Designing classifiers for subspecies of the Iris flower and handwritten digits

Project in TTT4275 - Estimation, Detection, and Classification

S. L. Breivik[a], M. Vårdal[a]

[a] *Department of Engineering Cybernetics, Norwegian University of Science and Technology, 7491 Trondheim.*

Summary

This project centers around the implementation of different classifiers for different tasks. First, a linear classifier for distinguishing between the three subspecies of the Iris flower Iris-setosa, Iris-versicolor, and Iris-virginica is implemented. Subsequently, the nearest neighbor (NN) and a k nearest neighbors (kNN) classifier is implemented for the MNIST dataset, which is a large dataset of images of handwritten digits and is one of the most popular datasets within the field of deep learning[1]. The linear classifier achieved an error rate of at best 1.1 and 3.3 percent for the training set and testing set, respectively. This was obtained when all four features sepal width, sepal length, petal width, and petal length were included in the training set. The error rate for the NN classifier was 3.1 and for the kNN classifier with clustering, 6.5 percent. As one can see from the results, the linear classifier works quite well for these particular subspecies of the Iris flower. Moreover, the NN and kNN classifiers can achieve decent accuracy despite the fact that they are sub-optimal classifiers. Similarly, since the Iris classifier was nearly linearly separable with only two features, it might save expenses by dropping the other two. Overall, this project illuminates some of the choices one has to consider when designing a classifier, such as the trade-off between accuracy and computational complexity. These choices must be made on a case-by-case basis, taking into account factors such as the level of precision required and the available hardware.

Contents

1. Introduction

The goal of this project is to develop an understanding of different types of classifiers and how they are implemented. Part of the task is to create a classifier for the classification of images of handwritten digits. This is a tool that can be used to automate tasks that would otherwise be done by human beings, for example in check processing or in digit recognition for the postal service.

In Chapter 2, the theory needed to understand the tasks is discussed. In Chapter 3, we take a look at the task we chose. Chapter 4 focuses on how we implemented our solution, in addition to a discussion of the results. Finally, in Chapter 5, we write a conclusion for the report. The code is displayed in Appendix A.

2. Theory

This chapter is a presentation of the theory needed to understand the task. We will first look at the theory necessary for understanding the linear classifier. Then, we will take a look at the theory underlying the nearest-neighbor classifier. This will also include the concepts of clustering and majority vote for the k-nearest neighbors. Most of the equations and theory are collected from a compendium about classification[2], handed out in the subject TTT4275 Estimation, detection, and Classification at NTNU. The figures in the theory part are made using geogebra classic.

2.1. The linear classifier

A classifier is trying to answer the question of which class a given instance belongs to. This is practically done

by comparing discriminant functions[1]. If we partition (divide into parts) the whole input space according to which class is most probable, then we have a classifier. Said mathematically this is given by the decision rule below in equation(1),

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

x (the thing we want to classify)is in the class (ω_j) that has the highest value for the discriminant function (g_j). The linear discriminant classifier is defined by the function[2, p. 9]

$$g_i(x) = \omega_i^T x + \omega_{io}, \qquad i = i, \ldots, C. \tag{2}$$

Here, ω_{io} is the offset for the class i. In the 2d-case this classifier is given by straight lines. If the number of classes C is greater than 2, then this can be written in the more compact form

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

where g and w_0 are column vectors of dimension C. The matrix W then has C rows and D columns, where D is the number of features in x[2]. We can make it more compact by performing the following two redefinitions,

$$\begin{bmatrix} W & \omega_o \end{bmatrix} \to W$$
$$\begin{bmatrix} x^T & 1 \end{bmatrix} \to x$$

The equation offset is now embedded in the feature vector 'x'. And we end up with a simple-looking expression,

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

An example of a linear classifier is given in Figure 1

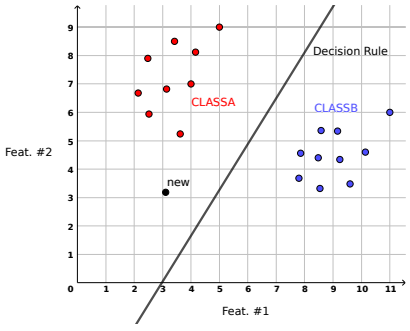


Figure 1: An example of a linear classifier, splitting the input space of two features, and deciding which classes new inputs belong to. For example, the black point, showing a new instance would be classified as Class A. A matrix W is specifying the decision rule. Feat. stands for feature and is a defining property of classes.

2.2. Nonlinear Problems

In most cases, the problem is not linearly separable. Then we have to make a more complex classifier. An example of this is shown in Figure 2

[1] functions which give the probability that an instance belongs to a given class.

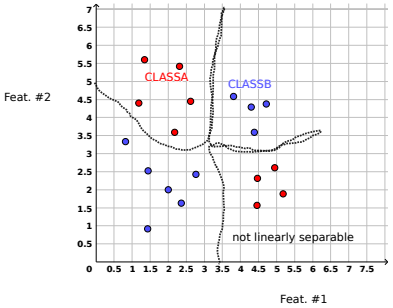


Figure 2: An example of a problem where the instances cannot be separated by a linear relation of the features.

2.3. Training of a linear classifier.

Before it is possible to talk about the training of a linear classifier, we first need to define how the performance of the classifier is to be measured. There are several ways of doing this, and one common way is to measure the performance in terms of the Minimum Square Error (MSE) of the classifier. In other words, we want to minimize the sum of the squares of the difference between a prediction from the classifier and the target value. Only now can we start determining what the classifier should look like. We define the loss function as follows,

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

The output vector $g_k = Wx_k$ corresponds to a single input x_k. We need to match this output to the target vector (which is the correct label) with the values 0 and 1. Since we need a smooth function, with a derivative, we use the sigmoid function. This is given in equation (6) and shown in Figure 3. This is more or less a curved step function.[2].

$$g_k = \frac{1}{1 + \exp^{-z_{ik}}}, \qquad z_{ik} = Wx_k. \tag{6}$$

In order for us to train our model, we need to know the steepest direction of decrease for the MSE. Therefore we calculate the gradient of the MSE with respect to the matrix W. This calculation uses the chain rule,

$$\nabla_W MSE = \sum_{k=1}^{N} \nabla_{g_k} MSE \, \nabla_{z_k} g_k \, \nabla_W z_k.$$

And if we use the following simplifications,

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

we get the following result,

$$\nabla_W MSE = \sum_{k=1}^{N}[(g_k - t_k) \circ g_k \circ (1 - g_k)]x_k^T. \tag{7}$$

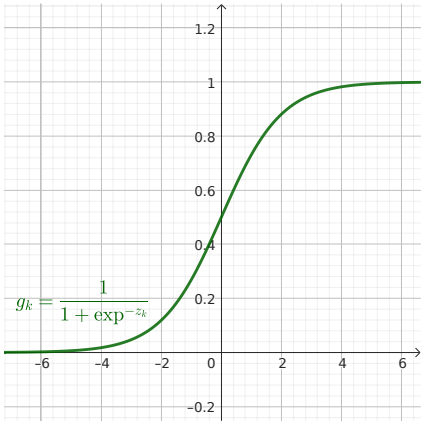


Figure 3: Plot of the sigmoid function, that converts a continuous value to a nearly binary value. The reason why we need a differentiable function is in order to ensure that the gradient is defined.

