TTT4275 Project Report

Tittel: Classification of Iris flowers and handwritten numbers	
Forfattere: Inge Eide Johnsen & Ole Edvard Kolvik Valøy	
Versjon: 1.0	Dato: April 30, 2021

Abstract

This projects consists of two parts, where both are programmed in Python:

1. Classification of Iris flowers into 3 different classes based on 4 measured features, sepal length, sepal width, petal length and petal width. This is done by a mean square error based linear classifier. The classifier is trained with 30 labeled examples of each flower, and it is able to classify another 20 flowers of each type with an error rate between 1.67 % and 3.33 %, depending on which of the samples are used for training and which are used for testing.

The classifier is also tested with some of the features removed, which most often results in higher error rates. If one only classifies by the sepal width, the error rate is over 40 %, making it the worst. Removing only this feature does not affect the error rate noticeably. The best single feature for our classifier is the petal width, which alone gives 8.33 % error rate on the verification set. Since we use a relatively small data set, with only 50 samples of each class, we see relatively large changes in error rate with small changes in data sets.

2. Classification of handwritten digits, both with a Nearest Neighbour classifier and with a K-Nearest Neighbours classifier. The first gives an error rate of 3.69 % with all 60,000 references and 10,000 test pictures. This is a very slow method, in our case it took over 3 hours to complete. Clustering the references into 640 clusters cut down the time by a factor 100, while the error rate less than tripled, to 8.23 %. Using the K-Nearest Neighbor method did not change the run time significantly, and increased the error rate. We do not know the exact reason for this, but possible factors will be discussed.

Contents

1 Introduction

This report describes two classification projects: The first project shows how one can classify iris flowers into three classes: Setosa, Versicolor and Virginica, based on the well-known Fisher's Iris data set with 50 labeled examples of the three classes, with an error rate under 10 %. Each example has four measured features of the flowers: petal width and length, and sepal width and length, as illustrated in figure Figure 1. One of the types has no overlap with the two others in two of the features, while the two last classes have an overlap in all features. This makes the classification problem one of the few close to linearly separable practical problems[4]. For this problem a mean square error based linear discriminant classifier will be used.

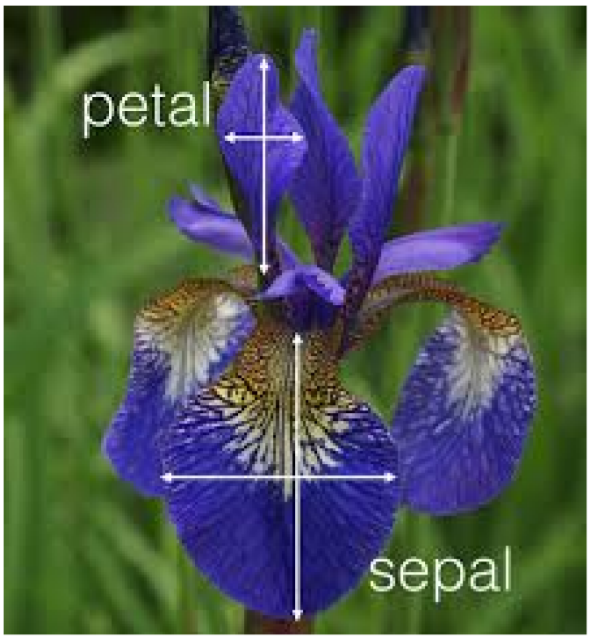

Figure 1: A iris flower with petal and sepal length and width marked. [2]

The second project is about classifying handwritten numbers from the MNIST database. This database contains in total 70,000 labeled pictures, of the numbers from 0 to 9. 60,000 are meant as references, and 10,000, written by other people, for testing. The pictures are 28x28 pixels in 8-bit gray-scale. All numbers are centered and scaled to a similar size. This problem is not linearly separable; there are similarities between many of the numbers[5]. This time, a template based classifier will be used.

2 Theory

2.1 Classifiers

A classifier takes in data about features of objects or measurements, and divides them into classes by given decision rules. Most classifiers do errors, and the error rate, ER, depends on how the data are, the amount of data, and how the decision rules are made.

There are three main types of classification problems: linearly separable, non-linearly separable and non-separable. For separable problems, it is possible to make a decision rule that divides the data points into distinct classes, where the measured features do not overlap, and thus it is theoretically possible to classify without errors. That is not the case for non-separable problems. For those, there are no linear decision rules capable of classifying with 100 % certainty. Nevertheless, it is often still possible to make good decisions, with low error rates.

Even for complex problems, one can often make fairly simple classifiers, trading error rates for calculation complexity. A linear classifier can in many cases be used for a non-separable problem with good results.

In this projects, a linear discriminant classifier and a template based classifier is used. A linear discriminant classifier has parameters that must be estimated based on a training data set with known classes. This process is called training. A template based classifier does not require training, as there are no parameters to train, because the test objects are compared to static templates.

2.2 Linear Discriminant Classifiers

The linear discriminant classifier has a discriminant function[1]

$$g = Wx + w_o \tag{1}$$

where w_o is an offset vector of size C (offset for each class), W is a CxD matrix, with C the number of classes and D the number of features. x is the test object vector.

For a linearly separable problem with two classes, a value in the middle of the highest value of the lowest class and the lowest value of the highest class can be used as a discriminant, because everything over that value will be one class, and everything under will be the other. Since most problems are non-separable, we often cannot discriminate like that. Nevertheless, we can make a value (or line or hyperplane) to discriminate the classes, with some errors. A common way to do so is to make decision planes that has the lowest possible mean square error (MSE) from the points in its respective class. The lowest MSE can be found from the

gradiant of the MSE[1] :

$$\nabla_W MSE = \sum_{k=1}^{N} \left[(g_k - t_k) \circ g_k \circ (1 - g_k) \right] x_k^T \tag{2}$$

where t_k is the true label, and the circles means element-wise multiplication. x_k is the training input. The predicted label, g_k, is a function of the input x_k and the training matrix W, shown in (3).

To discriminate between the classes, a Heaviside function, giving either 0 or 1, meaning either that class or not that class, would be ideal, but for later calculations an approximation to Heaviside, a Sigmoid function, is used, because this is differentiable. The Sigmoid function, given in (3), gives the output g_k, which can later be plugged into (2) to calculate the MSE.

$$g_{ik} = \frac{1}{1 + e^{-W x_{ik}}}, i = 1, \ldots, C \tag{3}$$

The goal is to find the W that gives the lowest MSE. To do so, we do many iterations of (2) and (3), and for every iteration, W are updated:

$$W_i = W_{i-1} - \alpha \nabla MSE \tag{4}$$

where α is the step factor or learning rate. A high α will possibly give high fluctuation in W, preventing W from converging. Too low α on the other hand, gives less fluctuations, but can give a slower learning rate, because the gradient is not valued high enough for each iteration, and a higher number of iterations is needed, giving higher calculation times.

2.3 K-Nearest Neighbour Classifiers

The K-Nearest Neighbour, KNN, algorithm is a type of template based classifier. It has a simple, but slow, decision rule where the input data x is compared to a set of reference data ref, where the k nearest matches is used to classify the input.

