TTT4275 Classification Project

Synnøve Arnesen and Jesper Nilsson Lybeck

April 29, 2025

Abstract

This report presents the design and evaluation of classification models for the Iris and MNIST datasets, developed as part of the course TTT4275 – Estimation, Detection and Classification.

In the Iris task, a linear classifier trained with minimum square error (MSE) achieved high accuracy when using all features. Removing the least linearly separable feature (sepal width) slightly increased misclassification, showing that even weak features can contribute to performance. When reduced to a single feature, the model required fewer training epochs but at the cost of degraded accuracy.

For MNIST, the nearest neighbour classifier with all templates yielded the best result with 96.91% accuracy, though at a high computational cost (7653 seconds). Using K-means clusters as templates for NN reduced runtime to 47 seconds, with a small accuracy reduction to 95.23%. This time measures only the NN run time, and not the K-means clustering time. The k-NN classifier (k=7) with the same templates, was equally fast (47 seconds), but with the lowest accuracy at 94.36%.

These results conclude that even simple classification methods can achieve strong results when properly tuned and supported by well-structured data. Key lessons include the impact of feature selection, the trade-off between speed and accuracy.

Contents

1 Introduction

This project report outlines the process of solving simple classification tasks. The goal of this project is to gain insight into the design and evaluation process of different classification approaches. The first part aims to produce a linear classifier that is able to differentiate between sub-species of the Iris flower. The Fisher Iris data[2] consists of a dataset where length and width of petal and sepal leaves are labelled with the correct subspecies. This dataset will be used for training and testing of the classifier. In the second part of the project, the task is to classify handwritten digits from the MNIST dataset[1]. For this part of the project, three different approaches are evaluated by execution time and error rate.

Classification theory has gained a great interest in a range of different fields. The problem of classifying data efficiently and precisely has broad applications in medicine, computer vision, insurance and banking, just to mention a few. The problem of classification can be considered as a general problem. In the first part of this project the application is biological taxonomy.

This report is structured as follows: Section 2 covers the theory, Section 3 describes the task, Section 4 presents the implementation and results, and Section 5 concludes the report. Since the project involves two different sub-projects, each of the sections cover first the Iris problem, and then the handwritten digits problem. A more detailed explanation of the task for the Iris problem and the handwritten digits problem is provided in section 3

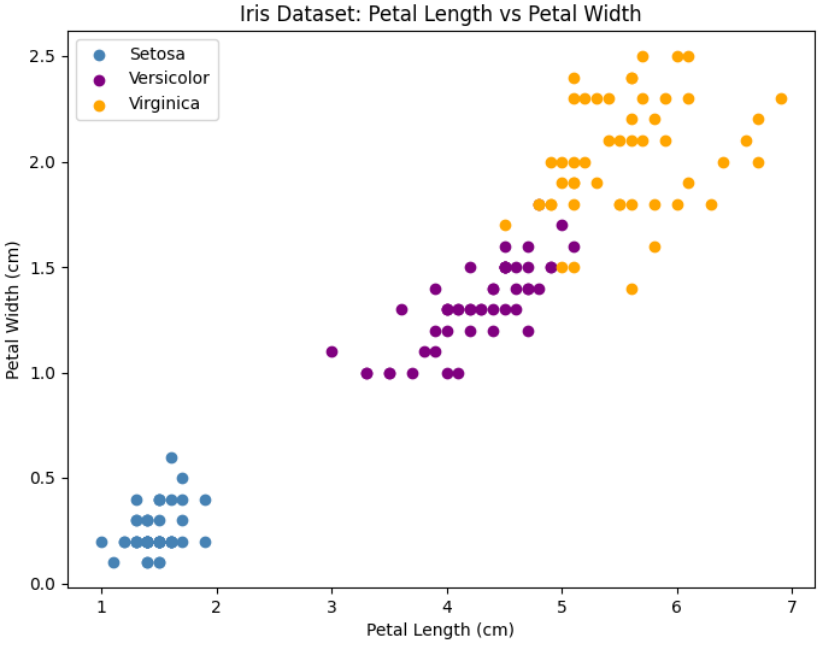

Figure 1: Iris Dataset Petal Length vs Petal Width

2 Theory

In this section the necessary background theory is described. The section provides a brief overview of the theory underlying the implementation presented in section 4, without going into detailed derivations.

In general, the classification problem can be summarized as follows. For a given observation $x \in \mathbb{R}^\ell$, where x is a vector of ℓ measured features, we want to assign x to one of m classes $\omega_1, \omega_2, \ldots, \omega_m$.

Geometrically, this requires partitioning the ℓ-dimensional observation space $\mathbb{R}^\ell$ into m disjoint regions $\Omega_1, \Omega_2, \ldots, \Omega_m$, such that:

$$x \in \Omega_i \implies x \text{ belongs to class } \omega_i.$$

2.1 Iris

For the first part of the project, we want to use a linear classifier for the Iris data. This means that the decision regions Ω_i are separated by *linear* surfaces. When looking at the histograms showing the distributions of the four features in 2, we note that petal width and petal length are features that shows promise for linear classification. Therefor we use those features as a basis and plot the dataset onto this in figure 1. Here it is quite clear that finding a linear partition of this space that separates between Iris Setosa, and the two other classes should be possible. However it will likely be more difficult to find a clear distinction between the versicolor and the virginica subspecies, at least based on petal-width and petal-length alone. The equations presented in the following section is from the course compendium in classification[5].The Iris task is described in more detail in section 3.1

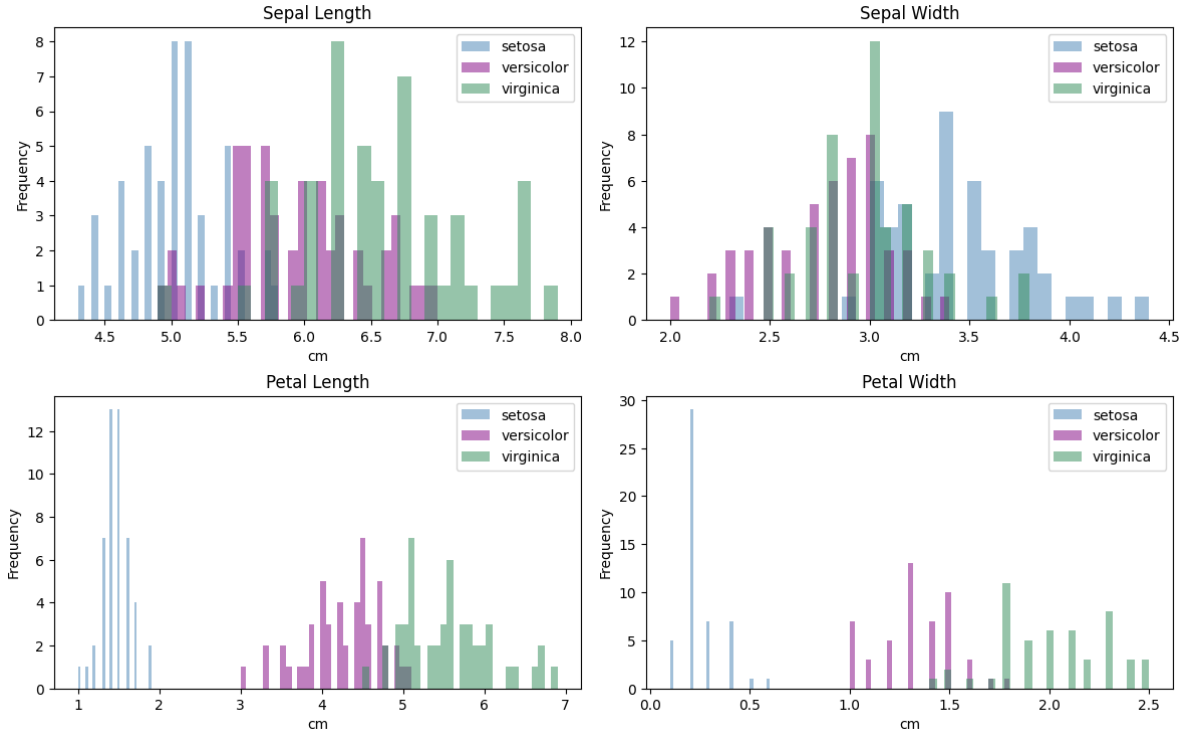

Figure 2: Histograms for each feature and class.

