NTNU
Kunnskap for en bedre verden

Department of Electronic Systems

TTT4275 - Estimation, Detection and Classification

Classification Project - Iris and MNIST

Authors:
Martin Borge Heir
Eskil Berg Ould-Saada

April, 2021

Abstract

This paper is written as part of the course TTT4275 - Estimation, Detection and Classification, and focuses on the classification part of the course. In the paper, two different methods of classification will be tested and discussed. In the first part, a Linear Discriminant Classifier will be used to classify three species of the Iris flower family. In the second part, a Nearest Neighbor Classifier will be used to classify hand-written digits, before the data will be clustered and a k-Nearest Neighbor Classifier will be used to classify the same digits. In the analysis, the focus will be the relationships and structures of the data. When classifying Iris flowers, it was deduced that the order in which the data was split between training and test sets had an impact on performance. Furthermore, it was deduced that the more features considered in the model, the higher accuracy was reached. In the end, the best result was a success percentage of 98.3%. When classifying the hand-written digits, the Nearest Neighbor Classifier achieved the highest accuracy of 96.31%. The runtime was drastically reduced after introducing clustering, while also having a slight performance drop of 0.8%. Lastly, the k-Nearest Neighbor Classifier with k=7 proved to have a performance loss of 1.8% while not improving runtime.

Contents

1 Introduction **1**

2 Theory **1**

2.1 Classification 1

2.2 Linear Classifier 2

2.3 Overfitting 2

2.4 Nearest Neighbor Classifier 2

2.5 Clustering 3

2.6 K-Nearest Neighbors Classifier 3

2.7 Counfusion matrix 3

3 Iris flower task **4**

3.1 The task 4

3.2 Implementation and results 4

3.3 Discussion 8

4 MNIST task - handwritten numbers **9**

4.1 The task 9

4.2 Implementation and results 9

4.3 Discussion 11

5 Conclusion **12**

Bibliography **13**

Appendix **14**

A IRIS Code 14

B MNIST Code 17

1 Introduction

The goal of this project in general is to learn more about classification and how different classifiers can be used to solve different problems.

Classification is one of the most vital concepts of information filtering. Improvements in machine learning field in the past decade have revolutionized digital classifiers with their high accuracy and general implementation. As a consequence, digital classifiers have gained massive popularity and is used everywhere in today's society.

In this project we will have a look at two simple classifiers, and see how they perform on two different classification problems. The first classifier we will look at is the *Linear Classifier* which we will use to classify flower species in the Iris family. The second classifier we will look at is the *Nearest Neighbor Classifier*, which we will use to classify handwritten numbers. For each of the two tasks, we will introduce the datasets, explain the tasks, give implementations and results before discussing the results. Before diving into each of these problems, the theory related to each of these classifiers will be presented.

2 Theory

In this section, we will introduce the theory necessary to understand all the methods used in this report. We will have a look at the theory behind classification, the linear classifier, the nearest neighbor classifier and different related concepts such as overfitting and clustering.

2.1 Classification

Most of us already have an intuitive understanding of classification as being "putting stuff into a set of predefined categories". For example, we classify fruit into different classes such as bananas and apples and animals into classes such as horses and dogs. When taking a statistical approach to classification however, problems can be divided into three main groups. Problems are either *linearly separable*, *non-linearly separable*, or *inseparable* [3]. Linear separability means that each instance of a category can be separated with a *hyper plane.* In two dimensions, such as the example in Figure 1 [4], a hyper plane is the same a straight line. Non-linearly separable problems however, requires a nonlinear geometry to separate the instances of each category. In our two dimensional example this is a circle. Inseparable problems on the other hand are, as the name suggests, hard to separate. In these problems the instances of each category will overlap to some extent which makes separating them difficult or in worst case impossible. Unfortunately, most of real world problems belongs in this category.

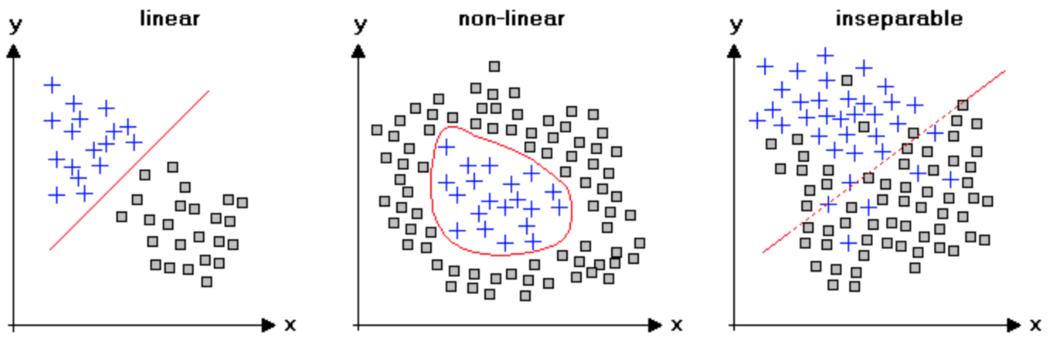


Figure 1: A two dimensional representation showing the geometric differences in separability for two different categories. Each axis represents a unique feature.

