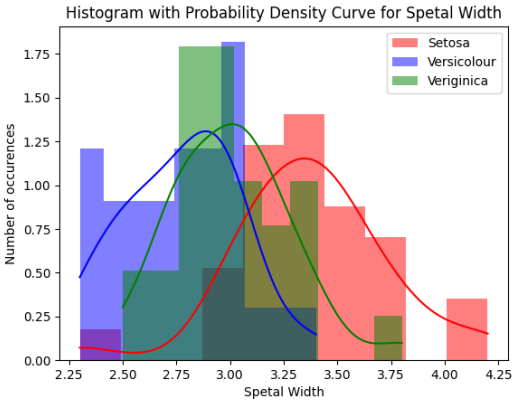
Using 30 first data	Septal Width	Septal Width	Petal Length	Petal Width	Total Error rate
True	x	x	x	x	4.17%
False	x	x	x	x	3.33%
True		x	x	x	4.17%
True			x	x	11.17%
True				x	13.61%

Table 2: Results from Iris classification for a given set of features. The total error rate includes both the test and training set

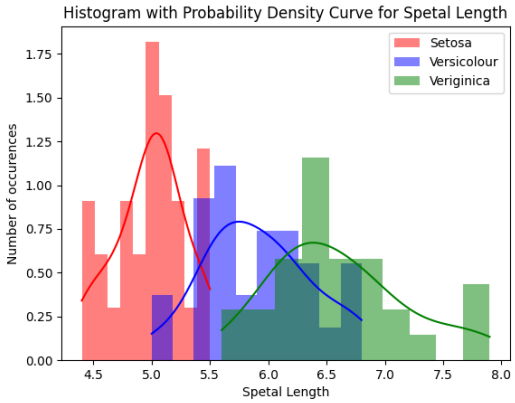
others may not contribute to improving the classifier's accuracy. To determine which features to include when training a LDC, we employed a feature analysis approach. Firstly, we utilized the Numpy library's .hist function to generate histograms for each feature. Subsequently, we identified the feature with the most overlap between classes, and eliminated it from the feature set. This process was repeated iteratively for each feature until we arrived at only one feature for our LDC. The decision to determine which feature to remove was based solely on our ability to visually identify overlap in the histograms.

As depicted in figure 10, there is significant overlap among all classes with respect to the spetal width feature. Thus, excluding it from the training set has minimal impact on the overall performance, given that this feature alone cannot linearly separate the classes. This is evident from the similarity in confusion matrices a and b shown in figure 11.

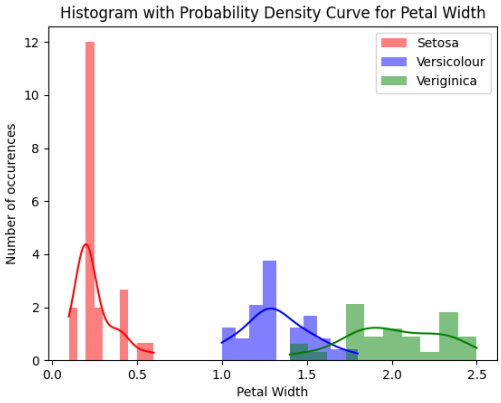
When both spetal width and spetal length were removed, the error rate doubled. This was expected since the LDC had fewer features to work with. However, it was surprising that the model performed better with three features removed compared to two. It is unclear if this trend holds for larger datasets since it is generally better to use two features than one. Looking at the table 2 we can see that if the training data is included when computing the error rate, the model performs better when using two features compared to one, meaning matrix d in figure 11 probably was a lucky coincidence. Nevertheless, it can be concluded that the strong correlation between petal length and what class a given iris belongs to enables a decent LDC to be created by using this as the sole feature.



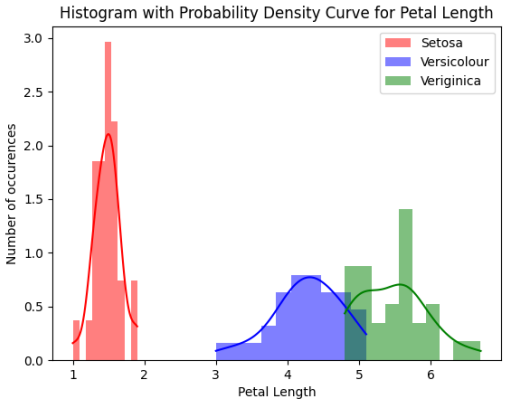
(a) Spetal Width



(b) Spetal Length



(c) Petal Width



(d) Petal Length

Figure 10: Histograms for each feature

4.2 Handwritten digits

The classification of handwritten digits was implemented in two parts. First, we used the nearest neighbor algorithm on the whole data set. In the second part, we introduced clustering and the K-nearest neighbor algorithm on the clustered data.

4.2.1 Splitting the data into training and testing sets

As discussed in 3, we utilized the MNIST database imported from the Python library keras.datasets which consists of images of handwritten digits. The database includes 60,000 training images, each with a resolution of 28x28 pixels, authored by 250 individuals, along with an additional 10,000 test images written by a separate set of 250 individuals. Each image is labeled with the corresponding digit it represents. We stored the data in four Numpy arrays.

4.2.2 Normalizing the data

The data is gray scaled with pixel values ranging from 0-255. As explained in the theory 2.5 we need to normalize this data by dividing every training and testing data value by 255.

4.3 Classification by Nearest Neighbor

To implement the Nearest Neighbor classifier described in 2.2.1, we created a Numpy array called "distances" and looped over every image in the training data for every image in the test data. During each iteration, we calculated the Euclidean distance between the test data and the current training data and appended it to the distances array. Once the loop finished, we classified the test data with the same label as its nearest neighbor. We used the Numpy function .argmin on the distances array to find the shortest distance.

As can be oberved from figure 12, this approach yielded a low error rate of only 3.0%, however this algorithm is somewhat dumb. By just calculating all the distances from a given test image to all the templates we end up having to perform 60,000*10,000 calculations. Since we are running the program on our personal laptops this leads to a runtime of over one hour. It does not help that Python is not a particularly fast language either. Despite its drawbacks, this is the best result we were able to achieve with minimal misclassification of images.

