TTT4275: Classification Project

Magnus Wiik 506637
Erlend Withammer-Ekerhovd 527340

April 28, 2022

Department of Engineering Cybernetics

1 Summary

In this report, the training of two different classifiers for two distinct problems are discussed.

First, the training of a linear classifier and fine-tuning of its step factor α were used to classify Iris flowers. The flowers were classified based on numerical values of four different features into three distinct flower types at a 3.33% error rate on unseen data. The error rate was highly dependent on the choice of α, as well as the choice of training and test data.

Second, two template based classifiers were designed to predict which handwritten numbers between 0-9 it was shown. This required a large amount of training data to get a satisfactory error rate. The data was drawn from the MNIST database, with a training set of 60000 images and a test set of 10000 images. The NN classifier ended up with an error rate of 3.09% and the KNN classifier with clustering had an error of 6.46%.

Contents

2 Introduction

This report is a discussion on two different classification tasks in the course TTT4275 Estimation, Detection and Classification. The chosen tasks were the Iris task with the problem of classifying Iris flower types by their numerical features, and the Digit task with the problem of classifying handwritten digits from images.

2.1 Idea

The two tasks are classic Machine Learning introductory tasks and teach the ideas and mathematics underlying prediction technology.

Both tasks use the idea of teaching an algorithm by making it do a prediction, and then checking the true answer afterwards and calculating the difference-error. However, the implementation of this prediction and checking differs between the tasks.

2.2 Interest to society

Machine Learning was developed in the 1960s[1] and is the first step in learning about how prediction machines work. In the early 2000nds and 2010ns[2][3] Neural Networks and Deep Neural Networks took over as the leading technologies within prediction.

As these technologies progress, the cost of prediction will fall drastically. Just as with the large fall in cost of computing with the computer revolution, this fall in price of a sought after commodity will likely cause a large shift in the economy and the lives of everyday people. More people understanding the ideas and mathematics underlying these technologies will likely be of great interest to society by making the transition to take advantage of cheap prediction smoother for businesses and governments, as argued in the book Prediction Machines: The Simple Economics of Artificial Intelligence by economists Ajay Agrawal, Joshua Gans and Avi Goldfarb[3].

3 Theory

Classification is the problem of putting the right objects in the right boxes. Most objects have vague definitions, and the features describing them might overlap. For example, a black cat and a white cat are both cats, while a black cat and a black dog are not both cats, even though the black animals might have more pixels in common in a picture. As such, different classification problems need different solutions, with differing complexity.

In the example of the black cat and the black dog, both animals have an overlapping feature, namely color. This makes differentiating them more difficult. However, not all classification problems have overlapping features, making these classifications easier. When a classification problem has features that do not overlap between the different groups, and it is possible to draw a line between these groups, the problem is called a linearly separable classification problem as showcased in Figure 1.

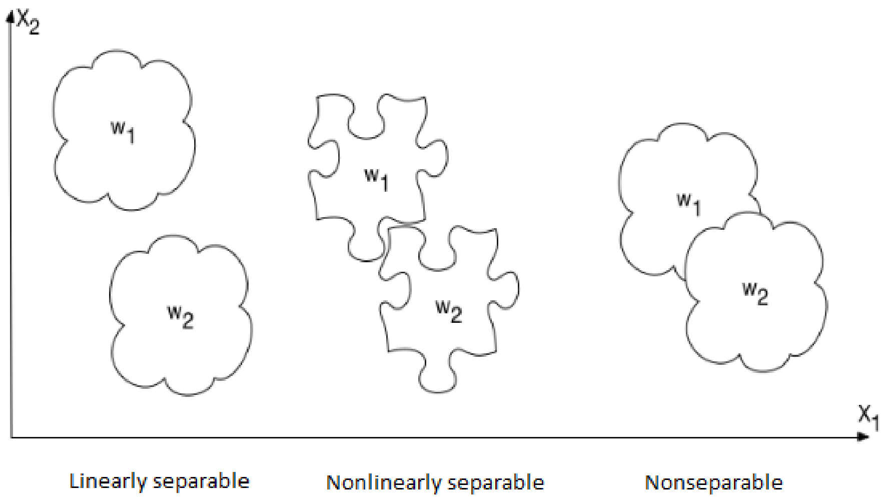


Figure 1: Difference between linearly separable, nonlinearly separable and nonseparable classes.

3.1 Linear classifier

The Iris flower separation task is close to linearly separable, causing a linear classifier to have an acceptable error rate. Though more complex classifiers might get a lower error rate, these would also be harder to implement, causing a tradeoff between accuracy and simplicity.

As stated in the Compendium[4], linear classifiers use a decision rule as in Equation 1 to classify objects.

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

In other words, the object x belongs to the group ω_j if the return from the decision rule for group j is the largest of all the returns from the different group decision rules in i.

As seen in the Compendium[4], the decision rule is defined as in Equation 2.

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

Where w_{io} is offset for the class w_i and C is the number of classes. Which can again be written in matrix form as in Equation 3.

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

Where w_o has dimensions $C \times 1$, W has dimensions $C \times D$, x has dimensions $D \times 1$, and D is the number of features.

It is then possible to merge these vectors to one matrix as $W = [W w_o]$ and $x = [x^T 1]^T$ resulting in Equation 4, with W having dimensions $C \times (D+1)$ and x having dimensions $(D+1) \times 1$.

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

To tune the decision matrix W to get the desired functionality, a cost function is also needed. A Minimum Square Error or MSE cost function, as shown in Equation 5, is the most popular variant[4].

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

Where g_k is a prediction as given in Equation 9 and t_k is the corresponding correct answer.

To minimize the MSE, an iterative process of taking steps in the opposite direction of the gradient of the MSE with respect to W can be done. The gradient of the MSE with respect to W is given in Equation 6.

$$\nabla_W MSE = \sum_{k=1}^{N} [(g_k - t_k) \circ g_k \circ (1 - g_k)] x_k^T \tag{6}$$

Where $(g_k - t_k) \circ g_k$ is the elementwise multiplication of the vectors. Minimizing MSE is the done by changing W as shown in Equation 8

Collecting all predictions g_k into one prediction matrix called g, all feature vectors x_k into a feature matrix x, and all correct answers t_k into an answer matrix t, Equation 6 can be reduced to Equation 7 to take advantage of linear algebra optimizations.

$$\nabla_W MSE = [(g - t) \circ g \circ (1 - g)]x^T \tag{7}$$

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

Where α is the step factor, and m is the current iteration. The step factor must be carefully chosen and tuned to the problem at hand.

Since the target vector t_k is a vector of zeros with a 1 at the correct class, i.e. $t_k = [1\ 0\ 0]^T$ for Setosa, a function for squishing the prediction Wx to the closed interval [0 , 1] is given in Equation 9. Thus, $g_k = [0.7\ 0.2\ 0.1]^T$ can be seen as the classifiers predicting a 70% chance of Setosa on the given x_k.

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

3.2 Template classifiers

In this subsection, the theory behind template based classifiers is explained. This is a method where the input x is matched with every template or reference in the training set. In the matching process, the similarity between the input and the template is measured. There are different ways to measure similarity. The method used in this project is the Euclidean distance, which is seen in Equation 10

$$d(x, ref_i) = (x - y_i)^T (x - y_i) \tag{10}$$

After measuring the input against every template, which class the input belongs to is determined based on a decision rule. In this project there are two decision rules of interesting.

The first classifier used the decision rule called "Nearest neighbor" or "NN", which finds the template that is the most similar to the input, choosing this as its prediction.

The second classifier used the decision rule called "K-nearest neighbor", more widely known as "KNN". This is a procedure where the $k > 1$ most similar templates to the input are found. The classification is determined based on a voting process between these K closest templates. The class with the most votes is the classifiers' prediction.

Training of the template based classifiers can be done through clustering of the training set data. This is a process where a new training set of the M

$<<$ N most representative templates for a class are chosen. This reduces the computational complexity, since the amount of data is reduced. However, this may increase the error rate, as there are now fewer data points.

4 The Tasks

4.1 The Iris Task

In this task, Iris flower types are to be separated by their features. The different Iris flower types were Setosa, Versicolor and Virginica. Using the Fisher Iris Dataset[5] four features of differing importance were used, namely sepal length, sepal width, petal length and petal width. The features were given as numerical values.

The goal of this task was to make a linear classifier with as few mistakes as possible, as discussed in Section 3. Furthermore, another central goal was to understand and get a practical experience with the underlying mathematics of the steepest descent algorithm.

As discussed in Section 3, the Iris types are close to linearly separable from each other, making this a good task for learning about linear classifiers. As part of the task, histograms showcasing the Iris types differences on each feature were made to clearly see linear separability, as seen in Figure 9 in Section 5.1.

4.2 The Digits Task

In this task, pictures of handwritten digits are to be classified based on similarity to former pictures of digits. The MNIST database has a dataset which contains in total 70000 pictures of handwritten numbers from 0 to 9, where 60000 of these are written by 250 people and the other 10000 are written by 250 other people. The pictures are a 28x28 matrix with 8-bit grayscale, so that every pixel has a value between 0-255. It should also be noted that the pictures have been preprocessed, where they have been centered and scaled.

The goal of this task is to program two different template based classifiers to distinguish between 10 classes, where every digit is a class. The classifiers used were the NN-based classifier and the KNN-based classifier, both using Euclidean distance. A clustering function was also used, to minimize the runtime of the classification process.

5 Implementations and results

5.1 The Iris task

In the Iris flower task, the gradient descent method explained in Section 3 was implemented. The dataset used was split up in a training-set and a test-set. The training-set was used to iteratively calculate a better W matrix for classifying the Iris flowers based on their features, and the test-set was used to test how well this matrix worked on unseen data. Using different parts of the dataset for testing and training was tested to compare performance. Also, removing features was tested to see the impact on classification performance.

The dataset consisted of 150 flowers with 4 features each. The flowers were separated into 3 different types called Setosa, Versicolor and Virginica of 50 flowers each. The data set was split into a training-set and a test-set with a 30-20 split, respectively.

