NTNU

Kunnskap for en bedre verden

TTT4275 - Estimation, detection and classification

Classification Project

Authors:
Zahra Parvinashtiani
Philip Shahdadfar

April 30, 2024

Summary

This report was written for the classification project, a part of the course TTT4275 - Estimation, Detection and Classification. The classification project consists of two parts. The first part being the classification of three species of the Iris flower and the second part being the classification of handwritten numbers from 0-9. The classifications is performed by designing classifiers using supervised learning. The classifiers in both parts of this project are first trained by feeding them sets of labeled data, called training sets, and then let it classify an unlabeled test set.

The classifier used for the Iris flower species is a linear classifier. The three types of flowers are classified by discriminating them based on four length and width features. By removing the most overlapping features one at a time, it can test whether the three flowers can be separated linearly. The results of the tests showed that each time a feature is removed, the error rate of the classification increases. In other words the classifier is more accurate, the more information it is given.

The handwritten numbers were classified using two types of a nearest neighbor classifier, NN and KNN. The classification is done by finding the distance between a test sample and the training set. The "nearest neighbor" to the sample determines the class, where the neighbor can be individual points or points grouped in clusters. The error and efficiency is tested for both algorithms to determine the preferred one for this task.

Contents

List of Figures

List of Tables

1 Introduction

This project covers two relevant tasks regarding the classification part of the course. The tasks of classifying closely related flower species and handwritten numbers teach practical ways of classification and give an idea of how one might use classification to solve problems in various fields.

The tasks in this project are the classifications of Iris flowers and handwritten numbers. The iris task has an objective of classifying three variants of the same flower. Each of the three flowers differ in size and will be separated into the correct group by finding a pattern in the size of their leaves. The classification of handwritten numbers will be performed by differentiating pixel images of numbers written by different people and giving each image the correct label (number). This task will be done by comparing how similar a set of numbers are to one another. The data for the iris task comes from a data file with size measurements. The handwritten numbers data is taken from a database called MNIST [1].

The MNIST database is a useful resource as it opens the possibility of digitally reading handwritten numbers and automatically determining which numbers they are in a couple seconds. If this quantity of handwritten numbers were to be classified manually by a person, it would take a long time. Human uncertainty would additionally lead to misclassifications as numbers handwritten by a set of people differ in form, and can be mistaken as another number.

This is the same case for the Iris task. Being able to differentiate the three flowers by a human can be a demanding task. Human classification can lead to errors since the flowers are somewhat similar in appearance and their dimensions miscalculated due to the lack of precision. Automating and digitalizing this process substantially lowers inaccuracy and time consumption.

This project is an example of how classification of available data can improve processes and tasks across various fields. Most people use automated classification algorithms on a daily basis without thinking about it. Smartphones are doing this constantly. By using smartphone cameras, it is easy to scan handwritten text and automatically acquire a digital version. Translating a restaurant menu to another language by simply holding a phone above the menu is an example of how classification can be applied to make difficult tasks easy. The recognition of specific plants from a photo on your phone is also made possible as a result of classifying processed data. Classification can be applied to countless fields for advancement. The goal of this project is give a fundamental introduction to this topic and machine learning.

This project report deals with the two mentioned tasks in multiple sections. The report will begin by introducing the relevant theory in Section 2, in order to design the required classifiers. The classification tasks themselves will be further explained in Section 3. Section 4 will navigate through implemention of the algorithms and final results. Lastly in Section 5 a conclusion will summarize the key findings and what to take away from this project.

2 Theory

This section will present and discuss the main elements and theory necessary in order to fully understand the implementation of the algorithms. The theory and equations explained in this section is taken from the classification compendium, [2], and class lectures [3]. Classification in this course can simply be described as assigning a piece of data to a specific group based on its characteristics. The tasks in this project use characteristic distinguishing algorithms, called classifiers, which are modeled by supervised learning. This type of machine learning is comparable to how humans learn to categorize things from experience. An example could be whether an individual will categorize the taste of a dish as ”good” or ”bad”, by comparing its appearance and smell to previously consumed dishes. The more dishes consumed will increase the likelihood of a ”correct” answer for that individual. In technical terms, supervised learning trains classifiers by using pre-classified sets of information to assign a class to unclassified sets of information. The following subsections will cover classification methods used for the iris task and the handwritten numbers task.

2.1 Linear Classifiers

In classification, there are several types of methods used depending on how the data is structured. The most straightforward method is by linearly separating data, resulting in no errors. Linear separability essentially means that we are able to categorically separate a set of data with a straight line between the available measurements. Figure 2.1 illustrates this problem compared to problems where data can be separated with curves (nonlinear separable problems), and where data cannot be separated by at all (non-separable problems). The linear and nonlinear are in theory able to perform separation without encountering errors. In cases where separation is not possible, errors cannot be avoided as a result of overlapping.

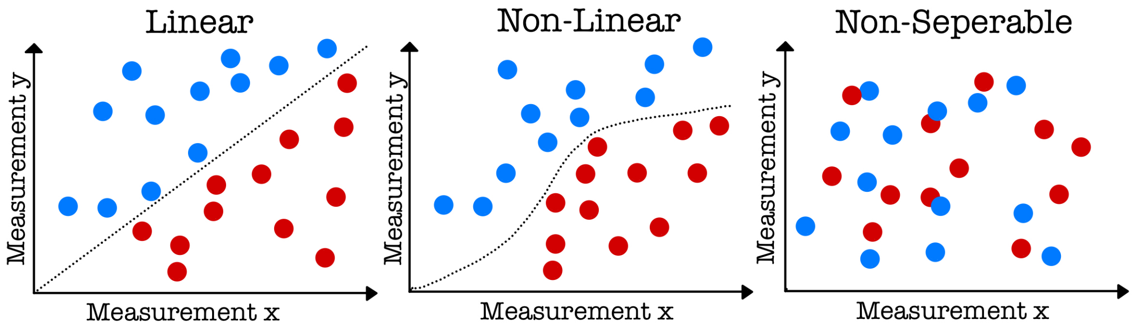


Figure 2.1: Graphical representation of linear separability, nonlinear separability and non-separable cases.

Classifiers that use linear separability are called linear discriminant classifiers (LDC), and defined by a discriminant function

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

with the decision rule

$$g_j(x) = \max_i g_i(x) \tag{2}$$

In $g_i(x)$, x is a vector of feature values, ω_{io} is the offset for class w_i, and i is a class number among the total number of classes C. In cases where the total number of classes C is greater than 2, $g(x)$ and ω_o are vectors dimensioned by C, resulting in a discriminant vector given in Equation 3.

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

This expression can be simplified to

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

if the offset ω_o is integrated into the the matrix W, becoming $W = \begin{bmatrix} W & \omega_o \end{bmatrix}$ and adding a 1 column to the features x, as $x = \begin{bmatrix} x^T & 1 \end{bmatrix}^T$. The simplified function in Equation 4 now holds the form, $g = \begin{bmatrix} W & \omega_o \end{bmatrix} \begin{bmatrix} x^T & 1 \end{bmatrix}^T$. The 1 column added to x is required to maintain the same matrix dimensions.

The next step prior to training the classifier, is to make sure it performs with minimal errors. This is achievable by calculating the difference between the classifier's predicted values and the actual input values. This difference is a reference meant for adjusting the parameters W of the classifier, which in turn improves the accuracy of the next prediction. The function that will be used for this classifier is the Minimum Square Error (MSE) and is shown in Equation 5.

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

MSE has a target vector t_k that indicates the correct class of training sample by returning a 1 for that class and 0 for the rest, $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$ indicating the sample is of the first class. The output vector g_k has a value between 0 and 1 and can be calculated using a squashing function given below. z_k is equal to Wx_k, where x_k is a single input of the sample number k.

$$g_{ik} = sigmoid(x_{ik}) = \frac{1}{1+e^{-z_{ik}}} \qquad i = 1, ..., C \tag{6}$$

The Minimum Square Error depends on the W matrix and can be reduced by adjusting the matrix. In which direction W needs to be adjusted is performed with a gradient method, resulting in the following equation,

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

where

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

Moving W in the opposite direction of the gradient as seen in equation 8,

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

improves the error. $W(m-1)$ is the value of W from the previous iteration, where m is the total number of iterations. To what magnitude we wish to adjust the value of W can be controlled using the step factor α. The step factor is a chosen constant number that acts as a sensitivity for the error reduction. The value of α is decided by trial and error until an appropriate value is found.

2.2 Nearest Neighbor Classifier

Nearest Neighbor (NN) classification is a template based decision rule. Unlike the linear classifier, NN-classifiers don't separate data, but use templates as a reference for the input data. The feature distance between templates and inputs is calculated to find which template has the most similar characteristics(is the "nearest neighbor"). The input is assigned the class of nearest template.

In addition to the NN classifier, a KNN classifier will be designed. The K Nearest Neighbor is a variant of the NN method, where more than one nearest neighbor is used as a reference. K is an integer that decides how many references are to be considered. If K = 3, the nearest three templates are considered and the input gets assigned the class of the majority. A visualization of KNN-classification is depicted in Figure 2.2. For K = 3 there are two references of class A and one of class C, which would result in A classification. For K = 7 and K = 11 however, the input would be given class C and class B, respectively.