Where $\circ$ stands for elementwise multiplication. We can now move the matrix W an appropriate step length α in the direction opposite of the gradient following the steepest descent algorithm[2],

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

where m is the iteration number. If the step length is chosen wisely and we update W according to equation(8), the solutions will converge towards a local (and possibly global) minimum for the MSE.

2.4. The nearest neighbor classifier

The nearest neighbor - NN classifier is based on a very intuitive idea. The idea is that we measure the "distance" between a set of features with unknown labels to all the samples from the training set with known labels. Then, we simply classify the unknown sample as the class of its nearest neighbor (the one for which the distance is smallest).

The "distance" from one data point to the next can be defined in a number of different ways. One of those ways is to define it as the Euclidean space.

The Euclidean distance between two vectors in $\mathbb{R}^n$, $p_1 = (x_1, x_2, \ldots, x_n)^T$ and $p_2 = (y_1, y_2, \ldots, y_n)^T$ is defined as follows,

$$d(p_1, p_2) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Figure 4 explains the intuition behind the concept. We have primarily two objections to this approach, which we will discuss later.

2.5. Clustering

A problem with the approach above is that it requires a lot of computation. All distances must be calculated each time we make a prediction. One way to try to get

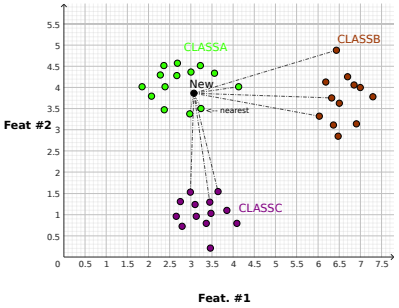


Figure 4: A new instance (the black point) is found. We then calculate the distance between all the points and the new point. Since CLASSA (green) has the nearest point, we then decide that the new class belongs to this class. Not all lines (dashed) showing the distances are displayed

around this is a process called "clustering". The concept is shown in Figure 5. All we have to do to produce clusters is to group together data points of the same class (can be several clusters per class) and find the center of each of them. That way, we can simplify the calculations by only calculating the distance to the cluster centers and not every single data point. This will reduce the accuracy to some extent, but this can be outweighed by the benefit of having a less computationally expensive system in many cases.

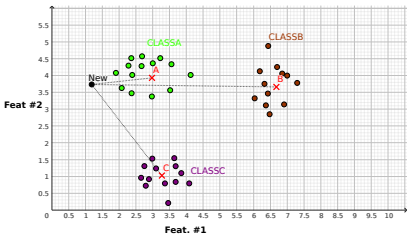


Figure 5: All of the classes are here reduced to their own cluster center. This reduces computation and to an extent accuracy. When deciding on the new point, we only need to take into account the cluster centers. So, since the new point in our case is closest to class A, it is labeled as such.

In order to produce the clusters, one uses algorithms. An example of this is given in [2, p. 18-19].

2.6. K-decision Nearest Neighbor

A flaw with our nearest neighbor classifier appears when we are unlucky with our new point being close to an outlier. This unfortunate event is shown in Figure 6. We instead want to make our system calculate the k-nearest points and then choose the class which has the most out of the k points.

2.7. Normalization of features

To avoid the impact of our decided unit of measurement, one can utilize algorithms to normalize the feature vector. So that if one feature has unit meters, whilst another one

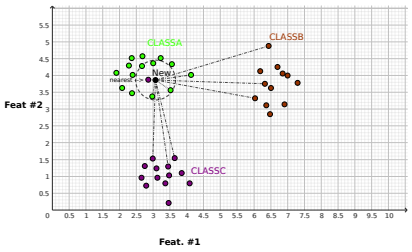


Figure 6: Because of an outlier near our new point, who clearly should be CLASSA is classified as CLASSC. If we instead have a voting system for the k nearest point, this problem disappears. This is shown by the circle choosing the k=5 nearest points. Since there are 4 Green points and only one purple, the Green will win the voting.

has the unit of centimeters, these differences will be taken into account.

2.8. Confusion matrix and error rate

A confusion matrix is a way of showing how a classifier has performed, by showing graphically how the outputs of the classifier compare to the target values. Ideally, everything should be on the diagonal, because that means that the predicted and true values are correct. Along the horizontal axis is the predicted value, whilst the true value is along the vertical axis. In order to rate how well a system behaves, one possible measure is the error rate, given in the following equation,

$$Err = \frac{Er}{N} \tag{9}$$

Where Err is the error rate, Er is the number of Errors and N is the number of samples. The measure is also often used with a subscript to clarify whether it is the testing or training set.

3. The task

In this chapter, we will present the task at hand in greater detail.

We chose the task where we were supposed to design classifiers for images of different Iris-subspecies and of different handwritten digits. The reason we chose this task, in particular, was because we both thought that classification was the most interesting topic. Furthermore, we thought that the one about the classification of images of handwritten digits seemed like the most exciting one at first glance.

To describe the task more specifically, the first main task was to design a linear classifier that could classify a set of features of a given Iris flower as one of three subspecies: Iris-setosa, Iris-versicolor or Iris-virginica. The features were sepal length, sepal width, petal length, and petal width. To do this, we were given a dataset consisting of 150 sets of the aforementioned features together with the corresponding subspecies - or target value. There were 50 sets of features and targets for each of the three subspecies.

First, we were going to train the classifier on the first 30 samples in the dataset for each subspecies, and test it on the remaining 20, and find the confusion matrix and error rate. After this, we were going to repeat what we had done, but with the last 30 samples as the training set and the first 20 as the testing set.

In the second and last part of the Iris task, we were going to produce histograms of all the features for the different subspecies. This was done so that we could try to eliminate the feature(s) with the most overlap between the subspecies in order to see if this increased the performance of the classifier. First, we eliminated one feature, then another, and after that, another one. Then we compared the results for the classifier based on four, three, two, and one feature(s), respectively.

The second main part of the task was to design some classifiers for classifying handwritten digits, based on the MNIST database. The MNIST database is a database containing 70'000 images of handwritten digits and the corresponding target values. The dataset is divided into a training set of 60'000 images and a test set of the remaining 10'000 images. The first classifier we were going to design for this dataset, was the simple nearest neighbor classifier. Then we investigated its performance via a confusion matrix and the error rate. Furthermore, we plotted a few of the misclassified digits and a few of the correctly classified digits. Furthermore, we implemented both clustering and rewrote the classifier into a k-nearest neighbors classifier. Additionally, we discussed how this changed the performance. The Python implementation can be found in the attached zip file.

4. Implementation and results

In this chapter, we will discuss the implementation of our solutions to the different tasks as well as the results. All of our solutions were written in Python and can be explored from the submitted files. It was written in notebook format, which gives us the opportunity to walk you through our solution, switching between markdown language and Python code.

