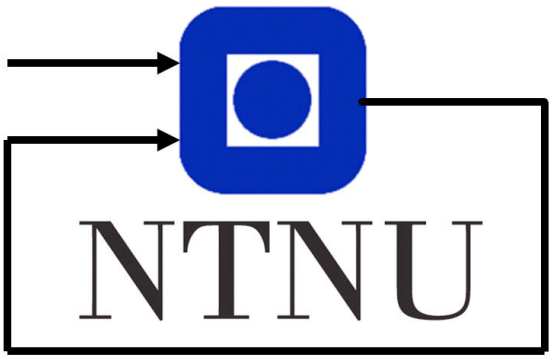
TTT4275 Classification project report - Iris and Handwritten numbers

Lyder K. Jacobsen
Mikael Shahly

April 30, 2024



Department of Engineering Cybernetics

Abstract

The following classification project is part of the course *TTK4275 Estimation, Detection and Classification*. The project consists of designing various classifiers and discussions of the respective performances. *Iris* and *Handwritten numbers* are the chosen tasks to be solved. The Iris task consists of training linear classifiers to be used on the "Fisher Iris data" database, while the Handwritten numbers tasks consists of designing template based classifiers to be used on the MNIST database.

In the Iris task, an error rate of 1.7% or 3.3%, depending on how the test and train set was defined, on the test set was achieved using all the features from the "Fisher Iris data" dataset. The linear classifier also performed very well using only the petal width feature from the dataset, attaining an error rate of 3.3%.

In the handwritten numbers task, when using the whole training set as templates and a NN based classifier, the computation time was very long, while the error rate also was low. When clustering the training set as templates, the computation time was vastly decreased, but the accuracy suffered. Furthermore, when introducing the kNN classifier, the error rate was slightly reduced.

Contents

Abstract **1**

1 Introduction **1**

2 Theory **1**
2.1 Relevant theory for both tasks 1
2.2 Iris task theory 2
2.3 Handwritten numbers task theory 3

3 Task **4**
3.1 Iris task 4
3.2 Handwritten numbers task 4

4 Implementation and results **5**
4.1 The Iris task implementation and results 5
4.2 The handwritten numbers task implementation and results 10

5 Conclusion **14**

References **14**

6 Appendix
6.1 Iris code . . . . .
6.2 Handwritten numbers code . . . . .

1 Introduction

In this project, the goal was to get hands-on experience designing varying classifiers including a linear classifier, nearest-neighbor classifier and k-nearest-neighbor classifier. The project also had a focus on discussing results, and how different approaches may affect the outcome. Knowing how to design and use different classifiers is an essential part of machine learning knowledge, which is a rapidly increasing science in many industries today. Computer vision, search engines, autonomous systems, economical predictions and power supply control are all examples of vastly different subjects that all make use of classification in some ways.

In chapter 2, theory that is relevant in order to understand the implementations and discussion of results is presented. In chapter 3, the tasks solved in the project are described. In chapter 4, matlab and pyhon implementations are explained. Furthermore, the results are presented and discussed. In chapter 5, we have written a conclusion summarising the results and commenting on the knowledge obtained throughout the project. Lastly, chapter 6 is an appendix where all code written for the project is provided.

2 Theory

This chapter is a presentation of the theory needed in order to understand the tasks that were solved. Two tasks were solved in the project, and respective relevant theory for each is provided below.

2.1 Relevant theory for both tasks

The classification models used to solve each task are both in the category of supervised learning. This means that the models use data with known class labels attached to each data point, referred to as training data. They then uses this data to learn how to perform classification on data with unknown classes[1].

For models based on supervised learning, an important concept that a they can underfit or overfit to the training data. Underfitting refers to a model which is to simple to capture the underlying pattern in the data. The model will therefore have a large error on both training and test data. Overfitting refers to a model that is too complex and captures noise or random fluctuations in the training data as if they were genuine patterns[1]. It will then fit extremely well to training data, but fail to generalize to the test data. The model will therefore have a small error on the training data, but have a large error on the test data. For all classification models made for the tasks, a trade off between these extremes must be made. A more mathematically rigorous approach to this concept is found in the bias-variance trade-off. Any error made by a classifier on a test set can be decomposed into an irreducible error, a bias term and a variance term [1]. A quick example can be made for a mean square error function.

$$E[(f(x) - \hat{f}(x))^2] = \text{bias}[\hat{f}(x)]^2 + var(\hat{f}(x)) + var(\epsilon) \tag{1}$$

Where $f(x)$ is the function that describes the true pattern of the classes, $\hat{f}(x)$ is the function the model predicts $f(x)$ to be and ϵ is the irreducible error.

models that underfit will have a large bias and low variance, while models that overfit will have a large variance and low bias. This shows directly that only way to have both

low variance and low bias (and therefore low error rate) is to strike a trade off between overfitting and underfitting to the training set.

2.2 Iris task theory

In the Iris task a linear classifier was built in order to perform the necessary classification. A linear classifier refers to classifiers that create a linear decision boundary in feature space, in order to separate different data points into different classes. A linear decision boundary is made by inputting the data into a linear discriminant function

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b \tag{2}$$

A data point can then be classified, based on it's location in the partitioned feature space. Note that a single discriminant function only partitions the feature space into two partitions. This means that only binary classification can be performed with a single discriminant function. In order classify between multiple classes, multiple discriminate functions must be used. As the Iris task contains three different classes, three separate linear discriminant functions are used. The following decision rule is then used to partition the feature space:

$$g_i(\boldsymbol{x}) = max(g_i(\boldsymbol{x})) \tag{3}$$

Each discriminant function is then given by

$$g_i(\boldsymbol{x}) = \boldsymbol{w}_i^T \boldsymbol{x} + b, \qquad i = 1, ..., C \tag{4}$$

Where C corresponds to the amount of classes. This can be written compactly in matrix form as:

$$\boldsymbol{g(x)} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{5}$$

Where $\boldsymbol{W}$ is a nxm matrix with n equal to amount of features and m is amount of classes in our dataset.

In order for the linear classifier to be accurate, it must be trained. This is done by selecting the weight matrix ($\boldsymbol{W}$) and bias vector ($\boldsymbol{b}$) that minimizes a cost function. The cost function quantifies the amount of error the classifier has on a dataset with known classes. For this task the mean square error (MSE) was used as a cost function. The MSE can be given as:

$$MSE = \frac{1}{N}\sum_{k=1}^{N}(\boldsymbol{g}(\boldsymbol{x}_k) - \boldsymbol{t}_k)^2 \tag{6}$$

where N is the number of data samples in our dataset and $\boldsymbol{t}_k$ is a hot-one encoded vector that represents the corresponding class of the data sample. So if the class of a data point is class 1 of three possible classes, we have $\boldsymbol{t}_k = [1, 0, 0]$.