The data was fed in to a function called makeTrainingAndTestDataClass, returning a TrainingAndTestDataClass (TTD). TTD contained trainingData, trainingTarget, testData and testTarget, where data was the input data and target was the correct answer of what type of flower it was. It was first used a TTD containing trainingData from the first 30 samples per class and testingData from the 20 last. It was then tested using another TTD containing the last 30 samples per class as trainingData and the first 20 as testingData.

Training the classifier was done by feeding the trainingData in as x in Equation 4. Doing this returned a matrix of predictions which, after going through the sigmoid function, contained vectors which could be viewed as containing the likelihood for each class being correct. This prediction matrix was then fed into Equation 7 as g, with the trainingTarget as t. The new W was then calculated using Equation 8. The training was done after 1000 iterations. The training process can be seen in Figure 2.

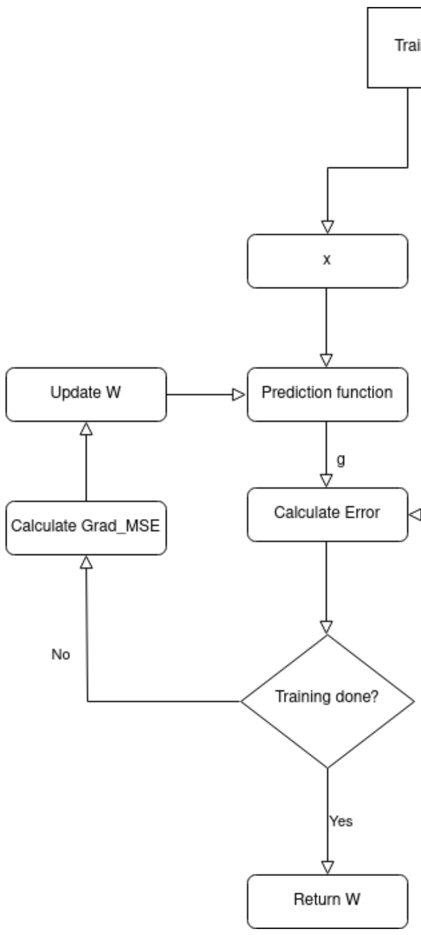


Figure 2: Flow diagram of training procedure.

An important part of training the classifier is tuning the step factor α. The tuning was done by iteratively increasing alpha and calculating the error percent rate. The resulting plot can be seen in Figure 3. The lowest achieved error rate for both sets were 3.33% for $\alpha \approx 0.008$.

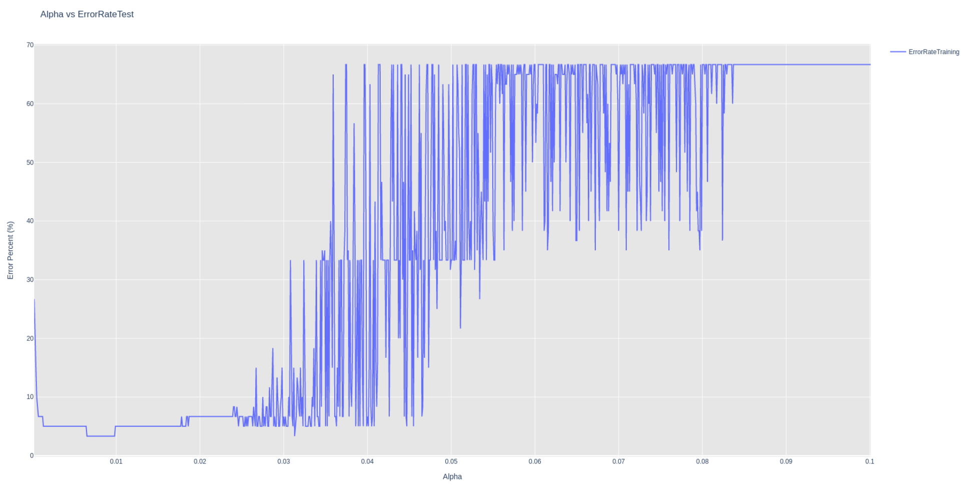


Figure 3: Error rate of the training-set plotted against different α.

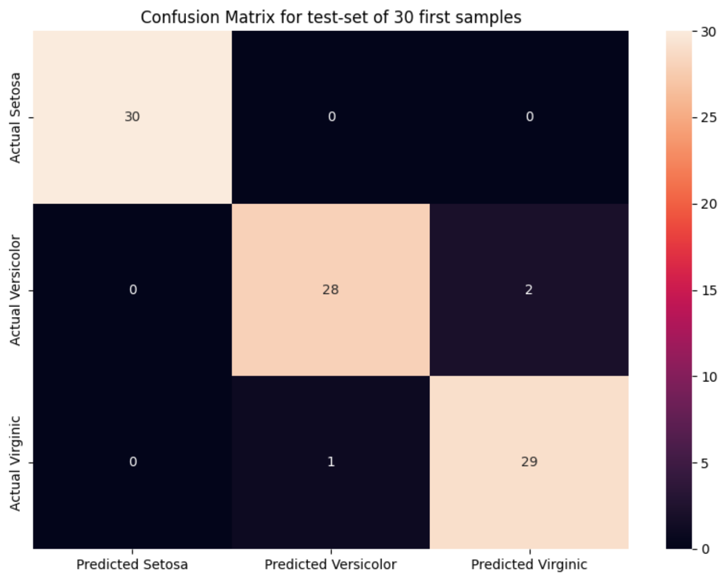


Figure 4: Confusion Matrix for the training-set of the first 30 samples.

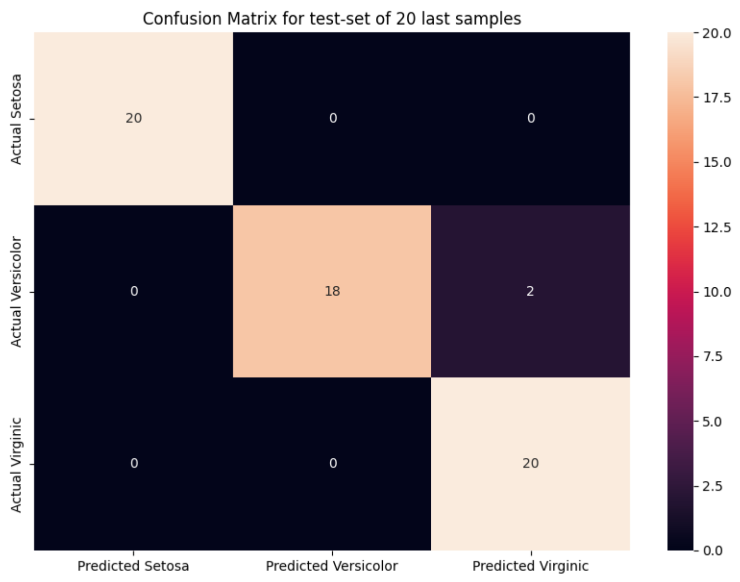


Figure 5: Confusion Matrix for the corresponding test-set of the first 30 samples training-set.

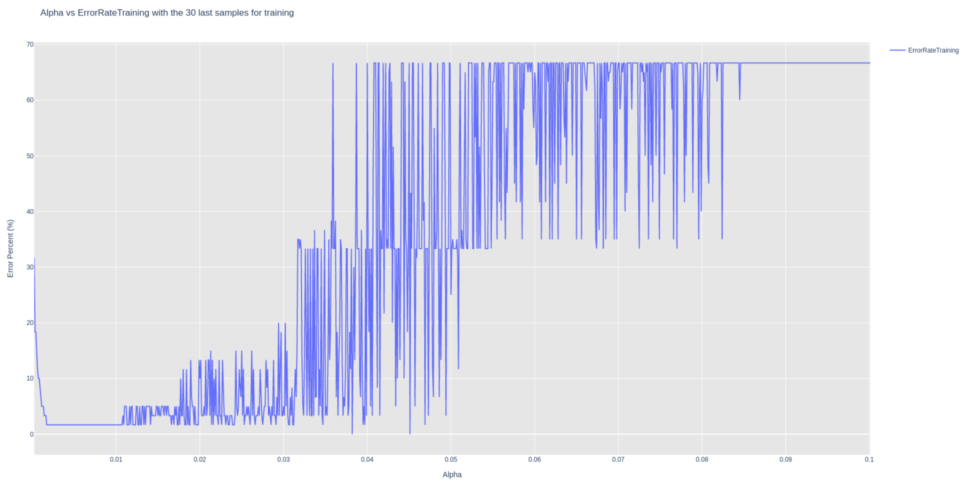


Figure 6: Error rate of the training-set plotted against different α, using the last 30 samples per class of the dataset as trainingData.

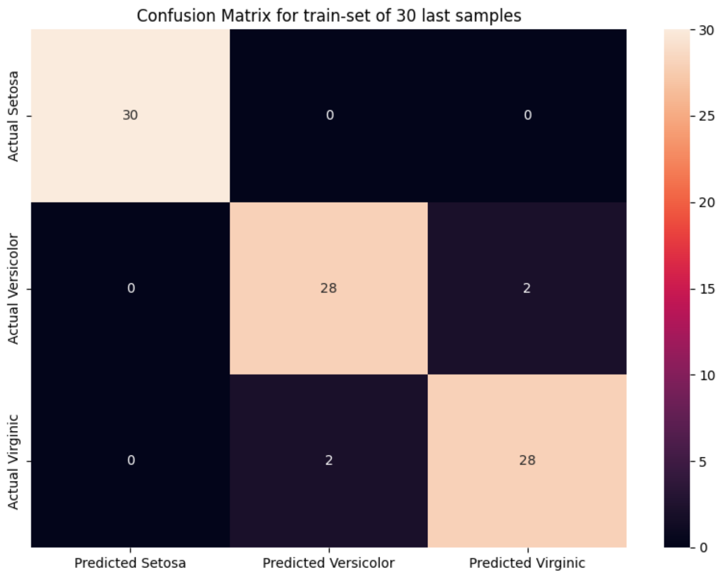


Figure 7: Confusion Matrix for the training-set of the last 30 samples.