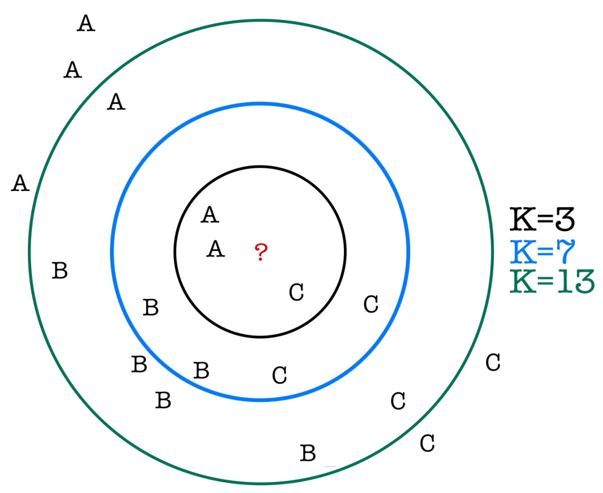


Figure 2.2: K Nearest Neighbors example with cases of K = 3, K = 7 and K = 11.

2.3 Euclidean Distance

In chapter 4 in [4], the partial Euclidean distance is defined as

$$D_r(a,b) = (\sum_{k=1}^{r}(a_k - b_k)^2)^{\frac{1}{2}} \tag{9}$$

where a and b are points in a r dimensioned space.

Equation 10 represents how to compute the squared Euclidean distance between a data point x and a reference point μ_{ik} in vector space. Here both are vectors and notation T denotes the transpose of the vector. Euclidean distance calculates the sum of the vectors x and μ_{ik}, providing a measure of distance without taking the square root.

$$d(x, ref_{ik}) = (x - \mu_{ik})^T(x - \mu_{ik}) \tag{10}$$

2.4 Clustering

Clustering is a method of unsupervised learning that groups a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups [5]. The clustering algorithm used in this paper is K-means which is one of the simplest and most commonly used clustering algorithms. This method divides the dataset into M cluster in which each data belongs to the cluster with the nearest mean.

The process involves randomly initializing k centroids and then choosing between: *(a)* assigning points to the nearest cluster centroid, and *(b)* recalculating the centroids as the mean of the points assigned to each cluster.

3 The classification tasks

This section will briefly introduce the two tasks, explaining the main objectives and the data sets that are used to perform classification. The descriptions of the tasks have been taken from [1].

3.1 Iris

The flowers Iris Setosa, Iris Versicolor and Iris Virginica will be classified from the dimensions of their leaves. The three flower variants will be used as classes in this task. The features, lengths and widths of the flowers' leaves, and their values are included in a data file as columns to their equivalent samples. The flowers have to types of leaves called sepals and petals. Since the features are lengths and widths of the leaves, each sample has the four features: sepal length, sepal width, petal length and petal width. The iris data file has a total of 150 samples, with 50 samples of each class. 90 samples from the iris dataset will be used to train a linear classifier and 60 samples used for testing the performance. The focus is to evaluate linear separability with respect to the features.

3.2 Handwritten numbers

In this task numbers from zero to nine handwritten by 500 different people will be classified through Nearest Neighbor algorithms. The MNIST database contains 60,000 labeled samples and 10,000 unlabeled samples. The samples are greyscale 28x28-pixel pictures with pixel values ranging from 0-255. Since the 60,000 samples are labeled with their corresponding number (class), they will be used for training each classifier. The 10,000 unlabeled samples will be used as a test set of inputs to assess the performance of the different classifiers.

4 Implementation and Results

The implementation of both tasks is explained in Subsection 4.1 and Subsection 4.3. The implementation has a different method for each task as they use different classification strategies, but also due to the choice of using different programming languages. The results of the experiments will be shown after the implementation in form of error rates and confusion matrices.

4.1 Implementation: Iris Task

The implementation of the linear classifier was done using Python in a Google Colab notebook for group collaboration. The choice of Python was due to the ease of reading and handling data from a **iris.data** data file. The following python packages were used:

- **numpy:** for stacking arrays, class mapping and calculations.
- **matplotlib:** for plotting histograms.
- **sklearn:** for displaying the confusion matrices.
- **seaborn:** for improved data visualization of the histogram plots, which are used to find the overlap between the features and classes.

The data is imported into a list of species (label of the iris flowers) and a list of the four feature. These lists are then sorted into 30 training sets and 20 testing sets. For the first experiment, the first 30 samples from each class will be used for training and the last 20 for testing. The step factor α from Equation 8 is tuned is set manually for a chosen number of iterations. For 1000 iterations, the chosen step factor value ended up being 0.01. This was once again tested, but this time the last 30 samples were used for training and the first 20 for testing the classifier.

Linear separability was evaluated by removing the most overlapping feature between the classes and testing the classifier with three features. This again performed with two features and lastly with one feature.

The error rates and confusion matrices for these experiments can be found in the following subsection. The Python code for this task is included in the Appendix Section A and the process is illustrated using a block diagram in Figure 4.1.

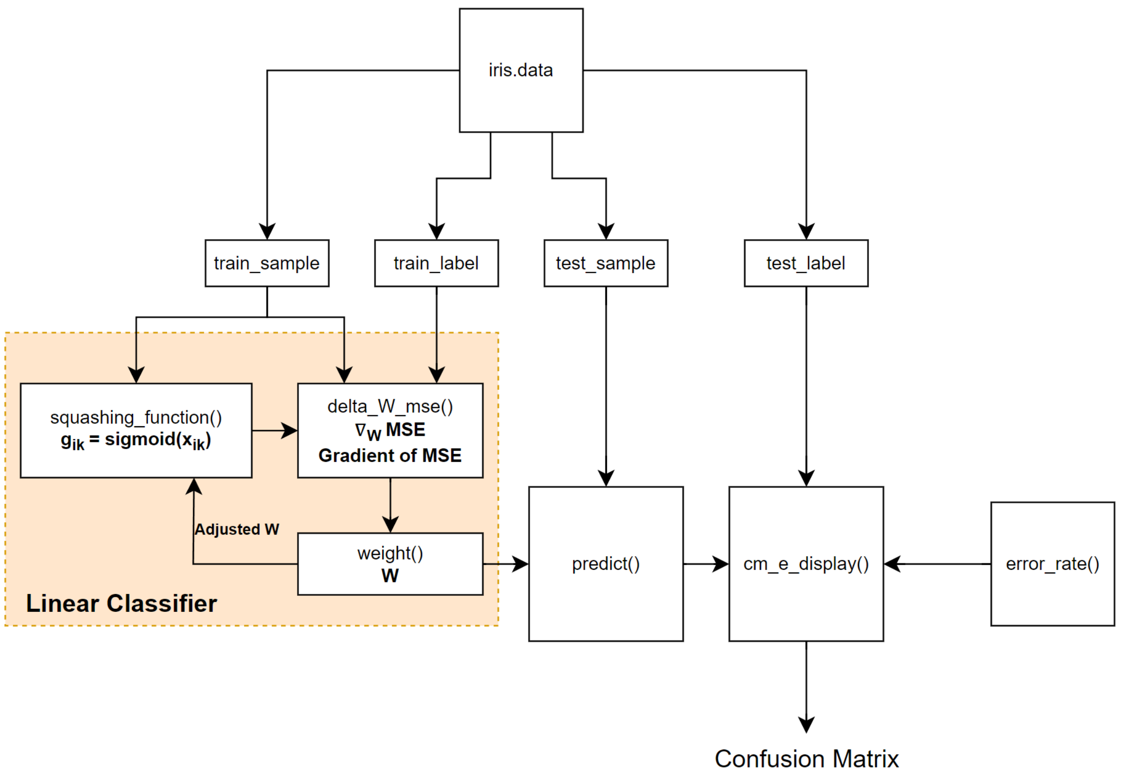


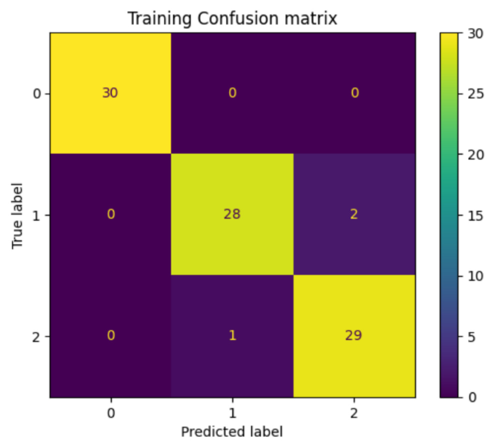
Figure 4.1: Block diagram of the Python implementation.

4.2 Results: Iris Task

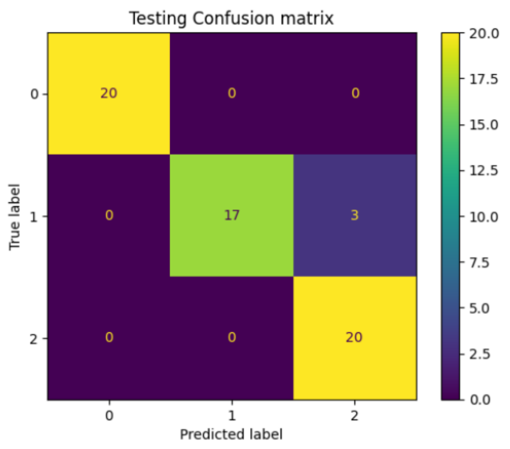
The first experiment resulted in the weighting matrix below.

$$W = \begin{pmatrix} 0.39064598 & 1.50732257 & -2.25165604 & -1.03725747 & 0.27496801 \\ 1.3989633 & -2.6662936 & -0.06956884 & -1.35028362 & 0.88490401 \\ -2.46961764 & -2.01296706 & 3.58374136 & 2.9449683 & -1.33004342 \end{pmatrix}$$

The columns represent each feature and the offset, w_0. The rows represents the three species. The values in the matrix indicate each feature's importance (”weight”) to the corresponding class. In this case the first column is the sepal length and the second row is the Versicolour class. Its value is are greater compared to the other classes in the other rows, thus the classifier knows that for larger sepal lengths the class is most likely Versicolour.