To find the best match, the Euclidean distance (5) between them is calculated. When x and ref is represented by two n-dimensional matrices, the element-wise distance is calculated and summed to give a quantitative number of their similarity, the lower the better. This is a very slow method, since one have to iterate through every reference to find the closest match.

$$d(x, ref) = \sqrt{(x - ref)^2} \tag{5}$$

The simplest KNN is a nearest neighbour, NN, were $k = 1$. In this case x is only classified by the closest match in the references. To get a lower error rate, the KNN can be used. The predicted class will now be the class of the majority (mode) of the k closest references. This will possibly give wrong answers where NN gives right answers, because the very nearest

neighbour is not weighted more than a possibly further neighbour, but most often it gives better results.

2.4 K-means Clustering

K-means clustering is a type of clustering that is used to group n points of data into k clusters. The k cluster centroids can then be looked upon as good representatives for the points around. In this way, you can drastically improve run-time by reducing the data-set.

More specifically, K-means clustering starts with a number of different centroid seeds, each containing k random cluster points. It then assigns all data to its nearest centroid point and calculates the variance of the clusters. This is then repeated for the next centroid seed. At last, the centroid seed that resulted in the smallest cluster variances is selected as the best clustering of the data. Finally the mean for each cluster is calculated and said to be the best representatives of the data set. [3]

2.5 Error rate

The error rate ER is given by

$$ER = \frac{FP + FN}{N} \tag{6}$$

where FP is the number of false positives, FN is the number of false negatives and N is the total number of measurements in the testing set. Off course, lower ER is better, but at the same time high calculation times can be problematic. Therefore, it is important to find an optimal learning rate, to get W converging with as few iterations as possible. To find a suitable value for the beforementioned step factor α, we can train the classifier for n different α, and observe where the error rate as a function of α is lowest.

A confusion matrix, as in Figure 2, is a good way to describe the performance of a classification model and can be used to find the error- and accuracy-rate of the model. The vertical axis contains the predicted class while the horizontal axis holds the true values. This makes it easy to identify false positives and negatives.

p\t	0	1	2
0	30	0	0
1	0	28	1
2	0	2	29

Figure 2: Example of confusion matrix with 90 samples. True values are horizontal and predicted values are vertical.

3 Implementation

3.1 Classifying Iris Flowers With a Linear Classifier

The linear classifier for this task is implemented in Python 3.7 with only the NumPy library, and matplotlib for plotting. The code is shown in Appendix B. A flowchart is presented in Figure 3.

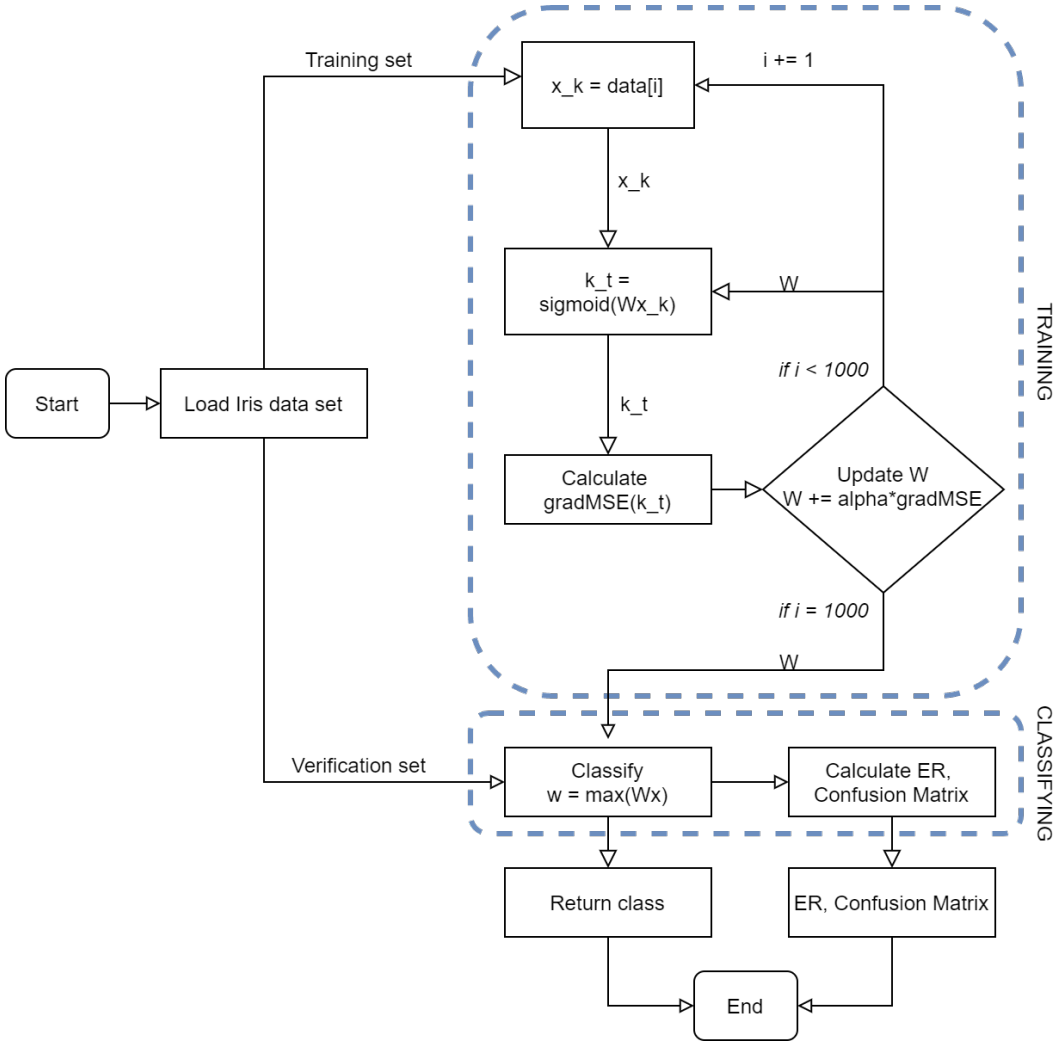

Figure 3: Flowchart for the linear classifier.

The function *importIris*() imports the binary data set, and converts it to a NumPy array with dimensions 3x50x4, respectively 3 classes with 50 samples of each class and 4 features for each example. An important part of this function is to add one element to all feature vectors. This is for the offset ω_o in (1). The returned arrays are the training set (ts, 3x30x5), the verification set (vs, 3x20x5) and the complete set (data, 3x50x5).

These sets are used for training in $trainModel()$, that takes inn a training set, $data$, the number of iterations, N, and the learning rate, α. It runs through (2), (3) and (4) and returns the trained 3x4 matrix W. The classification is done by $classifier()$, which takes in a trained W and a test set, and returns an array with the predicted labels in the same order as the input test set. In addition to doing classification, $confuMatrix()$ finds the error rate and makes a confusion matrix of a pre-trained W and a test set. There are also functions for finding a good learning rate α by (4), for plotting histograms of the features and for removing features to test with fewer features.

In the following sections and in the code, the name of the features and flower types will be like this:

Name	**Number**
Iris Setosa	Type 0
Iris Versicolor	Type 1
Iris Verginica	Type 2
Sepal length	Feature 0
Sepal width	Feature 1
Petal length	Feature 2
Petal width	Feature 3

3.2 Number classifying

These classifiers are also implemented in Python 3.7, with the help of the NumPy, Keras and sklearn libraries. NumPy is used to hold the data in matrices and manipulate it, Keras is used to import the MNIST database of handwritten digits, and sklearn is used to implement clustering. The code is roughly structured as the flowchart in Figure 4 shows.