2.2 Linear Classifier

While there are many different classifiers, perhaps one of the simplest ones is the Linear Discriminant Classifier (LDC). As the name suggests, this classifier is designed for solving problems that are linearly separable. Still, very few problems are actually 100% linearly separable, which is why this method seldom gives the best results. The decision rule for this classifier is simple. For an instance x, it is decided that it belongs to the category ω_j that has the greatest discriminant function $g_j(x)$. In mathematical terms this can be described as

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

where every category ω_i has a corresponding discriminant function $g_i(x)$. Since the classifier is linear, it is sufficient that the discriminant functions are linear functions too.

$$g_i(x) = w_i^T x + w_{io} \tag{2}$$

The question now becomes, how are the weights w_i,w_{i0} selected such that g_j is activated for an $x \in \omega_j$? For this we use a method called *supervised learning* [2]. In short this can be formulated as an optimization problem where we want to minimize an error function between a guess and the actual value. The problem then becomes to adjust the weights such that this error function is minimized. This can be done using standard optimization techniques such as *gradient descent* [1]. Representing the discriminant functions g_i on a compact matrix form W, the gradient of a mean square error cost function with regards to W becomes

$$\nabla_W MSE = \sum_{k=1}^{N} \nabla_{g_k} MSE \nabla_{z_k} g_k \nabla_W z_k \tag{3}$$

Where z_k is a simple sigmoid function. It is common to use an activation function like this to achieve normalized output.

In practise the weights are trained by iterating through a high number of instances and for every guess, the error function is minimized by changing the weights a step length of *alpha* in the descent direction. In mathematical terms this becomes

$$W(m+1) = W(m) - \alpha \nabla_W MSE, \tag{4}$$

The step length is often chosen manually and considered a tuning variable. When the training process is done, the optimal weights should have been found. A common way to verify the accuracy of a classifier is to have a separate data set called a *test set*, and measure the number of correct guesses using this. The reasons for using a completely new data set for verification are many, but the most common one is due to what is called *overfitting.*

2.3 Overfitting

Overfitting is a very common concept in machine learning. When optimizing weights such as described in the previous section, the model is fitted to the data set it is training with. This data set is often called the *training set.* If too many iterations are used or the training set contains little variation or too few examples, the model will specialize in the training set and lack generality. As a result, this will lead to the test set performing worse.

2.4 Nearest Neighbor Classifier

The Nearest Neighbor Classifier is a different classifier that does not need training weighs or other a priory computations. In this classifier every instance x is matched with a set of templates T.

Each class can have one or several belonging templates. The decition rule is simple: x belongs to the class that the most similar template belongs to. To measure similarity, a vector distance function such as the *euclidean distance* is often used. This is defined as

$$d\left(x, ref_{ik}\right) = \left(x - \mu_{ik}\right)^T \left(x - \mu_{ik}\right) \tag{5}$$

To design such a classifier however, templates need to be selected. The simplest case is to use training data directly as templates and match every new instance with all the training data. Unfortunately, this has some disadvantages, since matching an instance with all templates requires a lot of computing. In the next section, a method for reducing the number of templates to reduce required computing will be introduced.

2.5 Clustering

Clustering is a method where the number of templates are reduced into a few representing templates called clusters. The key idea can be seen in Figure 2 [8], where similar templates are grouped together to form a single template called a cluster. There exist many clustering algorithms, with one being the *k-means* algorithm which is used in the clustering implementation in this paper.

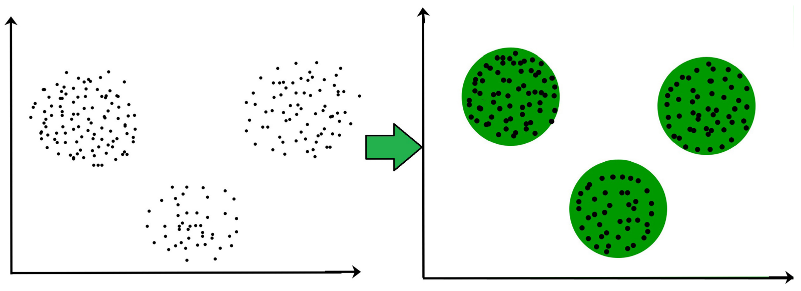


Figure 2: A simple example showing the key idea of clustering.