Figure 13 shows some correct and incorrect predictions by the NN classifier. As mentioned, this approach gets most images correct however there are some that are difficult to explain why it got wrong. It is for example fairly easy for humans to see that the second to last number is supposed to be an 8, and not a 5. This indicates that there are no 8's in the training data that are more similar to that particular image than there are 5's. In other cases like the last picture we can give the model more slack, as it is understandable that the given image could be confusing, although most humans would get that right also. An interesting takeaway from figure 12 is that it is worst at classifying 9's, mostly confusing it with 4's. This is probably because there is little that separates the actual handwritten digits from each other, making it more prone to errors.

4.4 Implementation of clustering

We optimized efficiency and computation time by implementing the K-Means clustering algorithm described in 2.2.3. Instead of calculating distances between a test image and all 60,000 training images, we created 64 clusters for each class and used them as our templates. Clustering is an unsupervised learning algorithm, which can be used on data without any prior knowledge. Since we have access to the labels of all the training data our approach falls under supervised learning, where we use the true labels to also label the clusters.

To cluster the training data, we employed the KMeans function from the sklearn.cluster library, which follows the algorithm described in section 2.2.3. Processing time may vary depending on the size of the training data and number of clusters selected. In our case, it took approximately two minutes. Figure 14 shows ten of the resulting clusters for each digit, which represents an average way of writing. For instance, one can observe that some participants in the MNIST database prefer to write a seven with a line through it while others don't. Despite reducing the templates from 60,000 to 640, the classification still performs well, as elaborated in the following chapter.

4.5 Classification by NN and K-NN on clustered data

By applying the same NN algorithm as in 4.3 on our clustered data we were able to reduce the number of computations needed drastically. This did yield a significantly shorter runtime, while not adding much to the error rate which in this case is 4.61%. This makes the nearest neighbor algorithm much more applicable in for example real time systems when clustering is used.

We can see from figure 15 that the results from this test is quite similar to using NN without clustering. The biggest confusion is between 9's and 4's and also some confusion between 5's and 3's. This is expected as there are few lines that separates these digits from each other.

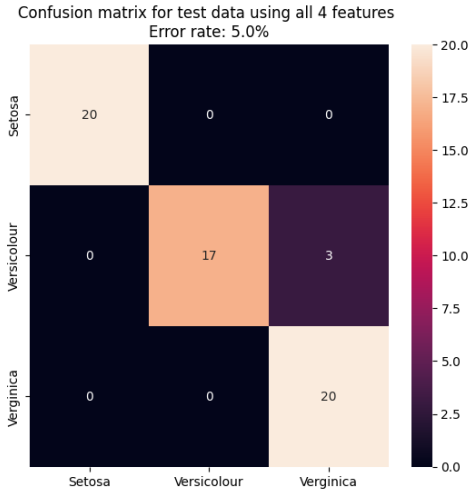
Clustering	K	Error	Classification time(h:m:s)
NO	1	3.0%	01:14:05
YES	1	4.61%	00:02:02
YES	3	5.71%	00:01:44
YES	7	6.49%	00:01:43

Table 3: Errors, and runtime of digit classification

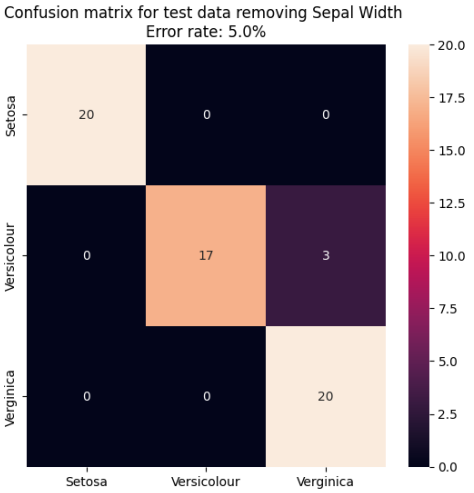
For future investigation, it would be interesting to see how using cluster means as templates compares to just selecting the same amount of templates at random.

To implement a K-Nearest Neighbor algorithm described 2.2.2 on the clustered data, we start off as with the NN case by calculating the Euclidean distance between the test data and the cluster means. Instead of using the .argmin function, we use .argsort to sort the indices of the distances. Afterward, we select the K closest clusters, where we chose K to be 3 and 7 in our implementation. We then determine which cluster index appears most frequently among the K closest. Finally, we assign the label of the most frequently appearing cluster to the test data. The resulting confusion matrix for KNN with clustering is shown in figure 16.

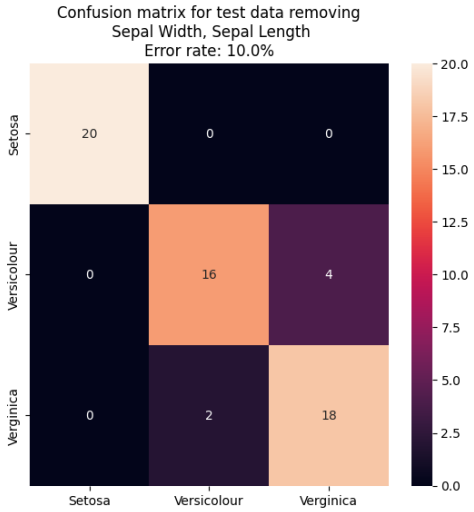
The performance of KNN with 3 and 7 neighbors on the clustered data is worse than when only using the nearest neighbor. Specifically, the error rate is 5.71% and 6.49% for K=3 and K=7, respectively. This was a bit surprising as we expected this algorithm to be more robust than just using NN. A possible explanation for this can be found by examining figure 14. We can see that there are similarities between 4's and 9's, and 3's and 5's. KNN aims to find the K nearest clusters of digits, but sometimes clusters representing different digits may be too close to each other, leading to mixed-up classifications. Interestingly, we observe that the error rate decreases when using fewer neighbors. In our case the error rate drops when reducing K, to a minimum of K=1 (NN with clustering). This suggests that for this type of classification, a larger number of clusters may be needed if we want to improve on the KNN algorithm.



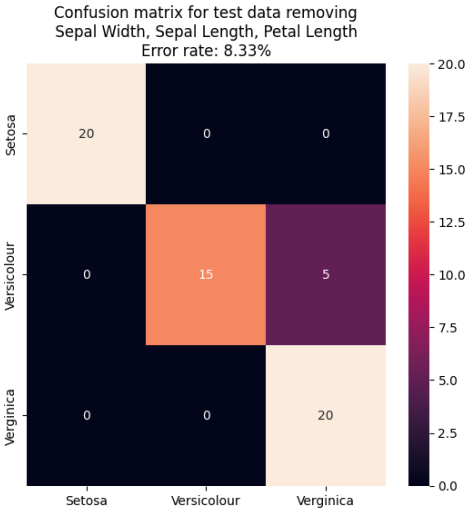
(a) 4 features



(b) 3 features, removed Spetal Width



(c) 2 features, removed Spetal Width, Length



(d) 1 features, removed Spetal Width, Length and Petal Length

Figure 11: Confusion matrices for removing features.