Error rate: 3.3% (3 errors / 90 samples)

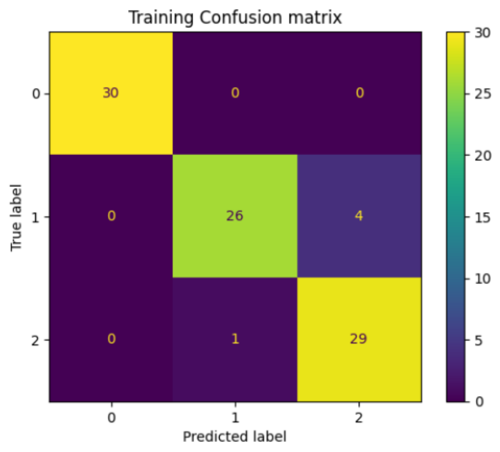


Error rate: 5% (3 errors / 60 samples)

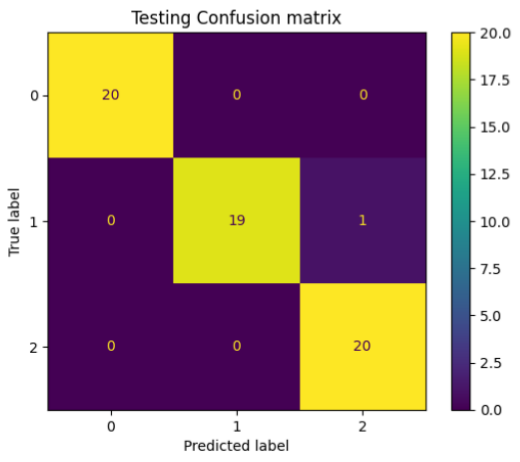
Figure 4.2: Confusion matrices for training (30 first samples) and testing (20 last samples).

Figure 4.2 show the confusion matrices for training and testing data of the first case. Based on the wrong predictions shown outside the main diagonals, the classifier has difficulty differentiating Iris Virginica and Iris Versicolor.

The second experiment resulted in the following error rates and confusion matrices:



Error rate: 5.6% (5 errors / 90 samples)



Error rate: 1.7% (1 error / 60 samples)

Figure 4.3: Confusion matrices for training (30 last samples) and testing (20 first samples).

The results of the two experiments show that the second case had a higher training error rate, but a lower testing error rate compared to the first case. Despite a improved testing error rate, there are too few samples in the dataset to conclude with which training case is better.

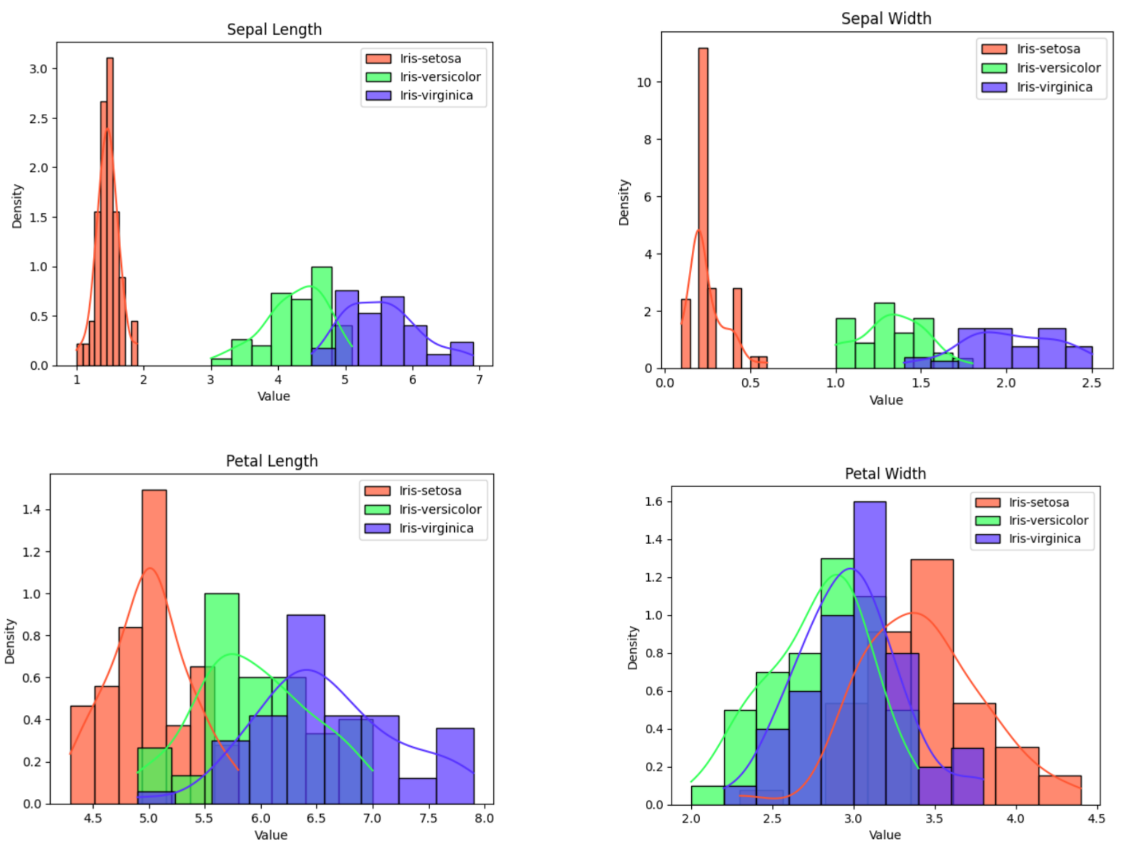
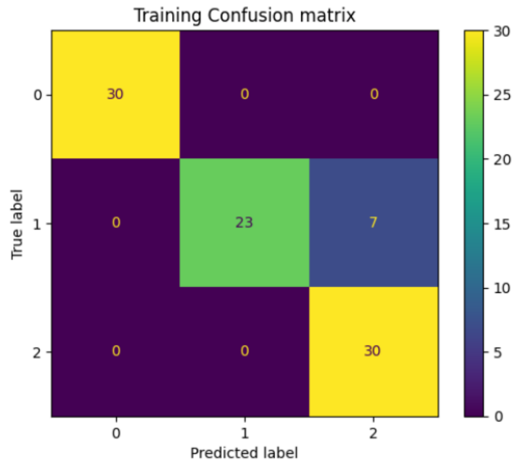


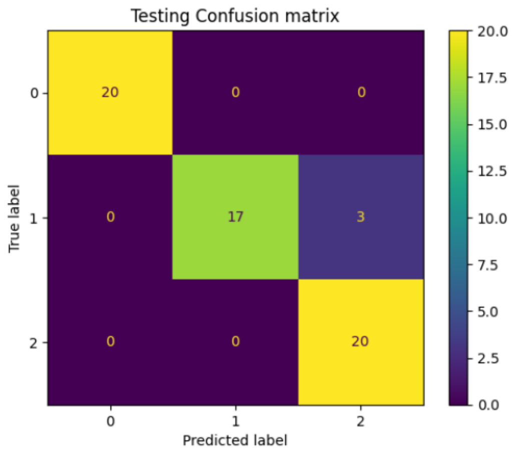
Figure 4.4: Histograms of each feature and classes.

The histograms in Figure 4.4 show the overlap of the features of each class. The overlap makes more difficult to distinguish the different classes by the feature. We can see that the petal width has most overlap between the classes. By removing this feature from training and test data, the following confusion matrices and error rates were achieved.



	0	1	2
0	30	0	0
1	0	23	7
2	0	0	30

Error rate: 7.8% (7 errors / 90 samples)

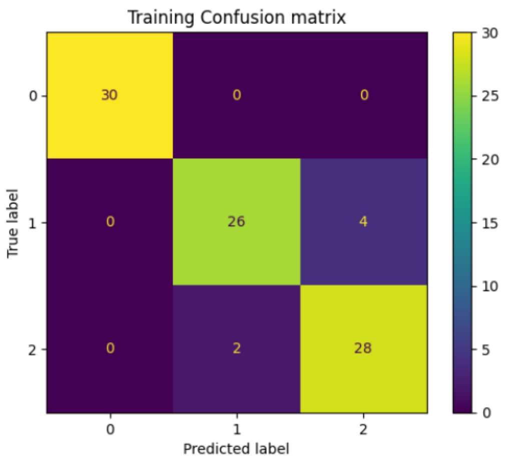


	0	1	2
0	20	0	0
1	0	17	3
2	0	0	20

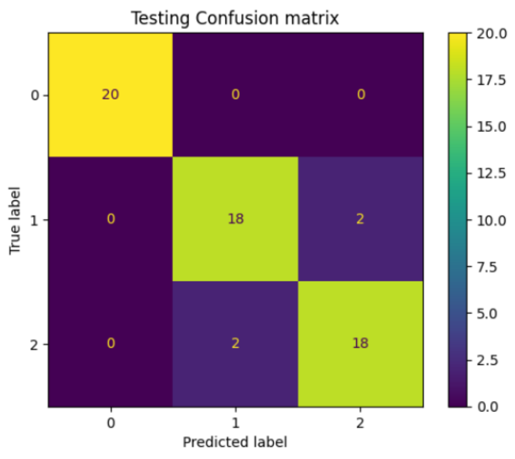
Error rate: 5.0% (3 errors / 60 samples)

Figure 4.5: Training and testing confusion matrices using the remaining three features. In this experiment training uses the first 30 samples and testing used the last 20 samples of each class.