The linear classifier in question is a discriminant classifier or *LDC*. This has the decision rule:

$$x \in \omega_j \iff g_j(x) = \max_{i \in \{1,\dots,m\}} g_i(x). \tag{1}$$

where:

$$g_i(x) = w_i^T x + w_{i_0}, \quad i = 1, \dots, m \tag{2}$$

In order to ease readabilty,this is written on the more compact form:

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

This vector has dimensionality equal to the number of classes m. In the case of the Iris data this is $m = 3$. We can redefine an augmented version of W and x, such that $[Ww_0] \to W$ and $[x^T 1]^T \to x$, then $g = Wx$. Training this model means adjusting the weights in a way that minimises a objective. This objective is the minimum square error or the *MSE*:

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

We sum over the dataset, computing the squared error, $g_k - t_k$, where t_k is the true class, and g_k is the discriminant vector. Since t_k is a one-hot encoded target vector and g_k is a vector of continuous values, we need some function to map g_k to either 1 or 0. Ideally, this would be the Heaviside step function. However, as will be evident later, we require differentiability of the MSE function. Therefore, we instead use the sigmoid (or logistic) function[1].:

$$g_{ik} = \text{sigmoid}(x_{ik}) = \frac{1}{1 + e^{-z_{ik}}} \quad i = 1, \dots, C \tag{5}$$

Here, z_{ik} represents the input to the sigmoid function(previously referred to as g_k). In this task we were to implement the classifier according to chapter 3.2 in the course compendium[5]. that meant using MSE, with sigmoid. This may not be the most common approach for multi class problems, but as will be evident later, still produced good results on the Iris data. Solving the optimization problem of finding the weights that minimises 4, is best approached numerically with a gradient based approach. The gradient of the MSE with respect to the weights can be found from the chain rule for derivatives. The goal being to iteratively update the weights W by moving in the direction of steepest descent in the parameterspace of the error (MSE). The gradient of the MSE with respect to the weights W is:

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

where:

$$\nabla_{g_k} \text{MSE} = g_k - t_k \tag{7}$$

$$\nabla_{z_k} g_k = g_k \circ (1 - g_k) \tag{8}$$

$$\nabla_W z_k = x_k^T \tag{9}$$

where $g_k \circ (1 - g_k)$ is the element-wise product.
inserting into 6, the following is obtained.

[1] A (maybe) more common approach would be to use *cross-entropy* with *Softmax* activation, as the softmax ensures the outputted result sums to 1 for all classes, yielding something than can be interpreted as a probability for classes. This is why we use the term "class score", not a probability for the output of the sigmoid function

$$\nabla_W \text{MSE} = \sum_{k=1}^{N} \left[(g_k - t_k) \circ g_k \circ (1 - g_k) \right] x_k^T \tag{10}$$

now that the gradient of 4 is found, the weights are iterativly updated with this formula:

$$W(m) = W(m-1) - \alpha \nabla_W MSE \tag{11}$$

α is called the learning rate or the step size, and this together with the number of training iterations are the tuning parameters for this model.

2.2 Handwritten numbers

For the classification of handwritten digits, a different approach will be taken. The handwritten numbers task is described in more detail in section 3.2. This part will use template based classification. The main idea here is that data that belongs together, is similar. A number of labeled samples, can be used to make *templates*. Then for a sample x this is classified by finding which template best matches x according to some evaluation criterion. x is then assumed to have the same class as the closest matching template. Three closely related methods will be used and evaluated based of accuracy and computational time. The first method is called *Nearest Neighbour*, which will be subsequently referred to as *NN*.

The measure of similiarity that will be used for methods is the Euclidean distance: The Euclidean distance between two points $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ in n-dimensional space is given by:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}. \tag{12}$$

In the NN approach, the following summarizes how a point x is classified. The pixel values ranges from $[0, 255]$, but for this explanation we let it be normalised to the interval $[0, 1]$. A sample from the dataset $\mathbf{X}$ is structured as:

$$\mathbf{X} \in [0,1]^{28 \times 28}, \quad \text{and} \quad X_{i,j} \text{ denotes the intensity of the pixel at row } i, \text{ column } j. \tag{13}$$

Here, $X_{i,j} = 0$ corresponds to a black background, while $X_{i,j} = 1$ represents a white pixel (digit stroke). In order to compute the euclidean distance as described in 12, $\mathbf{X}$ needs to be flattened. We let $x \in [0,1]^{784}$ be the flattened image.

2.3 Nearest Neighbor

For every data sample x in the dataset, the distance to every template sample is computed. x belongs to the same class as the template with the smallest distance from x. This approach is as the name suggests the *Nearest Neighbour* classification. As will be evident in the results presentation, this produces a accurate result, but at a high computational cost.

2.4 Clustering based approach

An alternative approach that adresses the high computational cost of computing the distance for every single point to all template points, is to create templates from clustering the data. The cluster centroids are instead used as templates. Then x belongs to the same class as the samples in the cluster that has the closest centroid point to x. Finding the centroids that are used for templates in this approach, is done with the K-means clustering algorithm. The K-means clustering algorithm can be summarized as follows

1. Decide on number of clusters, K
2. Randomly select K centroids
3. Assign each data point to a random cluster
4. For each data point, find the closest centroid and assign it to that centroid's cluster
5. For each cluster, calculate its new centroid by taking average position of all points in the cluster
6. repeat the last 2 steps, until the clusters and centroids no longer changes.

2.5 K nearest neighbors

The final approach we will use is closely related to nearest neighbor, but differs in the way that instead of finding the single nearest template, we find the nearest K neighboring templates. When the K nearest templates for the sample has been decided, a majority vote is taken. The sample x is classified with the same class that the majority of the K nearest template points belongs to. Implementation details and results for the three methods is presented in section4

3 The Task

In this section, the two classification tasks, Iris and MNIST are introduced and explored. The Iris task is presented first, followed by the MNIST task.

3.1 Iris

The Iris classification task is a classic problem in machine learning. It involves predicting which species an Iris flower belongs to based on four features: the length and width of the flower's petals and sepals. The dataset contains 150 examples, evenly split among three species: Setosa, Versicolor, and Virginica (see Figure 3).

To evaluate the classifier's performance, the model was trained on different subsets of the data and tested on unseen samples. Additionally, to assess the importance of each feature with respect to linear separability, the classifier was retrained using fewer features to explore how feature reduction affects the ability to separate the classes.

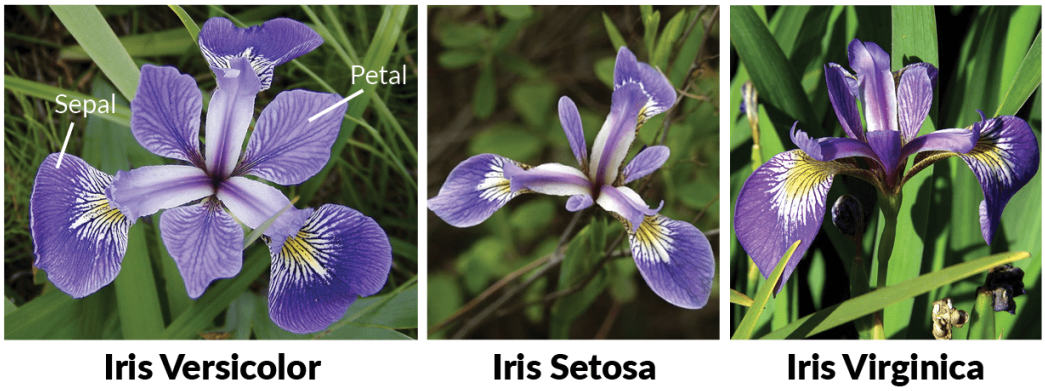

Figure 3: Iris flowers with labeled features (image taken from [3])

3.2 MNIST

The MNIST classification task involves recognizing handwritten digits (0–9) based on 28×28 grayscale images, as shown in 4. It is widely used as a benchmark in machine learning, containing 60,000 training and 10,000 test images written by different individuals to ensure realistic evaluation. The dataset's size and diversity make it a valuable tool for studying multi-class classification. Due to its popularity, a wide range of classifiers have been developed for this task, from simple distance-based models to advanced deep neural networks.

The task began with a nearest neighbor classifier using Euclidean distance, followed by K-means clustering to reduce templates per class, which were then used by both the K-means NN and a k-NN classifier (K = 7) for comparison.