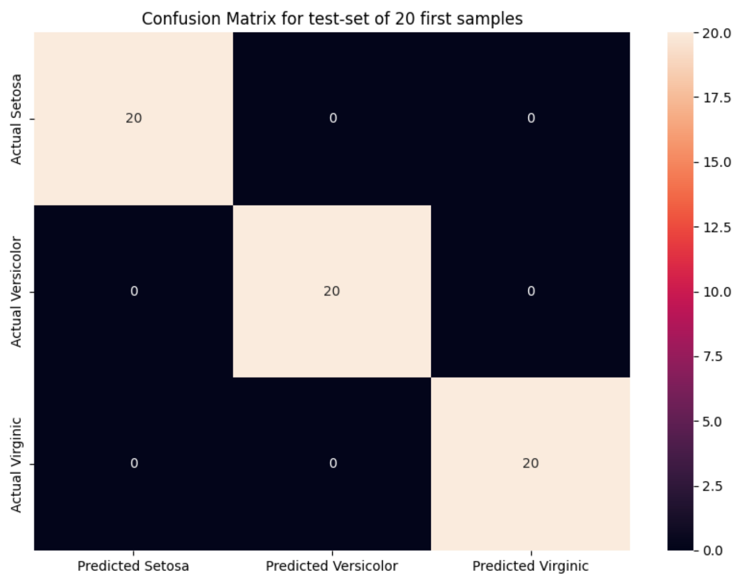


Figure 8: Confusion Matrix for the corresponding test-set of the last 30 samples training-set.

Also, using the last 30 samples as trainingData was tested. The resulting error rate against α plot can be seen in Figure 6, and had the lowest error rate of 0% for the test set and 5.56% for the training-set at $\alpha = 0.0382$. As can be seen from Figure 3 and 6 the choice of trainingData can be influential, especially for small datasets as some difficult data points will be put in either the training or test set. In the Iris flower dataset, some flowers are not possible for a linear classifier to classify correctly. As such, when these are not in the test-set, we can achieve 0% error rate. However, when they are not in the test-set, they will be in the training set and cause a higher error rate there. Hence, this was just a lucky coincidence and should not be expected to work well for other test sets.

The Iris flower data set contains as mentioned 4 features per flower. However, each feature is not equally useful, as some features have more overlap than others. Less distinct features are worse for classifying, and as such removing the worst features was interesting to test. The different features and their overlap can be seen in Figure 9. It is clear from the figure that Sepal Width has the most, Sepal Length the second most and Petal Length the third most overlap of the features. Thus, Petal Width has the most correlation with class.

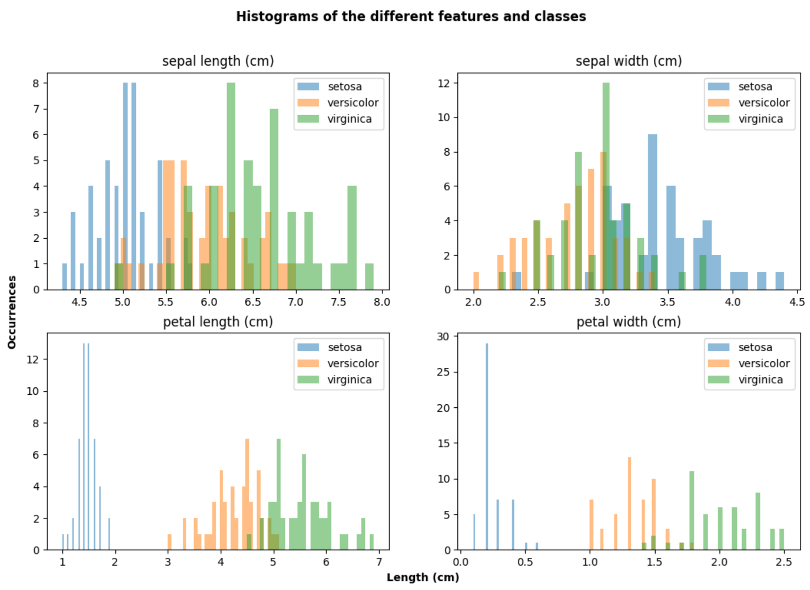


Figure 9: Histogram of different features and classes.

Removing features makes it harder to classify, as there are fewer features to distinguish the different classes from each other. However, it is again possible to be lucky with removing a tricky feature from the training-set, reducing the error rate there. The effect on error rate by α for removed features can be seen in Figure 10, 12 and 14. The error rates for the test-set and training-set for one, two and three removed features at the best step factors were $[5\%, 3.33\%]$, $[5\%, 5.56\%]$ and $[6.67\%, 4.44\%]$, respectively. As we can see from Figure 11, 13 and 15 the predictions got progressively worse by removing features. However, this effect was very small due to the high correlation between Petal Width and Iris flower type.

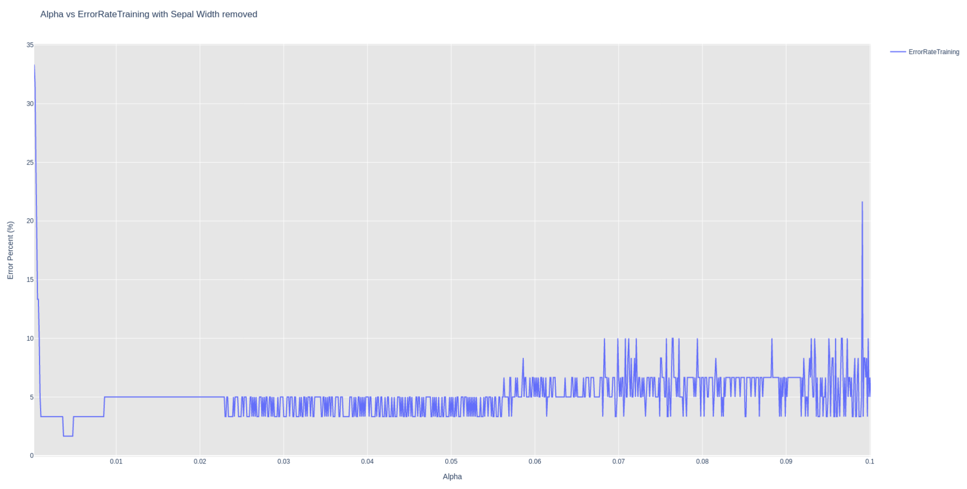


Figure 10: Error rate of the training-set plotted against different α, with Sepal Width removed.

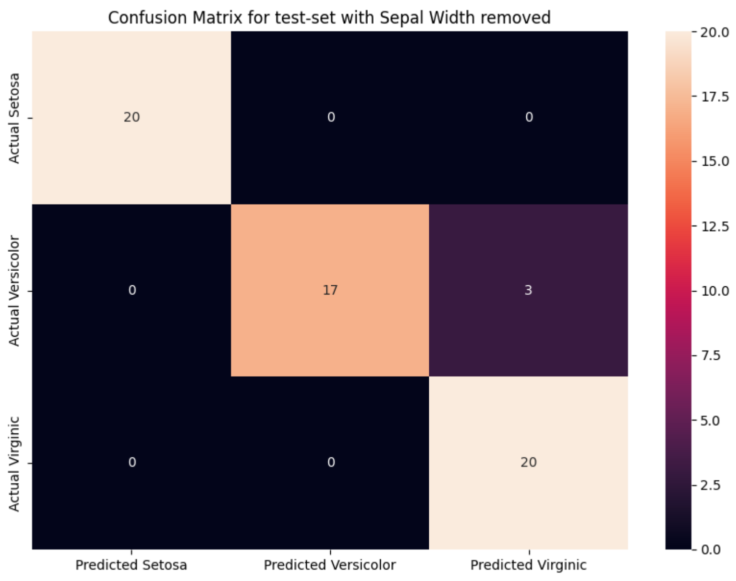


Figure 11: Confusion Matrix for the test-set, with Sepal Width removed.

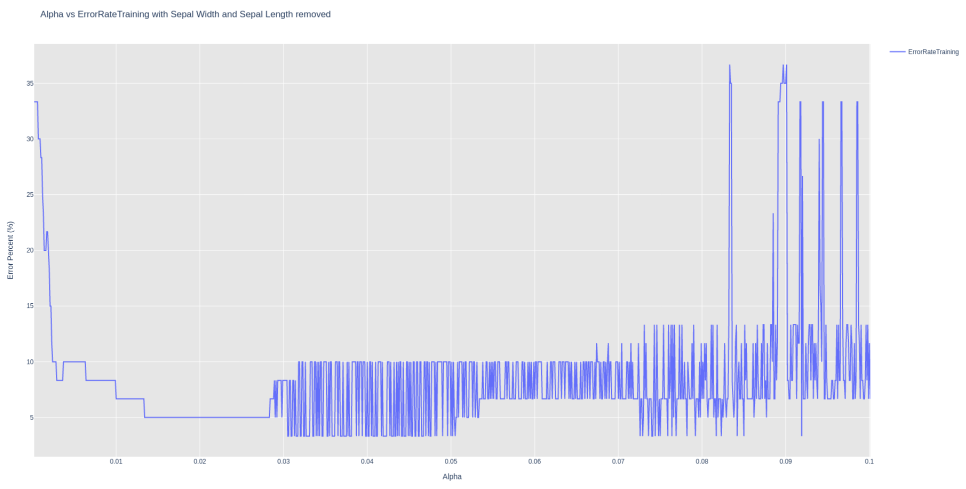


Figure 12: Error rate of the training-set plotted against different α, with Sepal Width and Sepal Length removed.