First, a Euclidean distance function is implemented. The MNIST data is stored in four unit8 arrays containing the training and test data and their labels. To be able to implement (5) with unit8 matrices, a neat trick is used. See $euclidianDistance()$ in Appendix C. When the euclidean distance is calculated for all the pictures, it is easy to find the k-nearest neighbours. Without clustering, 470 billion Euclidean distances is calculated in order to classify 10,000 pictures, this emphasises the need of clustering.

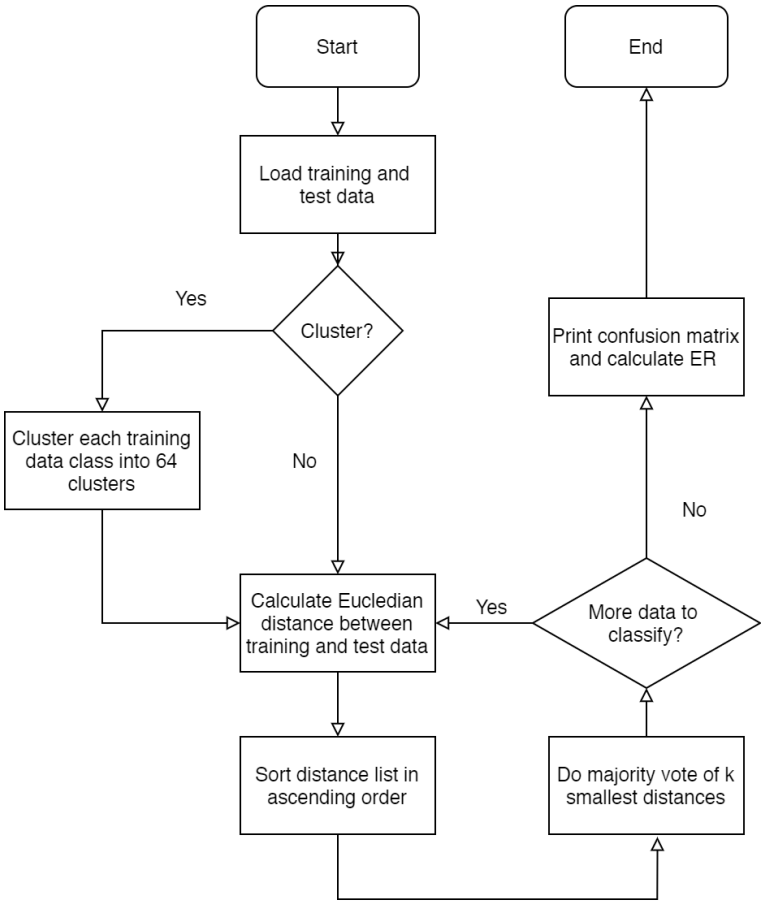

Figure 4: Flowchart of basic implementation of NN- and KNN-based classifiers.

Further on, clustering is implemented with the use of $sklearn.KMeans()$. The data is first sorted by class and divided into a 10x6000x28x28 matrix. It is then reshaped into 10 1D matrices which is fed into the KMeans function together with the desired number of clusters, 64, and the number of centroid seeds, 20. After 416 seconds, the result is a a matrix containing 640 pictures, centroid centers, that is used for much quicker classification. Figure 5 shows three excerpts from the generated cluster. They all represents a slightly different way to write the number two.

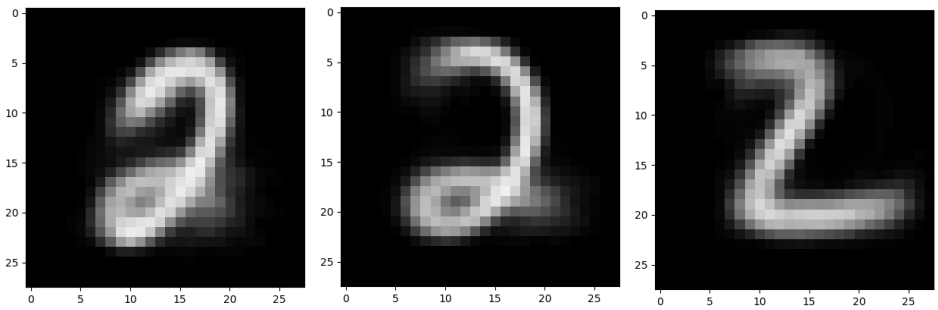

Figure 5: Three of the generated numbers in the cluster.

4 Results and discussion

4.1 Iris

To get good results, we must find a good learning rate α. The plot of error rate as a function of learning rate seen in Figure 6, shows that α between 0.003 and 0.025 gives low ER. Higher values will make the classifier diverge and give a more random ER, while lower values will require more iterations and make the classifier converge. The ER is not only dependent on α, but also on the test set and the training set. Different sets give different plots, but we found $\alpha = 0.009$ to give a generally good classifier.

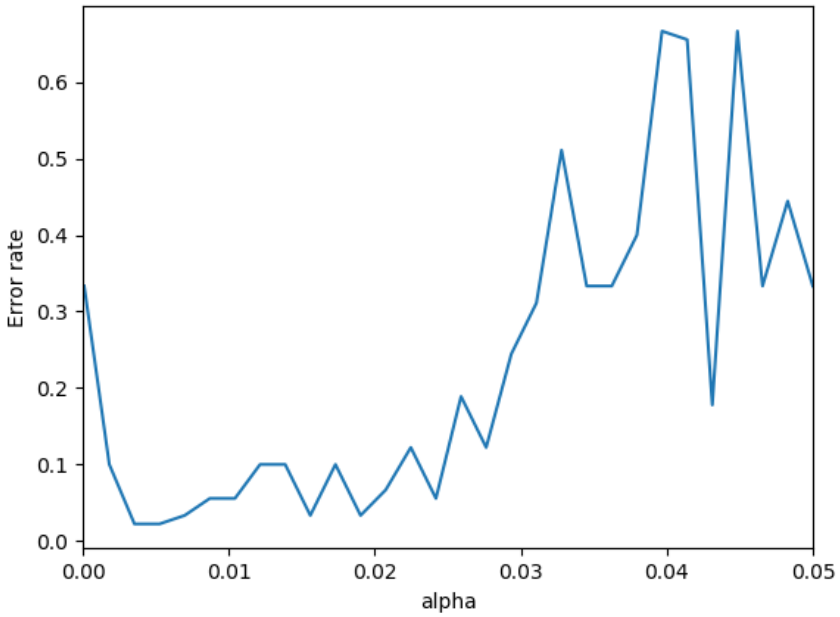

Figure 6: Error rate as a function of step factor α. This is run with the 30 first samples for training and 20 last samples for testing and 1000 iterations.

As can be seen in Figure 7, the MSE oscillates in the first 10-20 samples and converges close to 0 in the next 20 iterations. A higher α usually gives more oscillations. Removing features also gives oscillations for a higher number of iterations. 1000 iterations seems to give good results.