To minimize the cost function, gradient descent is performed in order to update $\boldsymbol{W}$ and $\boldsymbol{b}$. As the gradient of a function will always point in the direction of steepest ascent, the negative value of it will always point in the direction of steepest descent [2]. Which gives the following update rule for $\boldsymbol{W}$ and $\boldsymbol{b}$

$$\boldsymbol{W}_{j+1} = \boldsymbol{W}_j - \alpha \nabla_{\boldsymbol{W}} MSE \tag{7}$$

$$\boldsymbol{b}_{j+1} = \boldsymbol{b}_j - \alpha \nabla_{\boldsymbol{b}} MSE \tag{8}$$

This is repeated for j number of iterations, in order to achieve convergence. Both α (commonly referred to as learning rate) and the number of iterations will then be very

important hyperparameters, that must be tuned manually, in order to achieve the desired accuracy of the classifier.

Finally, as we one-hot encoded our class information in $\boldsymbol{t}_k$, it will only take on values in the range $(0, 1)$. Ideally, we would want equation 5 to also only output values in this range. To achieve this, we use a sigmoid function as an activation function. The sigmoid function acts as a differentiable approximation to the heavy side function and is given

$$\sigma_i(\boldsymbol{x}) = \frac{1}{1 + e^{g_i(\boldsymbol{x})}} \tag{9}$$

Doing this gives us the following gradients for the cost function.

$$\nabla_{\boldsymbol{W}} MSE = \frac{2}{N} \sum_{k=1}^{N} (\boldsymbol{\sigma_k} - \boldsymbol{t_k}) * (\boldsymbol{\sigma_k} * (1 - \boldsymbol{\sigma_k})) * \boldsymbol{x_k^T} \tag{10}$$

$$\nabla_{\boldsymbol{b}} MSE = \frac{2}{N} \sum_{k=1}^{N} (\boldsymbol{\sigma_k} - \boldsymbol{t_k}) * (\boldsymbol{\sigma_k} * (1 - \boldsymbol{\sigma_k})) \tag{11}$$

2.3 Handwritten numbers task theory

Classification using the nearest-neighbor rule is a conceptually simple method. Considering a training set $\mathcal{D}^n = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ of labeled prototypes, one can predict the label of a test point by assigning it the label of the nearest prototype[2]. In this context, the "nearest" point implies the the point in which the Euclidian distance to the test point is the smallest. The Euclidian distance can be calculated using

$$d = \sqrt{\sum_{i=0}^{n} (x_i - x_i')^2}, \tag{12}$$

where $\boldsymbol{x}$ is the test point and $\boldsymbol{x}'$ is the labeled prototype. Since we are comparing distances, to save computation, one can rather compare the squared Euclidian distances. This labeling results in the space being partitioned into sections around the prototypes in which a test point being in a certain partition assigns it the label of the prototype within the partition, see fig. 1.

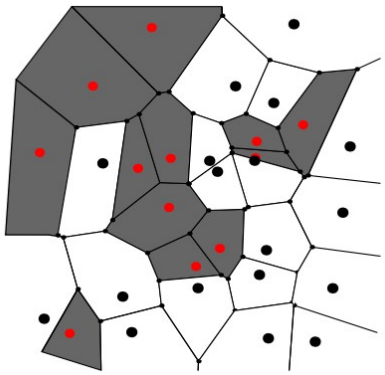


Figure 1: Partitioning as a result of the NN rule in the 2-dimensional case[2]

A natural extension of the NN rule is the kNN rule, k-Nearest-Neighbor. Similarly to NN, to classify a test point using kNN, one would use the Euclidian distance to identify

the k nearest training sample to the test point. The test point then is assigned the label in which is most frequent among the k nearest training samples. One can interpret this as growing a spherical region, centered at the test point, until it encloses k training samples. Then assigning it the label of majority[2]. The choice of k is vital when designing a kNN based classifier. The goal is for k to be as large as possible, but at the same time ensure that all of the k nearest neighbors to an arbitrary test point $\boldsymbol{x}$ are as close to $\boldsymbol{x}$ as possible. This results in the choice of k being a compromise between a large value and a value that is small enough so that the k nearest neighbors are all close enough to the test point.

To simplify computation and accelerate convergence, one can introduce the technique K-means clustering. This is a method dividing the training set into K clusters, where the goal is to minimize the sum of distances between the means of the samples within a cluster and the mean of the cluster itself. The algorithm of K-means clustering starts by choosing K random points as cluster centroids, then assigning the remaining points in the set to the closest cluster centroid. Furthermore, one recomputes the centroids, or means, of the clusters, and repeat the process[3]. Instead of the original training samples, one can now use the clusters as training samples in a classification method. Since the number of clusters should be chosen as a lot fewer than the number of training samples, the computational effort of the classification process should be vastly reduced.

3 Task

3.1 Iris task

In this task is based on the Iris dataset, which is a dataset containing information on the iris flower. There are three different iris flower types; Setosa, Versicolor and Virginica. The goal of the task was to build a linear classifier to classify between these flowers. To start the task, the first 30 samples for each class where used to train the classifier and the last 20 samples for each class was used as a test set. Afterwards, the last 30 samples for each class where used to train the classifier and the first 20 samples for each class was used as a test set. For both cases error rates and confusion matrices where calculated and compared. The features play a large role on the results of the classifier, therefore, in the next part of the task, the effect of the different features on the classifier was analyzed. The goal was then to check if removing features with a large overlap between the classes would result in a better error rate. Confusion matrices, error rates were again calculated and compared for each case. All results and discussion can be found in the next section.

3.2 Handwritten numbers task

This task revolves around designing both an NN-based and kNN-based classifier in order to predict the labels of the MNIST database test set using the training set. Furthermore, we used K-means clustering to reduce the set of templates and computational effort. In the first part, we designed a NN-based classifier using Euclidian distance in order to classify the test pixtures in MNIST. In the second part, we divided the training set into clusters using K-means clustering. Using the clusters, we once again used the NN classifier to classify the test set. Lastly, we designed a kNN classifier in order to classify the test set using the clusters. In all three cases, the confusion matrices and error rates were calculated, and the results will be discussed and compared in the next section.

4 Implementation and results

In this chapter, explanations of the python and matlab implementations are provided as well as the results of both tasks. Furthermore, the results will be compared and discussed in this chapter.

4.1 The Iris task implementation and results

In the Iris task a linear classifier, supporting classification of 3 different classes, was implemented in python using an MSE cost function. The weight matrix and bias vector was always initialized to zero when training the classifier through the task. During training a set value of 100000 iterations of gradient descent was always performed throughout the task.

To start the task, the learning rate was tuned on the MSE of the training set, until convergence was achieved. Figure 2, made with matplotlib.pyplot in python, shows the results of this tuning for different values of the learning rate.