4.1. Implementation of Iris

The way we implemented the linear classifier for the Iris task was by first organizing all the data in a class that we called "Dataset". This allowed us to structure the training and testing data in a tidy way. The training data was put into the Dataset.training dictionary where Dataset.training["features"] contained the features and Dataset.training["targets"] contained the targets. The same was done for the testing samples. After the initial data organizing was done, we used the function "TrainClassifier" to train our classifier. This function started with a matrix W that contained zeros and was updated 10'000 times with

a step length of 0.01 in accordance with equation (7) and equation (8). The reason behind these hyperparameters can be seen from Figure 7. The resulting W is printed to the Python console along with the first and last MSE.

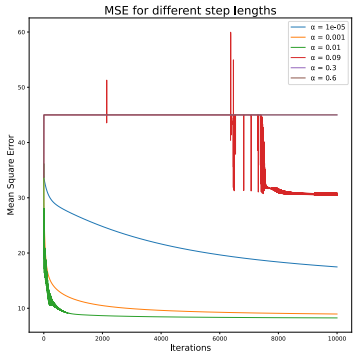


Figure 7: Experiment using the training set of the Iris-data. We observe that the step length should be 0.01. Because it was the value that worked the best. Along the y-axis is the Mean-squared-error, defined in equation (7). Along the x-axis is the corresponding iteration number. The Iris-training dataset is used. The reason why the red graph seems unstable is probably because it moves past the local minimum which the algorithm aims at, because of the step length being too big.

After this, what remained was to test the classifier we had trained. This was done by using our "TestClassifier" function. This function tested the classifier on the test samples, created confusion matrices, and calculated the error rate. Then we switched which samples we used for training and testing, in order to test if that made any difference. After this, we loaded all the data from the three classes and made histograms of them. Lastly, we trained and tested linear classifiers with decreasing number of features, by just loading the features we wanted. We used the numpy library for computation, and for the plotting, we used sklearn (confusion-matrices) and matplotlib.

4.2. Implementation of MNIST

In the MNIST task, we started by organizing the code in the same way as with the Iris data. That is, we downloaded the MNIST dataset through the Keras[2] library in Python and added it to an instance of our own Dataset class. The Dataset class in this task was defined quite similarly to the one from the Iris task. Then, we defined an NN_predict function that takes as input a set of images and returns a list of the predictions, one for each of the input images. The way it does this is to iterate through the image_predict input variable, which contains an array of all the images we want to predict. For each of the images, it calculates the Euclidean distance[3] from this image to all the others and classifies it as the class of that image for which the distance is the smallest. To avoid cumbersome calculations with very big distance matrices, we divide the training set into chunks of 1000 and do the calculations for each of the chunks respectively, and save the nearest neighbor in each chunk. After all the calculations are done for all the chunks, we simply find the absolute nearest neighbor by comparing all of the nearest neighbors from the different chunks.

To test this classifier, we made a confusion matrix and calculated the error rate, similar to what we did in the Iris task. This was done with the "conf_matrix_test" function which uses the "NN_predict" function to make a prediction for the images in the test set from the MNIST dataset. Additionally, we extracted the first four correctly classified images and the first four misclassified images and returned them. This was because we were going to plot them later.

Next, we wanted to implement clustering. The way we did this was by first sorting and normalizing the training set. Then we used an algorithm[4] for creating 64 clusters for each of the classes. Then we retrieved the center of each cluster and normalized it. At the end, we had a clusters list with all of the 640 (10 classes with 64 clusters) clusters. Then we could classify the test set by finding the cluster which minimized the Euclidean distance and finding the corresponding digit.

Lastly, we wanted to implement a k-nearest neighbor classifier. The only difference from the one before to this one was the decision rule. Instead of just finding the best cluster, we sorted the k=7 best clusters and had a majority vote. Therefore the test points belonged to the class with the most clusters among the best 7. In terms of libraries, we once again used numpy for computation, sklearn for metrics, and matplotlib for plots. In addition, we used keras in order to load the dataset, sklearn.cluster for the KMeans algorithm, and scipy for calculating Euclidean distance.

4.3. Results Iris

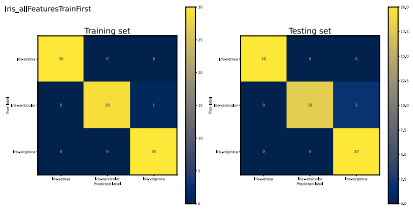
For the hyperparameters, we landed on a step length of $\alpha = 0.01$ and $n = 10000$ as the number of iterations. This can be seen from Figure 7. If the step length was too big, it would not hit the global minimum, even though it converges quickly. And if it was too small, it didn't converge fast enough[3, p. 271].

The results for the training and test set for the linear Iris classifier are represented by confusion matrices in Figure 8. The error rate was 0.011 and 0.033 for the training and test set, respectively. This shows that the linear classifier works quite well for distinguishing between Iris-setosa, Iris-versicolor, and Iris-virginica based on the four features sepal length, sepal width, petal length, and petal width.

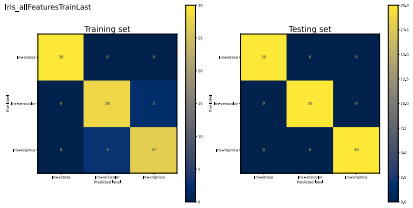
[2]This is also the reason why we need the tensorflow library to be installed for the loading to work. See requirements.txt in the submitted files

[3]This was done with the help of the python library scipy

[4]KMEANS from the sklearn.clusters library



(a) Results when using the first 30 instances for training and the last 20 for testing

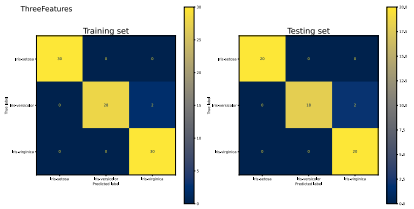


(b) Results when using the first 20 for testing and the last 30 for training.

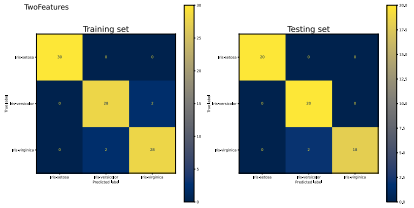
Figure 8: Confusion matrices for running with all four features included. This was primarily to test if it made any difference in which instances were used for training and testing. Often one would shuffle the instances before deciding which to use for training and testing.

We also checked whether using the last 30 samples for training instead of the first 30 and the first 20 samples for testing instead of the last 20 would alter the results. This is shown in Figure 8b. The ordering did not make a big impact.