2.6 K-Nearest Neighbors Classifier

In section 2.4, the decision rule was to select the single most similar template for every instance. Another decision rule is to select the k nearest templates for every instance, and decide on the class that has the highest number of belonging templates among these k templates. Such a classifier is often called a *K-Nearest Neighbors Classifier*. In many cases, such a decision rule may achieve better performance than the simple rule expressed earlier, differing on the choice of k.

2.7 Counfusion matrix

The confusion matrix is a common way of displaying the results after classifying a number of instances. An instance will be placed in the column corresponding to the class where the classifier predicts it belongs, and on the row corresponding to the true label of the instance. The table below serves as an example with an extended row and column for summing the instances.

Predicted : $\rightarrow$	Class 1	Class 2	Class 3	
True :$\downarrow$				
Class 1	47	3	0	Sum
Class 2	6	44	0	Sum
Class 3	0	2	48	Sum
	Sum	Sum	Sum	Total

3 Iris flower task

This section will be about classification of different flower species. In this task we will look at the features of three different species of the Iris flower; Iris Setosa, Iris Versicolor and Iris Virginica. The Iris dataset, often called Fisher's Iris data, is a set of 150 data samples, split evenly on the three Iris flower classes, where each sample includes the flower's sepal length, sepal width, petal length and petal width. A photo of the three flower species and what the sepals and petals are is presented in Figure 3 [5]. The dataset is one of the few practical data sets which are close to linearly separable, as the three species have distinct features making them easier to classify. Thus, it is often used as a beginner data set for machine learning purposes.

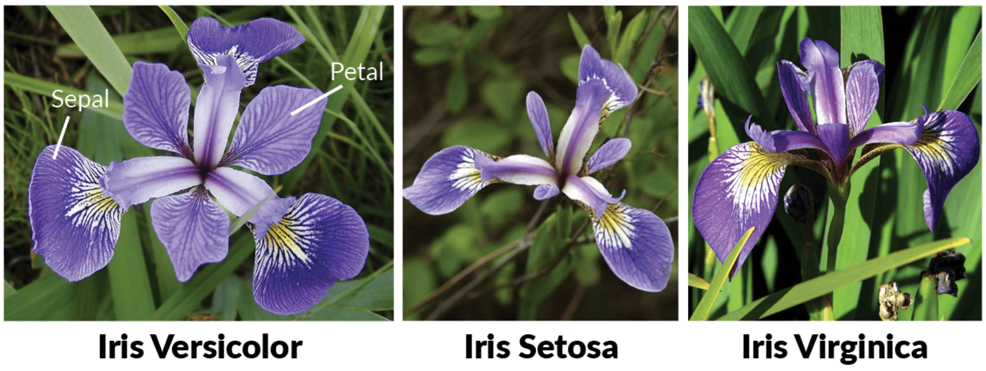


Figure 3: The three Iris flower species.

3.1 The task

This Iris task is twofold. In the first part, the dataset is split into training and test sets, and a linear classifier is trained on the training set to try to separate the three classes of the Iris flower. First, the first 30 samples of the data set are used for training while the last 20 are used to test. Second, the 30 last are used for training and the 20 first are used for testing. The two cases are to be compared, and confusion matrices and error rates for the sets will be discussed.

The second part of the task consists of investigating the linear separability of the dataset and how removing features from the dataset will affect the results of the linear classifier. A histogram of the different features and classes and a discussion of how the different features affect the classifier will be provided.

3.2 Implementation and results

In this section, the implementation of theory related to both parts of the Iris task will be presented, and the results will be provided underway. The implementation and results will follow the tasks chronologically, so that the results of one task will follow directly after the implementation of said task.