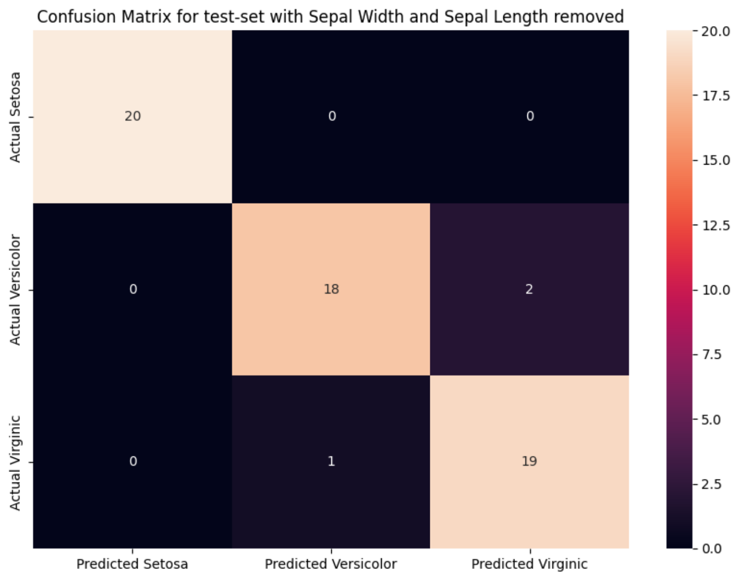


Figure 13: Confusion Matrix for the test-set, with Sepal Width and Sepal Length removed.

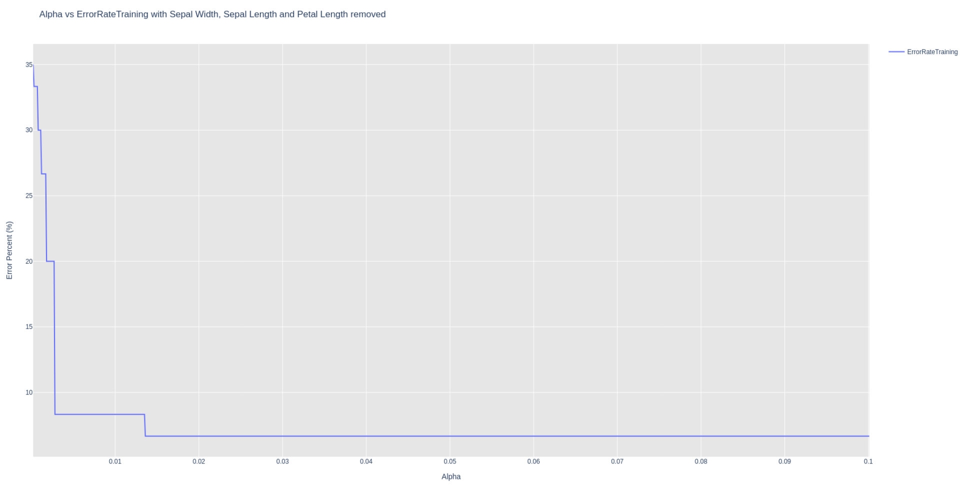


Figure 14: Error rate of the training-set plotted against different α, with Sepal Width, Sepal Length and Petal Length removed.

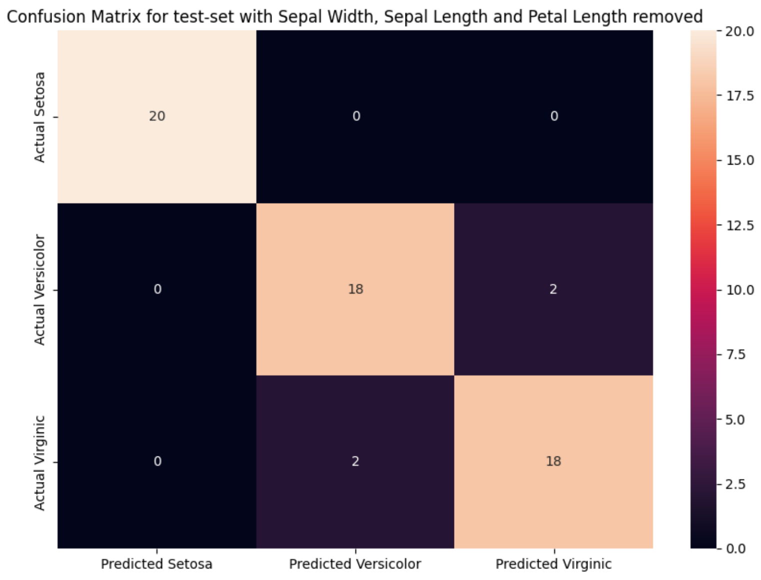


Figure 15: Confusion Matrix for the test-set, with Sepal Width, Sepal Length and Petal Length removed.

5.2 The Digits Task

In the first task a NN-classifier was designed and passed a training set containing 60000 images. It was also given a test set containing 10000 images. For every picture in the test set, it looped through the whole training set and calculated the Euclidean distances between all test and training images. Then the distances, train image and test image were put in an array and sorted in ascending order. The prediction was the first element, since this had the lowest distance to the test image. Since this was a very time-consuming process when the data set was large, the MNIST data set was split into chunks. The corresponding training- and test-chunks were 6000 and 1000 in size, respectively.

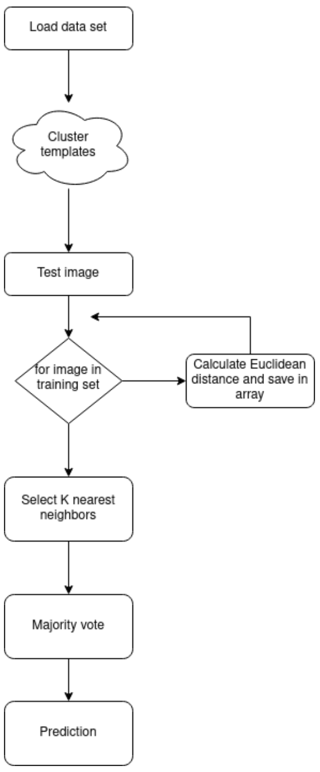
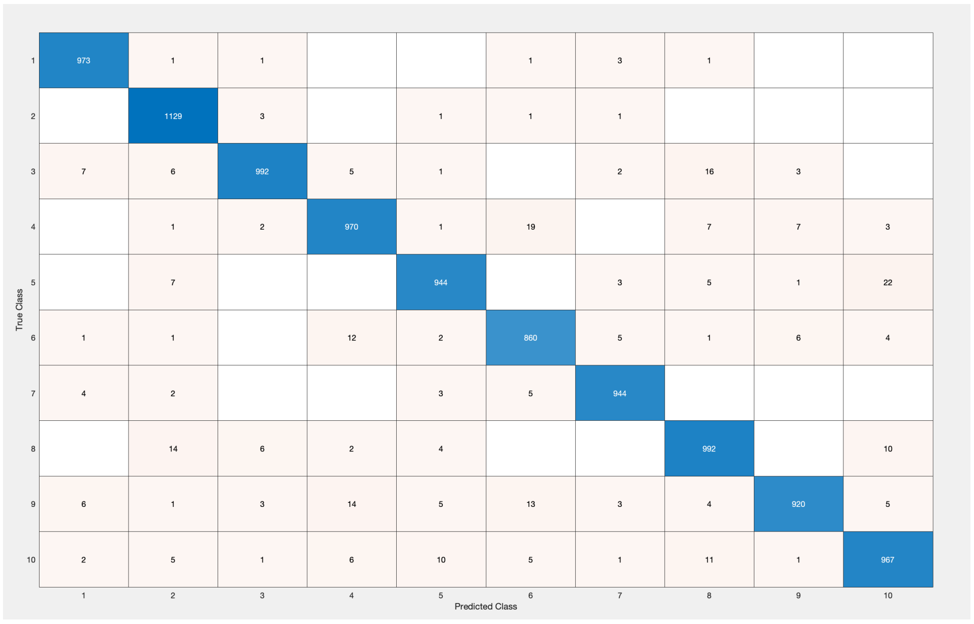

Load data set
Cluster templates
Test image
for image in training set
Calculate Euclidean distance and save in array
Select K nearest neighbors
Majority vote
Prediction

Figure 16: Flow diagram of the KNN classifier with clustering

In the second task, the focus was to improve the processing time by producing a smaller set of templates for each class. One way to do this is to pick some random images from the training set and make a new and smaller training set of these images. However, the problem is that this risks ending up with a training set which is not representative enough for the whole dataset. The proper way to reduce the number of templates is by clustering the training set. This is done by splitting the training set up in the templates for each class and passing these partitions, as well as the number of clusters you want, to the kmeans(partition, M) MATLAB function.

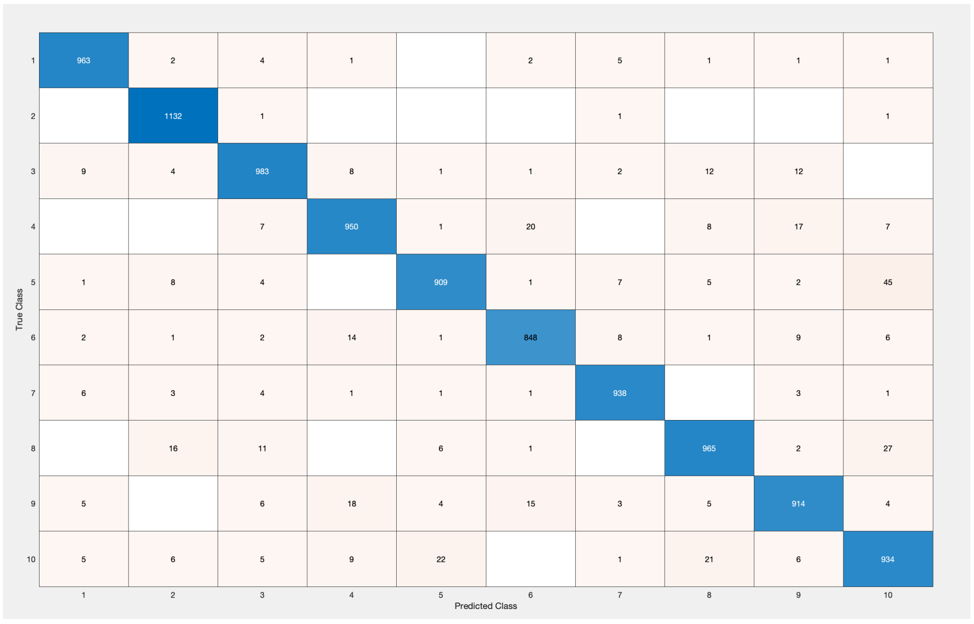
After implementing the clustering, it was time to make a KNN template based classifier. The only difference between and NN- and KNN classifier is that KNN fetches the K nearest neighbors and lets them have a majority vote. For this action, the MATLAB function mode(A), where A is an input array, was chosen. This function chooses the most frequent number in the array. If there is a tie between multiple numbers, it chooses the first occurring number.The flow of this implementation of the system is shown in Figure 16