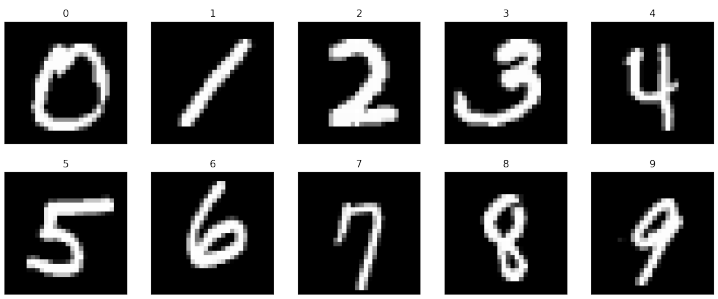

Figure 4: Examples of Mnist data (image taken from [4])

4 Implementation and Results

This section presents the implementation details and results of the classifiers developed for both the Iris and MNIST tasks. The goal is to evaluate the performance of different classification approaches and understand their strengths and limitations. The section is divided into three main parts: one focusing on a linear classifier applied to the Iris dataset, and the other two on distance-based classifiers used for MNIST digit recognition. Each subsection begins by outlining its objective, followed by implementation, visualizations, and a discussion of results.

4.1 Iris classification task

This subsection presents the implementation and evaluation of a linear classifier on the Iris dataset. The section is organized by the data preparation, model training process, and analysis of performance across different feature combinations and data splits.

In the Iris classification task, a linear discriminant classifier or *LDC* was developed and trained using *Minimum Square Error* (MSE) on the version of the Fisher Iris data provided by Scikit-learn [2]. The theory behind the process is explained in more detail in section 2.2, but quickly summarised here. The linear model outputs continuous values which are then passed through the sigmoid function to *squash* them into the interval $[0, 1]$, allowing them to be interpreted as class scores for multi-class classification. The prediction is made by selecting the class with the highest score. MSE was used as the objective function in finding the set of weights that minimizes this difference between the model predictions and the correct class label.

As illustrated in the flow diagram 5, the Iris dataset was first loaded and then split into training and testing sets, ensuring a balanced class representation. The target labels were converted into one-hot encoded vectors before training began.

The weights were initialized to zero, and training was carried out over multiple epochs. In each epoch, the training loss (MSE) was calculated, predictions were made for both the training and testing sets, and error rates were computed. The weights were then updated using gradient descent, based on the learning rate α.

The gradient of the minimum Square Error 10 was implemented in Python as shown in 1, and updated using the gradient descent rule 11. Additional implementation details can be found in the appendix.

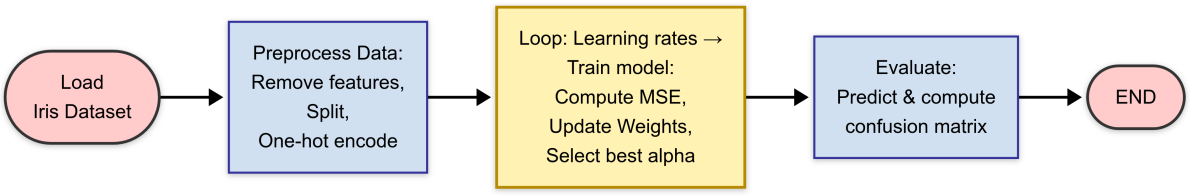

Figure 5: Flow-diagram iris classifier.

```python
def grad_MSE(data, weights, targets):
    MSE = 0
    g = discriminant_vec(data, weights)
    x_aug = np.hstack((data, np.ones((data.shape[0], 1))))
    error_term = (g - targets) * g * (1 - g)
    MSE = error_term.T @ x_aug
    return MSE
```

Listing 1: Gradient of the MSE loss function

Initially the first 30 samples for each class were chosen for training the model, while the last 20 samples were used for testing. Using this split, the number of iterations and the learning rate α were adjusted. The MSE was plotted with a selection of different values for α. As can be seen in figure 6, after around 2000 epochs the training loss converged. The best performance was achieved with $\alpha = 0.005$. The corresponding confusion matrix can be seen in figure 7.

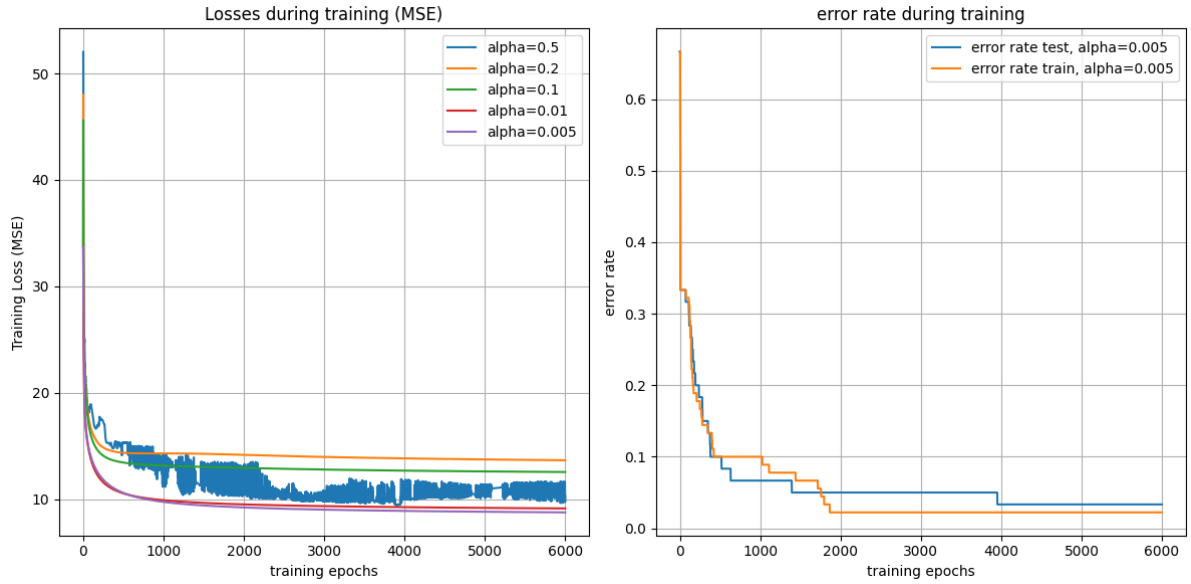

Figure 6: Losses (MSE) and error rate during training with the first 30 samples for training and the last 20 for testing.

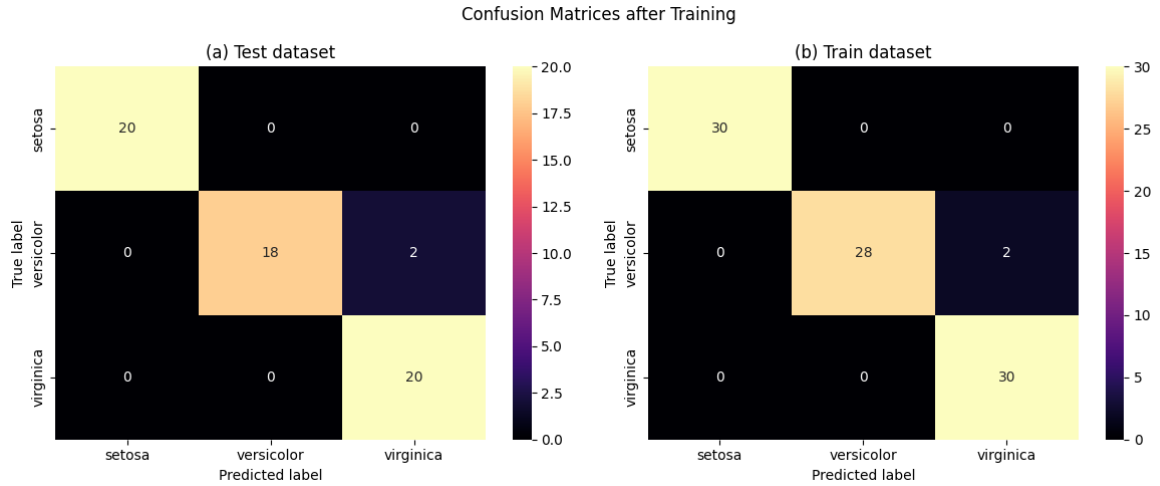

Figure 7: Confusion matrix after training with the first 30 samples for training and the last 20 for testing.