The next experiment is to remove the next most overlapping features until one feature remains. The most overlapping features after petal width are the petal length and sepal length. The error rates and confusion matrices for these cases can be seen in Figures 4.6 and 4.7, respectively.

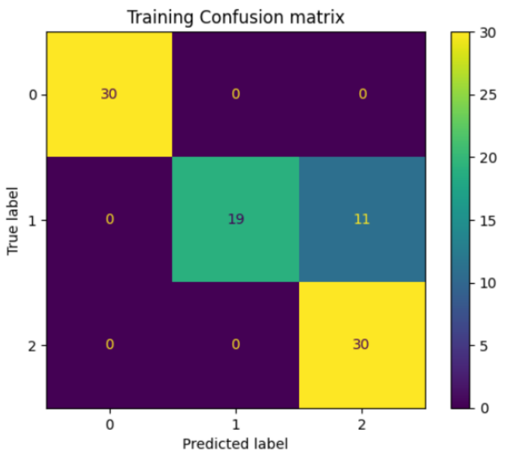


Error rate: 6.7%

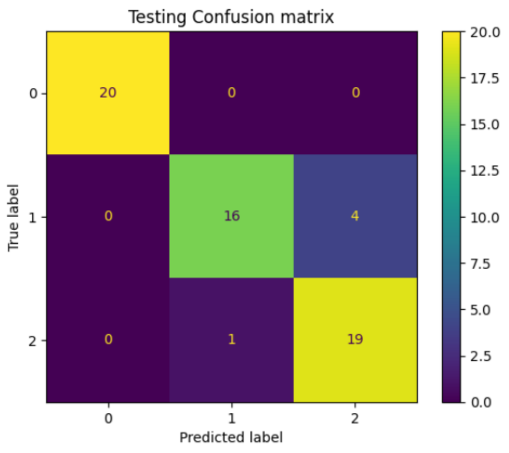


Error rate: 6.7%

Figure 4.6: Confusion matrices and error rates after removing the petal length.



Error rate: 12.2%



Error rate: 8.3%

Figure 4.7: Confusion matrices and error rates while only using sepal width as a feature (sepal length removed).

Due to Iris Setosa being distant from the other classes in sepal length and sepal width, the classifier can easily classify the flower in all cases, due to linear separability. For the other two classes, removing features doesn't allow a complete linear separation as Iris Versicolor and Iris Virginica do overlap slightly in the sepal and petal lengths. Removing the petal length and width seems to somewhat decrease the classifier's accuracy compared to when all features were used. While only training with the sepal width the misclassification rates have noticeably worsened. For these two classes, the sepal lengths are more distant compared to the sepal widths, which can be seen in the horizontal value axis of the first two histograms. This explains why the misclassification increases. In terms of performance, the time consumption for the different cases is minimal, which is why it appears all 4 features should be used for optimal classification.

4.3 Implementation: Handwritten Numbers Task

The data for this task was loaded into Matlab from the **data_all.mat** file. The choice of Matlab for this task is due to the large size of samples. Matlab is more a powerful language than Python when it comes to handling large sizes of data. Since 70,000 pictures will be processed and Matlab already has many useful preinstalled functions, it was more suitable for this task. The toolboxes used are the **Deep Learning Toolbox** for clustering. The NN and KNN classifiers are programmed as functions. The code files can be found in the Appendix Section B.

The first part of the is to design the NN classifier. The datasets from the MNIST database is split into chunks of 1000 to calculate matrices effectively by not using very long distances. The function **pdist2()** from the **Statistics and Machine Learning Toolbox** calculates the Euclidean distance between the set of samples. The confusion matrix was plotted using the function confusionchart().

In the second part, the performance of clusters is evaluated by clustering 6000 training vectors into 64 clusters and then compared to a KNN classifier with 7 nearest neighbors ($K = 7$). Figures 4.8, 4.9 and 4.10 show flowcharts of the algorithms.

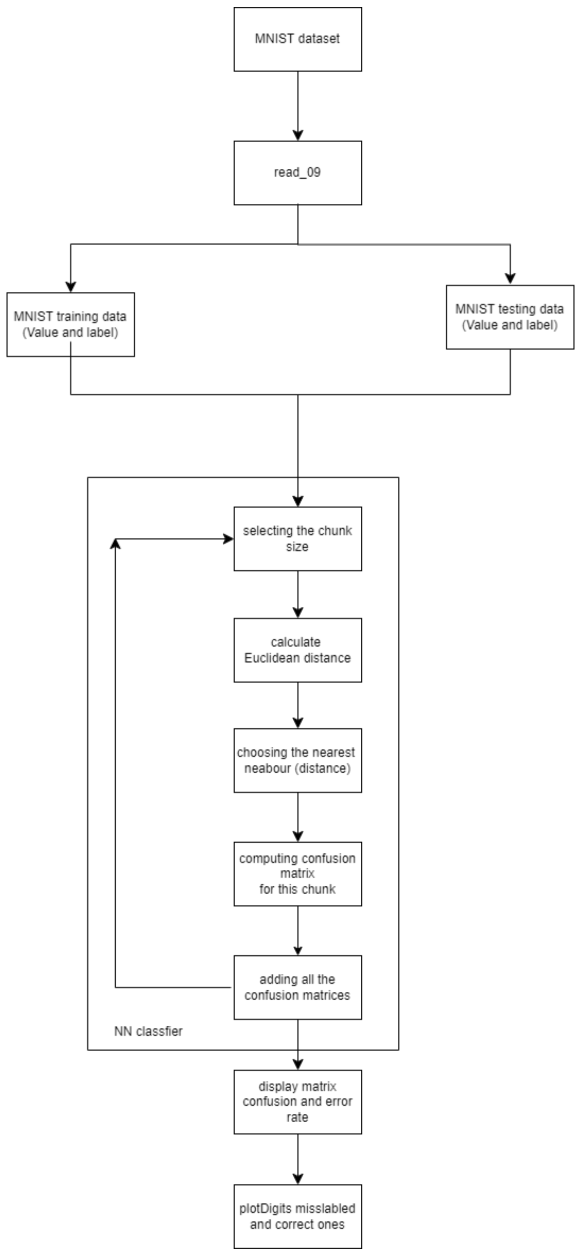

MNIST dataset
read_09
MNIST training data (Value and label)
MNIST testing data (Value and label)
selecting the chunk size
calculate Euclidean distance
choosing the nearest neabour (distance)
computing confusion matrix for this chunk
adding all the confusion matrices
NN classfier
display matrix confusion and error rate
plotDigits misslabled and correct ones

Figure 4.8: Flowchart of NN-classifier.