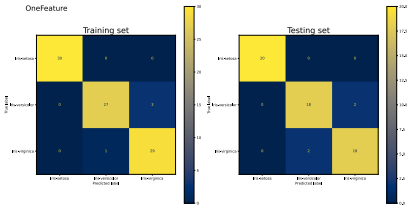
After the classifier had been tested with four features, we tried to see how eliminating one of the features would change the results. In order to decide which feature we were going to eliminate, we made a histogram of all the features for all the classes to see which had the most overlap. This is shown in Figure 10. It is evident from looking at the histograms that the feature "Sepal Width" has a great deal of overlap for the different classes. We then eliminated this feature, trained the classifier again, and got the results shown in Figure 9a. These results are somewhat similar to the results for four features, but the accuracy went down slightly. When only three features were used, the error rates were 0.022 and 0.033 for the training and test set, respectively. After this, we repeated the step of eliminating the feature with the most overlap and retrained the classifier. We did this three times until there was only one feature left. The features were removed in this order: "Sepal width", "Sepal length" and "Petal length". The corresponding results are shown in Figure 9. With only two features, the error rates were 0.044 and 0.033 for the training and testing set, respectively. With only one feature, the error rates were 0.044 and 0.067. This shows that the accuracy of the linear classifier drops when removing the features. However, the costs spared for measuring fewer features, or less computation could make it worth it in some cases. The error rates are displayed in Table 1



(a) Model with three features. So excluding the "sepal width"



(b) Model with two features. Namely the "petal length" and "petal width".



(c) Model using only one feature, the "petal width"

Figure 9: Confusion matrices for classifiers taking advantage of a different number of features. As we can see the performances did not drop too much.

Table 1: Iris classifier error rates. The different features are abbreviated. So 'S' stands for sepal, 'P' for petal, 'L' for length, and 'W' for width. This way PW stands for Petal width. The last two columns stand for error rates, given in percent. The subtext specifies which training set it belongs to. The first row belongs to all features with the first 30 examples for training, while the second belongs to all features with the last 30 examples for training

SL	SW	PL	PW	$Err_{train}[\%]$	$Err_{test}[\%]$
✓	✓	✓	✓	1	3
✓	✓	✓	✓	6	0
✓		✓	✓	2	3
		✓	✓	4	3
			✓	4	7

One thing that is worth mentioning is that all of the misclassifications are between the two subspecies Iris-versicolor and Iris-virginica. The other subspecies, Iris-setosa seems to be easy to distinguish from the other two for the linear classifier. If we once again take a look at the histogram in Figure 10, we can see that the overlap that occurs in the Sepal width and Sepal length features is most prominent between Iris-versicolor and Iris-virginica. This does make sense when considering that the classifier seemed to have an

easier time separating between Iris-setosa and something else than between Iris-versicolor and Iris-virginica. This also makes sense when we look at the scatter plot with two of the features, shown in Figure 11.

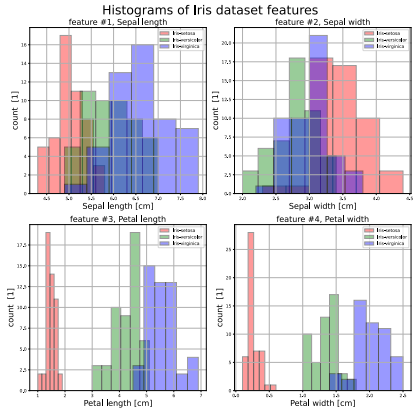


Figure 10: Histograms showing how the different features are distributed inside of the classes. As can be seen, the petal features are more spaced out than the sepal ones, and therefore more fitted for classification tasks.

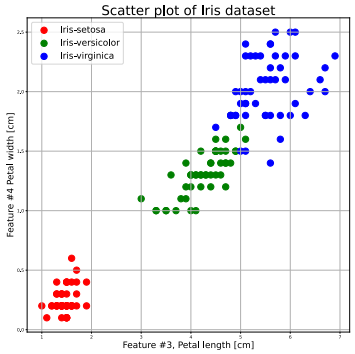


Figure 11: As can be seen the problem (with feature #3 and #4) seems almost linearly separable. It seems much easier to classify Iris-setosa, because it is completely separated. Along the axes, we can see the two features, which we decided were the most fitted. In reality, few problems are this close to being linearly separable. The plotted dots are all of the 150 instances in the IRIS set, divided into three classes.

4.4. Results MNIST

The results for the NN classifier for the MNIST dataset were quite good. The error rate was 3.1 percent, which means that it was able to correctly classify 9691 out of the 10'000 images in the test set. The confusion matrix for the NN classifier is shown in Figure 12. We also plotted four examples of misclassified images as well as four examples of correctly classified images. This is shown in Figure 13a and Figure 13b, respectively. If we once again take a look at the confusion matrix in Figure 12, we can see that the most common mistake that the classifier made was to mistake an image of the number four for an image of the number nine. As we can see, the error of misclassifying a seven for a one occurred 14 times, which is quite a lot. However, an image of a one was not once classified as a seven. This is interesting. In fact, the images of the number one were the ones that were most frequently correctly classified. They were classified correctly 1129 out of 1135 times, which corresponds to an accuracy of 99.5 percent. "The nearest-neighbor rule is a sub-optimal procedure; its use will usually lead to an error rate greater than the minimum possible, the Bayes rate"[3, p. 205]. So even though it is quite good, it can be better.

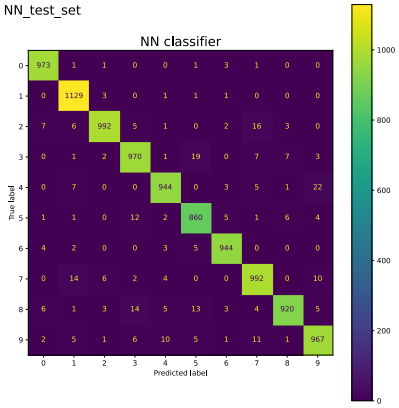
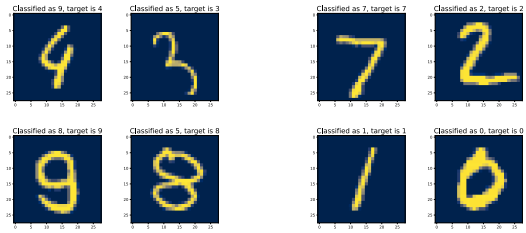


Figure 12: Confusion matrix for the nearest neighbor test set. It consists of 10'000 instances and we used a chunk size of 1000. The training set included 60'000 instances.



(a) Examples of four errors for the nearest neighbor classifier. The digits 9, 4, and 8 are similar, so it makes sense that these are misclassified.

(b) Examples of four correctly classified instances using the nearest neighbor classifier. The classifier was able to perform very well.

Figure 13: Examples of classifications using the nearest neighbor classifier. The training and test set included 60'000 and 10'000 instances.

With clustering the accuracy was supposed to drop slightly, and this is what we observed. The confusion matrix is shown in Figure 14. Nevertheless, the drop was small relative to the drop in computational complexity, see time usage in Table 2.

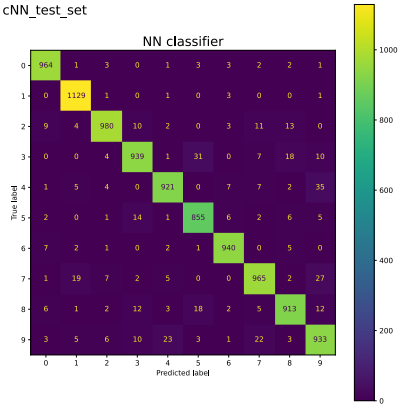


Figure 14: Confusion matrix for nearest neighbor test set with clustering. Every class had 64 clusters each. Still, the training and test set included 60'000 and 10'000 instances.