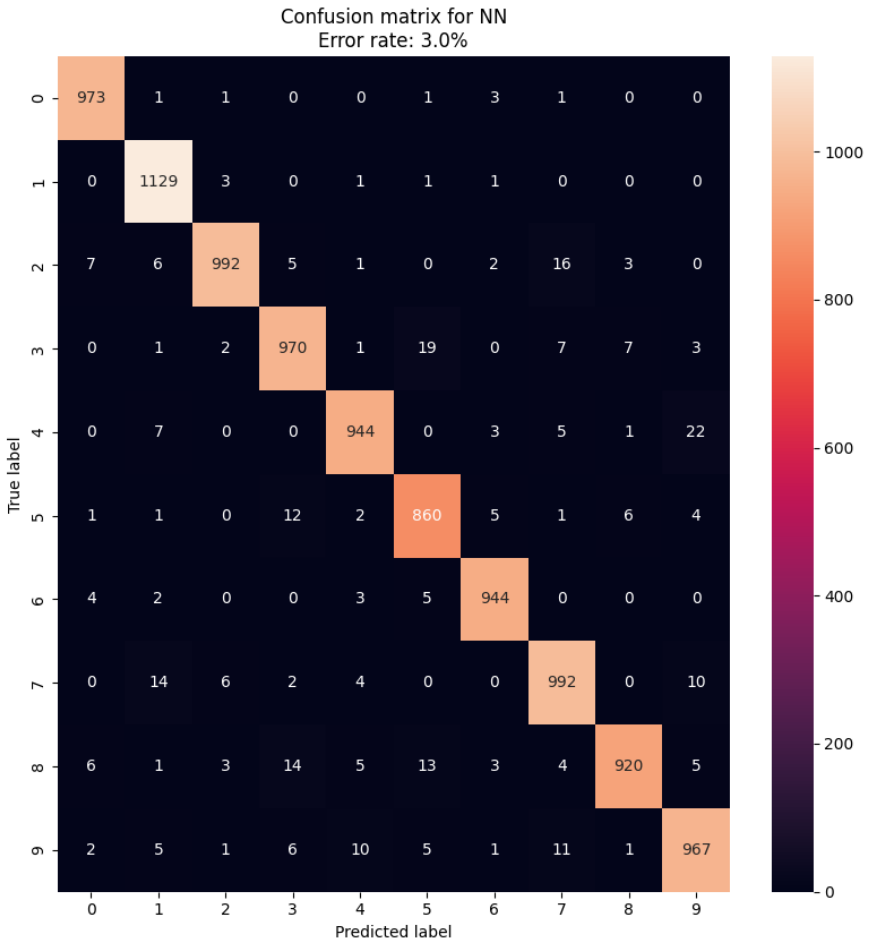


Figure 12: Confusion matrix using the whole training data as templates

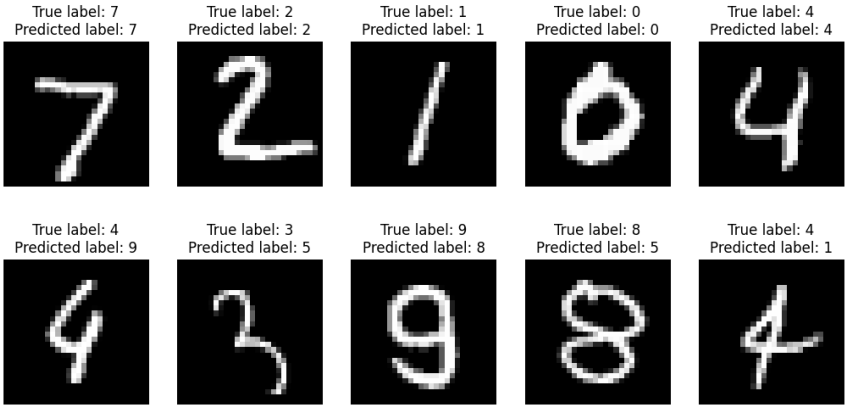


Figure 13: Some correctly and incorrectly classified images using NN

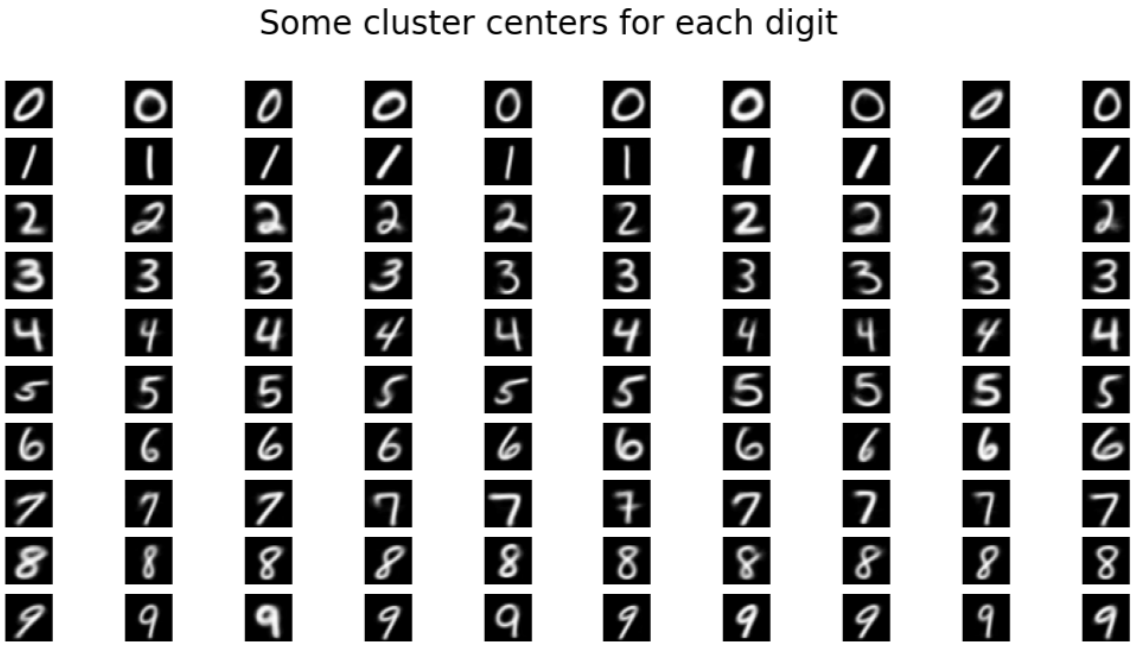


Figure 14: 60.000 training images converted into 640 clusters, here are some examples.