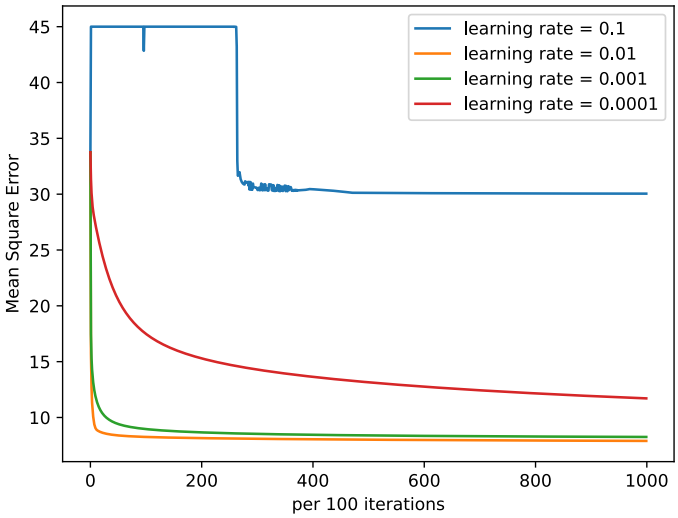
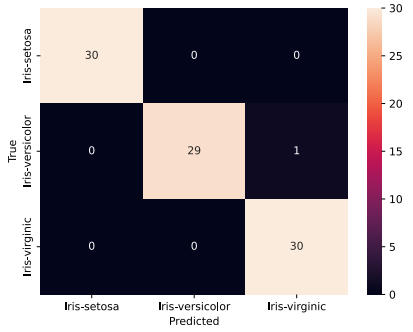


Figure 2: MSE on training set for different learning rate values per 100 iterations of gradient descent

As seen a learning rate of 0.01 resulted in the fastest convergence and also converged to the lowest MSE. This value was therefore used in the rest of the task.

Note that performing hyperparamater tuning on the training set like this is not ideal, as the classifier will be more likely to overfit to the training set from this. An alternative would be to tune on the test set, so that we tune based on the classifiers ability to generalize to unseen data. However, this leads to data leakage from the test set into the tuning phase. The best alternative would be to separate out a section of the training data as validation data, which will no longer be used for training, and tune on this data. This gives us the ability to tune based on the classifiers ability to generalize to unseen data, without causing data leakage from the test set. However, this is outside the scope of the task and is therefore not performed.

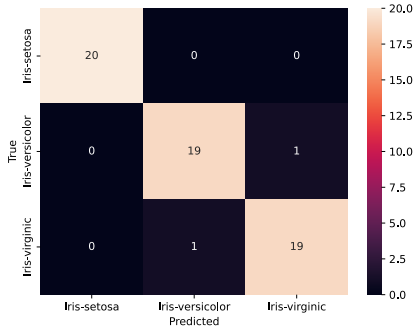
The resulting weight matrix of the classifier, with a learning rate of 0.01, and confusion matrix on both the train set and test set is shown below.



(a) Confusion matrix on train set with learning rate = 0.01 and iterations = 100 000

$$\boldsymbol{W} = \begin{bmatrix} 0.63569113 & 1.35593226 & -6.37697485 \\ 2.7015658 & -3.56154161 & -8.50572542 \\ -3.95071572 & -0.32461461 & 11.31619168 \\ -1.87660921 & -0.91500157 & 15.07095818 \end{bmatrix}$$

(b) Classifier weight matrix on train set



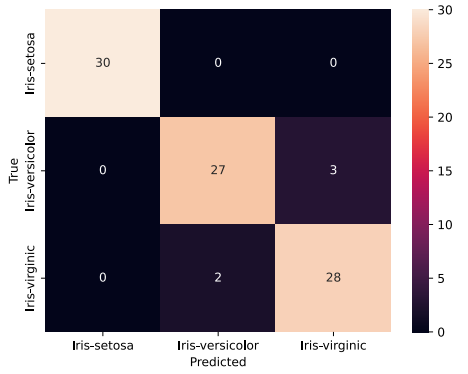
(c) Confusion matrix on test set with learning rate = 0.01 and iterations = 100 000

$$\boldsymbol{W} = \begin{bmatrix} 0.93268638 & -1.16283748 & -3.1667181 \\ 2.44789794 & -2.361613 & -5.18158121 \\ -4.39617185 & 7.96112976 & 6.85248145 \\ -2.067464 & -15.71373247 & 10.05874004 \end{bmatrix}$$

(d) Classifier weight matrix on test set

the corresponding error rate is $ER_{\text{train}} = \frac{1}{90} \times 100\% = 1.1\%$ and $ER_{\text{test}} = \frac{2}{60} \times 100\% = 3.3\%$.

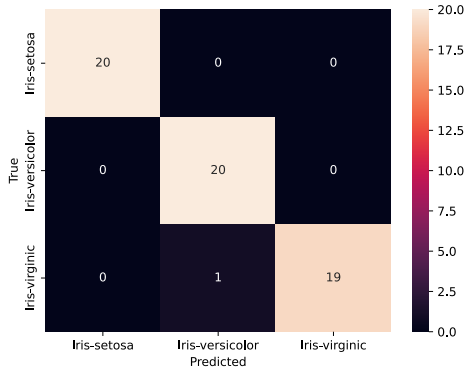
After this, the last 30 samples for each class where used to train the classifier and the first 20 samples for each class was used as a test set. The resulting weight matrix and confusion matrix for both the train and test data the model can be seen below.



(a) confusion matrix on train set with learning rate = 0.01 and iterations = 100 000

$$\boldsymbol{W} = \begin{bmatrix} 0.93268638 & -1.16283748 & -3.1667181 \\ 2.44789794 & -2.361613 & -5.18158121 \\ -4.39617185 & 7.96112976 & 6.85248145 \\ -2.067464 & -15.71373247 & 10.05874004 \end{bmatrix}$$

(b) Classifier weight matrix on train set



(c) confusion matrix on test set with learning rate = 0.01 and iterations = 100 000

$$\boldsymbol{W} = \begin{bmatrix} 0.85388036 & -1.28840976 & -2.78394998 \\ 2.6859322 & -3.14694731 & -4.88045766 \\ -4.39590973 & 2.48458169 & 6.27227436 \\ -2.1816855 & -4.37905939 & 9.83150251 \end{bmatrix}$$

(d) Classifier weight matrix on test set

The error rates are now $ER_{\text{train}} = 5.5\%$ and $ER_{\text{test}} = 1.7\%$. As seen, the error rate on both the train and test set is relatively equal regardless of the changes in the train and test data. This implies that the distributions of the features in these two cases is relatively equal. Its also worth noting that in the first case the error on the training set is lower then the test set, implying that the classifier does a slight overfitting to the training data in this case. In the second case the classifier has a lower error rate on the test set then the training set. This indicates that the classifier does a better job of avoiding to overfit to this specific training data and therefore generalizes better to unseen data.

The next part of the task was to analyze the effect of featuers on the classifier. To check overlap between features, histograms of each feature for each class was made. A kernal denisity estimation plot was also included, in order to get an idea of the distributions of the features. These plots where made with the seaborn library in python. In this part of the task, the first 30 samples of each class were always used for training and the remaining for testing.