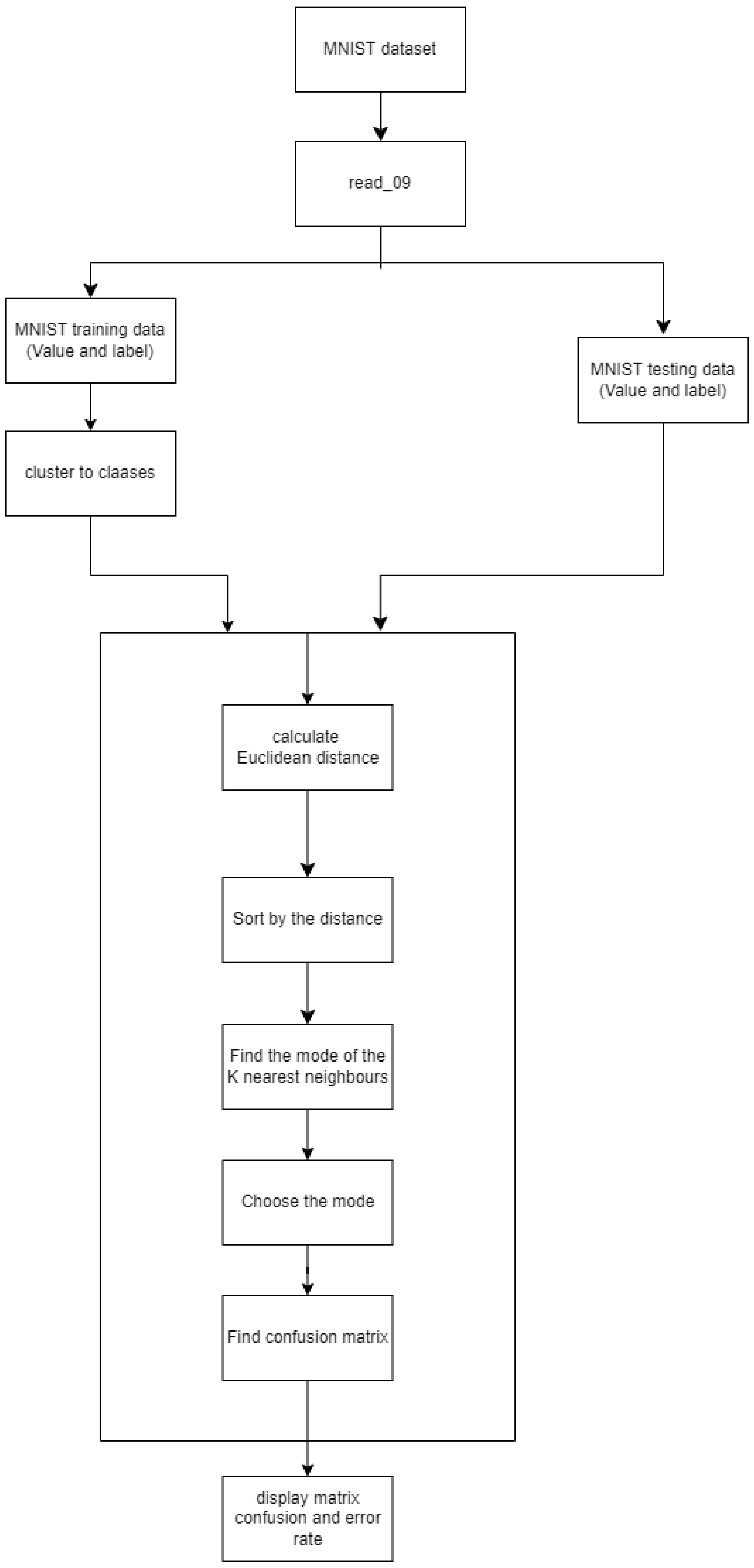

MNIST dataset
read_09
MNIST training data (Value and label)
MNIST testing data (Value and label)
cluster to claases
calculate Euclidean distance
Sort by the distance
Find the mode of the K nearest neighbours
Choose the mode
Find confusion matrix
display matrix confusion and error rate

Figure 4.9: Flowchart of K-nearest neighbors.

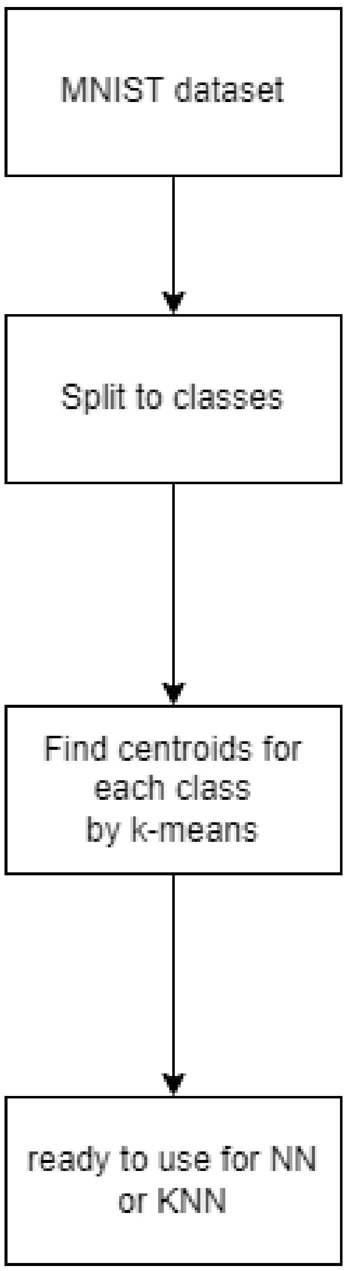
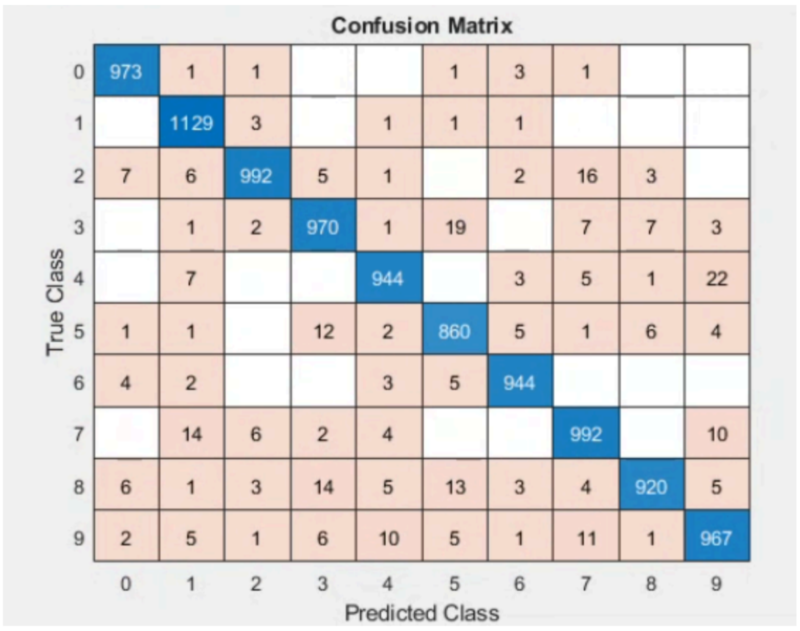

MNIST dataset
Split to classes
Find centroids for each class by k-means
ready to use for NN or KNN

Figure 4.10: Flowchart of clustering.

4.4 Results: Handwritten Numbers

The confusion matrix in the figure below shows that the NN classifier without clustering classifies majority of the digits correctly with an error rate of 3.09% in 8.878 seconds. The digit 8 was the hardest to predict as a consequence of overlap with other digits, especially with 3 and 5.

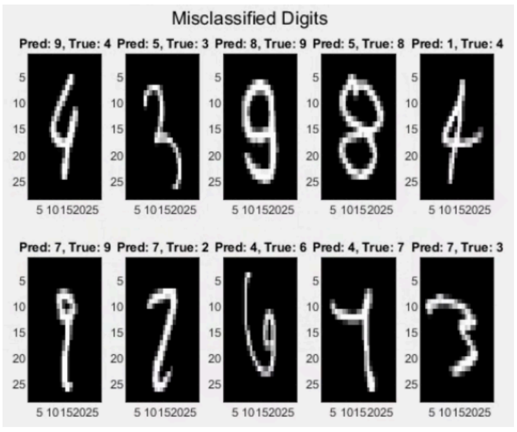


True Class \ Predicted Class	0	1	2	3	4	5	6	7	8	9
0	973	1	1			1	3	1		
1		1129	3		1	1	1			
2	7	6	992	5	1		2	16	3	
3		1	2	970	1	19		7	7	3
4		7			944		3	5	1	22
5	1	1		12	2	860	5	1	6	4
6	4	2			3	5	944			
7		14	6	2	4			992		10
8	6	1	3	14	5	13	3	4	920	5
9	2	5	1	6	10	5	1	11	1	967

Error rate: 3.09%

Figure 4.11: Confusion matrix for MNIST testing dataset

In figures 4.12 a set of correctly classified and misclassified numbers is revealed. Mistaking numbers such as 2 for a 7 is a regular occurrence due to their similarity in shape. The same goes for numbers 8 and 9, 1 and 7 and most commonly, as seen above, mistaking 4 and 9. For the correctly classified digits, the pictures are a lot clearer and easier to differentiate. This demonstrates the significance of the precision required in order to correctly classify numbers written by hundreds of individuals.



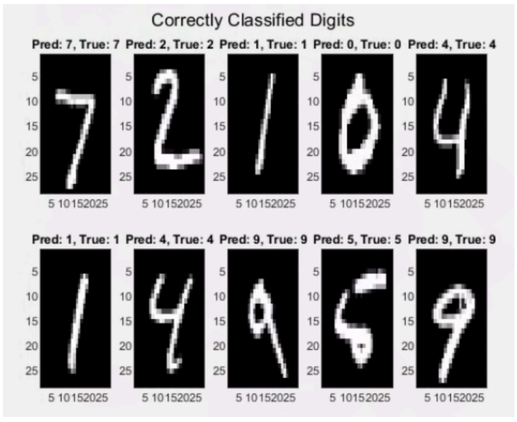
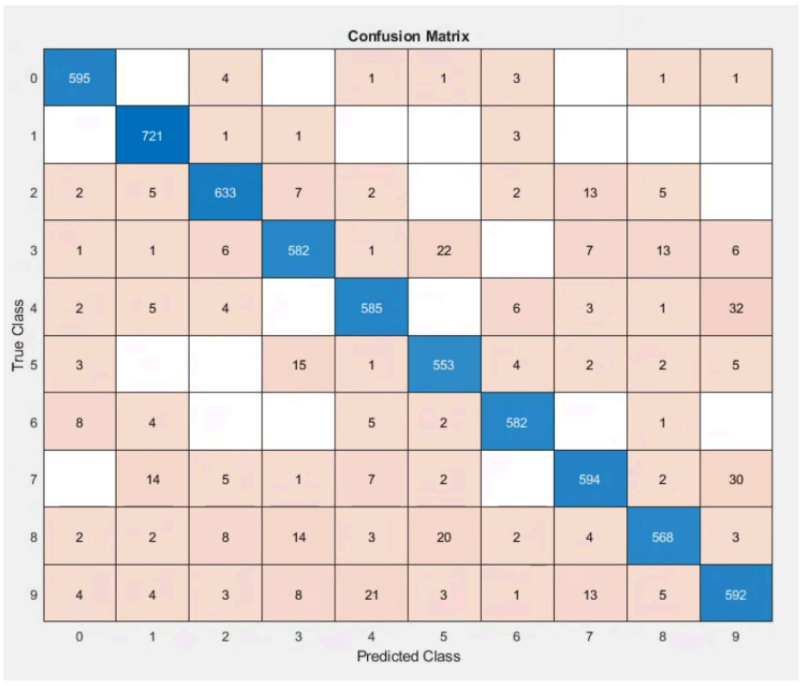


Figure 4.12: Correctly classified and misclassified digits using nearest neighbour.