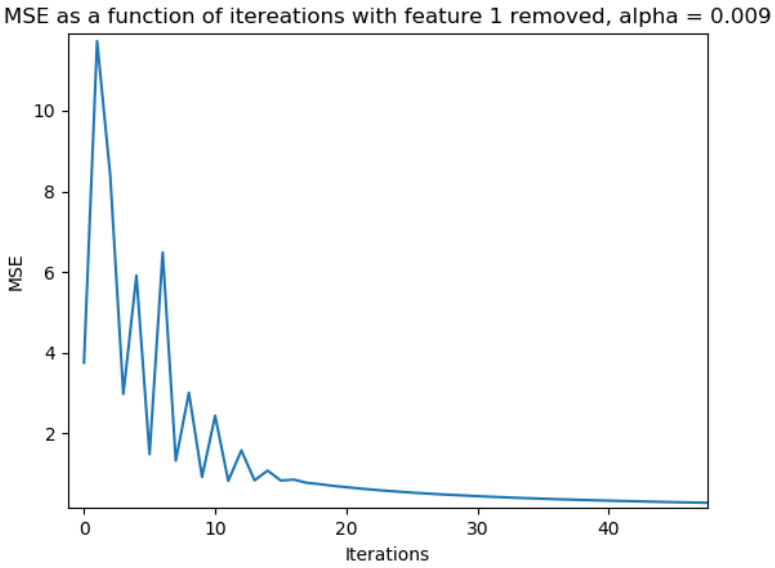

Figure 7: The MSE as a function of the number of iterations, with $\alpha = 0.009$.

Histograms for every feature are shown in Figure 8. It is clear that the petal of the Iris Setosa is very different from the two others. The petal measurements also have a smaller overlap than the sepal measurements for the two other versions. This raises the question of how the classifier will do with some features removed.

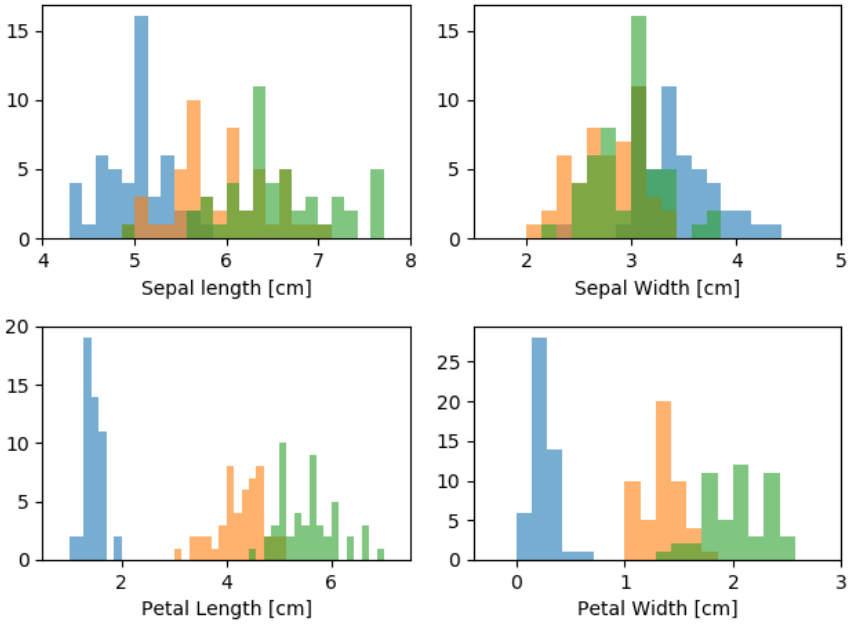

Figure 8: Histogram for all features. All 150 samples are included. Blue is Setosa, orange is Versicolor and green is Virginica. The second axis is the number of flowers.

The different results of removing features can be seen in Figure 9. The error rate for our data is most often higher with fewer features. That is not necessarily the case when removing features in general. Even though more features makes use of more data, adding features with too much overlap can confuse the classifier. In this case, it seems like non of the features overlap enough to make the classifier more confused. However, feature 1 does not add any noticeable improvement either, so the added data from one more feature does no more good than the increased confusion caused by the overlapping does bad.

As Figure 9 shows, when the features are used alone, feature 3 gives an ER of 8.3 % on the verification set. Feature 2 and 3 are clearly the easiest features to differentiate, while feature 0 and 1 give a high ER, especially when classifying Versicolor and Virginica. This corresponds well with the histograms in Figure 8.

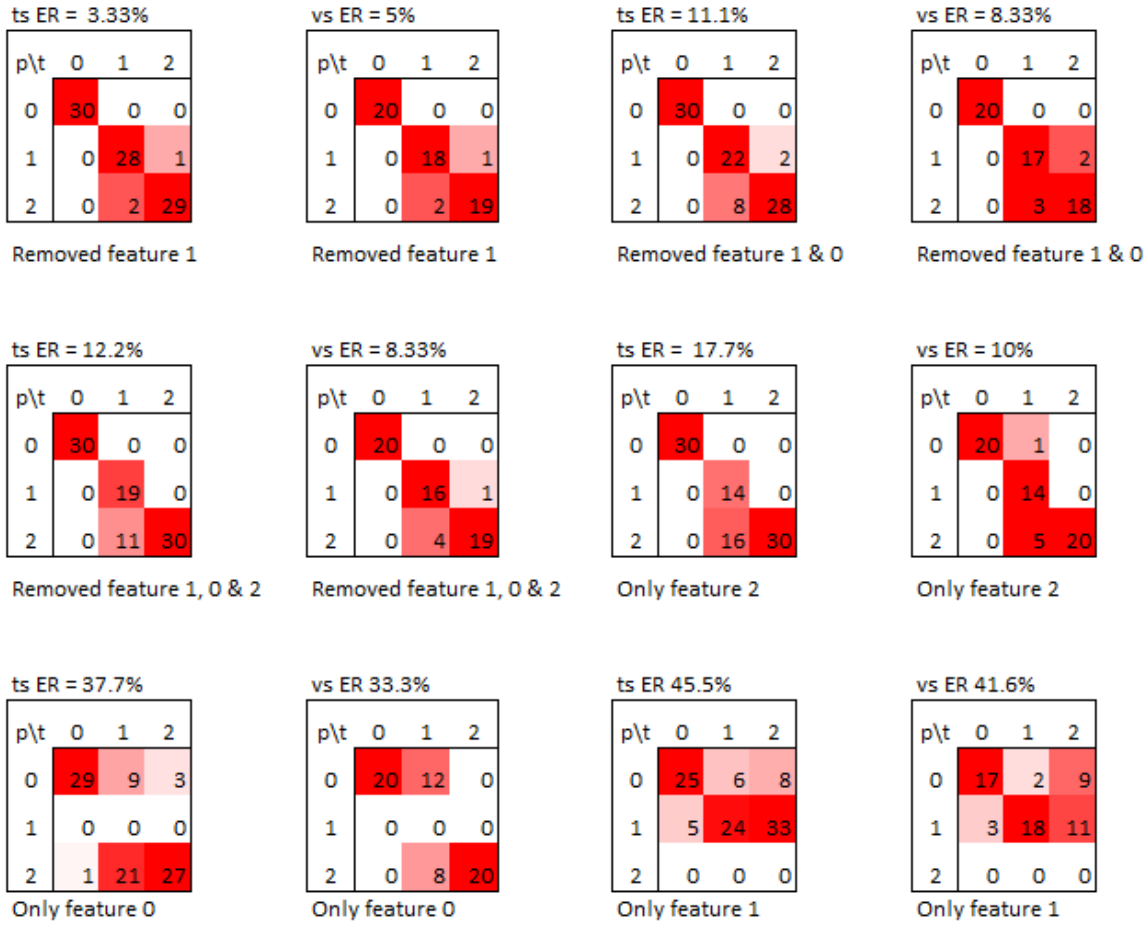

Figure 9: Confusion matrices for different choices of features. ts and vs means respectively tested with test set and tested with verification set. ER means error rate.