To assess how different training and testing splits affect model performance, the experiment was repeated using a new data split. For this test, the last 30 samples of each class were chosen for training, while the first 20 samples were used for testing. The learning rate and number of epochs were kept the same as in the previous experiment. As illustrated in Figure 8,

this configuration resulted in a slightly higher training error, misclassifying three more samples compared to the initial split.

Despite the change in training accuracy, the confusion matrices in Figure 9 reveal nearly identical performance on the test set. In the original split, two Versicolors were misclassified as Virginicas, while in the second split, only one Versicolor was misclassified, demonstrating the robustness of the classifier across different splits when class balance is maintained.

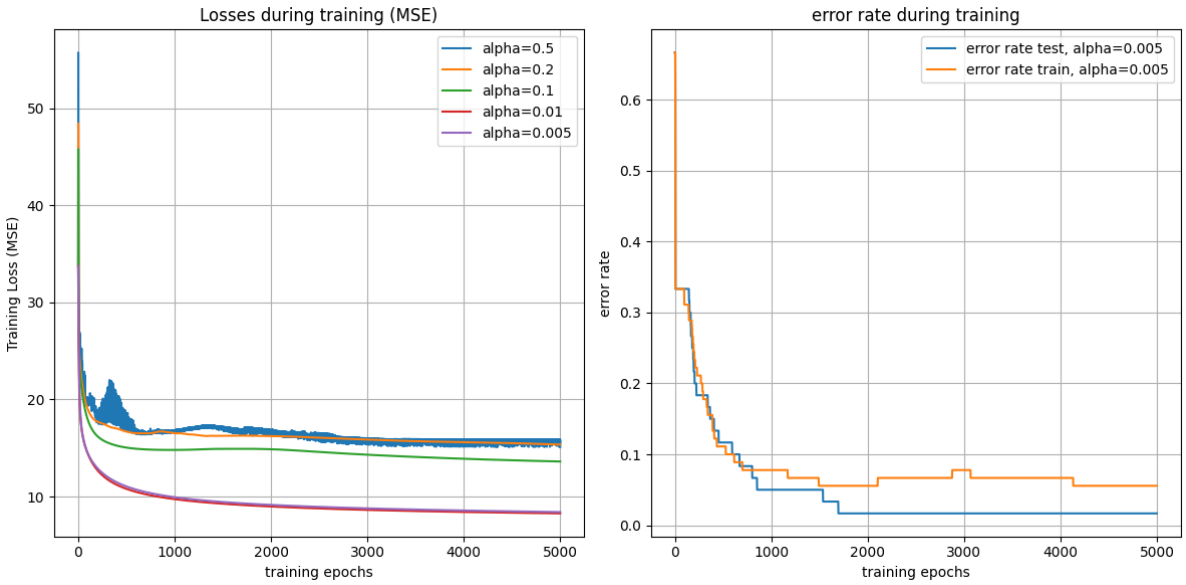

Figure 8: Losses (MSE) and error rate during training with the last 30 samples for training and the last 20 for testing.

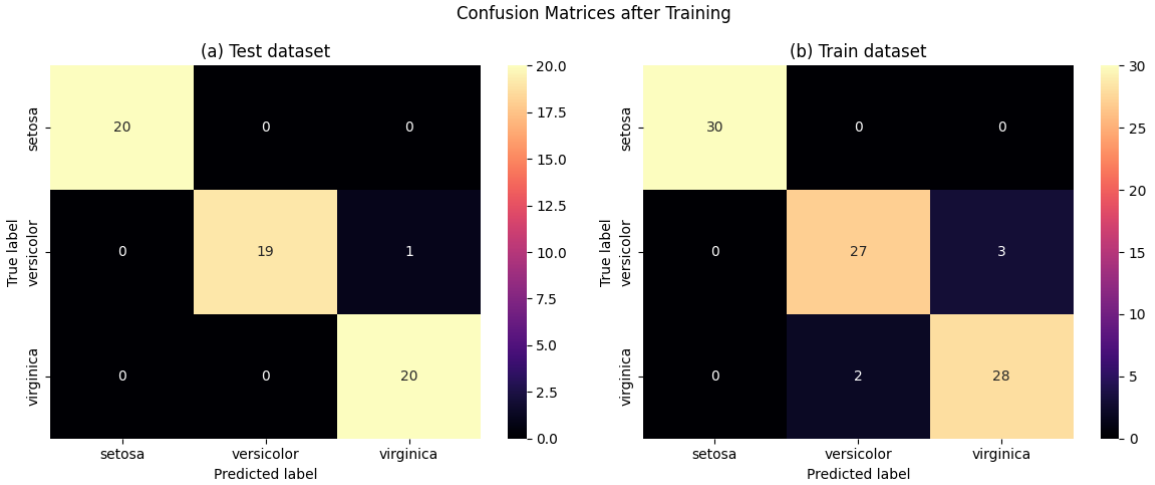

Figure 9: Confusion matrix after training with the last 30 samples for training and the first 20 for testing.

Additionally, the Iris task included an analysis on the importance of the features included during the training of the classifier. During this analysis the first 30 samples were chosen for training, while the last 20 samples were used for testing. Firstly, the features and their separation of classes were visualized with histograms as seen in figure 2.

These plots showcased that some of the features had more overlap between classes than others. Interestingly, the sepal width was the feature with the least linearly separable data. The sepal width was removed from the features and the classifier was trained and tested on the remaining three features. During this training the classifier needed a larger number of epochs. Initially it seemed that the removed feature did not contribute significantly to classification performance, as the data was not linearly separable. However, the model now had to work harder to find patterns in the remaining data. This slowed down our iterative gradient descent algorithm. Figure 10 shows the error rate flattening out at around 5000 epochs with the same learning rate as the previous test, 0.005. Looking at the confusion matrix figure 11, it is observed that the predictions were almost as accurate as the classifier with the full feature space. One more versicolor was classified wrongly as virginica.

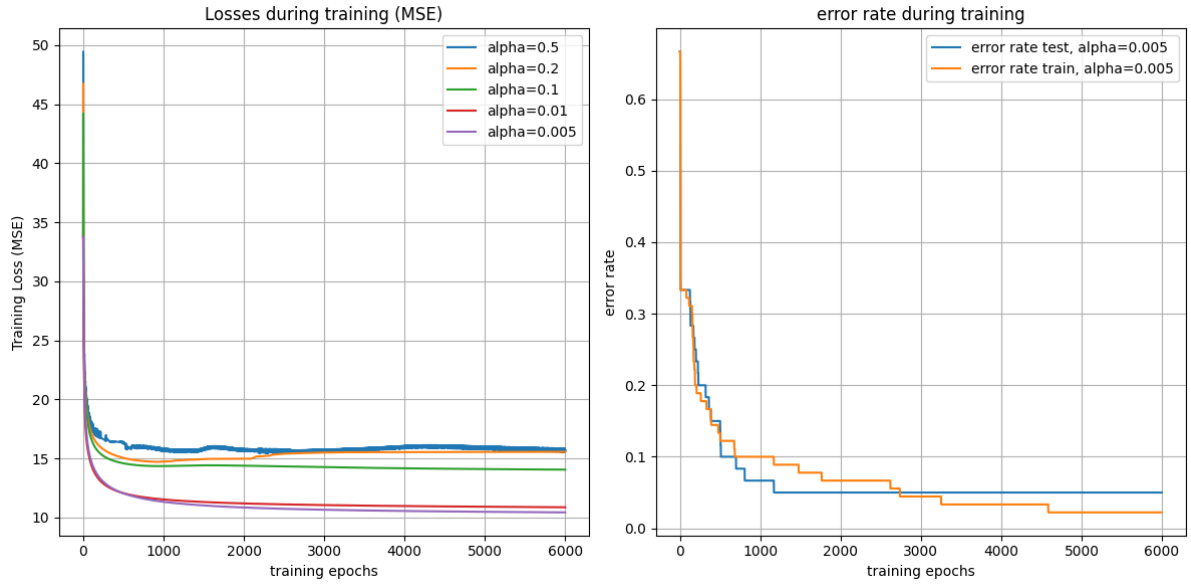

Figure 10: Losses (MSE) and error rate with removed feature sepal width. The first 30 samples for training and the last 20 for testing.