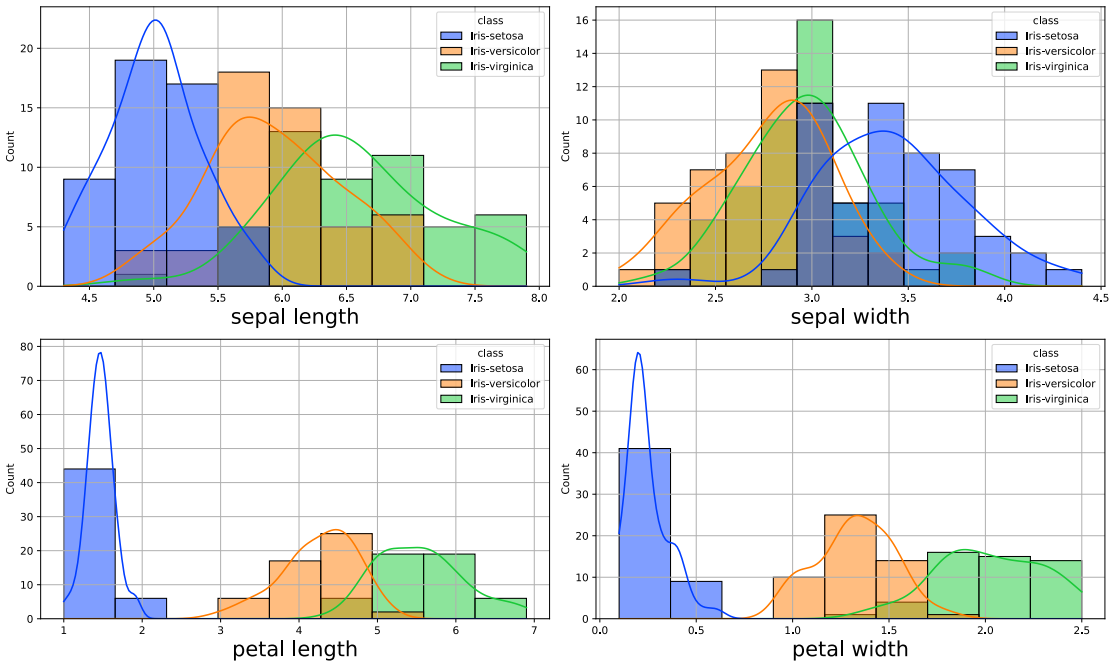
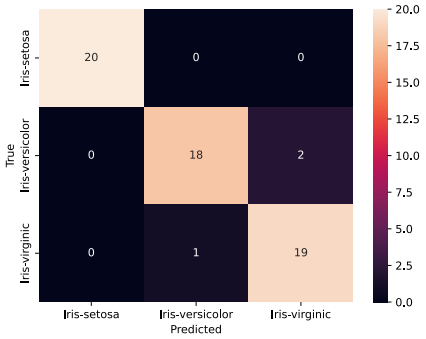


Figure 5: Histogram and KDE plot of each feature, hued by the classes.

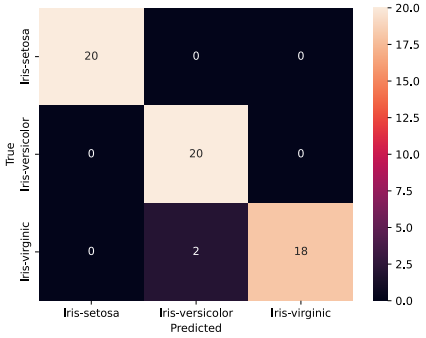
As seen in figure 5, *sepal width* is the feature with the most overlap. It is therefore removed and the model is retrained. Afterwards, both the *sepal width* and *sepal length* features were removed. Finally, all features except for *petal width* were removed. For all these instances, the model was retrained with the new feature combination. All confusion matrices, error rates and weights are shown in the figures below. Note, that the ratio between the weights associated with each feature give a good indication of the features importance. The higher the weight, the more effect the feature has on the model (as all feature are in the same scale [cm]).



(a) Confusion matrix for classifier with features ['sepal length', 'petal length', 'petal width']. $ER = 5\%$

$$\boldsymbol{W} = \begin{bmatrix} 0.19164457 & -0.18036816 & -0.0126138 \\ -0.34676975 & 0.40310664 & -0.05555327 \\ 0.06023067 & -0.67218157 & 0.61124765 \end{bmatrix}$$

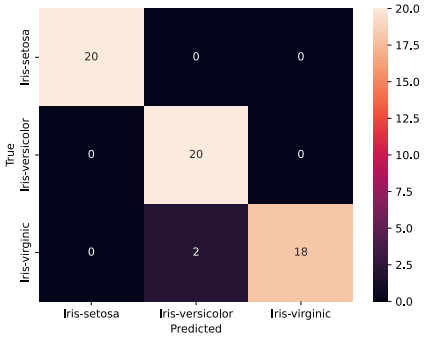
(b) Classifier weight matrix. Each row ordered ['sepal length', 'petal length', 'petal width']



(c) Confusion matrix for classifier with features ['petal length', 'petal width']. $ER = 3.3\%$

$$\boldsymbol{W} = \begin{bmatrix} -2.93009917 & 1.22701757 & 2.18869438 \\ -4.05182026 & -2.47820602 & 8.43369056 \end{bmatrix}$$

(d) Classifier weight matrix. each row ordered ['petal length', 'petal width']



(e) Confusion matrix for classifier with features ['petal width']. $ER = 3.3\%$

$$\boldsymbol{W} = \begin{bmatrix} -11.8947475 & 0.28291084 & 12.29386976 \end{bmatrix}$$

(f) Classifier weight matrix. row ordered ['petal width']

Figure 6: Confusion matrix, error rate and classifier weights on the test set for different feature combinations. Note that each row corresponds to a weight for a specific feature

The most important thing to note here is that the classifier always correctly classifies all data points with the class *Iris-Setosa*. This is because the *Iris-Setosa* class is linearly separable from the other classes, for all feature combinations. As we have a linear classifier, it will always correctly classify linearly separable classes.

It's also worth noting that the classifier performs equally well even with only the

petal width feature. As seen from the weight matrices, *petal width* is the most important feature for all the models. As seen from figure 5 *petal width* almost linearly separates the *Verticosa* and *Virginica* class. Therefore, It's reasonable that the linear classifier performs very well with only this feature, given that a linear classifiers performance is based on its ability to linearly separate the classes. Based on this, we can conclude that in a feature space consisting of just *petal width* it is possible to linearly separate the classes just as well as in a feature space consisting of all four features. This could imply that both costs and resources can be saved by not including the other features when creating the dataset.

However, keep in mind that this is the case for a linear classifier, using a more advanced classifier, such as a neural network or a boosted decision tree, a lower error rate could possibly be achieved with all features versus just *petal width* as the performance of these classifiers are not purely dependent on the classes being linearly separable.