The linear classifier, as explained in theory in subsection 2.2, was implemented in Python as follows.

```python
def train_linear_classifier(train_data,W,iterations,step_size,train_size):
    loss = np.zeros([iterations,1],dtype=float)
    for i in range(iterations):
        g_vec = forward_propagate(train_data, W,train_size)
        dmse = get_mse_derivative(train_data,len(data[0]),g_vec,target)
        loss[i] = mean_squared_error(g_vec,target)
        W = W - step_size*dmse
    return W, loss
```

The implementation of the other functions used in the linear classifier can be found in Appendix A with more documentation. To use this classifier, some parameters needed to be set. The training was done over 10000 iterations. The step length, called *step_size* above, for the linear classifier was chosen by plotting the mean square error for different step sizes until the training converged. It was chosen to be $\alpha = 0.01$, as it produced the highest accuracy, as can be seen in Figure 4.

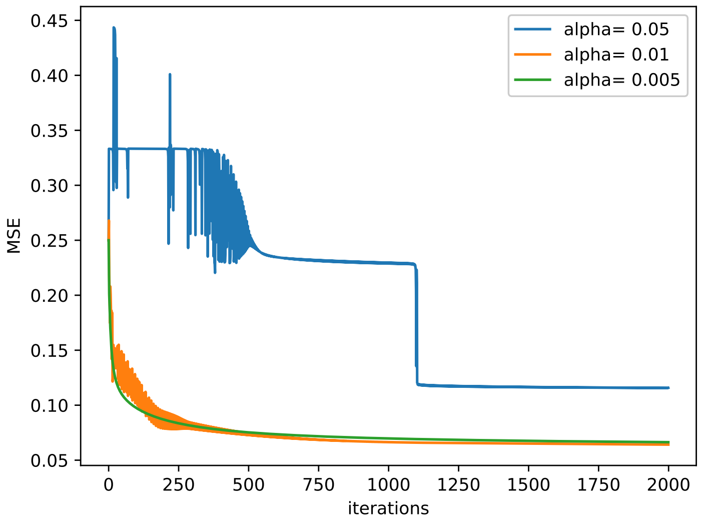
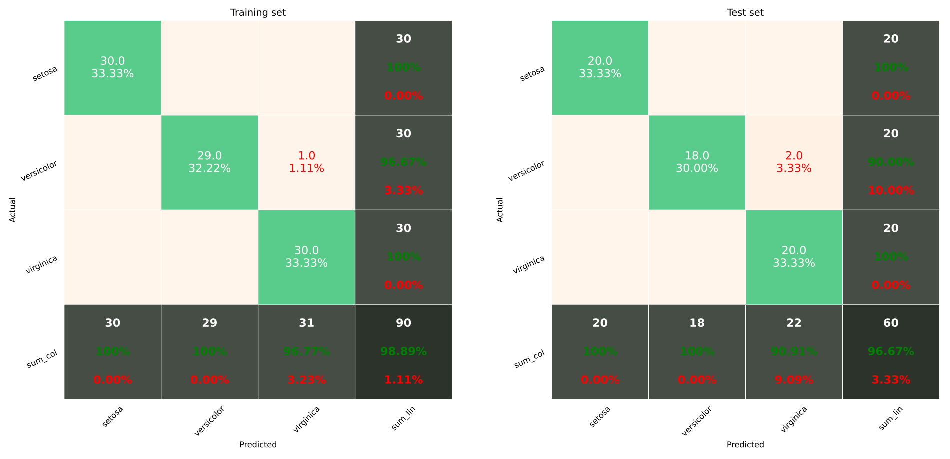


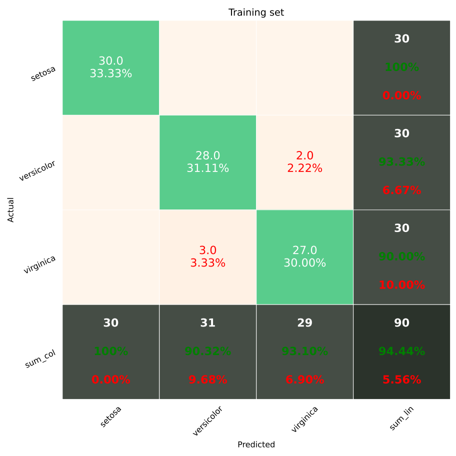
Figure 4: Plot of mean square error for different step lengths alpha

The results from the experiments are presented in the confusion matrices below. The confusion matrices when the first 30 samples are for training and the last 20 are for testing can be found in Figure 5, while the ones for when the 20 first samples are used for testing and the 30 last are used for training can be found in Figure 6. As observed from the figures, a test set success rate of 96.67% was obtained for the first case and 98.33% for the second case. This means that error rates of 0.023 and 0.017 were obtained for the respective cases, as seen in the lower right corner of the test set figures.