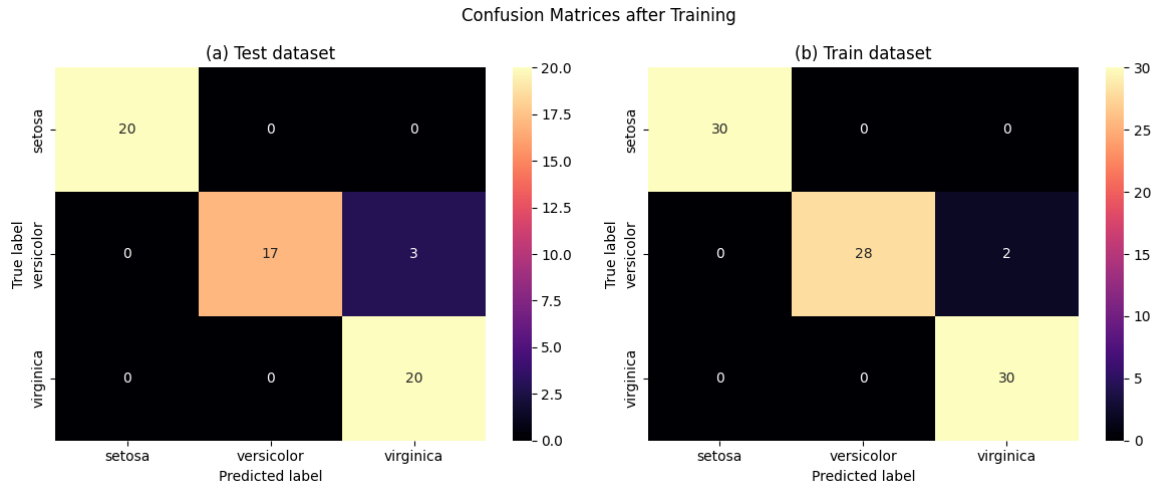

Figure 11: Confusion matrix after training with removed feature sepal width. The first 30 samples for training and the last 20 for testing

Next, the sepal length feature was also removed from the feature space, as it contained the most overlap between the classes among the three remaining features. With the remaining two features, petal width and length, the error rate converged even more slowly than with three features. The slower convergence was due to the smaller gradients used in the iterative optimization. Therefore the learning rate was bumped up to 0.01 from 0.005 to achieve somewhat efficient training. With the new learning rate, the error rate converged after approximately 5500 epochs, as seen in figure 12. Even though the features removed were not linearly separable, the classifier performed worse than when the features were included. The confusion matrix in figure 13 shows how the classifier misclassified 4 samples in both the training and test sets.

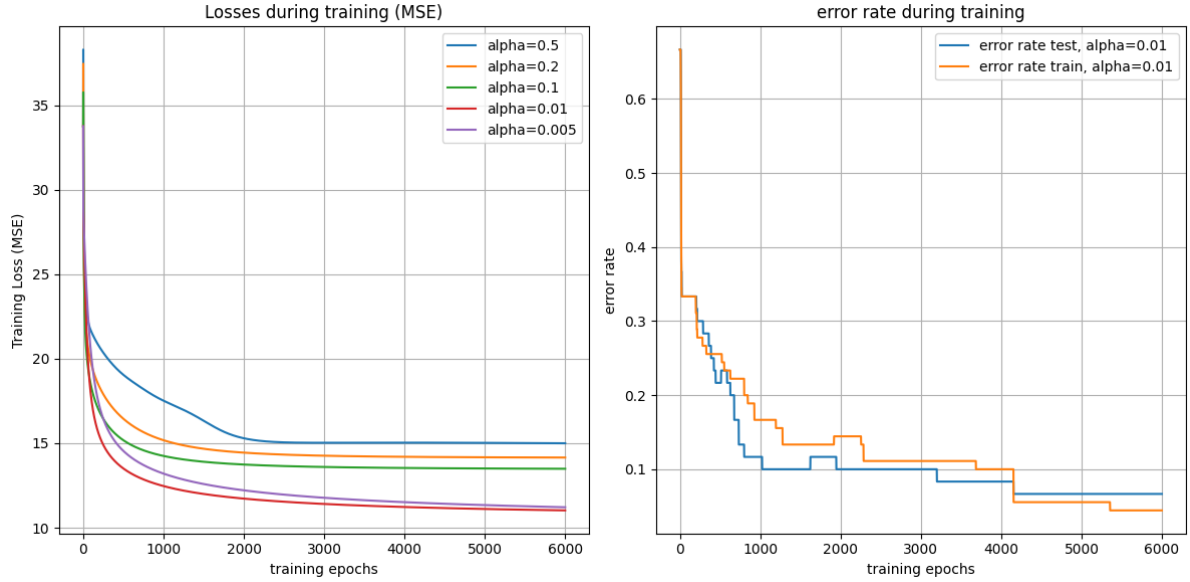

Figure 12: Losses (MSE) and error rate with removed feature sepal width and sepal length. The first 30 samples for training and the last 20 for testing.

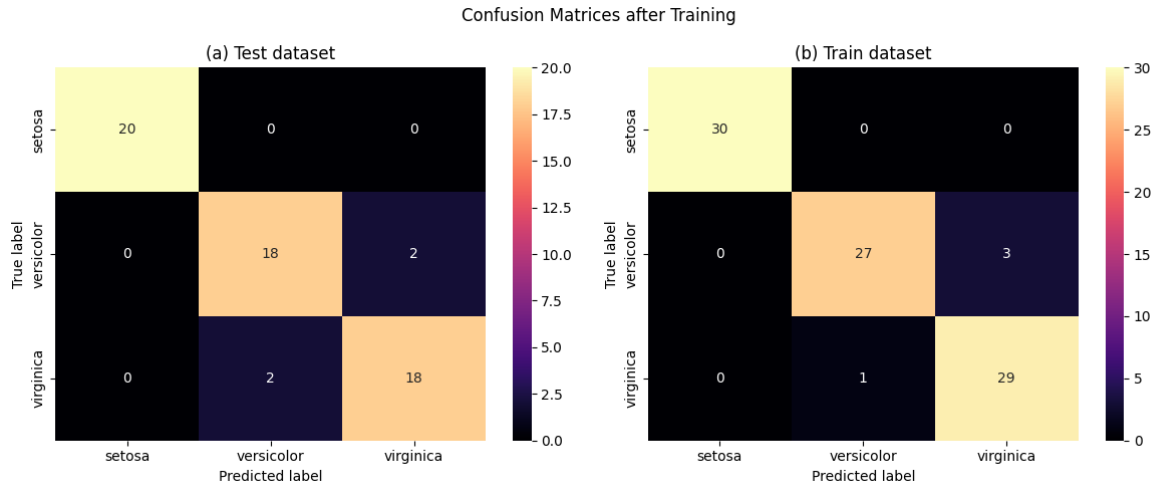

Figure 13: Confusion matrix after training with removed feature sepal width and sepal length. The first 30 samples for training and the last 20 for testing

Finally, we reduced the features to just petal width. The model now operated in a one-dimensional feature space, drastically simplifying the optimization. In higher-dimensional spaces, the gradients can point in many conflicting directions, making optimization more complex. By contrast, with a single dimension, the gradient updates are more stable, leading to faster convergence. Figure 14 shows how in just 1000 epochs the error rate flattens out. Inspecting the confusion matrix 15 the performance of the classifier had worsened from the initial classifier. However, it still managed to classify the same number of samples correctly as the model built on two features. In fact, the confusion matrix is identical to that of the previous test.