Another tweak to the nearest neighbor is the use of a majority vote for the k-closest instances. Once again the performance dropped a bit, but not too much. The results are displayed in Figure 15

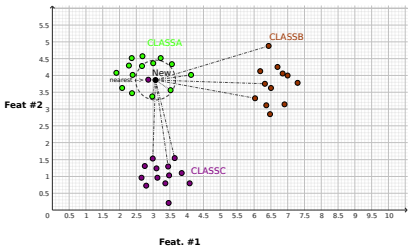


Figure 15: K-decision nearest neighbor confusion matrix. The model used the k=7 closest points. It also used clustering with 64 clusters for each of the classes. The training and test set had 60'000 and 10'000 instances.

As one can see from Figure 15, the performance did not drop so much with clustering and k-decision. In terms of computation time, however, it took much less time. This is without a doubt a good trade-off in most cases. Since we don't have endless resources, we must weigh the advantages and disadvantages against each other when designing classifiers.

5. Conclusion

In conclusion, the linear classifier worked quite well for distinguishing between Iris-Setosa, Iris-versicolor, and Iris-virginica. The error rate for the testing set was 1.1 percent for the training set and 3.3 percent for the testing set when the classifier was trained with the four features sepal width, sepal length, petal width, and petal height. This suggests that these three subspecies of the Iris flower have comes quite close to being linearly seperable when considering the abovementioned features. When testing the classifier with three, two, and one features, the accuracy went down.

Table 2: Results from MNIST Tasks. Displayed are the type of classifier (whether it is using clustering or majority vote). Also shown are the error rates and execution time. The time is completely dependent on which system it runs on, but since all tests were run on the same computer, the difference is the important part.

Clustering	k-value	$Err[\%]$	$t[s]$
	1	3	739
✓	1	5	95*
✓	7	7	93*

* It should be noted that the clustering time (210 seconds) is not included.

Especially when trained on only one feature, petal width. With only one feature, the error rate of the classifier had increased to 4.4 and 6.7 percent for the training set and the testing set, respectively. This shows that, in this case, the best performance was achieved when the classifier was trained on all of the four features in the dataset. The implementation of the linear classifier was centered around equations (7) and (8), from the theory section. The choice of the step length α in equation (8) was found through the trial and error method, which is shown in Figure 7. What we found was that a step length of 0.01 worked best.

The nearest neighbor classifier had an error rate of 3.1 percent for the test data in the MNIST dataset. As mentioned in the theory section, this type of classifier has some drawbacks. It requires a lot of calculation for each prediction and is prone to the error of being misclassified when it predicts the class of a data point that lies close to an outlier. Considering the "naivety" of the nearest neighbor approach to classification, the classifier did achieve decent accuracy when being applied to the test set of the MNIST dataset. The implementation of the nearest neighbor classifier was quite straightforward. In short, we downloaded the MNIST dataset from the Keras library in Python. Then we wrote the function "NN_predict", which for each of the images given as input, calculates the Euclidean distance from this image to all the others and classifies it as the class of the image which is its nearest neighbor.

This project has taught us a lot about the linear classifier and the NN/KNN classifiers. By implementing them in Python, we have gained a deeper understanding of exactly how they work. Additionally, we have seen how sub-optimal approaches can work quite well. For example, the NN classifier is by no means optimal, but it can work quite well in some cases. Another example is how clustering reduces the computational complexity significantly, without losing too much accuracy. Because we live in a limited world, these things definitely play a role when designing classifiers. We have learned that sacrifices can be made with regard to one particular area that yields significant gains in another. This is exemplified by the clustering approach

in the MNIST task. Here, a small sacrifice in accuracy yielded a significant benefit with regard to computational complexity. So that you can easier experiment with our project, the files are attached.

References

[1] Dar, P.: *25 open datasets for deep learning every data scientist must work with.* https://www.analyticsvidhya.com/blog/2018/03/comprehensive-collection-deep-learning-datasets/, 04.28.2018.

[2] TTT4275 Estimation, Detection and Classification, NTNU, M. H. Johnsen: *Classification.* 12.18.2017.

[3] Duda, Richard, Peter Hart, and David G.Stork: *Pattern Classification*, volume 2. January 2001, ISBN 0-471-05669-3.

Appendix A. Code

In the following pages, a pdf version of our Python implementation is shown. The files are also attached in the submitted zip file

iris

April 29, 2023

1 Code for the implementation of task IRIS

1.0.1 Importing libraries

[ ]:
```
# For computation
import numpy as np
# For confusion matrices and plotting
from sklearn import metrics
import matplotlib.pyplot as plt
```

1.1 1.a)

Defining classes and loading function

[ ]:
```
class Dataset:
    def __init__(self, instances):
        data = {"training": {"targets": [], "features":[]}, "testing":␣
↪{"targets": [], "features":[]}}
        labelToTarget = {"Iris-setosa": [1, 0, 0], "Iris-versicolor": [0, 1,␣
↪0], "Iris-virginica": [0, 0, 1]}
        for instance in instances:
            match instance.set:
                case 'training':
                    data["training"]["targets"].append(labelToTarget[instance.
↪label])
                    data["training"]["features"].append(instance.features)
                case 'testing':
                    data["testing"]["targets"].append(labelToTarget[instance.
↪label])
                    data["testing"]["features"].append(instance.features)

        # convert to numpy array
        data["training"]["targets"] = np.array(data["training"]["targets"]).
↪astype(float)
        data["training"]["features"] = np.array(data["training"]["features"]).
↪astype(float)
```

```
        data["testing"]["targets"] = np.array(data["testing"]["targets"]).
↪astype(float)
        data["testing"]["features"]  = np.array(data["testing"]["features"]).
↪astype(float)

        self.data = data
        self.classes_names = ["Iris-setosa", "Iris-versicolor",␣
↪"Iris-virginica"]
        self.feature_names = ['Sepal length [cm]', 'Sepal width [cm]', 'Petal␣
↪length [cm]', 'Petal width [cm]']
        self.colors = ['red', 'green', 'blue']
        self.DESCR = "Iris plants dataset"
```

[ ]:
```
class Instance:
    def __init__(self, features, label, set):
        self.features = features
        self.label = label
        self.set = set
```