4.2 The handwritten numbers task implementation and results

Initially, the MNIST dataset had to be imported into matlab. The handout file "read09.m" was run in order to create and format the data file, which in turn was used to extract the sets and labels. To implement the NN-based classifier, the set of 10000 test samples was iterated by groups of 1000 samples each, and each sample's distance to each of the 60000 training examples was computed using the matlab function `dist`. Then the index of the training sample with the smallest distance to the current test sample was extracted in order to get the label of said training sample.

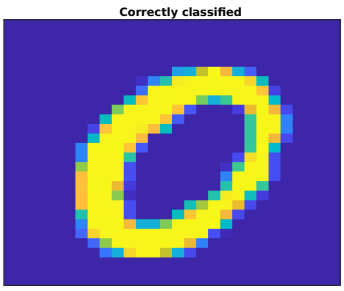
For the second part of the task, in order to cluster the training set, the matlab function `kmeans` was used. In order to make clusters of each number individually, the training labels were appended to the training samples in `X_train_tot` and was furthermore sorted by number. This made it easier to extract only samples of a certain number, or class, when clustering. Each number in the training set was divided into $M = 64$ clusters.

When using the NN classifier to predict the test labels using the clustered training set, the exact same method as explained above was utilized, but this time, the distance from test sample to each cluster was calculated.

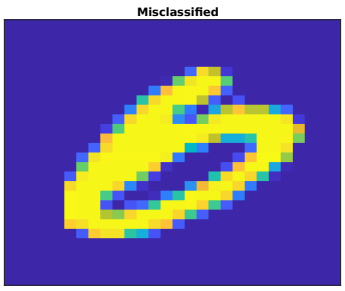
To implement the kNN classifier, a very similar method to the NN classifier was utilized. Now however, the index of the smallest distance in the distance vector `d` was extracted k times, while each time setting said distance to infinity in order to skip it next extraction. the labels of the k nearest training clusters were stored in `k_nearest`, and the matlab function `mode` was used in order to find the most frequent label. Lastly, the most frequent lable was assigned to the current test sample. In the task, $k = 7$ was used.

The confusion matrix of the NN classifier without clustering is shown in fig. 10a. Furthermore, the classifier had an error-rate of $ER = 3.09\%$. The computation process of this classification took relatively long at about 11 minutes and 20 seconds, which is why we introduce clustering later in the task. Although the error rate is low, it could be further reduced by using kNN with $k > 1$. Looking at the confusion matrix, one can argue that samples of the numbers 0, 1 and 6 are more isolated than the others considering the higher accuracy. Furthermore, 8 as some overlap with other numbers, especially 3 and 5, and therefore has a lower accuracy.

fig. 7, fig. 8 and fig. 9 show some misclassified and correctly classified pixtures of certain numbers side by side. The correctly classified cases all are quite clare indications of the number identified, although to varying degrees. The misclassified pixtures in fig. 7b and fig. 8b are clearly warped. As a human, the 0 in fig. 7b looks very close to a 6, which

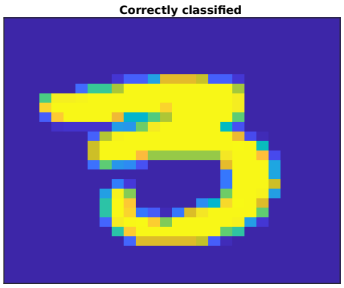


(a) 0 correctly classified

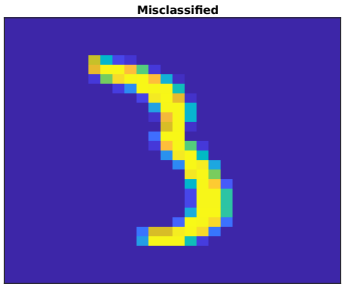


(b) 0 misclassified as 6

Figure 7: Example of correct and incorrect classifications of the number 0 using NN classifier without clustering



(a) 3 correctly classified



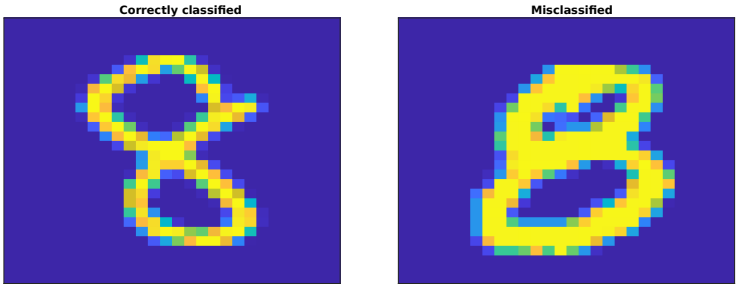
(b) 3 misclassified as 1

Figure 8: Example of correct and incorrect classifications of the number 3 using NN classifier without clustering

is what the classifier predicted as well. The 3 in fig. 8b is more difficult to identify. As humans, we would probably identify it as a 3, but could easily mistake it for a 1 or 7. The classifier predicted it as a 3, which we agree with to some degree. Lastly, the 8 in fig. 9b looks very much like an 8 through human eyes. The thicker lines however, makes the drawing less clear and results in the pixture losing a lot of blue color values. This could result in a lot of unintended values matching the values of other numbers, which is probably why the classifier misclassified it, in this case as a 3. Although, as humans we obviously do not agree with this prediction.

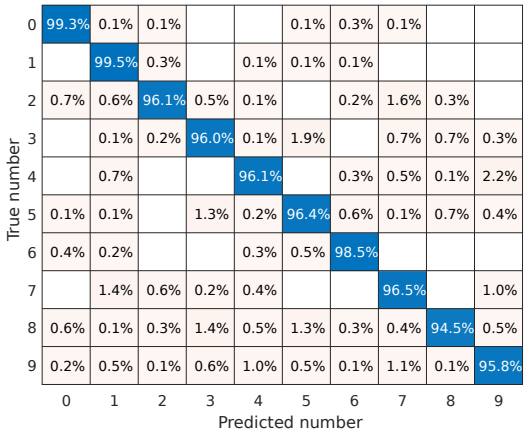
After clustering the training set, the confusion matrix using the NN classifier is shown in fig. 10b, with an error rate of $ER = 6.7\%$. The increased error rate makes sence considering the clustering vastly reduced the number of training samples. An immediate observation is that the accuracy in correctly classifying the number 1 dramatically dropped from 99.5% to 84.5%. Most of the accuracy drop was a result of misclassifying 1 as 2. An explanation for this may be that many samples of 1 and 2 may be close, but not overlapping. When clustering, the samples within a class are reduced and spread apart. Since 1 is missclassified as 2, but not the other way around, this is probably due to the clustering scattering the 1 samples than the 2 samples.