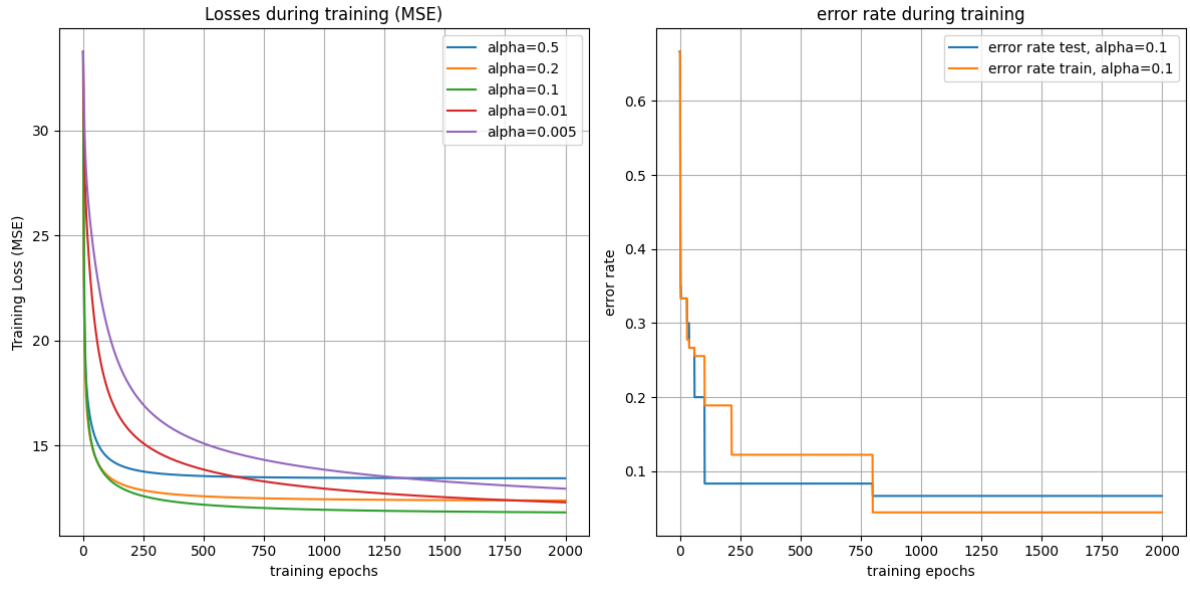

Figure 14: Losses (MSE) and error rate with removed feature sepal width, sepal length and petal length. The first 30 samples for training and the last 20 for testing.

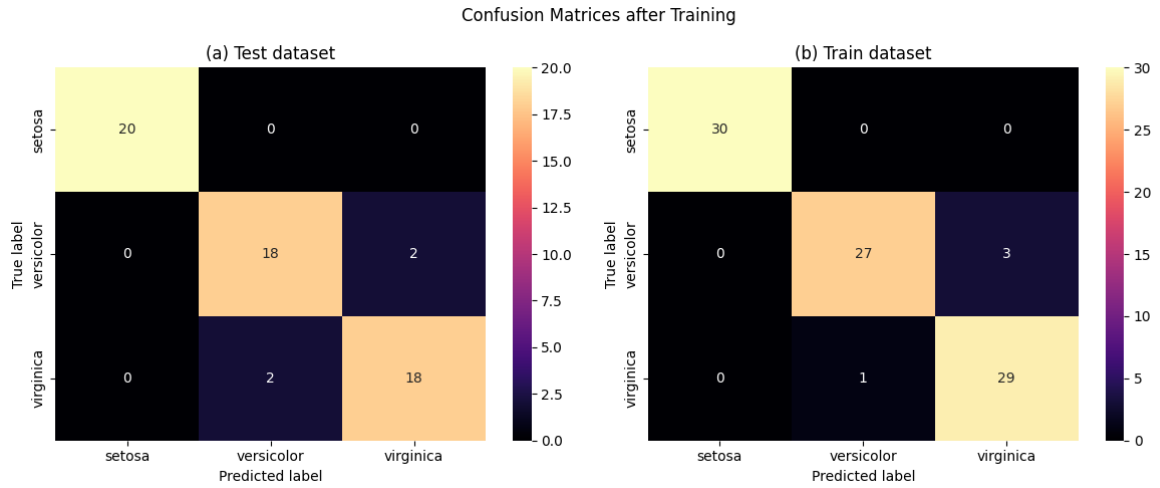

Figure 15: Confusion matrix after training with removed feature sepal width, sepal length and petal length. The first 30 samples for training and the last 20 for testing

4.2 MNIST with Nearest Neighbour Classifier

This subsection describes the application of a Nearest Neighbour (NN) classifier described in section 2.3, to the MNIST digit classification task. The performance is evaluated based on accuracy, error rate, and computational time. Examples of misclassification are included to highlight the classifier's behavior.

In the second part of this project, a Nearest Neighbor Classifier was implemented based of the theory presented in section 2.2. The dataset was accessed through Scikit-learn [1]. To manage the computational cost during the design phase, smaller segments of the dataset was used. After the program was tested on the smaller subset, the entire dataset was used to produce the results presented here. In addition, correctly and incorrectly classified images are visualized to give some insight into the the behaviour of the classifier. The python implementation can be seen in 2. The rest of the implemented code can be found in the appendix.

```
def nearest_neighbour(X, templates, template_labels, true_labels):
    correct_predictions = []
    incorrect_predictions = []
    predictions = []

    for x, true_label in zip(X, true_labels):
        distances = [distance(x, t) for t in templates]
        index = np.argmin(distances)
        predicted_label = template_labels[index]
        predictions.append(predicted_label)

        if predicted_label == true_label:
            correct_predictions.append(true_label)
        else:
            incorrect_predictions.append(true_label)

    return correct_predictions, incorrect_predictions, predictions
```

Listing 2: Nearest Neighbour Classifier

As shown i figure 16, the classifier achived high accuracy overall, with a total of 9691 correctly

classified digits and only 309 misclassified, resulting in an error rate of approximately 3.09%. This strong performance in identifying handwritten digits is illustrated in figure 17.

Digits such as 4 and 9 were often misclassified due to their similar shapes in handwritten form. Additionally, digits that were slanted, such as a leaning 3, were more prone to error. Misclassifications also occurred with heavily stylized digits, such as 9 with an extended tail, as seen in 17.

However, the classification process was computationally expensive, taking a total of 7653.60 seconds to complete.

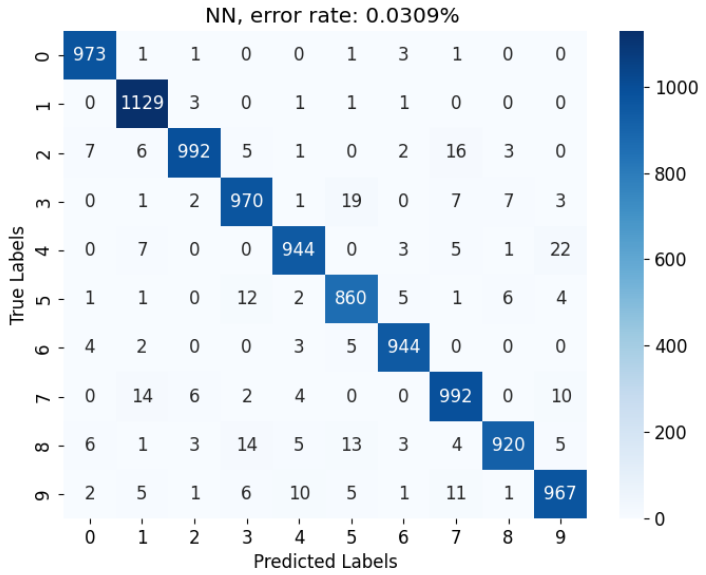

Figure 16: NN error rate

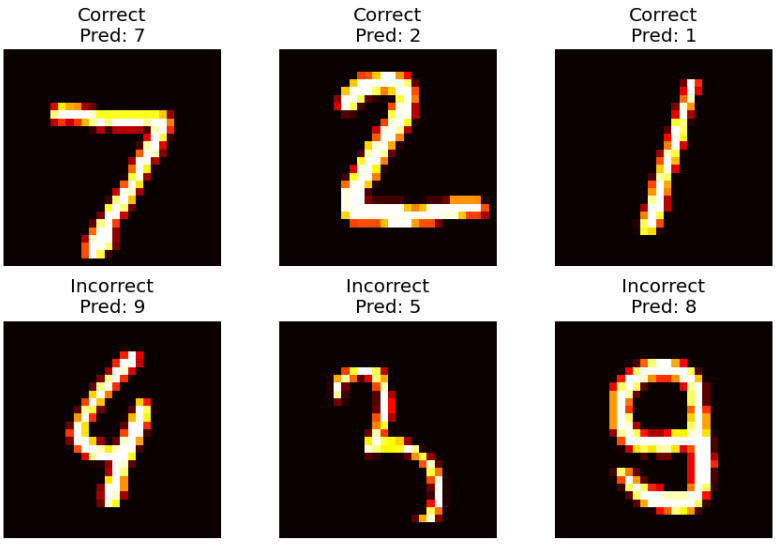

Figure 17: NN classification examples

4.3 MNIST with K-means clustering Nearest Neighbour Classifier and K-Nearest Neighbours Classifier

This subsection evaluates two additional classification methods on the MNIST dataset: Nearest Neighbour classifier using K-means-generated templates and a K-Nearest Neighbours (k-NN) classifier with $K = 7$ using the same clustered templates. These methods are more extensively described in section 2.4 and 2.5. The section compares these approaches to the standard NN model in terms of accuracy, error patterns, and runtime efficiency.