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

Figure 17: Confusion matrix of NN-classifier with 60000 training images

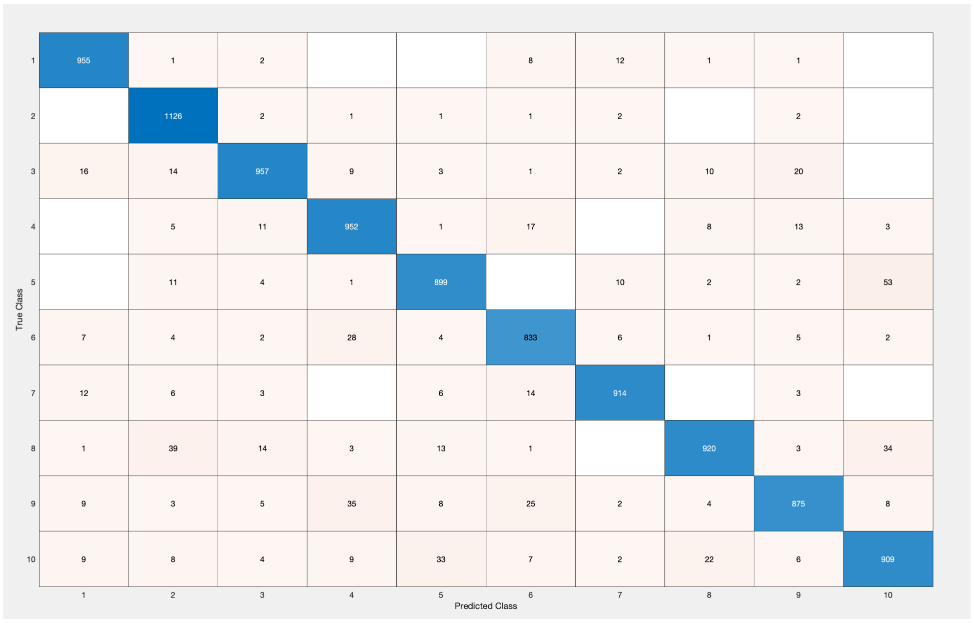
The results from the first task are presented in Figure 17. In the confusion matrix one can see the overall performance of the NN classifier with the whole training set of 60000 images. The process of classifying the training set took 23 minutes and the error rate was 3.09%. This is a decent result in light of a scientific paper in NCBI: "Nearest-neighbor classification using the normalized vector dot product applied to raw pixels classifies the MNIST and Fashion-MNIST data sets at an error rate of about 2.8% and 15% respectively"[6]. However, in the scientific paper they used principled coarse-graining and sampling to improve NN classification, which is outside the scope of this project. From the diagram, it can be seen that the classifier is best at guessing the numbers 0 and 1, which is probably because they have few of the same features which the other numbers have. On the other hand, it was worst at guessing the numbers 4, 8 and 9. An interesting takeaway is that the classifier mixed the numbers 4 and 9 in total 22 times, likely caused by there only being one line separating a 4 and a 9.



True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	10
1	963	2	4	1		2	5	1	1	1
2		1132	1				1			1
3	9	4	983	8	1	1	2	12	12	
4			7	950	1	20		8	17	7
5	1	8	4		909	1	7	5	2	45
6	2	1	2	14	1	848	8	1	9	6
7	6	3	4	1	1	1	938		3	1
8		16	11		6	1		965	2	27
9	5		6	18	4	15	3	5	914	4
10	5	6	5	9	22		1	21	6	934

Figure 18: Confusion matrix of NN-classifier with clustered training set

The results from the second task is presented is Figures 18 and 19. When comparing Figures 18 and 17, one can see that the overall performance is quite similar. However, there is a slight increase in error rate for the NN classifier with clustering. With clustering implemented, the processing time has decreased drastically from 23 minutes to 37 seconds, but the error rate is increased to 4.64%. The process time is much shorter because the clustering shrinks the training set from 60000 templates to 640 templates. However, this comes with a downside since the data is much smaller, which in turn increases the error rate. An interesting takeaway is that this algorithm is better at predicting the number 1 than previously. It went from having six missed number 1's to having three misses.



True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	10
1	955	1	2			8	12	1	1	
2		1126	2	1	1	1	2		2	
3	16	14	957	9	3	1	2	10	20	
4		5	11	952	1	17		8	13	3
5		11	4	1	899		10	2	2	53
6	7	4	2	28	4	833	6	1	5	2
7	12	6	3		6	14	914		3	
8	1	39	14	3	13	1		920	3	34
9	9	3	5	35	8	25	2	4	875	8
10	9	8	4	9	33	7	2	22	6	909

Figure 19: Confusion matrix of KNN-classifier with clustered training set

The results from the KNN classifier with clustering in Figure 19 is somewhat surprising since the error rate went to 6.31%. This is surprising because it would seem that a prediction backed by more data would return a more confident prediction. However, this was not the case. An interesting takeaway is that this algorithm has the largest increase in the mixing of the numbers 4 and 9. This means that the representativeness has decreased the most for both 4 and 9 in the clustering process.

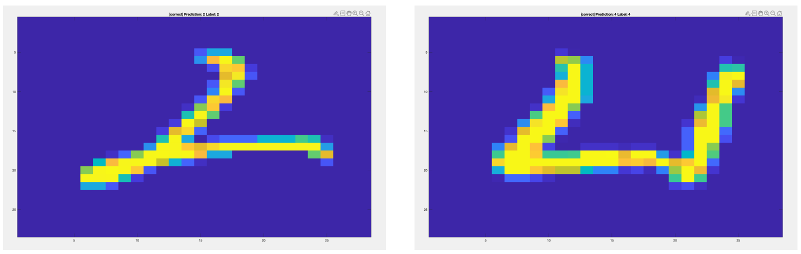


Figure 20: NN classifier correctly classified number 2 and 4

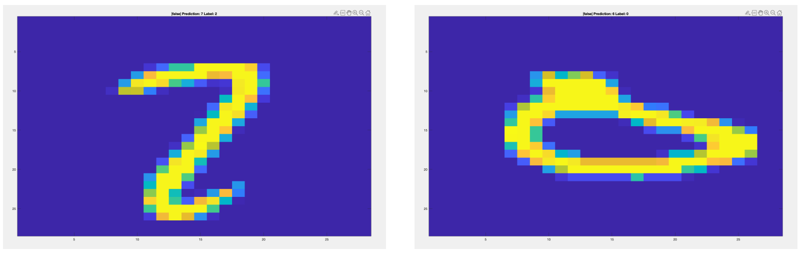


Figure 21: NN classifier misclassified number 2 and 0

In Figure 21 there are two images which the NN classifier has predicted correctly. However, for both of the images, it is not apparent which number it is representing. As a human, I would say the first images looks like a picture of the symbol alpha and the second on is a picture of the symbol omega or the letter u. In Figure 21 there are two images which the NN classifier misclassified. The algorithm predicted that the first image was a 7 and that the second image was a 6. However, as a human, I would say that it is apparent that the first one is a 2 and the second is a 0.

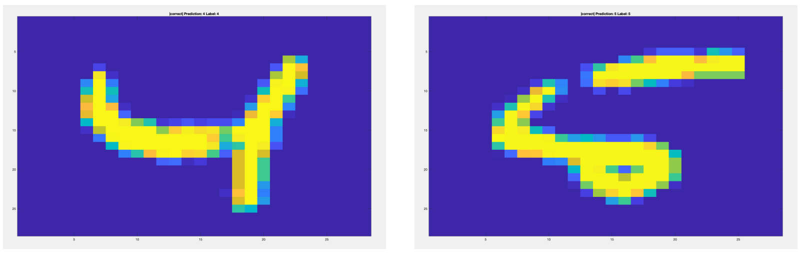


Figure 22: KNN classifier with clustering correctly classified number 4 and 5.

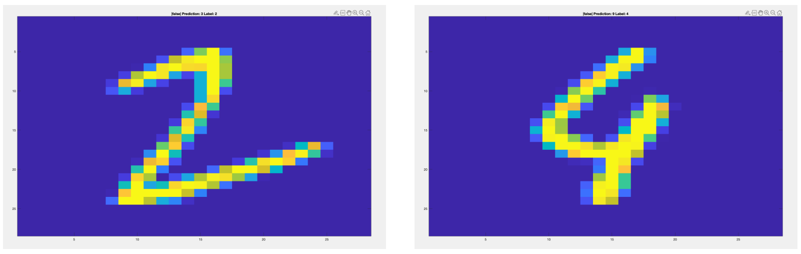


Figure 23: KNN classifier with clustering misclassified number 2 and 4.

In Figure 22 there are two images which the KNN classifier with clustering predicted correctly. However, the first images could also be mistaken for the number 7 and the second images could easily be seen as the number 6. In Figure 23 there are two images which the KNN classifier with clustering misclassified. It predicted that the first image was the number 3 and that the second was the number 9. As a human, I think the first image is an apparent number 2, but in the second image I agree with the program that this looks like the number 9.

6 Conclusion

6.1 The Iris Task

The Iris flower dataset gives a good introduction to machine learning. The close to linearly separable flower types makes it possible to create a linear classifier with sufficiently low error rate. Training the classifier using the steepest descent method results in a classifier with about 3.33% error rate on unseen data. However, this error rate is heavily dependent on the choice of the step factor α as well as how the dataset is split up in to a training and test set. Hence, one can be "lucky" and get a test set of just simple to classify flowers, resulting in a lower than expected error rate.

The dataset has 4 features per flower, some of which have a higher correlation with flower type than others. Removing the worst features, one by one, creates progressively worse error rates. However, since Petal Width has such a high correlation with flower type, even only using this feature results in a surprisingly low error rate of about 6.67% on unseen data.