By clustering the training data, the classification time is decreased to 4.411 seconds giving the confusion matrix:


Confusion Matrix

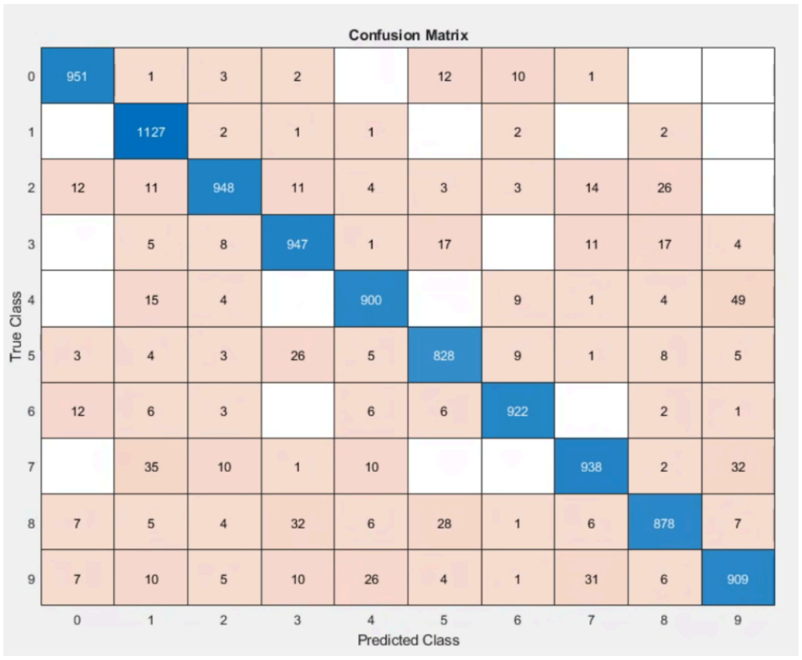
True Class / Predicted Class

	0	1	2	3	4	5	6	7	8	9
0	595		4		1	1	3		1	1
1		721	1	1			3			
2	2	5	633	7	2		2	13	5	
3	1	1	6	582	1	22		7	13	6
4	2	5	4		585		6	3	1	32
5	3			15	1	553	4	2	2	5
6	8	4			5	2	582		1	
7		14	5	1	7	2		594	2	30
8	2	2	8	14	3	20	2	4	568	3
9	4	4	3	8	21	3	1	13	5	592

Error rate: 6.17%

Figure 4.13: Confusion matrix for MNIST testing dataset after clustering using NN.

Although the time is halved, the error rate has doubled after clustering. The cause of this is the loss of information as the number of templates has been reduced. Instead of calculating the distance for every individual, the nearest mean distance is used, which leads to the lack of precision. The final experiment is to classify the numbers to the nearest 7 neighbors using clustering. The confusion matrix in Figure 4.14 shows that the KNN classifier ended up performing similarly to the NN classifier with a time of 4.819 seconds and error rate of 6.52%.


Confusion Matrix

True Class / Predicted Class

	0	1	2	3	4	5	6	7	8	9
0	951	1	3	2		12	10	1		
1		1127	2	1	1		2		2	
2	12	11	948	11	4	3	3	14	26	
3		5	8	947	1	17		11	17	4
4		15	4		900		9	1	4	49
5	3	4	3	26	5	828	9	1	8	5
6	12	6	3		6	6	922		2	1
7		35	10	1	10			938	2	32
8	7	5	4	32	6	28	1	6	878	7
9	7	10	5	10	26	4	1	31	6	909

Error rate: 6.52%

Figure 4.14: Confusion matrix for MNIST testing dataset after clustering using KNN

Although the nearest neighbour classifier (NN) without clustering correctly classified more digits, it required more time than the two other cases. Therefore, a choice needs to be made between an accurate classifier or an efficient classifier. The choice between minimal errors or rapidly classifying depends on the specific application. For this MNIST dataset and the code, the time variations were insignificant, which is why the choice of using the NN classifier without clustering appears to be the preferred option. For the sake of verifying the classification time, this task was repeated on another computer. The accuracy for all three cases were similar, but the the time consumption was longer. The error rates and classification time of the NN without clustering, NN with clustering and KNN with clustering were respectively, 3.09% and 79.984 seconds, 5.89% and 15.675 seconds, and lastly 6.52% and 13.480 seconds. The console output for the repeated task can be found in Appendix Section B.2.

5 Conclusion

In conclusion, we successfully addressed two distinct classification tasks, achieving satisfactory outcomes in both. The classifiers were implemented in Python and Matlab, and the data files for both tasks were downloaded from Blackboard under learning materials.

For the Iris classification, a Linear Discriminant Classifier was designed, implemented and trained. The classifier used four features and 90 samples for training, and achieved low error rates ranging from 1.7% to 5.6%. Additionally, an error rate of 8.3% using only one feature was achieved, which was considered satisfactory considering the overlap. When fewer features were used, the error rates increased and the runtime remained roughly the same. Thus, it is preferable to use all available features. Considering the small volume of data, the results are considered to be acceptable.

In the classification of MNIST handwritten digits in which the database consisted of 60 000 training samples and 10,000 samples for testing, a Template Based Classifier combined with the NN and KNN algorithm for K=7 was utilized, in cases of with and without clustering. Clustering led to significant reductions in processing time with the cost of with accuracy. In this task, the conclusion was that the first classifier with no clustering is preferred as it ended up with better accuracy, for a slightly increased processing time.

References

[1] Magne H. Johnsen. *Project descriptions and tasks in classification (Released on Blackboard).* Feb. 2018.

[2] Magne H. Johnsen. *Classification Compendium, (Blackboard Learning Materials: "Compendium - Part III - Classification").* Dec. 2017.

[3] Pierluigi Salvo Rossi Department of Electronic Systems NTNU. *Class Lectures, TTT4275 - Estimation, Detection and Classification.* 2024.

[4] D.G. Stork R.O. Duda P.E. Hart. *Pattern Classification.* Nov. 2012.

[5] Guojun Gan. *Data Clustering in C++.* 2011.

A Appendix A: IRIS

A.1 Code

iris.py

```python
# -*- coding: utf-8 -*-
"""iris.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1e4e12G0wq1dotNUKyxcAAGxCE0ElAIhQ
"""

import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
import seaborn as sns

from google.colab import drive
drive.mount('/content/drive')

class_mapping = {
    'Iris-setosa': np.array([1, 0, 0]),
    'Iris-versicolor': np.array([0, 1, 0]),
    'Iris-virginica': np.array([0, 0, 1])
}

file_path = 'Iris_TTT4275/iris.data'

def import_data():
    feature_list = []
    species_list = []
    with open(file_path, 'r') as data:
        rows = data.readlines()
        for row in rows:
            row_values = row.strip().split(',')
            features = [float(i) for i in row_values[:-1]]
            species = row_values[-1]
            feature_list.append(features)
            features.append(1.0)
            species_list.append(class_mapping.get(species))
    feature_array = np.stack(feature_list)
    species_array = np.stack(species_list)
    return feature_array, species_array

#Predicttion of each class returned as a vector
def squashing_function(iris_input, W):
    exponent = np.array([ np.exp(-(np.matmul(W, i))) for i in iris_input])
    sigmoid = 1/(1 + exponent)
    return sigmoid

```

```python
#Minimum Square Error function
def mse(g_k, t_k):
    error = sum((g_k-t_k)**2)/2
    return error

# Weight Matrix
def weight(previous, alpha, grad_mse):
    W = previous - alpha*grad_mse.T
    return W

from google.colab import drive
drive.mount('/content/drive')

#Gradient MSE
def delta_W_mse(g_k, t_k, iris_input):

    delta_zk_g = g_k * (1 - g_k)
    delta_gk_mse = g_k - t_k

    grad = np.dot(iris_input.T, delta_gk_mse * delta_zk_g)

    return grad

#Training linear classifier
def linear_classifier(samples,t_k , iterations=1000,alpha=0.01):

    mse_values = []
    W = np.zeros((3 , samples.shape[1]))
    for i in range(iterations):
      g_k = squashing_function(samples, W)
      grad_mse = delta_W_mse(g_k,t_k,samples)
      W = weight(W, alpha, grad_mse)
      mse_values.append(mse(g_k, t_k))

    return mse_values, W

def predict(X, W):
    probabilities = squashing_function(X, W)
    predictions = np.argmax(probabilities, axis=1)
    return predictions

def error_rate(confusion_matrix):
  pp_sum = 0
  for i in range(len(confusion_matrix)):
    pp_sum += confusion_matrix[i, i]
  error = 1 - pp_sum / np.sum(confusion_matrix)
  #print(f'error rate = {100 * error:.1f}%')
  return error

def cm_e_display(s,l,W):
  predicted = predict(s,W)
  actual = np.argmax(l, axis=1)

  confusion_matrix = metrics.confusion_matrix(actual, predicted)
  e = error_rate(confusion_matrix)
  cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
  cm_display.plot()
  if actual.shape[0] <  65 :
      plt.title('Testing Confusion matrix')
  else:
      plt.title('Training Confusion matrix')

  plt.show()
  print(f'error rate = {100 * e:.1f}%')

features, labels = import_data()

train_sample = []
test_sample = []

train_label = []
test_label = []