To reduce the computational burden, K-means clustering was applied to reduce the number of templates per class before running the Nearest Neighbourhood Classifier. The training set was divided into subsets for each digit, where every element had the same label. Subsequently, we ran the KMeans clustering on each subset, resulting in 64 cluster centroids for each digit. The 640 centroids produced for the entire dataset, was then used as templates in NN.

As a result, the runtime performance was significantly improved, reducing classification time to just 47.13 seconds. Note that this time measures only the NN classification, using 640 clusters as templates. The K-means clustering time is not included in this number. The number of correct predictions was 9523, with 477 incorrect classifications, resulting in an error rate of 4.77%, a slight drop in accuracy compared to the full-template approach 18.

Figure 19 illustrates typical examples of predictions made with the clustered templates. The classifier misclassified the unusually cropped 4, which could resemble the style of a 6. Another instance involves the digit 2 being misclassified as a 3, due to a faint bottom stroke, causing the digit to resemble certain handwritten styles of 3. Similarly, a misclassified 9, featuring a thin loop and downward curl, was interpreted as a 4. These examples highlight the limitations of the K-means clustering. K-means templates are more generalized and may not capture the subtle distinctions in rare or exaggerated writing styles.

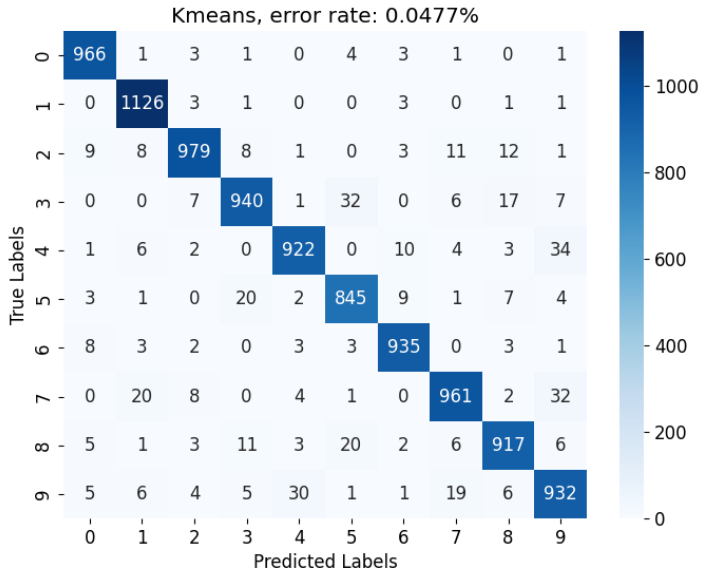

Figure 18: kmeans error rate

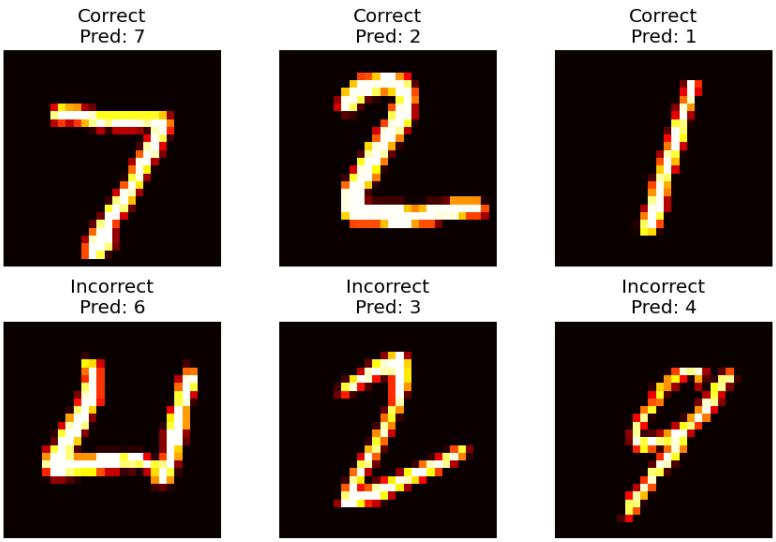

Figure 19: kmeans classification examples

Finally, a k-Nearest Neighbors (k-NN) classifier with $k = 7$ was implemented using the same clustered templates. This model achieved a correct classification count of 9436 out of 10,000 test samples, with an error rate of 5.64% and a runtime of 47.05 seconds. As shown in Figure 20, its accuracy was slightly lower than that of the K-means NN classifier, despite having a similar computation time. The extra cost of sorting and majority voting in k-NN is minimal in this case, because the total number of templates (64) relatively small and $k = 7$ keeps voting simple and efficient. Figure 21 illustrates examples of both correct and incorrect classifications made by the k-NN model.

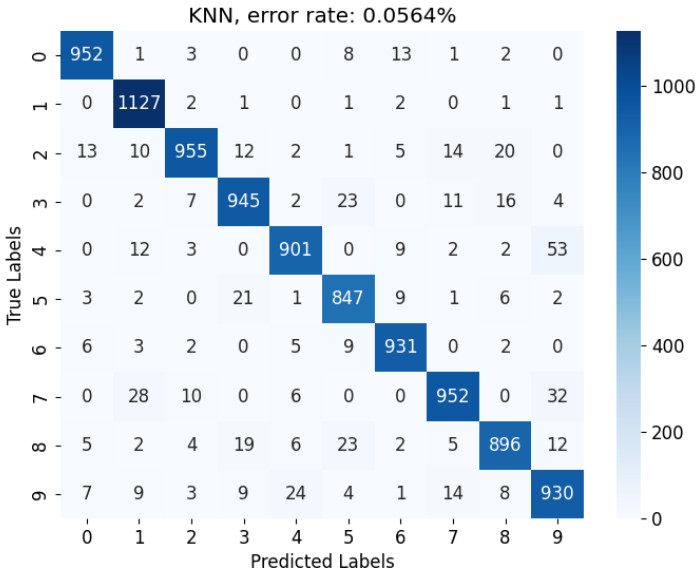

Figure 20: KNN error rate

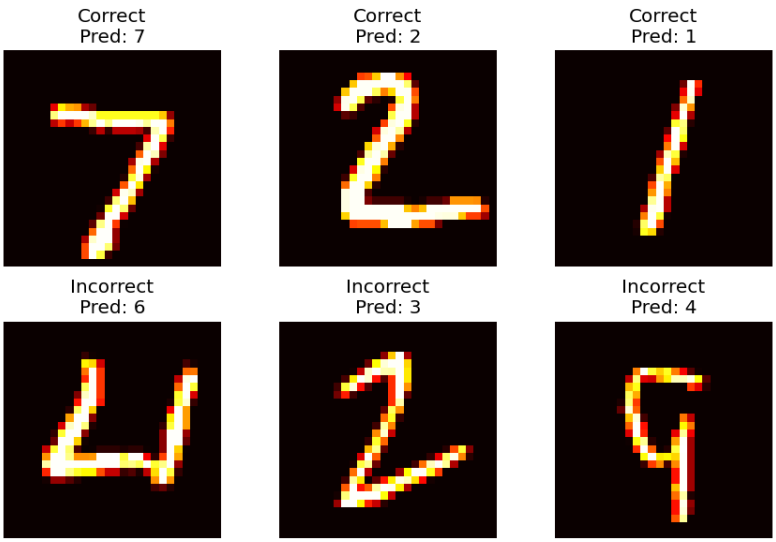

Figure 21: KNN classification examples

In summary, the Nearest Neighbour classifier using all templates provided the best accuracy, while the K-means and k-NN approaches offered competitive performance with significantly lower computational cost. A comparison of the results is shown in Table 1.