The Iris task has as such taught us about the underlying mathematics and practical implementation of linear classifiers. We have learned the importance of the choice of step factor to the error rate results of the steepest descent algorithm. Also, the importance of the choice of training and test sets have become clear. Thus, the Iris task has been a good introduction to machine learning and the practical applications of statistics.

6.2 The Digits Task

The MNIST dataset is easy to download and use in machine learning projects, and also gives a good introduction to image recognition. This is a dataset where a template based classifier such as a nearest-neighbor or k-nearest-neighbor proves to be successful. Creating classifiers using these methods for classification, yielded 3.09% for the NN classifier, 4.64% for NN classifier with clustering and 6.46% for KNN using clustering.

Even though the error rate is acceptably low, the program does make blunders which a human would not make. This is specially true for the number 4 and 9 which the algorithm mixes up on many of its attempts.

This task has learned us the inner workings of algorithms for how simple image recognition works, and the factors which determine how successful it is. We saw that a rather simple classifier such as Nearest neighbor with Euclidean distance gave impressive results.

References

[1] Machine learning. https://en.wikipedia.org/wiki/Machine_learning. Accessed: 2022-05-02.

[2] Artificial neural network. https://en.wikipedia.org/wiki/Artificial_neural_network. Accessed: 2022-05-02.

[3] J. Gans A. Agrawal and A. Goldfarb. *Prediction Machines: The Simple Economics of Artificial Intelligence., year = 2018.*

[4] Magne H. Johnsen. *Compendium -Part III- Classification.* 2017.

[5] Iris flower data set. https://en.wikipedia.org/wiki/Iris_flower_data_set. Accessed: 2022-05-02.

[6] Improving the accuracy of nearest-neighbor classification using principled construction and stochastic sampling of training-set centroids. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7911166/. Accessed: 2022-03-05.

A Iris Appendix

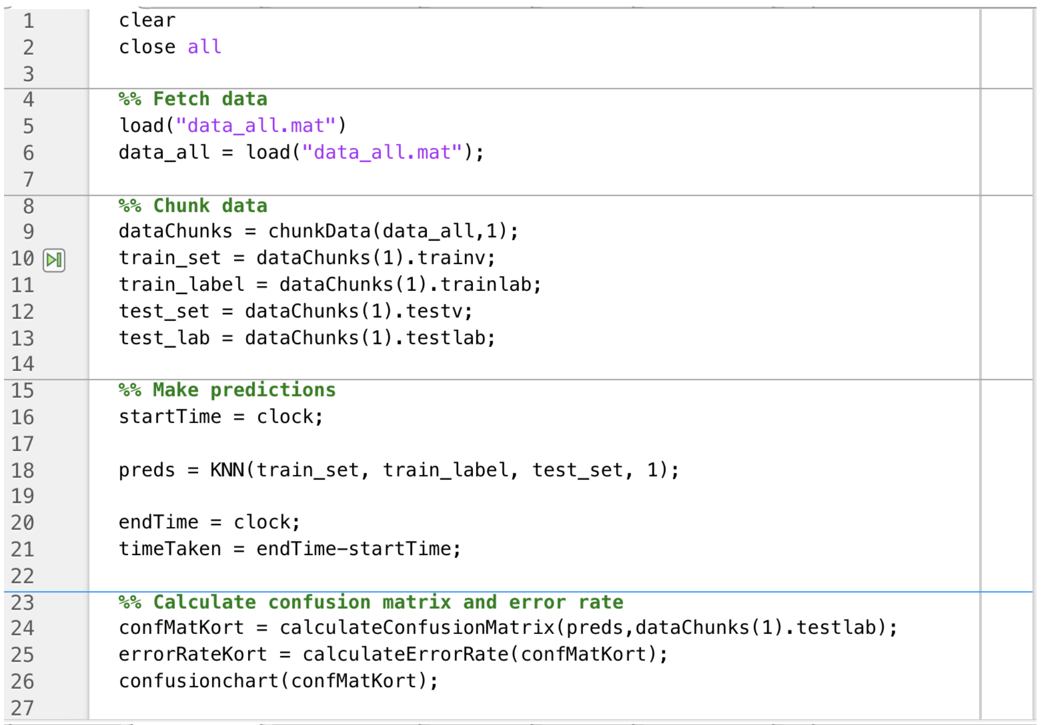
```python
from sklearn.datasets import load_iris
import numpy as np
import stat_helper as sh
import datetime as dt


iris = load_iris()

iris_classes =np.array(iris['target_names'])
iris_data = np.array(iris['data'])
iris_feature = np.array(iris['feature_names'])
iris_target = np.array(iris['target'])

def makePredictionMatrix(W, x):
    predictionMatrix = np.array(sh.sigmoid(np.matmul(W,x))).T
    return predictionMatrix

def trainWMatrix(W, TTD, alpha):
    n_classes = len(TTD.trainingData)
    gradW_MSE = 0
    for c in range(n_classes):
        x = np.c_[TTD.trainingData[c],np.ones(len(TTD.trainingData[c])).T].T
        t = TTD.trainingTarget[c]
        g = makePredictionMatrix(W, x)

        gradW_MSE += sh.calculate_grad_W_MSE(g,t,x)

    W-=alpha*gradW_MSE

    return W

def trainUntilSatisfactory(W, TTD, alpha, itt):
    for i in range(itt):
        W = trainWMatrix(W, TTD, alpha)
    return W

def makeConfusionMatricies(W, TTD):
    n_classes = len(TTD.testData)
    confusionMatrixTestSet = np.zeros((n_classes, n_classes))
    confusionMatrixTrainingSet = np.zeros((n_classes, n_classes))
    for c in range(n_classes):
        x_test = np.c_[TTD.testData[c],np.ones(len(TTD.testData[c])).T].T
        x_train = 
np.c_[TTD.trainingData[c],np.ones(len(TTD.trainingData[c])).T].T
        predictionTestSet = makePredictionMatrix(W, x_test)
        predictionTrainingSet = makePredictionMatrix(W, x_train)
        answearTestSet = TTD.testTarget[c]
        answearTrainingSet = TTD.trainingTarget[c]
        for i in range(len(TTD.testTarget[0])):
            confusionMatrixTestSet[np.argmax(answearTestSet[i])]
[np.argmax(predictionTestSet[i])] += 1
        for i in range(len(TTD.trainingTarget[0])):
            confusionMatrixTrainingSet[np.argmax(answearTrainingSet[i])]
[np.argmax(predictionTrainingSet[i])] += 1
    return confusionMatrixTestSet, confusionMatrixTrainingSet

def makePercentErrorRate(confusionMatrix):
    errorPercent = 0
    n_classes = len(confusionMatrix[0])
    n_pred = np.sum(confusionMatrix)
    for i in range(n_classes):
        for j in range(n_classes):
            if i != j:
                errorPercent += confusionMatrix[i][j]/n_pred
    return np.around(errorPercent*100,2)

class TrainingAndTestDataClass(object):
    def __init__(self):
        self.trainingData = []
        self.trainingTarget = []
        self.testData = []
        self.testTarget = []
    def AddToTTS(self, type, obj):
        match type:
            case "trainingData":
                self.trainingData.append(obj)
            case "trainingTarget":
                self.trainingTarget.append(obj)
            case "testData":
                self.testData.append(obj)
            case "testTarget":
                self.testTarget.append(obj)

def makeTrainingAndTestDataClass(data, target, trainingStart, trainingStop, classStart, classLength, n_classes):
    n_trainingData = trainingStop - trainingStart
    n_testData = classLength - n_trainingData
    n_trainingOffset = trainingStart-classStart
    TTD = TrainingAndTestDataClass()
    for c in range(n_classes):
        c_off = c*classLength
        trainingTarget = [target[c_off+trainingStart + i] for i in range(n_trainingData)]
        testTarget = [target[c_off+classStart + i] if i < n_trainingOffset else target[c_off+trainingStop-n_trainingOffset +i] for i in range(n_testData)]

        TTD.AddToTTS("trainingData",np.array([data[c_off+trainingStart + i] for i in range(n_trainingData)]))
        TTD.AddToTTS("trainingTarget",np.array(sh.fix_target(trainingTarget,n_classes)))
        TTD.AddToTTS("testData",np.array([data[c_off+classStart + i] if i < n_trainingOffset else data[c_off+trainingStop-n_trainingOffset + i] for i in range(n_testData)]))
        TTD.AddToTTS("testTarget",np.array(sh.fix_target(testTarget,n_classes)))

    return TTD

def runIrisTask(alphaStart, n_alphas, itt, TTD, file=0):
    alpErrList = np.array([[0.]*3 for i in range(n_alphas)])
    startTime = dt.datetime.now().replace(microsecond=0)
    n_classes = len(TTD.trainingData)
    n_features = len(TTD.trainingData[0][0])
    for i in range(n_alphas):
        if n_alphas > 1:
            alphaStart+=1*10**(-4)
        W = np.zeros((n_classes, n_features))
        bias = np.zeros((n_classes,))
        W = np.c_[W,bias]

        W = trainUntilSatisfactory(W, TTD, alphaStart, itt)

        confusionMatrixTestSet, confusionMatrixTrainingSet = makeConfusionMatricies(W, TTD)
        errorPercentTestSet = makePercentErrorRate(confusionMatrixTestSet)
        errorPercentTrainingSet = makePercentErrorRate(confusionMatrixTrainingSet)

        alpErrList[i][0] = alphaStart
        alpErrList[i][1] = errorPercentTestSet
        alpErrList[i][2] = errorPercentTrainingSet

        if i%100==0:
            print("Itterations ", i)
            print("Time taken: ", dt.datetime.now().replace(microsecond=0)-startTime)
    print("ConfusionMatrixTestSet: \n", confusionMatrixTestSet)
    print("ConfusionMatrixTrainingSet: \n", confusionMatrixTrainingSet)
    min = 100
    minitt= 0
    for i in range (n_alphas):
        if alpErrList[i][1] < min:
            min = alpErrList[i][1]
            minitt = i
    print("Best Alpha and ErrorMargin, with ErrorRates was: ", alpErrList[minitt])
    stopTime = dt.datetime.now().replace(microsecond=0)
    print("Time taken: ", stopTime-startTime)
    if file != 0:
        np.savetxt(file, alpErrList, delimiter=",")






alpha = 0.00370
n_a = 1
itt = 1000

newData, newFeatures = sh.removeListOfFeatures(iris_data, iris_feature, [0]) #leave list empty to include all
sh.hist(newData,newFeatures,iris_classes,"RemovedWorsedOneHist.png") #add a file as last input if you want to save
TTD = makeTrainingAndTestDataClass(newData, iris_target, 0, 30, 0, 50, 3)
runIrisTask(alpha,n_a, itt, TTD)
```

```python
import numpy as np
import matplotlib.pyplot as plt

def fix_target(target,n_classes):
    n_target = len(target)
    new_target = np.zeros((n_target,n_classes))
    for i in range(n_target):
        vector_target = np.zeros((1,n_classes))[0]
        vector_target[target[i]]+=1
        new_target[i]=vector_target
    new_target = np.reshape(new_target,(len(target),n_classes))
    return new_target

def sigmoid(z):
    return 1/(1+ np.exp(-z))

def calculate_MSE(gk,tk):
    return 0.5*np.matmul((gk-tk).T,(gk-tk))

def calculate_grad_W_MSE(g, t, x):
    return np.matmul(((g-t)*g*(1-g)).T,x.T)


def _removeFeature(data, features, featureToBeRemoved):
    n_features = len(features)
    newFeature = np.array([]) ##hard to preallocate string as you need to know the
size, bad for efficiancy but whatever
    n_data = len(data)
    newData = np.array([[0.]*(n_features-1) for j in range(n_data)])

    j = 0 #ugly hack to get correct index for newFeature
    for i in range(n_features):
        if i%n_features != featureToBeRemoved:
            newFeature = np.append(newFeature, features[i])
            j+=1

    for i in range(n_data):
        k = 0
        for j in range(n_features):
            if j%n_features != featureToBeRemoved:
                newData[i][k] = data[i][j]
                k+=1

    return newData, newFeature

def removeListOfFeatures(data, feature, featureRemoveList):
    newData = data
    newFeature = feature
    for i in range(len(featureRemoveList)):
        newIndex = np.where(newFeature == feature[i])[0][0]
        newData, newFeature = _removeFeature(newData, newFeature, newIndex)

    return newData, newFeature

def hist(data, features, classes, file = 0):
    histData = data.T
    fig = plt.figure()
    plt.suptitle('Histograms of the different features and classes', 
fontweight='bold')

    featuresLeftToPlot = len(features)
    for f in range(len(features)):
        if featuresLeftToPlot>=2:
            xdir = 2
        else:
            xdir = 1
        plt.subplot(int(np.ceil(len(features)/2)), xdir, f+1)
        for c in range(3):
            plt.hist(histData[f][c*50:(c+1)*50], bins=30, alpha=0.5, 
label=classes[c])

        plt.title(features[f])
        plt.legend(loc='upper right')

    # Adding a plot in the figure which will encapsulate all the subplots with 
axis showing only
    fig.add_subplot(1, 1, 1, frame_on=False)

    # Hiding the axis ticks and tick labels of the bigger plot
    plt.tick_params(labelcolor="none", bottom=False, left=False)

    #Make common x- and y-labels
    plt.xlabel('Length (cm)', fontweight='bold')
    plt.ylabel('Occurrences', fontweight='bold')

    if file != 0:
        plt.savefig(file)

    plt.show()
    return
```