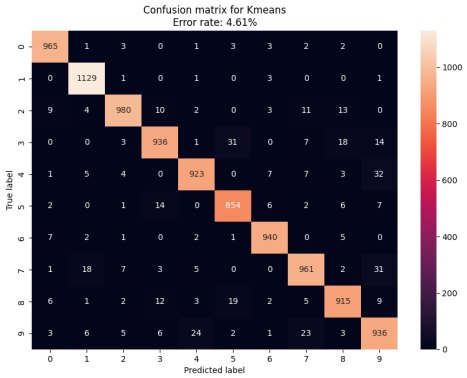
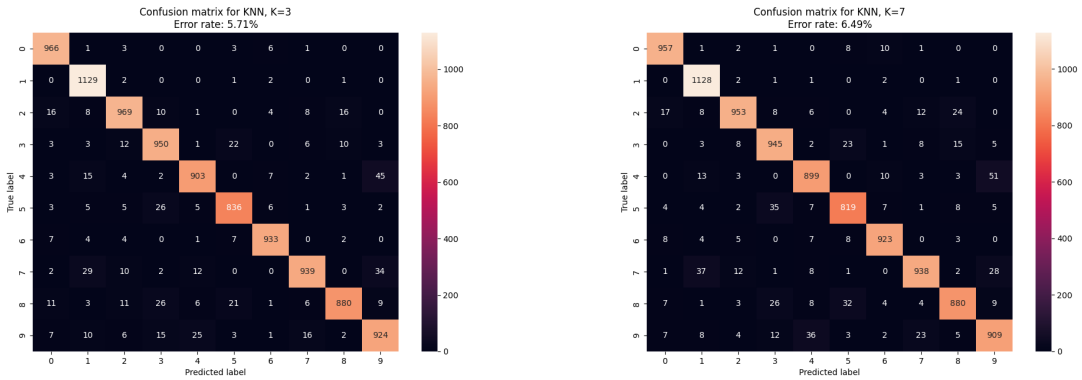


Figure 15: NN with clustering.



(a) K=3

(b) K=7

Figure 16: Confusion matrix KNN with clustering with different number K.

5 Conclusion

In the first part of the project, we implemented a Linear Discriminant Classifier to classify iris flowers based on their features. The LDC's error rate is reliant on multiple factors, including the learning rate, the number of iterations, and the selected features used for training. We found that employing all four features resulted in an error rate of 5.0%, while using only one feature led to an error rate of 8.33%. While the achieved error rate is acceptable, there is still room for improvement. Performing feature analysis, such as plotting histograms and identifying features with minimal overlap between classes, can aid in identifying the optimal features. A larger dataset may also improve the accuracy of the classifier, as we observed a difference in training on the first 30 versus the last 30 data samples. This suggests that we may have employed too few data samples, or that may have been some outliers that were harder to classify. In addition to the other factors, the choice of distance calculation method is crucial in determining the effectiveness of our approach. Although we used the Euclidean distance metric in this project, we could potentially achieve even better results by exploring alternative distance measures that take into account additional factors, such as variance.

In the second part, we classified handwritten digits using the nearest neighbor algorithm, achieving an error rate of 3.0%. However, this method required a high computational time of one hour. We found that by applying clustering, we could reduce the number of computations needed drastically while not sacrificing too much on the error rate of 4.61%. This will make the algorithm more applicable in real time systems, and when working on larger datasets, clustering can be done in advance meaning more time can be spent on testing which classification algorithm to apply on the clustered data. In our results, we found that using KNN performed worse than NN and conclude that using the nearest neighbor will be sufficient for this problem. For future investigation, it would be interesting to see if clustering is even needed for the Digits problem as the true labels of the training data are known. This means that it would be possible to pick some representative templates for each digit without having to apply the clustering algorithm.

This project has been a good introduction to using computer algorithms to solve classification problems. The algorithms implemented might not be the most advanced, but by programming most of the logic from scratch we got a great understanding of how the theory can be implemented in practice. The project has thus given us a great foundation for future work on classification problems.

References

[1] Magne H. Johnsen Tor A. Myrvoll, Stefan Werner. Estimation, detection and classification theory. Blackboard. Accessed: 2023-04-23.

[2] Chris Piech. K-means clustering. https://stanford.edu/~cpiech/cs221/handouts/kmeans.html, 2016. Accessed: April 23, 2023.

[3] Tavish Agarwal. Iris flowers classification using machine learning. https://www.analyticsvidhya.com/blog/2022/06/iris-flowers-classification-using-machine-learning/, 2022. Accessed: April 23, 2023.

A Iris Appendix

Attached below is the Python code for the iris task. It contains two files:
Iris_main.py
Iris_functions.py

Iris\Iris_main.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from Iris_functions import *
import time

# Parameters
D = 4                               # Number of features
C = 3                               # Number of classes
N_train = 30                        # Number of training data
N_test  = 20                        # Number of test data
first_30_to_train = True            # Use first 30 data points for training and last 20 for
testing
disabled_features = []              # Which features to disable
D = D - len(disabled_features)      # Update D

# Plot parameters
visualize_histogram = True         # Plot histograms of the data
visualize_confusion_matrix = False  # Plot confusion matrix
visualize_MSE = False               # Plot MSE vs iteration

# Load seperate iris data
setosa      = pd.read_csv('Iris_raw_data/class_1.csv', header=0)
versicolour = pd.read_csv('Iris_raw_data/class_2.csv', header=0)
verginica   = pd.read_csv('Iris_raw_data/class_3.csv', header=0)

# Remove unwanted features
setosa      = remove_features(setosa, disabled_features)
versicolour = remove_features(versicolour, disabled_features)
verginica   = remove_features(verginica, disabled_features)

# Create training and test data
train_data, test_data = create_train_test_data(setosa, versicolour, verginica, N_train, N_test,
first_30_to_train)

# Normalizing the data
max_features_val = np.array([train_data[:,i].max() for i in range(D)])
normal_train_data = train_data/max_features_val