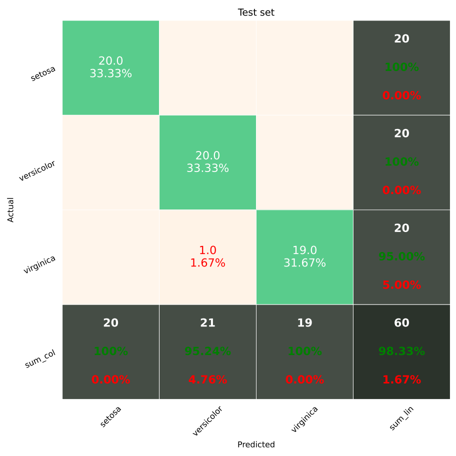


(a) (b)

Figure 5: Confusion matrices for the first case using 30 first samples for training, and 20 last for testing. The columns represent predicted labels while the rows represent actual labels



(a)



(b)

Figure 6: Confusion matrices for the second case using 30 last samples for training, and 20 first for testing

A histogram for each feature was then implemented, and can be seen in Figure 7, where the colours represent each of the three Iris classes. The histogram plot provides information about the separability for a given feature, useful for when trying to separate the classes. It is observed that the features with the most overlap between classes, in descending order, are sepal width, sepal length, petal length and petal width.

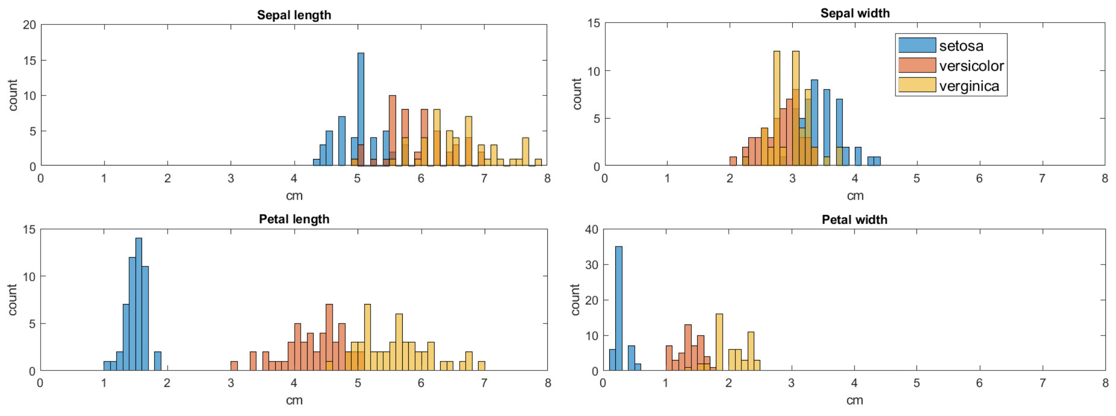
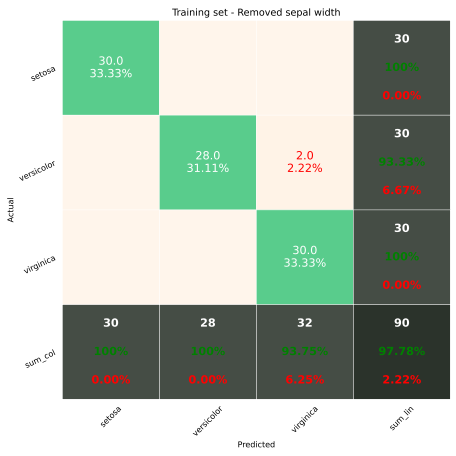
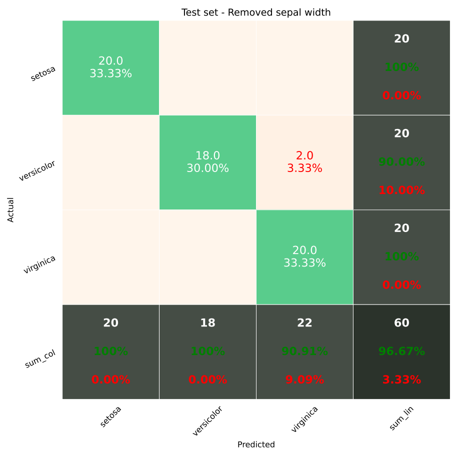


Figure 7: Histograms for each feature

In further experimentation, the most overlapping feature, sepal width, was removed to observe any change of performance in the linear classifier. The resulting confusion matrices after training the classifier with three features can be seen in Figure 8. This procedure was repeated two more times, by first removing the sepal length then the petal length, and the resulting confusion matrices be seen in Figures 9 and 10.