```

```python
train_sample.extend(features[0:30])
train_sample.extend(features[50:80])
train_sample.extend(features[100:130])

test_sample.extend(features[30:50])
test_sample.extend(features[80:100])
test_sample.extend(features[130:150])

train_label.extend(labels[0:30])
train_label.extend(labels[50:80])
train_label.extend(labels[100:130])

test_label.extend(labels[30:50])
test_label.extend(labels[80:100])
test_label.extend(labels[130:150])

t_l = np.stack(train_label)
t_s = np.stack(train_sample)
test_s = np.stack(test_sample)
test_l = np.stack(test_label)

mse_value, W = linear_classifier(t_s, t_l)

cm_e_display(test_s,test_l,W)

cm_e_display(t_s,t_l,W)

"""Part d: 20 first sampling and 30 last for training"""

features, labels = import_data()

tr_sample = []
te_sample = []

tr_label = []
te_label = []

tr_sample.extend(features[20:50])
tr_sample.extend(features[70:100])
tr_sample.extend(features[120:150])

te_sample.extend(features[0:20])
te_sample.extend(features[50:70])
te_sample.extend(features[100:120])

tr_label.extend(labels[20:50])
tr_label.extend(labels[70:100])
tr_label.extend(labels[120:150])

te_label.extend(labels[0:20])
te_label.extend(labels[50:70])
te_label.extend(labels[100:120])

t_l4 = np.stack(tr_label)
t_s4 = np.stack(tr_sample)
test_s4 = np.stack(te_sample)
test_l4 = np.stack(te_label)

mse_value4, W4 = linear_classifier(t_s, t_l)

cm_e_display(test_s4,test_l4,W4)

cm_e_display(t_s4,t_l4,W4)

"""# Part 2"""

color_palette = ['#FF5733', '#33FF57', '#5733FF']

# Define feature names
feature_names = {
    0: "Petal Length",
    1: "Petal Width",
    2: "Sepal Length",
    3: "Sepal Width",
    4: "Extra 1 feature"  # If you have an extra feature
}
# Plot histograms with KDE lines for each feature and species
for i in range(features.shape[1]):
    plt.figure()
    for idx, (species, color) in enumerate(zip(class_mapping.keys(), color_palette)
    ):
        data = features[labels[:, class_mapping[species] == 1][:, 0] == 1, i]
        sns.histplot(data, color=color, label=species, kde=True, alpha=0.7, stat='
    density')
    plt.title(f'{feature_names[i]}')
    plt.xlabel('Value')
    plt.ylabel('Density')
    plt.legend()
    plt.show()

ts_3f = np.delete(t_s, 1, 1)
test_s_3f = np.delete(test_s, 1, 1)
mse_value_3f, W3f = linear_classifier(ts_3f, t_l)

cm_e_display(test_s_3f,test_l,W3f)
cm_e_display(ts_3f,t_l,W3f)

"""with 2 features"""

ts_2f = np.delete(ts_3f, 0, 1)
test_s_2f = np.delete(test_s_3f, 0, 1)
mse_value_2f, W2f = linear_classifier(ts_2f, t_l)

cm_e_display(test_s_2f,test_l,W2f)
cm_e_display(ts_2f,t_l,W2f)

"""
with 1 feature"""

ts_1f = np.delete(ts_2f, 0, 1)
test_s_1f = np.delete(test_s_2f, 0, 1)
mse_value_1f, W1f = linear_classifier(ts_1f, t_l)

cm_e_display(test_s_1f,test_l,W1f)
cm_e_display(ts_1f,t_l,W1f)
```

B Appendix B: Handwritten Numbers

B.1 Code

e1a.m

```matlab
tic

load data_all.mat;

%Set chunk size
chunkSize = 1000;
%Number of chunks per chunk size
numChunks = ceil(size(testv, 1) / chunkSize);

%% N-Classifier using the Euclidian distance
[totalPredictions, totalC] = NN_classifier(trainv, trainlab, testv, testlab,
    numChunks, chunkSize);


%% Confusion Matrix and Error rate for the test set

disp(totalC);
%All diagonal (correct) in the matrix
totalCorrect = trace(totalC);
```

```matlab
%Sum of all samples in matrix
totalSamples = sum(totalC, 'all');
%Error rate for test
errorRate = 1 - (totalCorrect / totalSamples);

confusionMatrix_plot = confusionchart(totalC,{'0','1','2','3','4','5','6','7','8','
    9'});
confusionMatrix_plot.Title = 'Confusion Matrix';

fprintf('Total error rate: %.2f%%\n', errorRate * 100);




misclassifiedIndices = find(testlab ~= totalPredictions);
correctlyClassifiedIndices = find(testlab == totalPredictions);

% Plot misclassified and correctly classified images
plotDigits('Misclassified Digits', misclassifiedIndices, testv, totalPredictions,
    testlab);
plotDigits('Correctly Classified Digits', correctlyClassifiedIndices, testv,
    totalPredictions, testlab);


toc
```

NNclassifier.m

```matlab
%NN classifier

function [totalPredictions, totalC] = NN_classifier(trainingValue, trainingLabel,
    testValue, testLabel, numChunks, chunkSize)

        totalPredictions = [];
        %Confusion Matrix dimensions for all 10 written numbers
        totalC = zeros(10, 10);

        %Split up the chunks
    for k = 1:numChunks
        startIdx = (k - 1) * chunkSize + 1;
        endIdx = min(k * chunkSize, size(testValue, 1));

        %Chunk selection
        testImages = double(testValue(startIdx:endIdx, :));
        testLabels = testLabel(startIdx:endIdx);

        %Transpose
        trainImages = double(trainingValue);
        trainLabels = trainingLabel;

        %Euclidean distance
        Distance = pdist2(testImages, trainImages, 'euclidean');

        %Classify test image from nearest training image
        [~, minIndex] = min(Distance, [], 2);
        predictedLabel = trainLabels(minIndex);
        totalPredictions = [totalPredictions; predictedLabel];

        %Compute the confusion matrix for selected chunk
        C = confusionmat(testLabels, predictedLabel);



        %Accumulate all chunks
        totalC = totalC + C;

    end
```

plotDigits.m

```matlab
%Plotting
function plotDigits(titleStr, indices, testSet, estimatedLabels, actualLabels)
    figure;
    colormap(gray);
    sgtitle(titleStr);
    for i = 1:min(10, length(indices))
        subplot(2, 5, i);
        x = zeros(28, 28);
        x(:) = testSet(indices(i), :);
        image(x');
        title(sprintf('Pred: %d, True: %d', estimatedLabels(indices(i)),
    actualLabels(indices(i))));
    end
end
```

e2a.m

```matlab

tic

load data_all.mat;

%% 64 Clusters for each class
M = 64;
numClasses = 10;

%% Clustering

% Initialize classes for clustering
idxAllClasses = cell(numClasses, 1);
centroidsAllClasses = cell(numClasses, 1);

for i = 0:9
    % Select the training vectors for class i
    trainvClass = trainv(trainlab == i, :);

    % Perform k-means clustering
    [idxi, Ci] = kmeans(trainvClass, M);

    % Store the index and centroids
    idxAllClasses{i+1} = idxi;
    centroidsAllClasses{i+1} = Ci;

end

%Flatten cluster data
flattenedCentroids = zeros(M * numClasses, 784);
for i = 1:numClasses
    startRow = (i - 1) * M + 1;
    endRow = i * M;
    flattenedCentroids(startRow:endRow, :) = centroidsAllClasses{i};
end


%% Train label for cluster data 0-9 classes
labels = 0:9;
% Repeat each label 64 times
labelVector = repmat(labels, M, 1);
labelVector = labelVector(:);

%Classify with NN-classifier
[totalPredictions, totalC] = NN_classifier(flattenedCentroids, labelVector, testv,
    testlab, numChunks, M*numClasses);

%% Confusion Matrix Results
disp(totalC);
%All diagonal (correct) in the matrix
totalCorrect = trace(totalC);
%Sum of all samples in matrix
totalSamples = sum(totalC, 'all');
%Error rate for test
errorRate = 1 - (totalCorrect / totalSamples);

confusionMatrix_plot = confusionchart(totalC,{'0','1','2','3','4','5','6','7','8','
    9'});
confusionMatrix_plot.Title = 'Confusion Matrix';
```

```matlab

fprintf('Total error rate: %.2f%%\n', errorRate * 100);


toc
```