# Save max values to file
np.savetxt("Iris_max_values.txt", max_features_val)

# Target vectors one hot encoded
t1 = np.array([1, 0, 0])
t2 = np.array([0, 1, 0])
t3 = np.array([0, 0, 1])
label_train = np.vstack((np.tile(t1, (N_train, 1)), np.tile(t2, (N_train, 1)), np.tile(t3,
(N_train, 1))))
label_test = np.vstack((np.tile(t1, (N_test, 1)), np.tile(t2, (N_test, 1)), np.tile(t3, (N_test,
1))))

# Training the LDC
W = np.zeros((C, D+1))   # Initial Weights
```

```
training = True
iterations = 1000
learning_rate = 0.005

MSE_list = []
print("\nStarting training")
start_time = time.time()

for i in range(iterations):
    grad_W_MSE = 0
    MSE = 0
    for i in range(C*N_train):
        # Using 3.2 in compendium
        x_k  = np.array(train_data[i, :])   # Get data point
        x_k = np.append(x_k, 1)             # Add bias
        z_k = W @ x_k                       # Calculate weighted sum
        g_k = sigmoid(z_k)                  # Activation function
        t_k = label_train[i, :]             # Get data point label
        grad_W_MSE += grad_W_MSE_func(g_k, t_k, x_k, D)
        MSE += 0.5*(g_k-t_k).T @ (g_k-t_k)

    MSE_list.append(MSE)
    W = W - learning_rate*grad_W_MSE

end_time = time.time()
elapsed_time = round(end_time - start_time, 2)
print("Training time: ", elapsed_time, "s")
print("Training done\n")

# Set print options
np.set_printoptions(precision=2, suppress=True)
print("Weights:")
print(W)

# Plot functions
if visualize_MSE:
    plt.plot(MSE_list)
    plt.title("MSE vs iteration\nLearning rate: " + str(learning_rate) + ", Iterations: " +
str(iterations) + ", Time: " + str(elapsed_time) + "s")
    plt.xlabel("Iteration")
    plt.ylabel("MSE")
    plt.show()

# Find confusion matrix for training data
confusion_matrix_train = np.zeros((C, C))
for i in range(C*N_train):
    x_k  = np.array(train_data[i, :])
    x_k = np.append(x_k, 1)
    z_k = W @ x_k
    g_k = sigmoid(z_k)
    t_k = label_train[i, :]
    if np.argmax(g_k) == np.argmax(t_k):
        confusion_matrix_train[np.argmax(t_k), np.argmax(t_k)] += 1
    else:
        confusion_matrix_train[np.argmax(t_k), np.argmax(g_k)] += 1
```

```
print("Confusion matrix for training data: ")
print(confusion_matrix_train)

# Find confusion matrix for test data
confusion_matrix_test = np.zeros((C, C))
for i in range(C*N_test):
    x_k  = np.array(test_data[i, :])
    x_k = np.append(x_k, 1)
    z_k = W @ x_k
    g_k = sigmoid(z_k)
    t_k = label_test[i, :]
    if np.argmax(g_k) == np.argmax(t_k):
        confusion_matrix_test[np.argmax(t_k), np.argmax(t_k)] += 1
    else:
        confusion_matrix_test[np.argmax(t_k), np.argmax(g_k)] += 1

print("Confusion matrix for test data:")
print(confusion_matrix_test)

# Print accuracy for traing and test data
accuracy_train = np.sum(np.diag(confusion_matrix_train))/np.sum(confusion_matrix_train)
accuracy_test = np.sum(np.diag(confusion_matrix_test))/np.sum(confusion_matrix_test)
print_accuracy_for_confusion_matrix(confusion_matrix_train,"training")
print_accuracy_for_confusion_matrix(confusion_matrix_test,"test")

# END OF TASK 1
#-------------------------------------------------------------------------------------------------
---------------------
# START OF TASK 2

# Plotting confusion matrices for training and test data
error_rate_train_percent = round((1 - accuracy_train)*100, 2)
error_rate_test_percent = round((1 - accuracy_test)*100, 2)

label_names = ["Setosa", "Versicolour", "Verginica"]
if visualize_confusion_matrix:
    plot_confusion_matrix(confusion_matrix_train,confusion_matrix_test, label_names,
first_30_to_train,error_rate_train_percent,error_rate_test_percent)

if visualize_histogram and D == 4:
    #Plot 3 histograms for feature x for all classes
    plot_histograms(train_data,N_train)