Figure 10 shows that the ER is lower if we use the last 30 samples for training and the first 20 for verification compared to the when the first 30 samples are used for training and the last 20 for verification. To check if this is caused by the difference in the sizes of the sets

used for verification, the sizes where adjusted. With the same 25 samples (of each type) for training and testing, the ER is 2.67 %. With the same 25 for training, but the other 25 for verification, this rate was 5.33 %. The corresponding numbers for a 40/10 distribution of training/verification was 8.3 % and 0 %. This emphasises the importance of choosing the correct size of verification and training sets when the sample size is small, and also that the training set must be representative of the entire population.

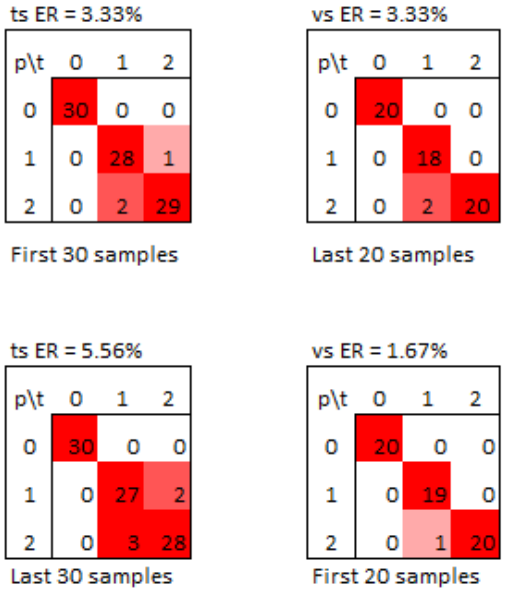

Figure 10: Confusion matrices for different choices of test set and verification set. ts and vs means respectively tested with test set and tested with verification set. ER means error rate.

4.2 Number classifying

Several different scenarios are tested to determine the accuracy for both the NN- and KNN-based classifiers. Table 1 shows a summary of the results for both, with and without clustering. All confusion matrices can be found in Appendix A. In general it can be seen that clustering greatly improves run time, but at the cost of accuracy.

Method	**Error rate (%)**	**Run Time (seconds)**
NN (no clustering)	3.69	9024
KNN (k=3, no clustering)	3.67	9037
NN (clustering)	8.23	90
KNN (k=3, clustering)	10.1	95
KNN (k=5, clustering)	11.1	93
KNN (k=7, clustering)	12.1	90

Table 1: Error rate and run time for different scenarios.

First, the NN-based classifier is run on the full MNIST dataset of 60,000 numbers and then verified with 10,000 numbers. The confusion matrix for this is shown in Figure 11. An error rate of 3.69% is achieved.

p\t	0	1	2	3	4	5	6	7	8	9
0	973	0	9	0	1	2	5	0	9	1
1	2	1129	8	2	9	1	2	20	5	5
2	1	3	987	4	0	0	1	4	6	1
3	0	0	6	965	0	17	0	2	21	7
4	0	1	1	1	937	2	2	4	4	13
5	1	1	0	21	0	848	5	0	18	5
6	2	1	2	0	3	9	943	0	3	1
7	1	0	17	9	4	1	0	989	4	9
8	0	0	2	4	1	5	0	0	894	1
9	0	0	0	4	27	7	0	9	10	966

Figure 11: NN confusion matrix for the full set without clustering. True values are in the top row, predicted values in the left column. Error rate: 3.69 %, run time: 12575 seconds.

Most of the errors happen for similar numbers, like 27 fours that are predicted to be nines, or the 20 sevens that are mistaken for ones. Figure 12 shows some of the misclassified images. It is not hard to understand why these are difficult to predict. The difference-column is interesting to look at because it can say something about how certain the classifier are on the guess. The lower the number - the better match and therefore the classifier can be more certain that the guess is correct. For normal correct guesses, the difference is in between 1000 and 10,000.

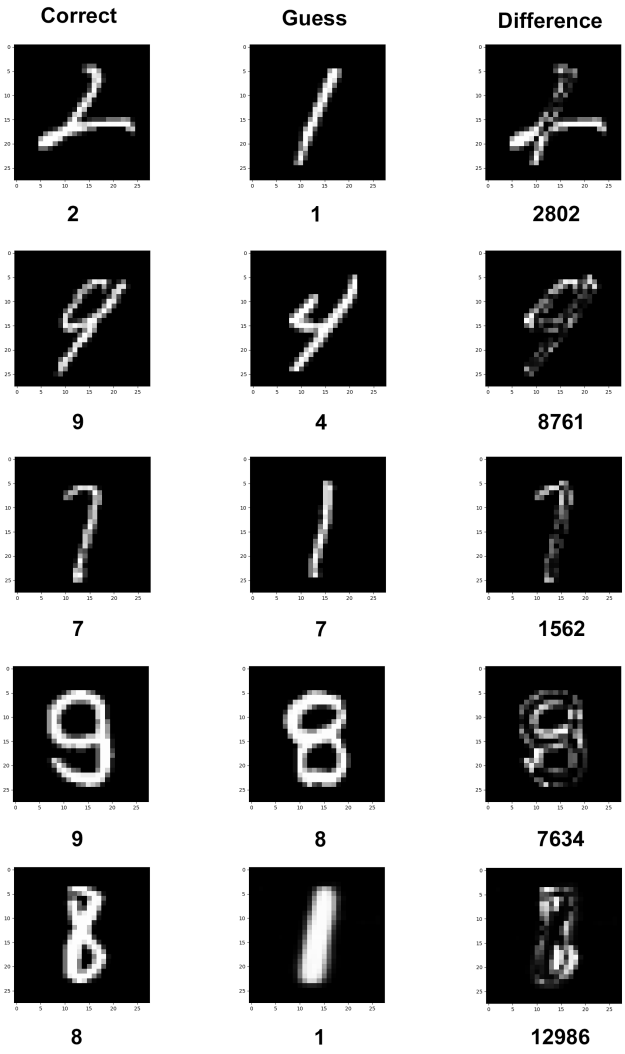

Figure 12: Misclassified pictures with the NN-based classifier. Column one shows pictures from the verification set and column two is the best match from the training set. Column three visualizes the Euclidian distance between the two pictures with the combined distance written under.

Some numbers are correctly classified despite being written poorly. Figure 13 gives three examples of this. Since the training set is so large, there are many examples of poorly writeen numbers. Thus, it can recognize these harder to understand numbers.