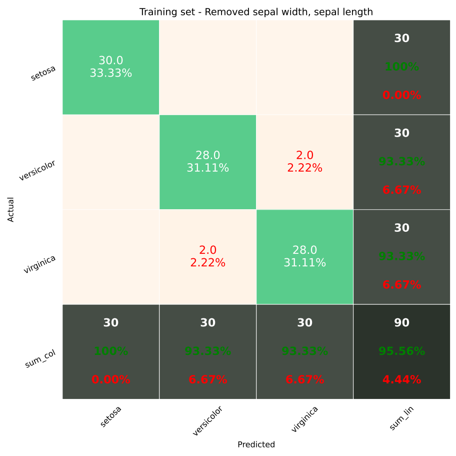


(a)

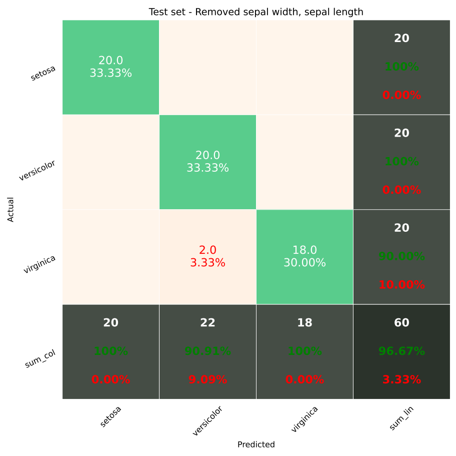


(b)

Figure 8: Confusion matrices after removal of one feature in data set

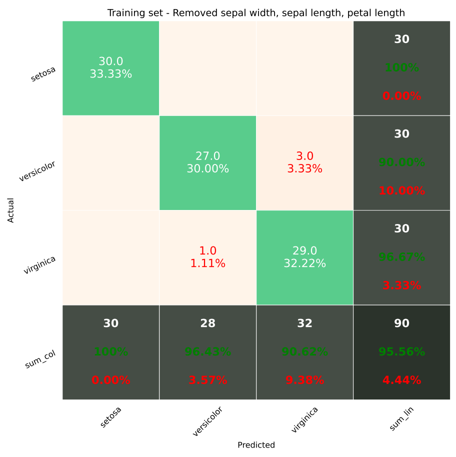


(a)

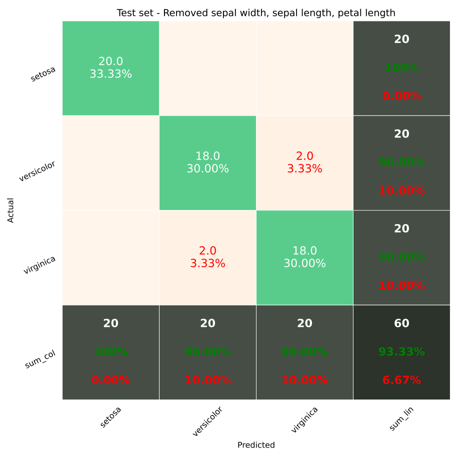


(b)

Figure 9: Confusion matrices after removal of two features in data set



(a)



(b)

Figure 10: Confusion matrices after removal of three features in data set

A comparison of error rates against features considered in the training of the linear classifier can be found in Table 1.

Number of features considered	Error rate on training set	Error rate on test set
4 features	0.011	0.033
3 features	0.022	0.033
2 features	0.044	0.033
1 feature	0.044	0.067

Table 1: Error rates for different number of features considered.

3.3 Discussion

In this section, the results obtained in the section above will be discussed. The discussion will be split between the two different Iris tasks. In the first part, the discussion will be about the two different ways the data was split in the task, and what impact they had on the results. Secondly, the discussion will be about the property of the features with respect to linear separability both as a whole and for the three separate classes.

When training the linear classifier with the first 30 samples as training samples, the error rates were very low for both the training and test sets, as seen in Figure 5. However, it is observed that the linear classifier obtained better results on the training data than the test data. This is logical, as the model is trained to fit the training data well, but large deviations here could also be an indication of overfitting. However, after testing the classifier with different parameters, it was concluded that the model generalized sufficienty, as the model worked equally as well for the test sets. Further, when using the 30 last samples for training and the first 20 for testing, it was observed that the model actually performed better on the test set than the training set, as seen in Figure 6. This could suggest that the first part of the Iris dataset is more linearly separable than the last part, resulting in better results for the linear classifier.