# Show all figures
plt.show()
```

Iris\Iris_functions.py

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from Iris_functions import *
import seaborn as sns
from scipy.stats import gaussian_kde

# Data processing and loading functions
def remove_features(data, disabled_features):
    data.columns = data.columns.str.strip()
    data = data.drop(columns=disabled_features)
    return data

def create_train_test_data(setosa, versicolour, verginica, N_train, N_test, first_30_to_train):
    if first_30_to_train:
        train_data = pd.concat([setosa[:N_train], versicolour[:N_train], verginica[:N_train]])
        test_data = pd.concat([setosa[N_train:N_train+N_test],
versicolour[N_train:N_train+N_test], verginica[N_train:N_train+N_test]])
        train_data = train_data.values
        test_data = test_data.values
    else:
        test_data = pd.concat([setosa[:N_test], versicolour[:N_test], verginica[:N_test]])
        train_data = pd.concat([setosa[N_test:N_test+N_train],
versicolour[N_test:N_test+N_train], verginica[N_test:N_test+N_train]])
        train_data = train_data.values
        test_data = test_data.values
    return train_data, test_data

# Functoins for training algorithm 3.1 in compendium
def sigmoid(x):
    return np.array(1 / (1 + np.exp(-x)))

def grad_W_MSE_func(g_k, t_k, x_k, D):
    A = (g_k - t_k)*g_k*(1-g_k)
    A = A.reshape(3, 1)
    B = x_k
    B = B.reshape(D+1, 1)
    return A @ B.T


# Plotting functions
def plot_MSE(MSE_list):
    plt.figure()
    plt.plot(MSE_list)
    plt.xlabel("Iteration")
    plt.ylabel("MSE")
    plt.title("MSE vs iteration")
    plt.show()

def print_accuracy_for_confusion_matrix(confusion_matrix, label_names):
    accuracy = round(np.trace(confusion_matrix)/np.sum(confusion_matrix),4)
    print("Accuracy for", label_names,"data:", accuracy)
    print("Error rate for",label_names,"data:", round(1-accuracy,4))
```

```
def plot_confusion_matrix(confusion_matrix_train,confusion_matrix_test, label_names,
first_30_to_train, error_rate_train, error_rate_test):
    class_labels = label_names
    df_cm_test = pd.DataFrame(confusion_matrix_test, index = [i for i in class_labels], columns =
[i for i in class_labels])
    plt.figure(figsize = (10,7))

    #Confusion matrix for test data
    if first_30_to_train:
        plt.title("Confusion matrix for test data using first 30\n" + "Error rate: " +
str(error_rate_test) + "%")
    else:
        plt.title("Confusion matrix for test data using last 30\n" + "Error rate: " +
str(error_rate_test) + "%")

    sns.heatmap(df_cm_test, annot=True)

    # Confusion matrix for train data
    df_cm_train = pd.DataFrame(confusion_matrix_train, index = [i for i in class_labels], columns
= [i for i in class_labels])
    plt.figure(figsize = (10,7))
    if first_30_to_train:
        plt.title("Confusion matrix for train data using first 30\n" + "Error rate: " +
str(error_rate_train) + "%")
    else:
        plt.title("Confusion matrix for train data using last 30\n" + "Error rate: " +
str(error_rate_train) + "%")
    sns.heatmap(df_cm_train, annot=True)

def plot_histograms(train_data, N_train):
    #Extract features from training data
    feature_1_class_1 = np.array(train_data[:N_train, 0])
    feature_2_class_1 = np.array(train_data[:N_train, 1])
    feature_3_class_1 = np.array(train_data[:N_train, 2])
    feature_4_class_1 = np.array(train_data[:N_train, 3])

    feature_1_class_2 = np.array(train_data[N_train:2*N_train, 0])
    feature_2_class_2 = np.array(train_data[N_train:2*N_train, 1])
    feature_3_class_2 = np.array(train_data[N_train:2*N_train, 2])
    feature_4_class_2 = np.array(train_data[N_train:2*N_train, 3])

    feature_1_class_3 = np.array(train_data[2*N_train:3*N_train, 0])
    feature_2_class_3 = np.array(train_data[2*N_train:3*N_train, 1])
    feature_3_class_3 = np.array(train_data[2*N_train:3*N_train, 2])
    feature_4_class_3 = np.array(train_data[2*N_train:3*N_train, 3])

    feature_plot_1 = [feature_1_class_1, feature_1_class_2, feature_1_class_3]
    feature_plot_2 = [feature_2_class_1, feature_2_class_2, feature_2_class_3]
    feature_plot_3 = [feature_3_class_1, feature_3_class_2, feature_3_class_3]
    feature_plot_4 = [feature_4_class_1, feature_4_class_2, feature_4_class_3]
    feature_plot_1_text = ['Setosa', 'Versicolour', 'Verginica']
    feature_plot_2_text = ['Setosa', 'Versicolour', 'Verginica']
    feature_plot_3_text = ['Setosa', 'Versicolour', 'Verginica']
    feature_plot_4_text = ['Setosa', 'Versicolour', 'Verginica']

    #Define the features and their corresponding labels
    features = [feature_plot_1, feature_plot_2, feature_plot_3, feature_plot_4]
```

```
    feature_labels = [feature_plot_1_text, feature_plot_2_text, feature_plot_3_text,
feature_plot_4_text]
    #Make list of color for each class to use in plots
    colors = ['red', 'blue', 'green']
    x_lable = ['Spetal Length', 'Spetal Width', 'Petal Length', 'Petal Width']
    #Loop through each feature
    for i, feature in enumerate(features):
        #Create a new figure for each feature
        plt.figure()

        #Loop through each class and plot histogram with probability density curve
        for j, data in enumerate(feature):
            plt.hist(data, density=True, alpha=0.5, label=feature_labels[i][j], color=colors[j])

            #Plot probability density curve
            kde = gaussian_kde(data)
            x_vals = np.linspace(min(data), max(data), 100)
            plt.plot(x_vals, kde(x_vals), color=colors[j])

        plt.title('Histogram with Probability Density Curve for ' + x_lable[i])
        plt.xlabel(x_lable[i])
        plt.ylabel('Number of occurences')
        plt.legend()
```

B Digits Appendix

Attached below is the Python code for the digit task. It contains two files:
Digits_main.py
Digits_functions.py

Digits\Digits_main.py

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from Digits_functions import *
from sklearn.cluster import KMeans
import time

# Print np array nicely
np.set_printoptions(precision=3, suppress=True)

# Parameters
N_train = 60000                  # Number of training samples
N_test  = 10000                  # Number of test samples
C = 10                           # Number of classes
K_neighbors = 7                  # Number of nearest neighbors
M_clusters  = 64                 # Number of clusters
N_pixels    = 784                # Number of pixels in image

# Classification methods
NN_classification      = False    # Use the nearest neighbor classifier
Kmeans_classification  = False    # Use NN with k-means clustering classifier
KNN_classification     = False    # Use k-nearest neighbor classifier with k-means clustering

# Plot parameters
visualize_confusion_matrix = True  # Visualize confusion images
visualize_NN_comparison    = True  # Visualize nearest neighbor comparison test, prediction
N_Comparisons = 5                  # Number of comparisons to visualize

# Load MNIST hand written digit data
(train_data, train_label), (test_data, test_label) = mnist.load_data()

# Select only N_train and N_test data and labels
train_data  = train_data[:N_train]
train_label = train_label[:N_train]
test_data   = test_data[:N_test]
test_label  = test_label[:N_test]

# Normalize data so grayscale values are between 0 and 1
train_data = train_data / 255
test_data  = test_data / 255

# Classify test data with nearest neighbor classifier ----------------------------------------
------------------------
if NN_classification:
    print("NN classification")

    print("Start training")
    time_start = time.time()

    classified_labels = []
    correct_labels_indexes = []
    failed_labels_indexes = []
```

```
    # Calculate distances for each test data to each training data
    for i in range(N_test):
        # Get test image
        test_image = test_data[i]

        distances = []
        for j in range(N_train):
            train_image = train_data[j]
            distance = euclidean_distance(test_image, train_image, N_pixels)
            distances.append(distance)

        # Find label with smallest distance
        closest_test_data_index = np.argmin(distances)
        label = train_label[closest_test_data_index]

        if label == test_label[i]:
            correct_labels_indexes.append(i)
        else:
            failed_labels_indexes.append(i)
        classified_labels.append(label)