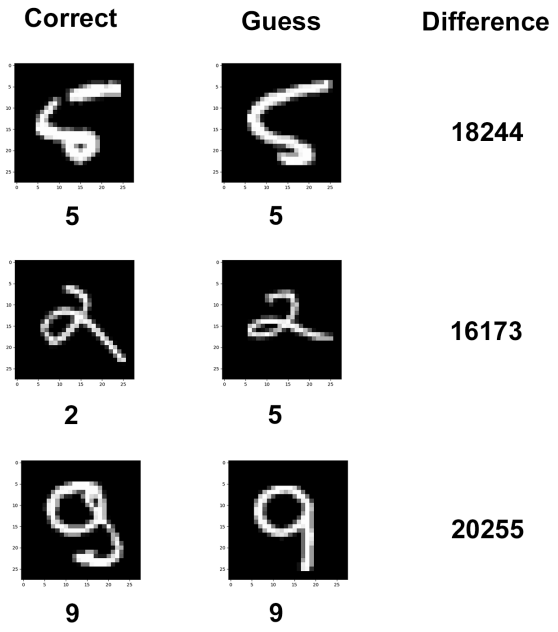

Figure 13: Correctly classified numbers with a high difference.

By introducing clustering, the run time was reduced by a factor of 100 to 90 seconds. Running the clustering algorithm takes 416 seconds, but this is a one time operation as the result can be saved to a file. As Figure 14 shows, the error rate is now 8.23 %. One can now also start to see that most of the misclassifications are numbers being predicted as ones, and eigths beeing classified as a number of wrong numbers. This can happen because many sloppy numbers can look like a one and also many of the ones in the data set is quite thick, so the resulting overlap will be big.

p\t	0	1	2	3	4	5	6	7	8	9
0	966	0	21	1	1	4	4	0	14	7
1	1	1132	88	17	34	21	9	50	28	12
2	1	1	864	1	0	0	0	1	4	2
3	0	1	8	933	0	22	0	0	28	6
4	0	0	3	0	861	2	4	6	11	21
5	4	0	1	26	1	805	3	0	43	1
6	7	1	6	1	11	17	936	0	7	1
7	1	0	31	13	5	4	0	945	11	30
8	0	0	9	12	1	2	1	0	806	0
9	0	0	1	6	68	15	1	26	22	929

Figure 14: NN confusion matrix for the full set with clustering. Error rate: 8.23 %, run time: 90 seconds.

At last, the tests are run with the KNN-based classifier on the cluster data set. Figure 15

shows the confusion matrix for k = 7. The error rate has risen to 12.1 %, but similar patterns can be seen as in the NN on the cluster set. For k = 3, the error rate is slightly smaller at 10.1 %.

p\t	0	1	2	3	4	5	6	7	8	9
0	955	0	27	3	1	9	6	1	18	12
1	2	1131	176	58	50	46	15	92	65	26
2	0	2	753	4	0	1	2	0	3	2
3	0	1	12	893	0	40	0	0	37	7
4	0	0	6	1	831	7	7	4	13	13
5	7	0	2	14	0	745	9	0	39	1
6	15	1	8	3	13	15	918	0	11	1
7	1	0	35	16	1	8	1	900	12	28
8	0	0	12	8	0	2	0	0	746	2
9	0	0	1	10	86	19	0	31	30	917

Figure 15: KNN confusion matrix for the full set with clustering., k = 1 Error rate: 12.1 %, run time: 90 seconds.

Without clustering, the KNN-based classifier with k = 3 only performs marginally better than the NN-based one. With clustering it performs worse for all k's. You would expect the error rate to gradually drop when k gets larger and eventually increase again. Seeing that our results does not follow this behaviour make us question the correctness of our KNN code. Another reason can be that there just is one correct match in the k suggested labels for poorly written numbers. This will result in the distance mode being wrong for the KNN. Also if the mode of the distance vector returns two labels, the smallest will be chosen. Here we could have implemented that the closest one is prioritized. The K-means clustering on the other hand, is performing as expected. The error rate increases because the training size decreases and that the cluster centroids in reality just is approximate representations of the original training set.

An improvement to the KNN can be to implement a weighted system were the closest points counts more when deciding the class. This can potentially solve the issues were KNN is worse than NN.

5 Conclusion

When trained with 30 labeled examples of each flower, with 1000 iterations and a learning rate $\alpha = 0.009$, the linear classifier was able to classify another 20 flowers of each type with an error rate of 3.33 %. The classifier was also tested with some of the features removed, which resulted in higher error rates. If one only classifies by the sepal width, the error rate is over 40 %, making it the worst. Removing only this feature does not affect the error rate noticeably. The best single feature is the petal width, which alone gives 8.33 % error rate on the verification set. Since we use a relatively small data set, with only 50 samples of each class, we see relatively large changes in error rate with small changes in the sample size, or with another sample set with the same size.

Classification of handwritten digits with a template based NN classifier gives an error rate of 3.69 % with all 60,000 references and 10,000 test pictures. This is a very slow method, in our case it took over 3 hours to complete. Clustering into 640 clusters cut down the time by a factor 100, and less than tripled the error rate to 8.23 %. Using the KNN method did not change the run time significantly, and increased the error rate. We do not know the exact reason for this, but we have identified several factors as discussed earlier.

References

[1] Magne H. Johnsen. *Classification*. NTNU.

[2] Magne H. Johnsen. *Project descriptions and tasks in classification*. NTNU.

[3] Jake VanderPlas. *In Depth: k-Means Clustering*. URL: https://jakevdp.github.io/PythonDataScienceHandbook/05.11-k-means.html.

[4] Wikipedia. *Iris flower data set*. URL: https://en.wikipedia.org/wiki/Iris_flower_data_set.

[5] Wikipedia. *MNIST database*. URL: https://en.wikipedia.org/wiki/MNIST_database.

A Confusion Matrices for Number Classifier

p\t	0	1	2	3	4	5	6	7	8	9
0	973	0	9	0	1	2	5	0	9	1
1	2	1129	8	2	9	1	2	20	5	5
2	1	3	987	4	0	0	1	4	6	1
3	0	0	6	965	0	17	0	2	21	7
4	0	1	1	1	937	2	2	4	4	13
5	1	1	0	21	0	848	5	0	18	5
6	2	1	2	0	3	9	943	0	3	1
7	1	0	17	9	4	1	0	989	4	9
8	0	0	2	4	1	5	0	0	894	1
9	0	0	0	4	27	7	0	9	10	966

Figure 16: NN confusion matrix for the full sets without clustering. Error rate: 3.69 %, run time: 12575 seconds.

p\t	0	1	2	3	4	5	6	7	8	9
0	966	0	21	1	1	4	4	0	14	7
1	1	1132	88	17	34	21	9	50	28	12
2	1	1	864	1	0	0	0	1	4	2
3	0	1	8	933	0	22	0	0	28	6
4	0	0	3	0	861	2	4	6	11	21
5	4	0	1	26	1	805	3	0	43	1
6	7	1	6	1	11	17	936	0	7	1
7	1	0	31	13	5	4	0	945	11	30
8	0	0	9	12	1	2	1	0	806	0
9	0	0	1	6	68	15	1	26	22	929

Figure 17: KNN confusion matrix for the full sets with clustering. Error rate: 8.23 %, run time 90 seconds.

p\t	0	1	2	3	4	5	6	7	8	9
0	967	0	29	3	3	7	8	1	18	9
1	1	1132	130	37	52	31	11	71	43	19
2	0	1	812	5	0	1	2	0	6	2
3	1	1	9	924	0	32	0	0	46	7
4	0	0	2	1	834	8	5	7	10	21
5	4	0	1	12	1	783	4	0	33	2
6	6	1	7	1	12	14	928	0	6	1
7	1	0	30	11	1	2	0	921	15	25
8	0	0	11	10	0	1	0	0	768	2
9	0	0	1	6	79	13	0	28	29	921