[ ]:
```
def loadDataSet(features_list: list):
    Instances = []
    path = "IRIS_TTT4275/iris.data"
    n_classes = 3
    n_training = 30
    n_testing  = 20
    with open(path) as file:
        for _ in range(n_classes):
            for _ in range(n_training):
                line = file.readline()
                line = line.split(',')
                features = []
                for i in features_list:
                    features.append(line[i])
                label = line[-1].strip("\n")
                training_instance = Instance(features=features, label=label,␣
↪set="training")
                Instances.append(training_instance)
            for _ in range(n_testing):
                line = file.readline()
                line = line.split(',')
                features = []
                for i in features_list:
                    features.append(line[i])
                label = line[-1].strip("\n")
                testing_instance = Instance(features=features, label=label,␣
↪set="testing")
```

```
                Instances.append(testing_instance)
    return Dataset(Instances)
```

loading the iris-dataset with all features

[ ]:
```
IRIS_Dataset = loadDataSet([0, 1, 2, 3])
```

1.1.1 b) Training a linear classifier

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

[ ]:
```
def TrainClassifier(dataset: Dataset, step_length: float, n_iterations, W_0: np.
↪ndarray):
    W = W_0
    MSEs = []
    n_classes = 3
    n_training = 30
    for _ in range(n_iterations):
        Gradient_MSE = np.zeros(W_0.shape)
        MSE = 0
        for index in range(n_classes * n_training):
            x_k = dataset.data["training"]["features"][index]
            x_k = np.append(x_k, 1)
            z_k = np.dot(W, x_k)
            g_k = sigmoid(z_k)
            t_k = dataset.data["training"]["targets"][index]
            MSE += 1/2 * np.dot((g_k - t_k).T,(g_k - t_k))
            Gradient_MSE += np.outer((g_k - t_k)*g_k*(np.ones((1,3))-g_k), x_k.
↪T)
        W = W - step_length * Gradient_MSE
        MSEs.append(MSE)
    print(f"MSE in first iteration {MSEs[0]}, and last iteration {MSEs[-1]}\n")
    print(f"Our W is \n{W}\n")
    return W, MSEs
```

For deciding the step-length, plotted in the report. Uncomment to see it

[ ]:
```
# MSEs_with_diff_step_lengths = []
# step_lengths = [0.00001, 0.001, 0.01, 0.09, 0.3, 0.6]
# n = 10000
# for alpha in step_lengths:
#     MSEs_with_diff_step_lengths.append(TrainClassifier(IRIS_Dataset, alpha,␣
↪n, np.zeros((3, 5)))[1])
# plt.figure(figsize=(10, 10))
# for index ,MSEs in enumerate(MSEs_with_diff_step_lengths):
#     plt.plot(MSEs, label=f' = {str(step_lengths[index])}')
```

```
# plt.legend(fontsize=12)
# plt.title("MSE for different step lengths", fontsize=20)
# plt.xlabel('Iterations', fontsize=15)
# plt.ylabel('Mean Square Error', fontsize=15)
# plt.savefig(f"svg_figures/MSEvsAlpha.svg")
# plt.show()
```

Hyperparameters for the best classifier See the one above

[ ]:
```
alpha = 0.01
n = 10000
```

[ ]:
```
W_0 = np.zeros((3, 5))
W, _ = TrainClassifier(IRIS_Dataset, alpha, n, W_0)
```

1.1.2 1c) training part, confusion matrix, error rate

[ ]:
```
def TestClassifier(dataset: Dataset, W: np.ndarray):
    n_classes = 3
    n_training = 30
    n_testing = 20
    ConfMatrices = {"training": np.zeros((n_classes, n_classes)), "testing": np.
 ↪zeros((n_classes, n_classes)), "fileName": dataset.DESCR}
    errors = {"training": 0, "testing": 0}
    for index in range(n_classes * n_testing):
        x_k = IRIS_Dataset.data["testing"]["features"][index]
        x_k = np.append(x_k, 1)
        t_k = IRIS_Dataset.data["testing"]["targets"][index]
        g_k = np.dot(W, x_k)
        ConfMatrices["testing"][np.argmax(t_k), np.argmax(g_k)] += 1
        if np.argmax(t_k) != np.argmax(g_k):
            errors["testing"] += 1

    for index in range(n_classes * n_training):
        x_k = IRIS_Dataset.data["training"]["features"][index]
        x_k = np.append(x_k, 1)
        t_k = IRIS_Dataset.data["training"]["targets"][index]
        g_k = np.dot(W, x_k)
        ConfMatrices["training"][np.argmax(t_k), np.argmax(g_k)] += 1
        if np.argmax(t_k) != np.argmax(g_k):
            errors["training"] += 1

    print(f"Error-rates: \nTraining: {errors['training']/
 ↪(n_training*n_classes)}, Testing: {errors['testing']/
 ↪(n_classes*n_testing)}\n")
    return ConfMatrices
```

[ ]:
```
def PlotConfusionMatrix(matrix, fileName: str):
    plt.figure()

    fig, ax = plt.subplots(1, 2, figsize=(16,8))

    ax[0].set_title("Training set",fontsize=22)
    ax[1].set_title("Testing set", fontsize=22)
    metrics.ConfusionMatrixDisplay(confusion_matrix=matrix["training"],
                                   display_labels=IRIS_Dataset.classes_names,
                                   ).plot(ax=ax[0], cmap="cividis")
    metrics.ConfusionMatrixDisplay(confusion_matrix=matrix["testing"],
                                   display_labels=IRIS_Dataset.classes_names,
                                   ).plot(ax=ax[1], cmap="cividis")
    plt.tight_layout()
    plt.suptitle(f"{fileName}", fontsize=20, x=0.11)
    plt.savefig(f"svg_figures/{fileName}.svg")
    plt.show()
```

[ ]:
```
ConfMatrix = TestClassifier(IRIS_Dataset, W)
PlotConfusionMatrix(ConfMatrix, "Iris_allFeaturesTrainFirst")
```

1.1.3 d) Switching ordering.

[ ]:
```
# a little bit hard coded. Since we should only use it once
path = "IRIS_TTT4275/iris.data"
n_classes = 3
n_training = 30
n_testing  = 20

Instances = []

with open(path) as file:
    for _ in range(n_classes):
        for _ in range(n_testing):
            line = file.readline()
            line = line.split(',')
            features = line[0:-1]
            label = line[-1].strip("\n")
            testing_instance = Instance(features=features, label=label,␣
 ↪set="testing")
            Instances.append(testing_instance)

        for _ in range(n_training):
            line = file.readline()
            line = line.split(',')
            features = line[0:-1]
            label = line[-1].strip("\n")
```

```
            training_instance = Instance(features=features, label=label,␣
 ↪set="training")
            Instances.append(training_instance)
IRIS_Dataset = Dataset(Instances)
```

[ ]:
```
W_0 = np.zeros((3, 5))

W = W_0
MSEs = []
for i in range(n):
    Gradient_MSE = np.zeros(W_0.shape)
    MSE = 0
    for index in range(n_classes * n_training):
        x_k = IRIS_Dataset.data["training"]["features"][index]
        x_k = np.append(x_k, 1)
        z_k = np.dot(W, x_k)
        g_k = sigmoid(z_k)
        t_k = IRIS_Dataset.data["training"]["targets"][index]
        MSE += 1/2 * np.dot((g_k - t_k).T,(g_k - t_k))
        Gradient_MSE += np.outer((g_k - t_k)*g_k*(np.ones((1,3))-g_k), x_k.T)
    W = W - alpha * Gradient_MSE
    MSEs.append(MSE)
print(f"MSE in first iteration {MSEs[0]}, and last iteration {MSEs[-1]}\n")
print(f"Our W is {W}\n")
```