Looking at the results, it is observed that the classifier only misclassified Iris Virginica as Iris Versicolor and that Iris Seposa was never misclassified. This would suggest that Iris Seposa is more linearly separable from the other classes which are overlapping to a small degree. From the histograms in Figure 7, this assumption was confirmed, as the features of the Setosa species separates itself from the others to a large degree for multiple features. In order for a linear classifier to be most efficient, the classes upon which it trains should have linearly separable features. The fact that two of the flower species overlaps for multiple features in the histogram is therefore problematic for the performance of the linear classifier, and it is intuitive to think that a removal of these could potentially better the performance of the classifier. However, compared to the results with all features considered in Figure 5, it is observed that the performance is deteriorating when removing more and more features from the data set, especially on the training sets. The reason why this is happening is because a higher number of features actually increases the overall separability. It can be thought of like this: if only a single feature was considered with some overlap between the classes, these overlaps could potentially be separated by looking at another feature. This effect is why several features are used and not just one. Even though an added feature is less linearly separable than what is already included,the overall separability could potentially increase.

4 MNIST task - handwritten numbers

The following section will cover different tasks using what is called the MNIST dataset. The MNIST dataset is a dataset constructed from scanned documents in the National Institute of Standards and Technology (NIST), the American version of the Norsk Forskningsråd (NFR). Images of written digits were taken from scanned documents, normalized in size and centered. This made it excellent for evaluating models, allowing for focus on the machine learning with very little preparation required. For someone new to digital classifiers, this exercise is often used as a first. It consists of 70000 labeled 28x28 pixel images of hand-written digits, where each pixel has a grayscale value between 0 and 255. The dataset consists of 60000 training or template images written by 250 persons and 10000 test images written by 250 other persons. Each digit between 0 and 9 has its own class. Some examples of hand-written digits can be found in Figure 11 [6].

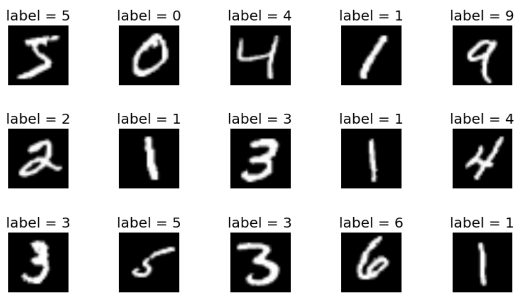


Figure 11: Examples of hand-written numbers in the MNIST database with their labels.

4.1 The task

The MNIST task is twofold. In general the task at hand is to test different classification models on the 10000 test images. For the first part, a Nearest Neighbor Classifier (1-NN), as described in subsection 2.4, is to be implemented using the Euclidean distance. After testing with smaller chunks of data, classification of the whole testing set against the whole training set is to be done and commented. Furthermore, some correctly and incorrectly classified digits will be provided and commented.

The second part consists of clustering the data for each digit into 64 clusters and use it as templates for the 1-NN Classifier. The theory behind this is explained in subsection 2.5. Furthermore, a k-nearest neighbor classifier, as explained in subsection 2.6, is to be designed and compared with the 1-NN classifier.

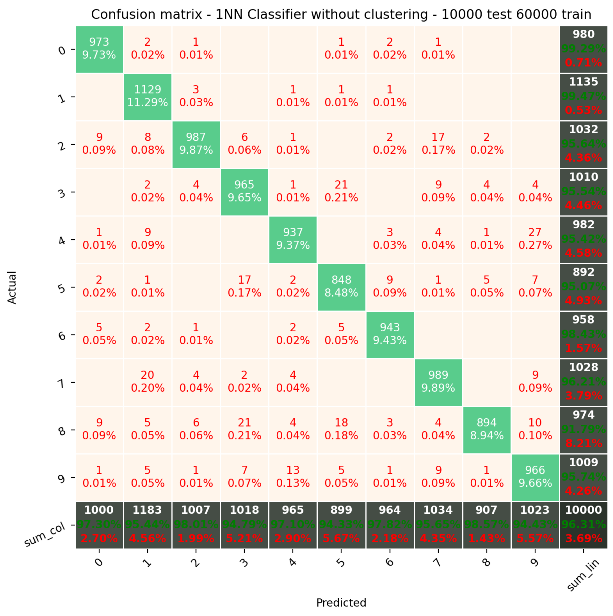
4.2 Implementation and results

In this section, the implementation of theory related to both parts of the MNIST task will be presented, and the results will be provided. The implementation and the results will follow the tasks chronologically, so that the results of one task will follow directly after the implementation of said task.