    # Print training time
    time_end = time.time()
    training_time = int(time_end - time_start)
    print_time(time_start, time_end)

    # Find confusion matrix
    confusion_matrix = confusion_matrix_func(classified_labels, test_label, C)
    print("Confusion matrix: ")
    print(confusion_matrix)

    # Print error rate
    error_rate = error_rate_func(confusion_matrix)
    print("Error rate: ", error_rate*100, "%")

    # Save confusion matrix to file
    save_to_file("NN/CM_NN_", confusion_matrix, error_rate, N_train, N_test, training_time)

    # Visualize confusion matrix
    plot_confusion_matrix("NN", confusion_matrix, error_rate, visualize_confusion_matrix)

    # Visualize nearest neighbor comparison test, prediction
    plot_NN_comparison(test_data, test_label, classified_labels, correct_labels_indexes,
failed_labels_indexes, N_Comparisons, visualize_NN_comparison)

# Classify test data with Kmeans classifier-----------------------------------------------------
-------------------------------------------------------------
if Kmeans_classification:
    print("K-means classification")

    print("Start training")
    time_start = time.time()

    # Perform k-means clustering on training data
    start_training = False
    if start_training:
```

```
        # Create 64 clusters for each unique label from training data
        kmeans_centers = np.empty((0, N_pixels))
        cluster_labels = np.empty((0, 1))

        for i in range(C):
            # Get indices for label i
            label_indices = np.where(train_label == i)[0]
            # Get data for label i
            label_data = train_data[label_indices]
            # Perform k-means clustering on label data
            kmeans = KMeans(n_clusters=M_clusters,
random_state=0).fit(label_data.reshape(len(label_indices), N_pixels))
            # Store cluster centers
            kmeans_centers = np.append(kmeans_centers, kmeans.cluster_centers_, axis=0)
            # Append M_clusters cluster labels to cluster_labels
            cluster_labels = np.append(cluster_labels, np.full((M_clusters, 1), i), axis=0)

        #Store cluster labels for training data and cluster centers in a file in a folder called
"kmeans_trained"
        np.savetxt("kmeans_trained/cluster_labels.txt", cluster_labels, fmt="%d")
        np.savetxt("kmeans_trained/cluster_centers.txt", kmeans_centers, fmt="%f")

    # Load cluster labels and cluster centers from file
    cluster_labels = np.loadtxt("kmeans_trained/cluster_labels.txt", dtype=int)
    kmeans_centers = np.loadtxt("kmeans_trained/cluster_centers.txt", dtype=float)

    # Put 10 clusters from each label in a np array
    clusters_to_plot = np.empty((0, N_pixels))
    for i in range(10):
        # Get indices for label i
        label_indices = np.where(cluster_labels == i)[0]
        # Get data for label i
        label_data = kmeans_centers[label_indices]
        # Append 10 cluster centers from label i to clusters_to_plot
        clusters_to_plot = np.append(clusters_to_plot, label_data[:10], axis=0)

    # Plot some cluster centers
    plot_cluster_centers(clusters_to_plot)

    # Classify test data with nearest neighbor classifier
    classified_labels = []

    # Calculate distances for each test data
    for i in range(N_test):
        # Get test image
        test_image = test_data[i]

        distances = []
        for j in range(len(kmeans_centers)):
            mean_image = kmeans_centers[j]
            distance = euclidean_distance(test_image, mean_image, N_pixels)
            distances.append(distance)

        # Find label with smallest distance
        label = np.argmin(distances)
        label = cluster_labels[label]
        classified_labels.append(label)
```

```
    # Print training time
    time_end = time.time()
    training_time = int(time_end - time_start)
    print_time(time_start, time_end)

    # Find confusion matrix
    confusion_matrix = confusion_matrix_func(classified_labels, test_label, C)
    print(confusion_matrix)

    # Print error rate
    error_rate = error_rate_func(confusion_matrix)
    print("Error rate: ", error_rate*100, "%")

    # Save confusion matrix to file
    save_to_file("Kmeans/CM_Kmeans_", confusion_matrix ,error_rate, N_train, N_test,
training_time)

    # Plot confusion matrix
    plot_confusion_matrix("Kmeans", confusion_matrix, error_rate, visualize_confusion_matrix)

# Classify test data with KNN classifier-------------------------------------------------------------
-------------------------------------------------------------
if KNN_classification:
    print("KNN classification")

    print("Start training")
    time_start = time.time()

    # Perform k-means clustering on training data
    start_training = True
    if start_training:

        # Create 64 clusters for each unique label from training data
        kmeans_centers = np.empty((0, N_pixels))
        cluster_labels = np.empty((0, 1))

        for i in range(C):
            # Get indices for label i
            label_indices = np.where(train_label == i)[0]
            # Get data for label i
            label_data = train_data[label_indices]
            # Perform k-means clustering on label data
            kmeans = KMeans(n_clusters=M_clusters,
random_state=0).fit(label_data.reshape(len(label_indices), N_pixels))
            # Store cluster centers
            kmeans_centers = np.append(kmeans_centers, kmeans.cluster_centers_, axis=0)
            # Append M_clusters cluster labels to cluster_labels
            cluster_labels = np.append(cluster_labels, np.full((M_clusters, 1), i), axis=0)

        #Store cluster labels for training data and cluster centers in a file in a folder called
"kmeans_trained"
        np.savetxt("kmeans_trained/cluster_labels.txt", cluster_labels, fmt="%d")
        np.savetxt("kmeans_trained/cluster_centers.txt", kmeans_centers, fmt="%f")

    # Load cluster labels and cluster centers from file
    cluster_labels = np.loadtxt("kmeans_trained/cluster_labels.txt", dtype=int)
```

```
    kmeans_centers = np.loadtxt("kmeans_trained/cluster_centers.txt", dtype=float)

    # Classify test data using K-nearest neighbor classifier
    classified_labels = []

    for i in range(N_test):
        # Get test image
        test_image = test_data[i]

        distances = np.zeros(len(kmeans_centers))
        for j in range(len(kmeans_centers)):
            mean_image = kmeans_centers[j]
            distance = euclidean_distance(test_image, mean_image, N_pixels)
            distances[j] = distance

        nearest_neighbors = np.argsort(distances)[:K_neighbors]

        nearest_neighbors_labels = []
        for neighbor in nearest_neighbors:
            nearest_neighbors_labels.append(cluster_labels[neighbor])