With the clustering taking around 30 seconds, and classification taking around 8 seconds, the processing time was drastically reduced from around 11 minutes to 38 seconds when introducing clustering. However, as mentioned, this impacted the accuracy. Since

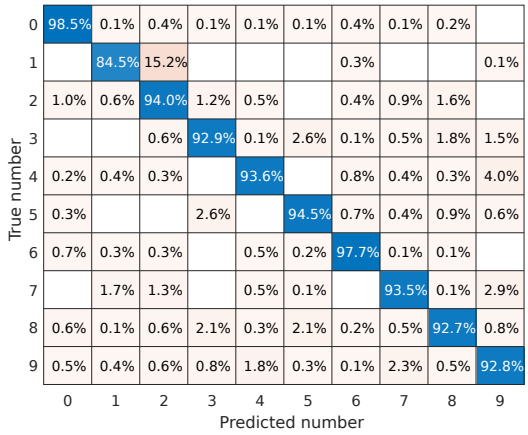


(a) 8 correctly identified (b) 8 misclassified as 3

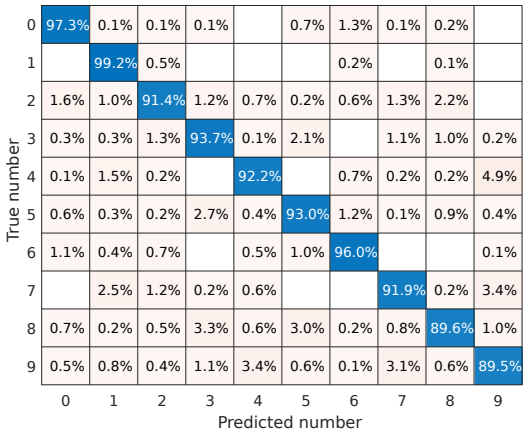
Figure 9: Example of correct and incorrect classifications of the number 8 using NN classifier without clustering



(a) Confusionmatrix of NN classifier without clustering



(b) Confusionmatrix of NN classifier with clustering



(c) Confusionmatrix of kNN classifier with clustering, $k = 7$

Figure 10: Confusion matrices of NN and kNN based classifiers with and without clustering

the NN partitioning is very specific, the accuracy may suffer due to overfitting. To try to increase the accuracy, we will now introduce kNN classification.

Using the kNN-based classifier, figure fig. 10c shows the confusion matrix while the classifier obtained an error rate of $ER = 6.55\%$, which is a slight improvement compared to NN. Comparing the confusion matrices, it is clear that increasing k from 1 to 7, solved the problem with identifying 1 as 2. Worth to notice, is the accuracy drop of some of the other numbers, most notably the larger ones. This may be a consequence of the distance needed to find k neighbors is too large, leading to a more generalized partition, which in turn could lead to under fitting. Matlab `mode` tiebreakers favoring the smallest number may also contribute to this.

5 Conclusion

In the Iris task, using a linear classifier with a sigmoid activation function resulted in very promising error rates. An error rate on the test set of 3.3% was achieved, using the first 30 samples per class for training and last 20 samples per class for testing. Similarly, an error rate of 1.7% on the test set was achieved using the last 30 samples per class for training and the first 20 samples per class for testing. This indicated that the distribution of features was relatively equal in both cases.

Afterwards, using the first 30 samples per class for training and last 20 samples per class for testing, an error rate of 3.3% was achieved on the test set using only the petal width feature. This was reasonable as a linear classifier's performance depends on the classes being linearly separable, and the petal width feature almost manages to linearly separate the classes in the dataset. This implied that in a feature space consisting of just *petal width* it is possible to linearly separate the classes just as well as in a feature space consisting of all four features.

In all cases, the classifier always correctly classifies the "Iris-setosa" class. This is due to this class being linearly separable from the other classes in all cases.

In the Handwritten numbers task, using a NN based classifier on the whole training set yielded promising results with an error rate of 3.09%. However, due to this decision rule being very specific, the accuracy may had suffered as a result of over fitting. Therefore, using the kNN based classifier with $k > 1$, the accuracy could have been improved. The computation process was expensive and slow, due to the large amount of samples in the database.

To decrease the processing time, the amount of templates were reduced using k means clustering. This understandably increased the error rate to 6.7% compared to using each sample as templates. However, the processing time was significantly reduced. Once again, the accuracy suffered due to overfitting using the NN based classifier. In order to solve this problem, a kNN classifier using $k = 7$ was implemented on the clustered training set. This addition slightly improved the error rate down to 6.55%. When classifying some numbers, the error rate was significantly reduced, while for some other numbers, the error rate either just slightly improved or even slightly worsened.

The tasks given in the project did in fact lead to a much deeper understanding of the respective classifiers, and improved our implementation skills. Specifically, we learned both theory and implementation methods for designing both linear discriminant based classifiers, and template based classifiers using the k nearest neighbours rule in addition

to choosing templates using k means clustering. Considering this, the quality of the tasks was undoubtedly high. However, parts of certain tasks, especially in the Iris task, were to some extent unclear. This lead to our group having to make some assumptions in order to solve them.

References

[1] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R.* Springer, 2013.

[2] Stork, D. G. Duda, R. O., Hart, P. E. *Pattern Classification.* Springer, second edition, 2000.

[3] P. Sharma. The ultimate guide to k-means clustering: Definition, methods and applications. `https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/`. Accessed: 18.04.2024.

6 Appendix

6.1 Iris code

All code related to the Iris tasks implemented in Python.