Figure 18: KNN confusion matrix for the full sets with clustering and $k = 3$. Error rate: 10.1 %, run time: 95 seconds.

p\t	0	1	2	3	4	5	6	7	8	9
0	955	0	27	3	1	9	6	1	18	12
1	2	1131	176	58	50	46	15	92	65	26
2	0	2	753	4	0	1	2	0	3	2
3	0	1	12	893	0	40	0	0	37	7
4	0	0	6	1	831	7	7	4	13	13
5	7	0	2	14	0	745	9	0	39	1
6	15	1	8	3	13	15	918	0	11	1
7	1	0	35	16	1	8	1	900	12	28
8	0	0	12	8	0	2	0	0	746	2
9	0	0	1	10	86	19	0	31	30	917

Figure 19: KNN confusion matrix for the full sets with clustering and $k = 7$. Error rate: 12.1 %, run time: 90 seconds

B Python Code for Iris Task

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

def importIris(filename,slice1, slice2):
    rawData = np.genfromtxt(filename, dtype=str, delimiter=",") # feature*example
    data = np.empty((3,50,4)) # classes*examples*features
    indices = [0,0,0] #

    for row in rawData: # Removing labels
        if row[len(row)-1] == "Iris-setosa":
            data[0,indices[0]] = row[0:4]
            indices[0] +=1
        elif row[len(row)-1] == "Iris-versicolor":
            data[1,indices[1]] = row[0:4]
            indices[1] +=1
        elif row[len(row)-1] == "Iris-virginica":
            data[2,indices[2]] = row[0:4]
            indices[2] +=1

    # Add 1 to end of features, for the offsets
    classes, samples, features = data.shape
    data = np.dstack([data, np.ones([classes, samples, 1])])

    ts = data[:,:slice1] # Training set
    vs = data[:, slice1:(slice1+slice2)] # Testing set
    return ts, vs, data

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

def trainModel(data, N, lr): #training data, N = iterations, lr = alpha = learning rate
    classes, samples, features = data.shape
    W = np.zeros((classes, features)) # the matrix to train
    t_k = np.array([[1,0,0],[0,1,0],[0,0,1]]) # The ideal probability matrix
    g_k = np.zeros((classes)) # The input values we want to minimise MSE for
    g_k[0] = 1 # always start with 1

    MSEset = []
    for k in range(N):
        MSE = 0
        gMSE = 0
        for i in range(samples):
            x_k = data[:,i]
            z_k = np.matmul(W,x_k.T) # The guessed value
            g_k = sigmoid(z_k) #

            grad1 =  np.multiply(np.multiply((g_k-t_k), g_k),(1-g_k))
            gMSE += np.matmul(grad1,  x_k) # The gradiant of the MSE

            MSE += 1/2 * np.multiply((g_k-t_k).T,(g_k-t_k)) #MSE only for plotting
        MSEset.append(MSE)

```

```
        W -= lr*gMSE # The trained matrix
    MSEset = np.array(MSEset)
    return W #, MSEset

def classifier(W, test_set): # W must be the already trained matrix
    for s in test_set[2,:]: # Every element checked with W
        omega = int(np.argmax(np.matmul(W,s.T))) # The best class
    return omega

def confuMatrix(W, test_set): # Classifies, finds error rate and confusion matrix
    classes, samples, features = test_set.shape
    confusion_matrix = np.zeros((classes, classes))
    for flower in range(classes):
        for s in test_set[flower,:]: # Every element checked with W
            omega = int(np.argmax(np.matmul(W,s.T)))
            confusion_matrix[omega][flower] += 1

    errorRate = (confusion_matrix[0,1]+confusion_matrix[0,2] + confusion_matrix[1,0] \
    +  confusion_matrix[1,2] + confusion_matrix[2,0]\
    + confusion_matrix[2,1]) / (confusion_matrix.sum())
    return confusion_matrix, errorRate

def calculateAlpha(start, end, n, ts, trainSize): #
    alphaSet = np.linspace(start, end, n)
    errorSet = []

    for a in alphaSet:
        W = trainModel(ts,trainSize, a)
        CF, ER = confuMatrix(W, ts) # Could have used another data set
        errorSet.append(ER)

    errorSet = np.array(errorSet)
    alpha = np.argmin(errorSet)

    plt.scatter(alphaSet, errorSet)
    plt.xlabel("alpha")
    plt.ylabel("Error rate")
    plt.show()
    print("Alpha: ", alpha)
    return errorSet, alpha

def plotHistogram(set, feature):
    bins = np.arange(0, 8, 1/7)
    featureNames = ["Sepal length", "Sepal Width", "Petal Length", "Petal Width"]
    fig, axs = plt.subplots(3,1)
    for (i, ax) in enumerate(axs.flat):
        ax.hist(set[i,:,feature].tolist(),bins=bins)
        ax.set_ylabel("Class %d" %(i))
        #ax.set_ylim((0,13))
    plt.xlabel("%s [cm]" % (featureNames[feature]))
    plt.show()

def removeFeature(set, feature):
    return np.delete(set, feature, axis = 2)
```

```
107
108 #task 1a
109 ts1, vs1, data1 = importIris('Code\iris\iris.data', 30, 20)
110 print("Task 1a)")
111 print("Training samples:",ts1.shape)
112 print("Verification samples:",vs1.shape)
113
114 #1b
115 #calculateAlpha(0.01,0.1,30,ts1,1000) # valgt 0.009
116 #calculateAlpha(0.0001,0.05,30,ts1,500) # valgt 0.009
117 #ts1 = removeFeature(ts1, 1)
118 #vs1 = removeFeature(vs1, 1)
119 #ts1 = removeFeature(ts1, 0)
120 #vs1 = removeFeature(vs1, 0)
121
122 #1c
123
124 W1= trainModel(ts1, 1000, 0.009)
125 CMts1, ERts1 = confuMatrix(W1,ts1)
126 CMvs1, ERvs1 = confuMatrix(W1,vs1)
127 print("All features")
128 print("ts = 30, vs = 20, ts used for training")
129 print("Training set error rate: ", ERts1)
130 print(CMts1)
131 print("Verification set error rate: ", ERvs1)
132 print(CMvs1)
133 """
134
135 plt.plot(MSEset[:,0,0])
136 plt.xlabel("Iterations")
137 plt.ylabel("MSE")
138 plt.title("MSE as a function of itereations with feature 1 removed, alpha =
        0.009")
139 plt.show()
140
141 #1d
142 vs2, ts2, data2 = importIris('Code\iris\iris.data', 20, 30)
143
144 W2 = trainModel(ts2, 1000, 0.009)
145 CMts2, ERts2 = confuMatrix(W2,ts2)
146 CMvs2, ERvs2 = confuMatrix(W2,vs2)
147 print("\nts = 20, vs = 30, last 30 used for training")
148 print("Training set error rate: ", ERts2, "\n",CMts2)
149 print("Verification set error rate: ", ERvs2, "\n",CMvs2)
150
151 """
152
153 #2a
154 #plotHistogramAll(data1)
155 #plotHistogram(data1, 0)
156 #plotHistogram(data1, 1)
157 #plotHistogram(data1, 2)
158 #plotHistogram(data1, 3)
159 #
160
161 ts1 = removeFeature(ts1, 1)
162 vs1 = removeFeature(vs1, 1)
```