Table 1: Comparison of Classifier Performance on the MNIST Test Set

Classifier	**Accuracy (%)**	**Error Rate (%)**	**Runtime (s)**
Nearest Neighbour (Full Templates)	96.91	3.09	7653.60
Nearest Neighbour (K-means Templates)	95.23	4.77	47.13
k-NN Classifier ($k = 7$) (K-means Templates)	94.36	5.64	47.05

5 Conclusion

This section summarizes the theoretical concepts, implementation steps, and evaluation results for both the Iris and MNIST classification tasks. Highlighting the strengths and limitations of each method, and reflects on the lessons learned from model performance and design.

5.1 Summary

This project explored the design, implementation, and evaluation of classifiers for two well-known datasets: Iris and MNIST. Theoretical concepts such as linear classification using minimal square error (MSE) and distance-based methods like Nearest Neighbour (NN), K-means NN, and k-Nearest Neighbours (k-NN) were translated into practical implementations in Python. Through experimentation and visualization, the classifiers were evaluated on both accuracy and computational efficiency.

5.2 Iris

In the Iris task, a linear classifier was trained using the MSE loss and gradient descent. Tests with different training/testing splits showed that classification performance remained consistent as long as the data distribution across classes was balanced. In evaluating feature importance, the removal of non-linearly separable features initially seemed unimportant. However, the tests proved that the removal of the features made the classification task more challenging. With reduced input information, the model required more iterations and higher learning rates to achieve effective training and convergence. Additionally, the performance of the classifier went down, proving that even nonlinearly separable data are of more importance than anticipated.

5.3 MNIST

Through the classification process of the MNIST data it became clear that misclassifications happened due to similar shapes, slanted digits and heavily stylized versions of digits. These type of errors reflect the classifier's reliance on pixel-level similarity and highlight the limitations of using Euclidean distance on raw image data.

The standard NN classifier, using all training samples as templates, achieved the highest accuracy, as it could represent a wider range of digit styles. However, this came at the cost of very long computation time. The K-means NN classifier significantly reduced this cost by clustering the training set into a fixed number of representative centroids. Although notably less accurate, it offered a practical trade-off between speed and precision. The k-NN classifier with $k = 7$ used the same clustered templates as K-means NN, but predicted by majority vote among the nearest neighbors. This added flexibility came with slightly lower accuracy, though runtime remained comparable, as summarized in Table 1.

5.4 Lessons learned

Overall, the tasks were well-designed for understanding core classification concepts. They provided valuable insight into how theoretical ideas translate into working models. A key lesson learned is the importance of balancing model complexity, data representation, and computational efficiency. Even simple classifiers can perform well with the right data and tuning, but performance often depends on careful handling of input features and dataset characteristics.

References

[1] Scikit-learn developers. *MNIST dataset documentation*. Accessed 2025. https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html

[2] Scikit-learn developers. *Iris dataset documentation*. Accessed 2025. https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

[3] Rafael Santos. *The Whole Story About the Iris Data Set*. Instituto Nacional de Pesquisas Espaciais (INPE). Accessed April 22, 2025. http://www.lac.inpe.br/~rafael.santos/Docs/CAP394/WholeStory-Iris.html

[4] A.B. Paudel. *MNIST Sequence and Feature Extraction*. Accessed April 23, 2025. https://abpaudel.com/blog/mnist-sequence-feature-extraction/

[5] Johnsen, M. H. (2017). *Compendium - Part III - Classification*. Lecture Notes, Norwegian University of Science and Technology. December 18, 2017.

A Python Code for Iris Classification Task

Listing 3: Core Iris classification functions used in this project.

```
# Sigmoid function
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def discriminant_vec(data, weights):
    # Add a bias term (column of ones) to the data
    x_aug = np.hstack((data, np.ones((data.shape[0], 1))))
    z = x_aug @ weights.T
    g = sigmoid(z)
    return g

# Gradient of MSE
def grad_MSE(data,weights,targets):
    MSE =0
    g = discriminant_vec(data,weights)
    x_aug = np.hstack((data, np.ones((data.shape[0], 1))))
    error_term = (g - targets) * g * (1 - g)
    MSE = error_term.T @ x_aug
    return MSE

# Remove features by index
def remove_features(X, indices_to_remove):
    return np.delete(X, indices_to_remove, axis=1)

# Split into training and testing sets
def get_data_split(option, X, y):
    if option == 1:
        X_train = np.concatenate([X[:30],X[50:80],X[100:130]])
        y_train = np.concatenate([y[:30],y[50:80],y[100:130]])
        X_test = np.concatenate([X[30:50],X[80:100],X[130:]])
        y_test = np.concatenate([y[30:50],y[80:100],y[130:]])
    elif option == 2:
        X_train = np.concatenate([X[20:50],X[70:100],X[120:]])
        y_train = np.concatenate([y[20:50],y[70:100],y[120:]])
        X_test = np.concatenate([X[:20],X[50:70],X[100:120]])
        y_test = np.concatenate([y[:20],y[50:70],y[100:120]])
    return X_train, y_train, X_test, y_test

# One-hot encode y_train manually
def one_hot_encode(y, num_classes):
    return np.eye(num_classes)[y]

# Predict class labels for the test set
def predict(X, W):
    probs = discriminant_vec(X, W)
    return np.argmax(probs, axis=1)

# Train the model using gradient descent and MSE loss
def fit(X_train, y_train, X_test, y_test, alpha, epochs):
    W = np.zeros((num_classes, X_train.shape[1] + 1))  # Initialize
        weights to zero
    train_loss = np.zeros((epochs,))
    error_rate_train = np.zeros((epochs,))
    error_rate_test = np.zeros((epochs,))
```

```
    for epoch in range(epochs):
        train_loss[epoch] = 0.5 * np.sum((discriminant_vec(X_train, W)
            - y_train) ** 2)
        y_pred = predict(X_test, W)
        y_true = np.argmax(y_test, axis=1)
        error_rate_test[epoch] = 1- accuracy_score(y_true, y_pred,
            normalize=True)
        error_rate_train[epoch] = 1- accuracy_score(np.argmax(y_train,
            axis=1), predict(X_train, W), normalize=True)

        for i in range(X_train.shape[0]):
            x_i = X_train[i].reshape(1, -1)
            t_i = y_train[i].reshape(1, -1)
            W = W - alpha * grad_MSE(x_i, W, t_i)
    return W, train_loss, error_rate_train, error_rate_test
```

B Python Code for MNIST Classification Task

Listing 4: Core MNIST classification functions used in this project.

```
def distance(x,template):
    return np.linalg.norm(x.flatten() - template.flatten())

def nearest_neighbour(X, templates, template_labels, true_labels):
    correct_predictions = []
    incorrect_predictions = []
    predictions = []

    for x, true_label in zip(X, true_labels):  # Iterate over images
       and their true labels
        # Compute distances to all templates
        distances = [distance(x, t) for t in templates]
        index = np.argmin(distances)

        # Determine the predicted label based on the closest template
        predicted_label = template_labels[index]
        predictions.append(predicted_label)
        if predicted_label == true_label:
            correct_predictions.append(true_label)
        else:
            incorrect_predictions.append(true_label)

    return correct_predictions, incorrect_predictions, predictions

def k_nearest_neighbour(X, templates, template_labels, true_labels, k
   =7):
    correct_predictions = []
    incorrect_predictions = []
    predictions = []

    for x, true_label in zip(X, true_labels):  # Iterate over images
       and their true labels
        # Compute distances to all templates
        distances = [distance(x.flatten(), t) for t in templates]
        k_indices = np.argsort(distances)[:k]
        k_labels = [template_labels[i] for i in k_indices]

        # Determine the predicted label using majority voting
        predicted_label = Counter(k_labels).most_common(1)[0][0]
        predictions.append(predicted_label)
        if predicted_label == true_label:
            correct_predictions.append(true_label)
        else:
            incorrect_predictions.append(true_label)

    return correct_predictions, incorrect_predictions, predictions

def create_templates(X_templates, y_templates, num_templates=64):
    templates = []
    template_labels = []
    for digit in range(10):
        class_samples = [x.flatten() for x, y in zip(X_templates,
           y_templates) if y == digit]
        n_clusters = min(num_templates, len(class_samples))
```

```
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        kmeans.fit(class_samples)
        cluster_centers = kmeans.cluster_centers_
        templates.extend(cluster_centers)
        template_labels.extend([digit] * num_templates)
    return np.array(templates), np.array(template_labels)
```