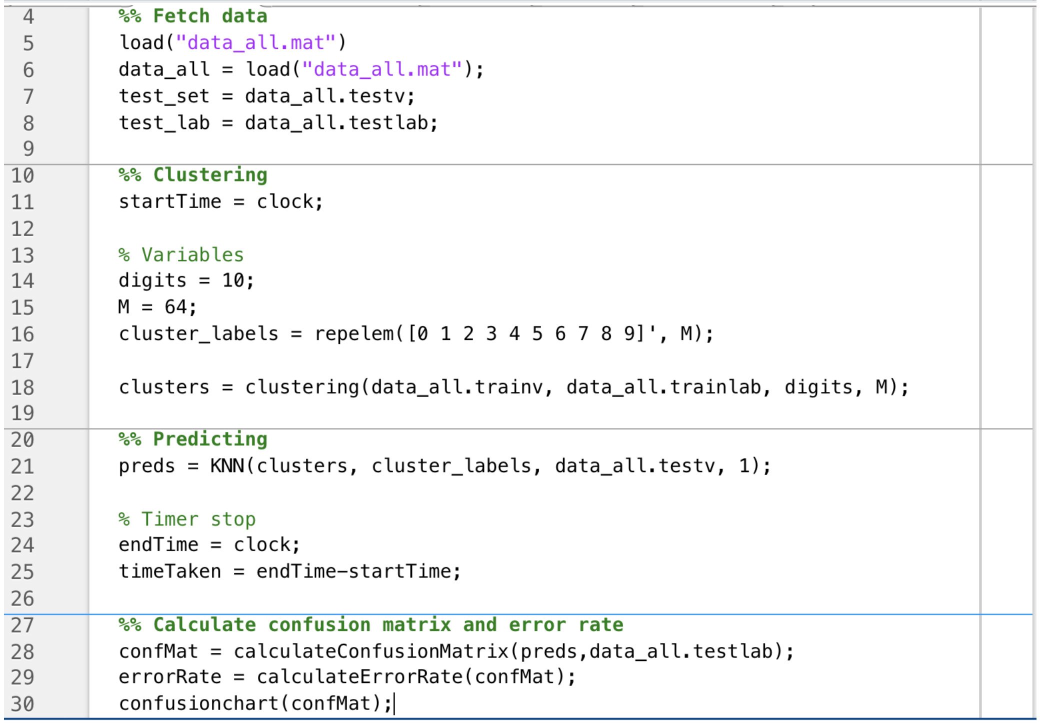
```python
from matplotlib.axis import YAxis
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import plotly.graph_objects as go

plt.rcParams["figure.figsize"] = [15, 3.50]
#plt.rcParams["figure.autolayout"] = True

#headers = ['Alpha','dirt1', 'ErrorRateTraining', 'dirt2','dirt3']
headers = ['Alpha',  'ErrorRateTraining', 'ErrorRateTest']



df = pd.read_csv('iris/worsed_0_features_removed.csv', names=headers)
df.drop('ErrorRateTest', inplace=True, axis=1)


fig = go.Figure(go.Scatter(x = df['Alpha'], y = df['ErrorRateTraining'],
                  name='ErrorRateTraining'))

fig.update_layout(title='Alpha vs ErrorRateTraining',
                   plot_bgcolor='rgb(230, 230,230)',
xaxis_title="Alpha",yaxis_title="Error Percent (%)",
                   showlegend=True)

fig.show()

plt.show()
```

```python
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt


flowers = ["Predicted Setosa", "Predicted Versicolor","Predicted Virginic"]
flowers2 = ["Actual Setosa", "Actual Versicolor","Actual Virginic"]


array = [[30.,  0.,  0.],
 [ 0., 27.,  3.],
 [ 0.,  1., 29.]]
df_cm = pd.DataFrame(array, index = [i for i in flowers2],
                  columns = [i for i in flowers])
plt.figure(figsize = (10,7))
plt.title('Confusion Matrix for train-set with Sepal Width, Sepal Length and Petal Length removed')
sn.heatmap(df_cm, annot=True)
plt.show()
```

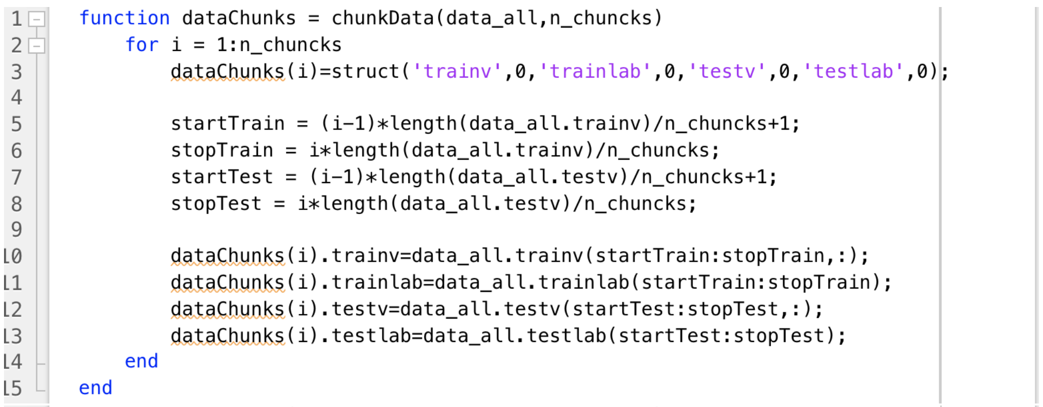
B Digits Appendix



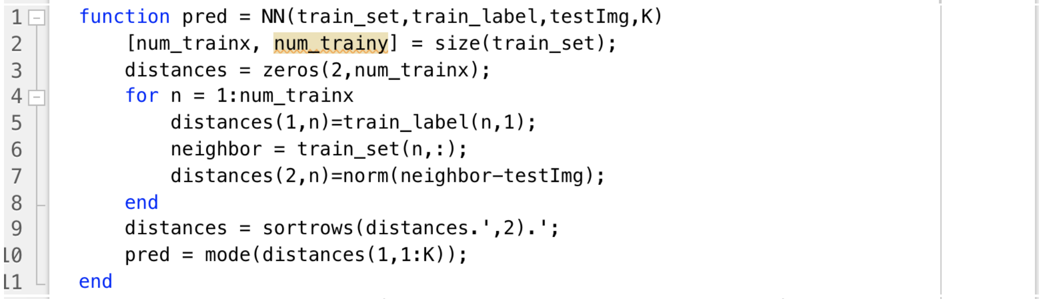
```matlab
clear
close all

%% Fetch data
load("data_all.mat")
data_all = load("data_all.mat");

%% Chunk data
dataChunks = chunkData(data_all,1);
train_set = dataChunks(1).trainv;
train_label = dataChunks(1).trainlab;
test_set = dataChunks(1).testv;
test_lab = dataChunks(1).testlab;

%% Make predictions
startTime = clock;

preds = KNN(train_set, train_label, test_set, 1);

endTime = clock;
timeTaken = endTime-startTime;

%% Calculate confusion matrix and error rate
confMatKort = calculateConfusionMatrix(preds,dataChunks(1).testlab);
errorRateKort = calculateErrorRate(confMatKort);
confusionchart(confMatKort);