[ ]:
```
ConfMatrix = {"training": np.zeros((n_classes, n_classes)), "testing": np.
 ↪zeros((n_classes, n_classes)), "fileName": IRIS_Dataset.DESCR}
errors = {"training": 0, "testing": 0}
for index in range(n_classes * n_testing):
    x_k = IRIS_Dataset.data["testing"]["features"][index]
    x_k = np.append(x_k, 1)
    t_k = IRIS_Dataset.data["testing"]["targets"][index]
    g_k = np.dot(W, x_k)
    ConfMatrix["testing"][np.argmax(t_k), np.argmax(g_k)] += 1
    if np.argmax(t_k) != np.argmax(g_k):
        errors["testing"] += 1

for index in range(n_classes * n_training):
    x_k = IRIS_Dataset.data["training"]["features"][index]
    x_k = np.append(x_k, 1)
    t_k = IRIS_Dataset.data["training"]["targets"][index]
    g_k = np.dot(W, x_k)
    ConfMatrix["training"][np.argmax(t_k), np.argmax(g_k)] += 1
    if np.argmax(t_k) != np.argmax(g_k):
        errors["training"] += 1
print(f"Error-rates: \nTraining: {errors['training']/(n_training*n_classes)},␣
 ↪Testing: {errors['testing']/(n_classes*n_testing)}\n")
```

```
PlotConfusionMatrix(ConfMatrix, "Iris_allFeaturesTrainLast")
```

1.1.4 2a) Histograms

[ ]:
```
# loading data
n_examples = 50
paths = ["IRIS_TTT4275/class_1", "IRIS_TTT4275/class_2", "IRIS_TTT4275/class_3"]
features = {paths[0]: [[], [], [], []], paths[1]:[[], [], [], []], paths[2]:
 ↪[[], [], [], []]}
for i in range(len(paths)):
    with open(paths[i]) as file:
        for j in range(n_examples):
            line = file.readline().strip('\n').split(',')
            for k in range(len(line)):
                features[paths[i]][k].append(float(line[k]))
```

[ ]:
```
# displaying data
n_bars = 6
opacity = 0.4
plt.figure(figsize=(13,13))
plt.suptitle("Histograms of Iris dataset features", fontsize=28)
for index, feature in enumerate(IRIS_Dataset.feature_names):
    plt.subplot(2, 2, index+1)
    for class_nr in range(n_classes):
        plt.title(f"feature #{index+1}, {feature[:-5]}", fontsize=18)
        plt.hist(features[paths[class_nr]][index], label=IRIS_Dataset.
 ↪classes_names[class_nr], bins=n_bars, alpha=opacity, edgecolor='black',␣
 ↪color=IRIS_Dataset.colors[class_nr])
    plt.legend()
    plt.xlabel(feature, fontsize=18)
    plt.ylabel('count  [1]',fontsize=18)
    plt.grid()
plt.tight_layout()
plt.savefig("svg_figures/Iris_histograms.svg")
plt.show()
```

As Can be seen by the plots, feature 'Sepal width' is caotic and should be excluded. This is done below. Afterwards we scratch sepal length, the petal length

[ ]:
```
IRIS_Dataset = loadDataSet([0, 2, 3])
W_0 = np.zeros((3, 4))
W, _ = TrainClassifier(IRIS_Dataset, alpha, n, W_0)
ConfMatrix = TestClassifier(IRIS_Dataset, W)
PlotConfusionMatrix(ConfMatrix, "ThreeFeatures")
```

2b) with two features Petal width & length

[ ]:
```
IRIS_Dataset = loadDataSet([2, 3])
W_0 = np.zeros((3, 3))
W, _ = TrainClassifier(IRIS_Dataset, alpha, n, W_0)
ConfMatrix = TestClassifier(IRIS_Dataset, W)
PlotConfusionMatrix(ConfMatrix, "TwoFeatures")
```

And with only one feature (Petal width):

[ ]:
```
IRIS_Dataset = loadDataSet([3])
W_0 = np.zeros((3, 2))
W, _ = TrainClassifier(IRIS_Dataset, alpha, n, W_0)
ConfMatrix = TestClassifier(IRIS_Dataset, W)
PlotConfusionMatrix(ConfMatrix, "OneFeature")
```

1.1.5 2d) see report part.

Making of scatter plot, also commented out

[ ]:
```
# n_classes = 3
# plt.figure(figsize=(13,13))
# for index in range(n_classes):
#     plt.scatter(features[paths[index]][2], features[paths[index]][3],␣
 ↪label=IRIS_Dataset.classes_names[index], color=IRIS_Dataset.colors[index],␣
 ↪s=200)
# plt.legend(fontsize=20)
# plt.grid()
# plt.xlabel("Feature #3, Petal length [cm]", fontsize=20)
# plt.ylabel("Feature #4 Petal width [cm]", fontsize=20)
# plt.title("Scatter plot of Iris dataset", fontsize=30)
# plt.savefig("svg_figures/Iris_scatter.svg")
# plt.show()
```

mnist

April 29, 2023

1 Code for the implementation of task MNIST

1.0.1 Importing libraries

[ ]:
```
import numpy as np
from sklearn import metrics
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from scipy.spatial import distance
from keras.datasets import mnist
import time
```

1.0.2 1a) Creating NN classifier with chunksize 1000 and plot confusion matrix and print error rate

loading of data and declaring classes

[ ]:
```
class Dataset:
    def __init__(self, train_X, train_y, test_X, test_y):
        data = {"training": {"targets": [], "features":[]}, "testing":␣
 ↪{"targets": [], "features":[]}}

        data["training"]["features"] = train_X
        data["training"]["targets"] = train_y
        data["testing"]["features"] = test_X
        data["testing"]["targets"]  = test_y

        self.data = data
        self.DESCR = "MNISK dataset"
        self.n_training = 60000
        self.n_testing = 10000
        self.chunk_size = 1000
        self.n_chunks = self.n_training//self.chunk_size
        self.classes_names = ['0','1','2','3','4','5','6','7','8','9']
```

[ ]:
```
# loading the MNIST dataset
(train_X, train_y), (test_X, test_y) = mnist.load_data()
# reshape to fit for our class.
```

```
train_X = np.reshape(train_X, (train_X.shape[0], train_X.shape[1] * train_X.
 ↪shape[2]))
test_X = np.reshape(test_X, (test_X.shape[0], test_X.shape[1] * test_X.
 ↪shape[2]))
MNIST_Dataset = Dataset(train_X, train_y, test_X, test_y)
```

[ ]:
```
def NN_predict(images_predict, n_chunks, chunk_size, train_set):
    predicted = []
    time_s = time.time()
    for image in images_predict:
        min_dict = {"indices": [], "values": []}
        for i in range(n_chunks):
            # Divide into chunks
            reduced_test_set = train_set["features"][i*chunk_size:
 ↪(i+1)*chunk_size]
            distances = np.ravel(distance.cdist([image], reduced_test_set,␣
 ↪'euclidean'))
            # find the minimum euclidean distance from each chunk
            min_cluster_index = np.argmin(distances)
            min_global_index = i*chunk_size+min_cluster_index
            min_dict["indices"].append(min_global_index)
            min_dict["values"].append(distances[min_cluster_index])
        # find the minimum euclidean distance from all chunks
        min_dict_NN_index = np.argmin(min_dict["values"])
        NN_global_index = min_dict["indices"][min_dict_NN_index]
        predicted.append(train_set["targets"][NN_global_index])
    print(f"time for Predicting: {time.time()-time_s}")
    return predicted
```