e2b.m

```matlab
tic

load data_all.mat;

%Set chunk size
chunkSize = 1000;
%Number of chunks per chunk size
numChunks = ceil(size(testv, 1) / chunkSize);

M=64;

%% N-Classifier using the Euclidian distance
[totalPredictions, totalC] = NN_class(trainv, trainlab, testv, testlab, numChunks,
    chunkSize, M);


%% Confusion Matrix and Error rate for the test set

disp(totalC);
%All diagonal (correct) in the matrix
totalCorrect = trace(totalC);
%Sum of all samples in matrix
totalSamples = sum(totalC, 'all');
%Error rate for test
errorRate = 1 - (totalCorrect / totalSamples);

confusionMatrix_plot = confusionchart(totalC,{'0','1','2','3','4','5','6','7','8','
    9'});
confusionMatrix_plot.Title = 'Confusion Matrix';

fprintf('Total error rate: %.2f%%\n', errorRate * 100);




misclassifiedIndices = find(testlab ~= totalPredictions);
correctlyClassifiedIndices = find(testlab == totalPredictions);

% Plot misclassified and correctly classified images
plotDigits('Misclassified Digits', misclassifiedIndices, testv, totalPredictions,
    testlab);
plotDigits('Correctly Classified Digits', correctlyClassifiedIndices, testv,
    totalPredictions, testlab);


toc
```

e2c.m

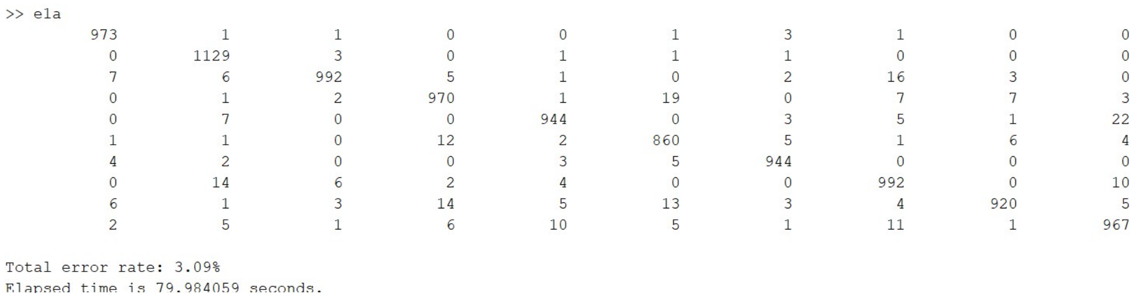
```matlab

tic

load data_all.mat;


M = 64;

numClasses = 10;

% Initialize cell arrays to hold the indices and centroids for each class.
idxAllClasses = cell(numClasses, 1);
centroidsAllClasses = cell(numClasses, 1);

% Perform clustering for each class.
for i = 0:numClasses-1
```

```matlab
    % Select the training vectors for class i
    trainvClass = trainv(trainlab == i, :);

    % Perform k-means clustering
    [idxi, Ci] = kmeans(trainvClass, M);

    % Store the indices and centroids
    idxAllClasses{i+1} = idxi;
    centroidsAllClasses{i+1} = Ci;

end


numFeatures = 784;
flattenedCentroids = zeros(M * numClasses, numFeatures);

for i = 1:numClasses
    startRow = (i - 1) * M + 1;
    endRow = i * M;
    flattenedCentroids(startRow:endRow, :) = centroidsAllClasses{i};
end


labels = 0:9;  % The labels you want to repeat
labelVector = repmat(labels, M, 1);  % Repeat each label 64 times
labelVector = labelVector(:);

[totalPredictions, totalC] = KNN_classifier(flattenedCentroids, labelVector, testv,
     testlab, 7);

disp(totalC);
%All diagonal (correct) in the matrix
totalCorrect = trace(totalC);
%Sum of all samples in matrix
totalSamples = sum(totalC, 'all');
%Error rate for test
errorRate = 1 - (totalCorrect / totalSamples);

confusionMatrix_plot = confusionchart(totalC,{'0','1','2','3','4','5','6','7','8','
    9'});
confusionMatrix_plot.Title = 'Confusion Matrix';

fprintf('Total error rate: %.2f%%\n', errorRate * 100);


toc
```

KNNclassifier.m

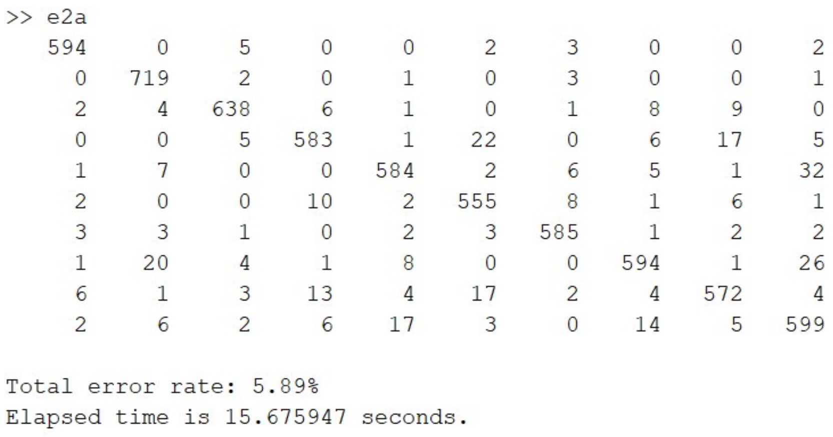
```matlab
function [totalPredictions, totalC] = KNN_classifier(trainingValue, trainingLabel,
    testValue, testLabel, K)


    numTestVectors = size(testValue, 1);
    totalPredictions = zeros(numTestVectors, 1);

    %Confusion Matrix dimensions for all 10 written numbers
    totalC = zeros(10, 10);

    %Euclidean distance
    distances = pdist2(testValue, trainingValue, 'euclidean');

    %K nearest neighbors
    for i = 1:numTestVectors
        [~, sortedIndices] = sort(distances(i, :), 'ascend');
        nearestNeighbors = trainingLabel(sortedIndices(1:K));

        %Classify test image from nearest neighbor
        predictedLabel = mode(nearestNeighbors);
        totalPredictions(i) = predictedLabel;
    end

    %Compute confusion matrix
    totalC = confusionmat(testLabel , totalPredictions);
end
```

B.2 Repeated results



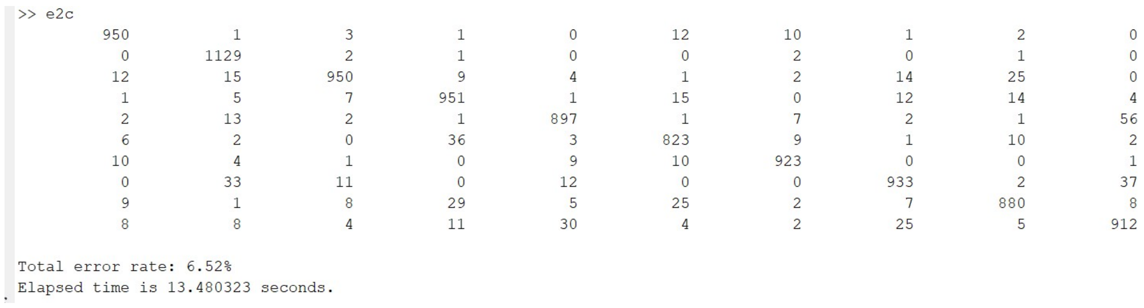
```
>> e1a
       973           1           1           0           0           1           3           1           0           0
         0        1129           3           0           1           1           1           0           0           0
         7           6         992           5           1           0           2          16           3           0
         0           1           2         970           1          19           0           7           7           3
         0           7           0           0         944           0           3           5           1          22
         1           1           0          12           2         860           5           1           6           4
         4           2           0           0           3           5         944           0           0           0
         0          14           6           2           4           0           0         992           0          10
         6           1           3          14           5          13           3           4         920           5
         2           5           1           6          10           5           1          11           1         967

Total error rate: 3.09%
Elapsed time is 79.984059 seconds.
```

Figure B.1: Confusion matrix, error rate and time for NN classifier without clustering



```
>> e2a
   594     0     5     0     0     2     3     0     0     2
     0   719     2     0     1     0     3     0     0     1
     2     4   638     6     1     0     1     8     9     0
     0     0     5   583     1    22     0     6    17     5
     1     7     0     0   584     2     6     5     1    32
     2     0     0    10     2   555     8     1     6     1
     3     3     1     0     2     3   585     1     2     2
     1    20     4     1     8     0     0   594     1    26
     6     1     3    13     4    17     2     4   572     4
     2     6     2     6    17     3     0    14     5   599

Total error rate: 5.89%
Elapsed time is 15.675947 seconds.
```

Figure B.2: Confusion matrix, error rate and time for NN classifier with clustering



```
>> e2c
       950           1           3           1           0          12          10           1           2           0
         0        1129           2           1           0           0           2           0           1           0
        12          15         950           9           4           1           2          14          25           0
         1           5           7         951           1          15           0          12          14           4
         2          13           2           1         897           1           7           2           1          56
         6           2           0          36           3         823           9           1          10           2
        10           4           1           0           9          10         923           0           0           1
         0          33          11           0          12           0           0         933           2          37
         9           1           8          29           5          25           2           7         880           8
         8           8           4          11          30           4           2          25           5         912

Total error rate: 6.52%
Elapsed time is 13.480323 seconds.
```

Figure B.3: Confusion matrix, error rate and time for KNN classifier with clustering