        # Find label with most occurences
        label = np.argmax(np.bincount(nearest_neighbors_labels))
        classified_labels.append(label)

    # Print training time
    time_end = time.time()
    training_time = int(time_end - time_start)
    print_time(time_start, time_end)

    # Find confusion matrix
    confusion_matrix = confusion_matrix_func(classified_labels, test_label, C)
    print(confusion_matrix)

    error_rate = error_rate_func(confusion_matrix)
    print("Error rate: ", error_rate*100, "%")

    # Save confusion matrix to file
    save_to_file("KNN/CM_KNN_", confusion_matrix ,error_rate, N_train, N_test, training_time)

    # Plot confusion matrix
    plot_confusion_matrix("KNN, K=" + str(K_neighbors),confusion_matrix, error_rate,
visualize_confusion_matrix)

# ------------------------------------------------------------------------------------------
----------------------
plt.show()
```

Digits\Digits_functions.py

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import math

# Calculate mean value of training data for each label
def mean_digit_value_image(train_data, train_label, C, N_pixels):
    mean_data = np.zeros((C, N_pixels))
    for i in range(C):
        mean_data[i] = np.mean(train_data[train_label == i], axis=0).reshape(N_pixels)
    return mean_data

# Calculate euclidean distance
def euclidean_distance(x, mean, N_pixels):
    mean = mean.reshape(N_pixels, 1)
    x = x.reshape(N_pixels, 1)
    return ((x - mean).T).dot(x - mean)

def confusion_matrix_func(classified_labels, test_label, C):
    confusion_matrix = np.zeros((C, C))

    for i in range(len(classified_labels)):
        confusion_matrix[test_label[i], classified_labels[i]] += 1
    return confusion_matrix

# Calculate mahalanobis distance
def mahalanobis_distance(x, mean, cov, N_pixels):
    mean = mean.reshape(N_pixels, 1)
    x = x.reshape(N_pixels, 1)
    return ((x - mean).T).dot(np.linalg.inv(cov)).dot(x - mean)

# Find error rate
def error_rate_func(confusion_matrix):
    error = np.trace(confusion_matrix)
    return round(1 - (error / np.sum(confusion_matrix)),5)

# Plot functions
def plot_digit(data_set, index):
    plt.imshow(data_set[index], cmap=plt.get_cmap('gray'))

def plot_confusion_matrix(titleCF,confusion_matrix, error_rate, visualize):
    if visualize:
        plt.figure(figsize = (10,7))
        plt.title('Confusion matrix for ' + titleCF + '\n'+'Error rate:
'+str(error_rate*100)+'%')
        sns.heatmap(confusion_matrix, annot=True, fmt='.0f')
        plt.xlabel('Predicted label')
        plt.ylabel('True label')
        plt.show()

def plot_classified_image(test_image, mean_image):
    plt.subplot(1, 2, 1)
    plt.imshow(test_image, cmap=plt.get_cmap('gray'))
```

```
    plt.title('Test image')
    plt.subplot(1, 2, 2)
    plt.imshow(mean_image, cmap=plt.get_cmap('gray'))
    plt.title('Mean image')
    plt.show()

def compare_test_images(N_plots, test_data, mean_data, classified_labels, labels_indexes):
    plt.figure()
    for i in range(N_plots):

        lab_index = labels_indexes[i]

        test_image = test_data[lab_index]
        predicted_image = mean_data[classified_labels[lab_index]].reshape(28, 28)
        difference_image = test_image - predicted_image

        plt.subplot(N_plots,3,3*i+1)
        plt.imshow(test_image, cmap=plt.get_cmap('gray'))
        if i == 0:
            plt.title('Test image')

        plt.subplot(N_plots,3,3*i+2)
        plt.imshow(predicted_image, cmap=plt.get_cmap('gray'))
        if i == 0:
            plt.title('Predicted image')

        plt.subplot(N_plots,3,3*i+3)
        plt.imshow(difference_image, cmap=plt.get_cmap('gray'))
        if i == 0:
            plt.title('Difference image')

# Plot some clusters centers
def plot_cluster_centers(centers):
    plt.figure(figsize=(10, 100))
    for i in range(100):
        plt.subplot(10, 10, i+1)
        plt.imshow(centers[i].reshape(28, 28), cmap=plt.get_cmap('gray'))
        plt.axis('off')
    # Add title
    plt.suptitle("Some cluster centers for each digit", fontsize=20)

    plt.show()

# Print training time nicely
def print_time(start_time, end_time):
    time = end_time - start_time
    hours = int(time // 3600)
    minutes = int((time % 3600) // 60)
    seconds = int(time % 60)
    print("Time: {:02d}:{:02d}:{:02d}".format(hours, minutes, seconds))


# Save to file
def save_to_file(file_name, confusion_matrix, error_rate, N_train, N_test, time):
    file_title = "Plots_and_results/"+ file_name +"N_train_" + str(N_train) + "_N_test_" +
str(N_test) + ".txt"
```

```
        with open(file_title, 'w') as f:
            f.write("Confusion matrix:\n")
            f.write(str(confusion_matrix))
            f.write("\nError rate: "+str(error_rate*100)+"%\n")
            f.write("Time: "+str(time)+"\n")

        print("Saved to file: " + file_title)

    # plot NN comparison
    def plot_NN_comparison(test_data, test_label, classified_labels, correct_labels_indexes,
    failed_labels_indexes, N_Comparisons, visualize_NN_comparison):
        N_Comparisons = min(N_Comparisons, len(correct_labels_indexes), len(failed_labels_indexes))
        if visualize_NN_comparison:
            plt.figure(figsize=(10, 10))
            for i in range(N_Comparisons):
                plt.subplot(2, N_Comparisons, i+1)
                plt.imshow(test_data[correct_labels_indexes[i]], cmap=plt.get_cmap('gray'))
                plt.title("True label: " + str(test_label[correct_labels_indexes[i]]) + "\nPredicted
    label: " + str(classified_labels[correct_labels_indexes[i]]))
                plt.axis('off')

                plt.subplot(2, N_Comparisons, i+1+N_Comparisons)
                plt.imshow(test_data[failed_labels_indexes[i]], cmap=plt.get_cmap('gray'))
                plt.title("True label: " + str(test_label[failed_labels_indexes[i]]) + "\nPredicted
    label: " + str(classified_labels[failed_labels_indexes[i]]))
                plt.axis('off')
            plt.show()
```