[ ]:
```
def PlotConfusionMatrix(matrix, fileName: str):
    fig, ax = plt.subplots(figsize=(8, 8))
    metrics.ConfusionMatrixDisplay(confusion_matrix=matrix,
                                   display_labels=MNIST_Dataset.classes_names,
                                   ).plot(cmap="viridis", ax=ax)
    plt.title("NN classifier", fontsize=18)
    plt.tight_layout()
    plt.suptitle(f"{fileName}", fontsize=18, x=0.1)
    plt.savefig(f"svg_figures/{fileName}.svg")
    plt.show()
```

[ ]:
```
def conf_matrix_test(MNIST_Dataset: Dataset):
    n_classes = 10
    data = MNIST_Dataset.data
    testing_data = data["testing"]["features"]
    # Compute the prediction
    prediction = NN_predict(testing_data, MNIST_Dataset.n_chunks, MNIST_Dataset.
 ↪chunk_size, data["training"])
```

```
    # For statistics
    conf_matrix = np.zeros((n_classes, n_classes)).astype(int)
    n_misclassified = 0
    misclassified_examples = []
    correctly_classified_examples = []
    for i in range(len(prediction)):
        target = data["testing"]["targets"][i]
        pred = prediction[i]
        conf_matrix[target, pred] += 1
        if target != pred and len(misclassified_examples) < 4:
            n_misclassified += 1
            misclassified_examples.append([testing_data[i], pred, target])
        elif target != pred:
            n_misclassified += 1
        elif target == pred and len(correctly_classified_examples) < 4:
            correctly_classified_examples.append([testing_data[i], pred,␣
↪target])

    error_rate = float(n_misclassified) / len(testing_data)
    return conf_matrix, error_rate, misclassified_examples,␣
↪correctly_classified_examples
```

[ ]:
```
confusion_matrix, error_rate, misclassified_examples,␣
↪correctly_classified_examples = conf_matrix_test(MNIST_Dataset)
print("Error rate: " + str(error_rate))
PlotConfusionMatrix(confusion_matrix, "NN_test_set")
```

1.0.3 1b, c) Plot some examples

[ ]:
```
def PlotExamples(examples, fileName: str):
    fig, axs = plt.subplots(2, 2, figsize=(10, 10))

    for i, (image, predicted, target) in enumerate(examples):
        ax = axs[i//2, i%2]
        ax.set_title(f"Classified as {predicted}, target is {target}",␣
↪fontsize=18)
        ax.imshow(np.reshape(image, (28, 28)), cmap='cividis')

    fig.subplots_adjust(hspace=0.4)
    plt.savefig(f"svg_figures/{fileName}.svg")
    # using svg so that the graphics looks good in latex
    plt.show()
```

[ ]:
```
PlotExamples(misclassified_examples, "misclassified")
PlotExamples(correctly_classified_examples, "Correctly_classified")
```

1.0.4 2a) Clustering

[ ]:
```
def CreateClusters(n_clusters, train_x, train_y):
    time_s = time.time()
    # sorting the data
    n_classes = 10
    tuples = [(train_x[i], train_y[i]) for i in range(len(train_x))]
    tuples = sorted(tuples, key=lambda x: x[1])
    train_x = np.array([t[0] for t in tuples])
    train_y = np.array([t[1] for t in tuples])
    # flatten (normalize)
    train_x = train_x.flatten().reshape(train_x.shape)
    # using kmeans to create clusters
    kmeans = KMeans(n_clusters=n_clusters, random_state=0)
    # trick to find the break points for when the classes change
    break_points = [0]
    for i in range(len(train_y)-1):
        if train_y[i] != train_y[i+1]:
            break_points.append(i+1)
    break_points.append(len(train_x)-1)

    clusters = np.empty((n_classes, n_clusters, train_x.shape[1]))
    for i in range(n_classes):
        clusters[i] = kmeans.fit(train_x[break_points[i]:break_points[i+1]]).
↪cluster_centers_
    # clusters have all of the cluster centers for each class (64 each)
    clusters = clusters.flatten().reshape(n_classes*n_clusters, train_x.
↪shape[1])
    print(f"time for Clustering: {time.time()-time_s}")
    return clusters
```

[ ]:
```
def cnn_test(clusters, test_x, test_y):
    n_classes = 10
    test_x = test_x.flatten().reshape(test_x.shape)
    # for saving statistics
    conf_matrix = np.zeros((n_classes, n_classes)).astype(int)
    errors = 0
    time_s2 = time.time()
    for index, img in enumerate(test_x):
        distances = []
        for cluster in clusters:
            distances.append(distance.euclidean(img, cluster))
        # Divide index by 64 to get the class (64 cluster per class)
        pred = np.argmin(distances) // 64
        conf_matrix[test_y[index], pred] += 1
        if pred != test_y[index]:
            errors += 1
```

```
    print(f"time for Predicting: {time.time()-time_s2}")
    print("Error rate: " + str(errors/len(test_y)))
    PlotConfusionMatrix(conf_matrix, "cNN_test_set")
```

[ ]:
```
clusters = CreateClusters(64, MNIST_Dataset.data["training"]["features"],␣
↪MNIST_Dataset.data["training"]["targets"])
cnn_test(clusters, MNIST_Dataset.data["testing"]["features"], MNIST_Dataset.
↪data["testing"]["targets"])
```

The performance is a little bit poorer, but the time usage is quite substantially better

1.0.5 2c) Designing a KNN classifier

[ ]:
```
def cknn_test(clusters, k, test_x, test_y):
    n_classes = 10
    test_x = test_x.flatten().reshape(test_x.shape)
    conf_matrix = np.zeros((n_classes, n_classes)).astype(int)
    errors = 0
    time_s2 = time.time()
    for index, img in enumerate(test_x):
        distances = []
        for cluster in clusters:
            distances.append(distance.euclidean(img, cluster))
        # Basically same as the last one but with k neighbors. Therefore we
        # need a different rule for voting
        top_k = np.argsort(distances)[:k]
        votes = [0] * n_classes
        for val in top_k:
            votes[(val // 64)] += 1
        pred = np.argmax(votes)
        conf_matrix[test_y[index], pred] += 1
        if pred != test_y[index]:
            errors += 1
    print(f"time for Predicting: {time.time()-time_s2}")
    print("Error rate: " + str(errors/len(test_y)))
    PlotConfusionMatrix(conf_matrix, "cKNN_test_set")
```

[ ]:
```
cknn_test(clusters, 7, MNIST_Dataset.data["testing"]["features"], MNIST_Dataset.
↪data["testing"]["targets"])
```

Once again it is a little bit poorer than the NN, but takes a lot less time.