```



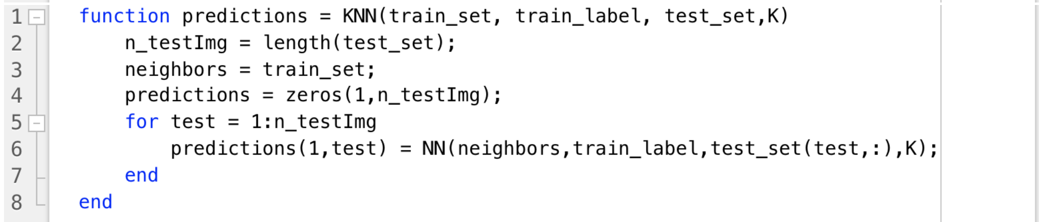
```matlab
%% Fetch data
load("data_all.mat")
data_all = load("data_all.mat");
test_set = data_all.testv;
test_lab = data_all.testlab;

%% Clustering
startTime = clock;

% Variables
digits = 10;
M = 64;
cluster_labels = repelem([0 1 2 3 4 5 6 7 8 9]', M);

clusters = clustering(data_all.trainv, data_all.trainlab, digits, M);

%% Predicting
preds = KNN(clusters, cluster_labels, data_all.testv, 1);

% Timer stop
endTime = clock;
timeTaken = endTime-startTime;

%% Calculate confusion matrix and error rate
confMat = calculateConfusionMatrix(preds,data_all.testlab);
errorRate = calculateErrorRate(confMat);
confusionchart(confMat);
```



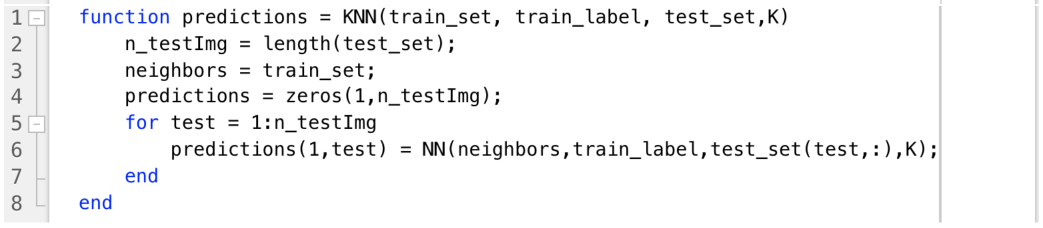
```
function dataChunks = chunkData(data_all,n_chuncks)
    for i = 1:n_chuncks
        dataChunks(i)=struct('trainv',0,'trainlab',0,'testv',0,'testlab',0);

        startTrain = (i-1)*length(data_all.trainv)/n_chuncks+1;
        stopTrain = i*length(data_all.trainv)/n_chuncks;
        startTest = (i-1)*length(data_all.testv)/n_chuncks+1;
        stopTest = i*length(data_all.testv)/n_chuncks;

        dataChunks(i).trainv=data_all.trainv(startTrain:stopTrain,:);
        dataChunks(i).trainlab=data_all.trainlab(startTrain:stopTrain);
        dataChunks(i).testv=data_all.testv(startTest:stopTest,:);
        dataChunks(i).testlab=data_all.testlab(startTest:stopTest);
    end
end
```



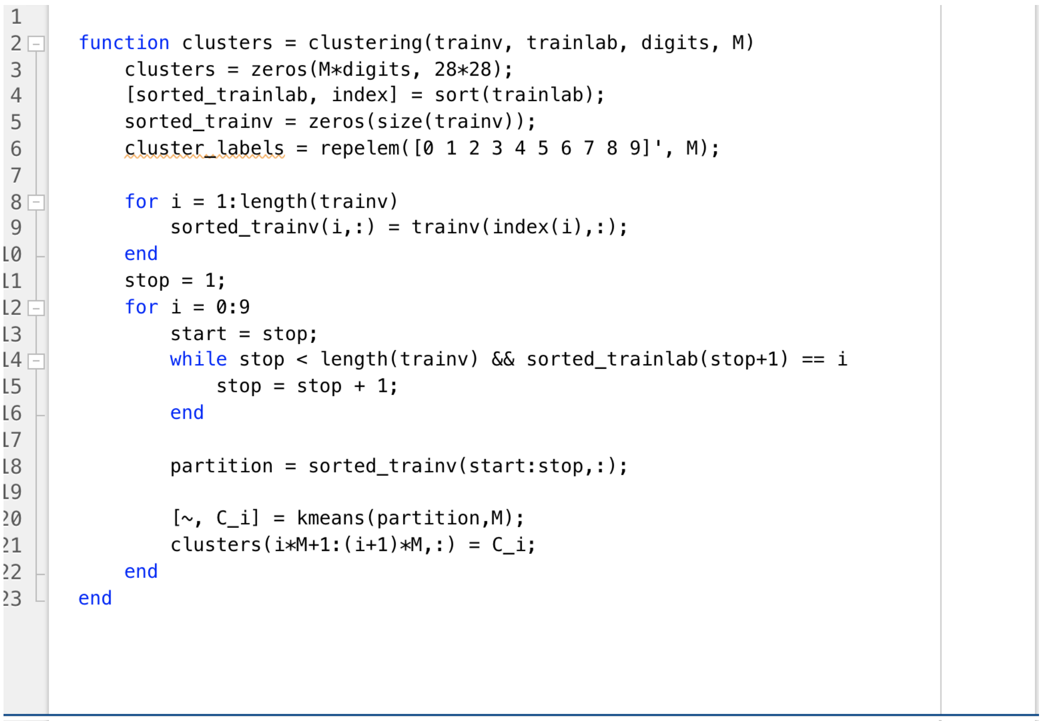
```
function pred = NN(train_set,train_label,testImg,K)
    [num_trainx, num_trainy] = size(train_set);
    distances = zeros(2,num_trainx);
    for n = 1:num_trainx
        distances(1,n)=train_label(n,1);
        neighbor = train_set(n,:);
        distances(2,n)=norm(neighbor-testImg);
    end
    distances = sortrows(distances.',2).';
    pred = mode(distances(1,1:K));
end
```



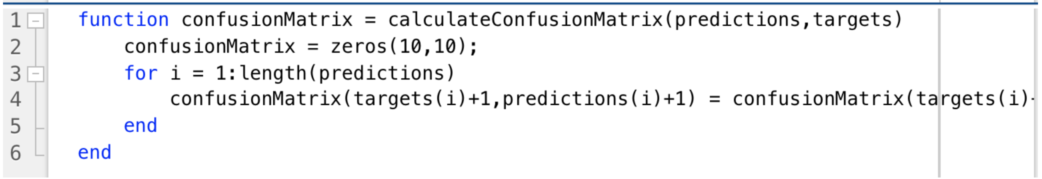
```
function predictions = KNN(train_set, train_label, test_set,K)
    n_testImg = length(test_set);
    neighbors = train_set;
    predictions = zeros(1,n_testImg);
    for test = 1:n_testImg
        predictions(1,test) = NN(neighbors,train_label,test_set(test,:),K);
    end
end
```



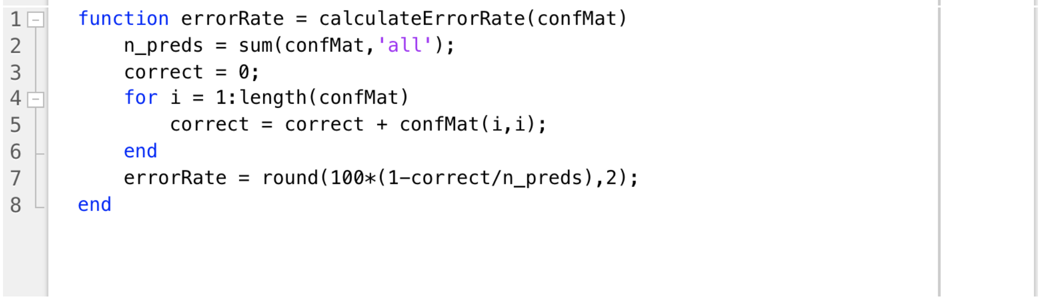
```
function predictions = KNN(train_set, train_label, test_set,K)
    n_testImg = length(test_set);
    neighbors = train_set;
    predictions = zeros(1,n_testImg);
    for test = 1:n_testImg
        predictions(1,test) = NN(neighbors,train_label,test_set(test,:),K);
    end
end
```



```

function clusters = clustering(trainv, trainlab, digits, M)
    clusters = zeros(M*digits, 28*28);
    [sorted_trainlab, index] = sort(trainlab);
    sorted_trainv = zeros(size(trainv));
    cluster_labels = repelem([0 1 2 3 4 5 6 7 8 9]', M);

    for i = 1:length(trainv)
        sorted_trainv(i,:) = trainv(index(i),:);
    end
    stop = 1;
    for i = 0:9
        start = stop;
        while stop < length(trainv) && sorted_trainlab(stop+1) == i
            stop = stop + 1;
        end

        partition = sorted_trainv(start:stop,:);

        [~, C_i] = kmeans(partition,M);
        clusters(i*M+1:(i+1)*M,:) = C_i;
    end
end
```



```
function confusionMatrix = calculateConfusionMatrix(predictions,targets)
    confusionMatrix = zeros(10,10);
    for i = 1:length(predictions)
        confusionMatrix(targets(i)+1,predictions(i)+1) = confusionMatrix(targets(i)
    end
end
```



```
function errorRate = calculateErrorRate(confMat)
    n_preds = sum(confMat,'all');
    correct = 0;
    for i = 1:length(confMat)
        correct = correct + confMat(i,i);
    end
    errorRate = round(100*(1-correct/n_preds),2);
end
```