implements a class "LinearClassifier" that implements a linear classification model as described in 2.2. It implements two different discriminant functions that can be used to fit and predict on 2 separate linear classification models. The first is a linear discriminant function and the second is a linear discriminant function with a sigmoid activation function function.

```
import numpy as np
import pandas as pd

class LinearClassifier:

    def __init__(self, learning_rate=0.008, iterations=10000):
        """
        sets the hyperparamaters for the model
        Args:
            learning_rate (float): specifies the learning rate used
            ↪  in
              gradient descent to fit the model.
            iterations (int): specifies the amount of iterations used
            ↪  in
              gradient descent to fit the model.

        """
        self.learning_rate = learning_rate
        self.iterations = iterations

    def fit_sigmoid(self, X, y):
        """
        Estimates weights and biases for the classifier using a
        ↪  sigmoid activation function
        Args:
            X (array<m,n>): a matrix of floats with
                m rows (#samples) and n columns (#features)
            y (matrix<m>): a vector of ints containing
                m label values
        """
        #initialize paramaters
        n_samples, n_features = X.shape
        n_classes = len(np.unique(y))

        self.W = np.zeros((n_features, n_classes))
        self.B = np.zeros(n_classes)

        #convert lables to a matrix<m, n_classes> with 1 column for
        ↪  each class
```

```
36          y_dummy = pd.get_dummies(y, prefix='class')
37          y_dummy = y_dummy.values
38
39          alpha_vals = []
40
41          #perform gradient descent
42          for i in range(self.iterations):
43              Z_test = np.dot(X, self.W)
44              B_test = self.B
45              Z = np.dot(X, self.W) + self.B
46              MSE_sigmoid = (1/2) * np.sum((sigmoid(Z) - y_dummy) ** 2)
47
48              dW = np.dot(X.T, ((sigmoid(Z) * (1 - sigmoid(Z))) *
                ↪  (sigmoid(Z) - y_dummy)))
49              dB = np.sum((sigmoid(Z) * (1 - sigmoid(Z))) * (sigmoid(Z)
                ↪  - y_dummy), axis=0)
50
51              self.W = self.W - self.learning_rate * dW
52              self.B = self.B - self.learning_rate * dB
53
54              if (i%100 == 0):
55                  print(f"cost after {i} iterations of gradient descent
                    ↪  is: {MSE_sigmoid}")
56                  alpha_vals.append(MSE_sigmoid)
57          return alpha_vals
58
59      def fit_linear(self, X, y):
60          """
61          Estimates weights and biases for a linear classifier without a
            ↪  sigmoid activation function
62          Args:
63              X (array<m,n>): a matrix of floats with
64                  m rows (#samples) and n columns (#features)
65              y (matrix<m>): a vector of ints containing
66                  m label values
67          """
68
69          #initialize paramaters
70          n_samples, n_features = X.shape
71          n_classes = len(np.unique(y))
72          self.W = np.random.randn(n_features, n_classes)
73          self.B = np.random.randn(n_classes)
74
75          #convert lables to a matrix<m, n_classes> with 1 column for
            ↪  each class
76          y_dummy = pd.get_dummies(y, prefix='class')
77          y_dummy = y_dummy.values
78
79          #perform gradient descent
```

```
        for i in range(self.iterations):
            Z = np.dot(X, self.W) + self.B
            MSE = (1/(n_samples)) * np.sum((Z - y_dummy) ** 2 )

            dW = (2 / (n_samples)) * np.dot(X.T, (Z - y_dummy))
            dB = (2 / (n_samples)) * np.sum((Z - y_dummy), axis=0)

            self.W = self.W - self.learning_rate * dW
            self.B = self.B - self.learning_rate * dB

            if (i%100 == 0):
                print(f"cost after {i} iterations of gradient descent
                ↪  is: {MSE}")

    def predict_class_probability(self, X):
        """
        Generates probability predictions if using the model trained
        ↪  with a sigmoid activation function

        Note: should be called after .fit_sigmoid()

        Args:
            X (array<m,n>): a matrix of floats with
                m rows (#samples) and n columns (#features)

        Returns:
            A length m array of floats in the range [0, 1]
            with probability-like predictions
        """
        z = np.dot(X, self.W) + self.B
        return sigmoid(z)

    def predict_class_sigmoid(self, X):
        """
        Generates predictions

        Note: should be called after .fit_sigmoid()

        Args:
            X (array<m,n>): a matrix of floats with
                m rows (#samples) and n columns (#features)

        Returns:
            A length m array of floats in the range [0, 1]
            with class predictions
        """
        z = np.dot(X, self.W) + self.B
        # Predict the class based on the maximum value along each row
```

```
        predictions = np.argmax(sigmoid(z), axis=1) + 1  # Adding 1 to
        ↪  make the prediction 1, 2, 3 npot 0,1,2
        return predictions

    def predict_class_linear(self, X):
        """
        Generates predictions

        Note: should be called after .fit_linear()

        Args:
            X (array<m,n>): a matrix of floats with
                m rows (#samples) and n columns (#features)

        Returns:
            A length m array of floats in the range [0, 1]
            with class predictions
        """

        Z = np.dot(X, self.W) + self.B

        # Predict the class based on the maximum value along each row
        predictions = np.argmax(Z, axis=1) + 1  # Adding 1 to make the
        ↪  prediction 1, 2, 3 not 0,1,2
        return predictions

def sigmoid(x):
    """
    Applies the logistic function element-wise

    Args:
        x (float or array): input to the logistic function
            the function is vectorized, so it is acceptible
            to pass an array of any shape.

    Returns:
        Element-wise sigmoid activations of the input
    """

    return 1. / (1. + np.exp(-x))

```

Below is all code needed to load inn and preprocess the Iris dataset, so that it can be used for our classification task.

```

iris_df = pd.read_csv('iris.data', delimiter=',', header=None)
iris_df.columns = ['sepal length', 'sepal width', 'petal length',
↪  'petal width', 'class']
```

```
4
5   #encode the labels to be able to use in classification model
6   mapping = {
7       'Iris-setosa' : 1,
8       'Iris-versicolor' : 2,
9       'Iris-virginica' : 3
10      }
11  iris_df_categorical = iris_df.copy()
12  iris_df['class'] = iris_df['class'].map(mapping)
13
14  def train_test_split_a(iris_df):
15      """
16      function for splitting data into a train / test set.
17      splits first 30 samples for each class into train
18      and last 20 samples for each class into test
19
20       Args:
21              iris_df (pandas:dataFrame<m,n>):
22                  pandas dataframe with our dataset
23      Returns:
24              dataset split into train and test datasets
25      """
26      iris_train_df = pd.concat([iris_df[0:30], iris_df[50:80],
        ↪  iris_df[100:130]])
27      iris_test_df = pd.concat([iris_df[30:50], iris_df[80:100],
        ↪  iris_df[130:150]])
28
29      iris_train_df.reset_index(drop=True, inplace=True)
30      iris_test_df.reset_index(drop=True, inplace=True)
31
32      X_train = iris_train_df.drop("class", axis=1)
33      X_test = iris_test_df.drop("class", axis=1)
34      y_train = iris_train_a_df["class"]
35      y_test = iris_test_a_df["class"]
36
37      return X_train, y_train, X_test, y_test
38
39  def train_test_split_d(iris_df):
40      """
41      function for splitting data into a train / test set.
42      splits last 30 samples for each class into train
43      and first 20 samples for each class into test
44
45       Args:
46              iris_df (pandas:dataFrame<m,n>):
47                  pandas dataframe with our dataset
48      Returns:
49              dataset split into train and test datasets
50      """
```

```
    iris_train_d_df = pd.concat([iris_df[30:60], iris_df[70:100], iris_df[120:150]])
    iris_test_d_df = pd.concat([iris_df[0:20], iris_df[50:70], iris_df[100:120]])

    iris_train_d_df.reset_index(drop=True, inplace=True)
    iris_test_d_df.reset_index(drop=True, inplace=True)

    X_train = iris_train_d_df.drop("class", axis=1)
    X_test = iris_test_d_df.drop("class", axis=1)

    y_train = iris_train_d_df["class"]
    y_test = iris_test_d_df["class"]

    return X_train, y_train, X_test, y_test

#splitting data into train / test data
X_train_a, y_train_a, X_test_a, y_test_a = train_test_split_a(iris_df)
X_train_d, y_train_d, X_test_d, y_test_d = train_test_split_d(iris_df)
```

Below is code used to train and perform classification on prepared train and test data

```
model_a = lr.LogisticRegression(learning_rate=0.01, iterations=100000, seed=10)
model_a.fit_linear(X_train_a, y_train_a)
pred_a = model_a.predict_class_linear(X_test_a)
```