```
163
164 W1 = trainModel(ts1, 1000, 0.009)
165
166 CMts1, ERts1 = confuMatrix(W1,ts1)
167 CMvs1, ERvs1 = confuMatrix(W1,vs1)
168 print("\nRemoved feature 1")
169 print("Training set error rate: ", ERts1)
170 print(CMts1)
171 print("Verification set error rate: ", ERvs1)
172
173
174 ts1 = removeFeature(ts1, 0)
175 vs1 = removeFeature(vs1, 0)
176
177 W1 = trainModel(ts1, 1000, 0.009)
178
179 CMts1, ERts1 = confuMatrix(W1,ts1)
180 CMvs1, ERvs1 = confuMatrix(W1,vs1)
181 print("\nRemoved feature 0")
182 print("Training set error rate: ", ERts1)
183 print(CMts1)
184 print("Verification set error rate: ", ERvs1)
185 print(CMvs1)
186
187 ts1 = removeFeature(ts1, 0)
188 vs1 = removeFeature(vs1, 0)
189
190 W1 = trainModel(ts1, 1000, 0.009)
191
192 CMts1, ERts1 = confuMatrix(W1,ts1)
193 CMvs1, ERvs1 = confuMatrix(W1,vs1)
194 print("\nRemoved feature 2")
195 print("Training set error rate: ", ERts1)
196 print(CMts1)
197 print("Verification set error rate: ", ERvs1)
198 print(CMvs1)
```

C Python Code for Number classification

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

def printNumber(first_image):
    first_image = np.array(first_image, dtype='float')
    pixels = first_image.reshape((28, 28))
    plt.imshow(pixels, cmap='gray', vmin=0, vmax=255)
    plt.show()

def eucledianDistance(img1, img2):
    a = img1-img2 # Since we import uint8, this will give zero for img1[i]-img2[i] < 0
    b = np.uint8(img1<img2) * 254 + 1 # smart trick
    return np.sum(a*b)

def NNpredictor(test, ref, refLables, trueLables):  # No need for teslabels (not without clustering)

    predictedLables = []
    i = 0
    for testPicture in test:
        distance = []  # Distance array, from testpic to all refpics
        correctLable = trueLables[i]
        for refPicture in ref:
            distance.append(eucledianDistance(refPicture, testPicture))

        testLable = refLables[np.argmin(distance)]  # Index to lowest distance
        predictedLables.append(testLable)  # List with the
        """
        #fway of display different numbers that meet the requirement
        if correctLable 1= testLable:
            print("Correct label", correctLable, " predicted lable: ", testLable)
            print(eucledianDistance(testPicture, ref[np.argmin(distance)]))
            printNumber(testPicture)
            printNumber(ref[np.argmin(distance)])
            printNumber(differenceImage(testPicture,ref[np.argmin(distance)]))

        i+=1
        """
    return predictedLables

def KNN(test, ref, testLables, refLables, k):
    predictedLables = []
    for testPicture in test:
        distance = []  # Distance array, from testpic to all refpics
        for refPicture in ref:
            distance.append(eucledianDistance(refPicture, testPicture)) # distances
        knnIndex = np.argpartition(distance,k)[:k] # List with index to k lowest distances
        knnValue = np.take(refLables, knnIndex)
```

```
51          predictedLables.append(np.bincount(knnValue).argmax())  # Mode of KNN-
    list
52      return predictedLables
53
54
55  def confusionMatrix(test, testLables, ref, refLables, k=0):
56      if k==0:
57          predictedLables = NNpredictor(test, ref, refLables, testLables)
58      else:
59          predictedLables = KNN(test, ref, testLables, refLables, k)
60      #print("actual label: %s,\n   predicted: %s \n" % (trueLables, np.array(
    predictedLables)))
61      confuMatrix = np.zeros((10, 10))
62      for i in range(len(predictedLables)):
63          confuMatrix[predictedLables[i], testLables[i]] += 1
64      correct = 0
65      for el in range(10):
66          correct += confuMatrix[el][el]
67      errorRate = (len(predictedLables) - correct) / len(predictedLables)
68
69      return confuMatrix, errorRate
70
71
72  def differenceImage(img1, img2):
73      a = img1-img2 # Since we import uint8, this will give zero for img1[i]-
    img2[i] < 0
74      b = np.uint8(img1<img2) * 254 + 1 # smart trick
75      return a*b
76
77
78
79  def getClusters(ref, refLables, nClusters=64):
80
81      #clusters = [] # [(label, kmeans[])] # 10*64*28*28
82      clusterData = [] # [kmeans] 10*64*28*28
83      classes = range(len(np.unique(refLables))) #creates a list [0:9]
84
85      for c in classes: # For every number 0-9
86          i = np.where(refLables == c)[0] #indices for the number
87          data = ref[i] # All pictures of number c
88          data = data.reshape((data.shape[0], -1)) #make it 1d
89          kmeans = KMeans(n_clusters=nClusters, n_init=20, n_jobs=1) # mean-
    picture
90
91          kmeans.fit_predict(data) # cluster centers and index
92          cluster = kmeans.cluster_centers_ #
93          cluster = cluster.reshape((64,28,28))
94          clusterData.extend(cluster)
95
96      clusterData = np.array(clusterData)
97      clusterData = np.uint8(clusterData)
98      #create a 10*64 array with 0-9
99
100     clusterLables = []
101     for c in range(10):
102         clusterLables.extend([c]*nClusters)
103     clusterLables = np.array(clusterLables)
```

```
104
105     return clusterData, clusterLables
106
107
108 #Saves array to file
109 def saveFile(arr, name):
110     np.save(name, arr)
111
112 def loadFile(name):
113     loaded_arr = np.load(name)
114     return loaded_arr
115
116 #Starts timer
117 t1 = time.time()
118
119 #Load dataset (60000x28x28)
120 (train_x, train_y), (test_x, test_y) = mnist.load_data()
121
122
123 #task 1
124 #conMatrix, ER= confusionMatrix(test_x[:10], test_y[:10],  train_x, train_y,0)
125 #print(conMatrix, "\n ER: ", ER)
126
127 #task 2
128
129 N = 6000 # Number of pictures we want to use in each training
130 M = 10 # Number of classes
131 nClusters = 64 # Number of clusters for each class
132
133 #clusterData, clusterLables = getClusters(train_x, train_y, nClusters)
134
135 clusterData = loadFile('clusterData.npy')
136 clusterLables = loadFile('clusterLables.npy')
137
138 #printNumber(clusterData[129])
139 #printNumber(clusterData[140])
140 #printNumber(clusterData[180])
141 #printNumber(clusterData[170])
142
143
144
145 conMatrix, ER= confusionMatrix(test_x, test_y,  clusterData, clusterLables,2)
146 np.set_printoptions(precision=3)
147 print(conMatrix)
148 print("Error",ER)
149
150 #task 2c
151
152 #conMatrix, ER= confusionMatrix(test_x[10:30], test_y[10:30],  train_x,
    train_y,7)
153 #print(conMatrix, "\n ER: ", ER)
154
155 t2 = time.time()
156 t = t2 - t1
157 print("Runtime: %.20f" % t)
158 #saveFile(clusterLables, 'clusterLables.npy')
```