Below is code for calculating and plotting confusion matrixes and error rates of a classifier based on test data.

```


# Calculate confusion matrix
confusion_matrix_df = pd.DataFrame(data=confusion_matrix(y_test, y_pred),
                                   index=["True Iris-setosa", "True Iris-versicolor", "True Iris-virginic"],
                                   columns=["Predicted Iris-setosa", "Predicted Iris-versicolor", "Predicted Iris-virginic"]
                                   )

# Plot confusion matrix
fig = plt.figure()
sns.heatmap(confusion_matrix_df, annot=True)
plt.show()

```

```
# Calculate accuracy and error rate
accuracy = accuracy_score(y_test, y_pred)
error_rate =  1 - accuracy
```

```
#creating histograms with Kde plots for Iris dataset
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(20, 20), dpi=200)
fig.suptitle('Histogram with KDE plots for each feature', fontsize=40,
↪  y=1)
axes = axes.flatten()
for i, column in enumerate(iris_df_categorical.columns):
    if column == "class":
        continue
    sns.histplot(data=iris_df_categorical, x = column, kde=True,
    ↪  ax=axes[i], hue ="class")
    axes[i].set_xlabel(column, fontsize = 30)
    axes[i].grid(True)
plt.tight_layout()
plt.savefig('IRIS_Histogram.svg', format='svg')
plt.show()
```

below is the code for plotting the MSE for different learning rates

```
fig = plt.figure(dpi=200)

# Plot the data, alpha_... refers to the MSE output per 100 iterations
↪  for the linear classifiers with different learning rates
x = [i for i in range(1000)]
plt.plot(x, alpha_01, label="learning rate = 0.1")
plt.plot(x, alpha_001, label="learning rate = 0.01")
plt.plot(x, alpha_0001, label="learning rate = 0.001")
plt.plot(x, alpha_00001, label="learning rate = 0.0001")

# Add legend
plt.legend()
plt.ylabel("Mean Square Error")
plt.xlabel("per 100 iterations")


plt.savefig('alpha tuning.svg', format='svg')
# Show the plot
plt.show()
```

6.2 Handwritten numbers code

This task was in its entirety implemented in Matlab. All code written in the project is provided below. The parameter `use_clusters` may be set according to if the training set

should be divided into clusters or not. Furthermore, the parameter `k` may be set according to the intended number of closest neighbors to be considered in the kNN classifier. If one intends to use a NN classifier, set `k = 1`. The following code is used for both clustering and classifying.

```
data_all = load('data_all.mat');
X_test = data_all.testv;
y_test = data_all.testlab;
X_train = data_all.trainv;
y_train = data_all.trainlab;

use_clusters = true;
if use_clusters

    X_train_tot = sortrows([X_train y_train], 28*28+1);
    M = 64; %Number of clusters per class

    %Clustering of each class
    [ind1, C1] = kmeans(X_train_tot(1:5923, 1:28*28), M);
    [ind2, C2] = kmeans(X_train_tot(5924:12665, 1:28*28), M);
    [ind3, C3] = kmeans(X_train_tot(5925:18623, 1:28*28), M);
    [ind4, C4] = kmeans(X_train_tot(18624:24754, 1:28*28), M)
       ;
    [ind5, C5] = kmeans(X_train_tot(24755:30596, 1:28*28), M)
       ;
    [ind6, C6] = kmeans(X_train_tot(30597:36017, 1:28*28), M)
       ;
    [ind7, C7] = kmeans(X_train_tot(36018:41935, 1:28*28), M)
       ;
    [ind8, C8] = kmeans(X_train_tot(41936:48200, 1:28*28), M)
       ;
    [ind9, C9] = kmeans(X_train_tot(48201:54051, 1:28*28), M)
       ;
    [ind10, C10] = kmeans(X_train_tot(54052:60000, 1:28*28),
       M);

    C = [C1;C2;C3;C4;C5;C6;C7;C8;C9;C10]; %Cluster matrix

    %Class of each cluster
    y_clusters = [zeros(M, 1); ones(M, 1); 2*ones(M, 1); 3*
       ones(M, 1); 4*ones(M, 1); 5*ones(M, 1);
        6*ones(M, 1); 7*ones(M, 1); 8*ones(M, 1); 9*ones(M,
           1)];
end

%Classification
k = 1; %Number of nearest neighbors to compare
group_size = 1000; %Size of goups of test samples
y_pred = zeros(10000, 1);
if use_clusters
```

```
    for i = 0:length(X_test)/group_size-1
        d = dist(X_test(group_size*i+1:group_size*(i+1),:), C
            .');
        for j = 1:group_size
            k_nearest = zeros(k, 1); %Classes of nearest
                neighbors
            for n=1:k
                [minDist, indx] = min(d(j,:));
                d(j,indx) = inf; %Next neighbor must be
                    different
                k_nearest(n) = y_clusters(indx);
            end
            y_pred(group_size*i+j) = mode(k_nearest); %Take
                most frequent class of neighbors
        end
    end
else
    for i = 0:length(X_test)/group_size-1
        d = dist(X_test(group_size*i+1:group_size*(i+1),:),
            X_train.');
        for j = 1:group_size
            [minDist, indx] = min(d(j,:));
            y_pred(group_size*i+j) = y_train(indx);
        end
    end
end
```

The following code is used for evaluating the predictions made in the code above. Furthermore, it calculates the error rate and plots the confusion matrix. Furthermore, an example of both a random correctly classified number and a misclassified number is plotted. The number to plot can be chosen by setting the variable `wanted_number` correspondingly.

```
%Test predictions
evaluated_predictions = 1*ones(10000, 1); %1 if correct, 0 if
    incorrect
n_correct = 0;

for i=1:length(y_pred)
    if y_pred(i) ~= y_test(i)
        evaluated_predictions(i) = 0;
    else
        n_correct = n_correct + 1;
    end
end

error_rate = 1 - n_correct/length(y_test);

%Confusion matrix
figure()
```

```
CMat = confusionmat(y_test, y_pred);
confusionchart(CMat, categorical({'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}), 'Normalization', 'row-normalized');
xlabel("Predicted number")
ylabel("True number")

%Find index of random correct and incorrect
%classifications of the number 'wanted_number'
random_incorrect_indx = 0;
random_correct_indx = 0;
wanted_number = 5;
max_tries = 1000000;
for j = 1:max_tries
    i = randi([1 length(y_pred)]);
    if evaluated_predictions(i) == 0 && y_test(i) == wanted_number
        random_incorrect_indx = i;
        break;
    end
end

for j = 1:max_tries
    i = randi([1 10000]);
    if evaluated_predictions(i) == 1 && y_test(i) == wanted_number
        random_correct_indx = i;
        break;
    end
end

%Plot correctly and incorrectly
%classified images

figure()
m = zeros(28,28);
m(:) = X_test(random_incorrect_indx,:);
image(flip(imrotate(m, 90)));
title("Misclassified")
set(gca,'xtick',[])
set(gca,'ytick',[])

figure()
m = zeros(28,28);
m(:) = X_test(random_correct_indx,:);
image(flip(imrotate(m, 90)));
title("Correctly classified")
set(gca,'xtick',[])
set(gca,'ytick',[])
```