The implementation of the first part of the task, that is the Nearest Neighbor Classifier, was done in Python as follows below. The following code snippet only includes the lines of code necessary for the classifier, while the full code with more comments can be found in Appendix B.

```python
def nearest_neighbor_classifier(train_img,train_labels,test_img,test_labels,N_TRAIN,N_TEST):
    confusion_matrix = np.zeros((10,10),dtype=int)
    start = time.time()
    # Iterating through every test image
    for i in range(N_TEST):
        prediction = 0
        min = float('inf')
        # Comparing distance of test image to every training image
        for j in range(N_TRAIN):
            d = euclid_distance(test_img[i], train_img[j])
            if d < min:
                min = d
                prediction = train_labels[j]
        confusion_matrix[test_labels[i]][prediction] += 1
    end = time.time()
    print(f"Runtime of program is {(end-start)/60} minutes.")
    return
```

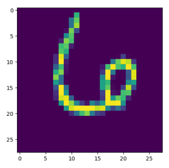
After testing the classifier with smaller chunks of data to reduce the classification runtime, the full 10000 image test set was compared to the full 60000 image training set. This was a lengthy affair, running for approximately 118 minutes, or close to two hours. The method ended up having a classification success of 96.31%, or that is to say an error rate of 0.0369, as seen in Figure 12.

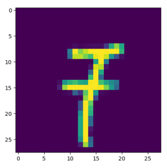


	0	1	2	3	4	5	6	7	8	9	sum_lin
0	973 9.73%	2 0.02%	1 0.01%			1 0.01%	2 0.02%	1 0.01%			980 99.29% 0.71%
1		1129 11.29%	3 0.03%		1 0.01%	1 0.01%	1 0.01%				1135 99.47% 0.53%
2	9 0.09%	8 0.08%	987 9.87%	6 0.06%	1 0.01%		2 0.02%	17 0.17%	2 0.02%		1032 95.64% 4.36%
3		2 0.02%	4 0.04%	965 9.65%	1 0.01%	21 0.21%		9 0.09%	4 0.04%	4 0.04%	1010 95.54% 4.46%
4	1 0.01%	9 0.09%			937 9.37%		3 0.03%	4 0.04%	1 0.01%	27 0.27%	982 95.42% 4.58%
5	2 0.02%	1 0.01%		17 0.17%	2 0.02%	848 8.48%	9 0.09%	1 0.01%	5 0.05%	7 0.07%	892 95.07% 4.93%
6	5 0.05%	2 0.02%	1 0.01%		2 0.02%	5 0.05%	943 9.43%				958 98.43% 1.57%
7		20 0.20%	4 0.04%	2 0.02%	4 0.04%			989 9.89%		9 0.09%	1028 96.21% 3.79%
8	9 0.09%	5 0.05%	6 0.06%	21 0.21%	4 0.04%	18 0.18%	3 0.03%	4 0.04%	894 8.94%	10 0.10%	974 91.79% 8.21%
9	1 0.01%	5 0.05%	1 0.01%	7 0.07%	13 0.13%	5 0.05%	1 0.01%	9 0.09%	1 0.01%	966 9.66%	1009 95.74% 4.26%
sum_col	1000 97.30% 2.70%	1183 95.44% 4.56%	1007 98.01% 1.99%	1018 94.79% 5.21%	965 97.10% 2.90%	899 94.33% 5.67%	964 97.82% 2.18%	1034 95.65% 4.35%	907 98.57% 1.43%	1023 94.43% 5.57%	10000 96.31% 3.69%

Figure 12: Confusion matrix for the 10000 test images against the 60000 training images used as templates.

Some of the correctly classified images can be found in Figure 13, while some of the incorrectly classified images can be found in Figure 14. This was done by storing sets of indices of the testing image and the predicted closest template image.


(a) Classified as 6


(b) Classified as 7

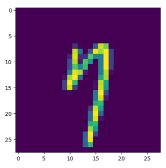
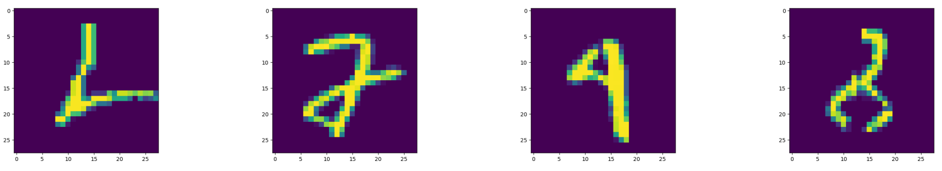

(c) Classified as 7

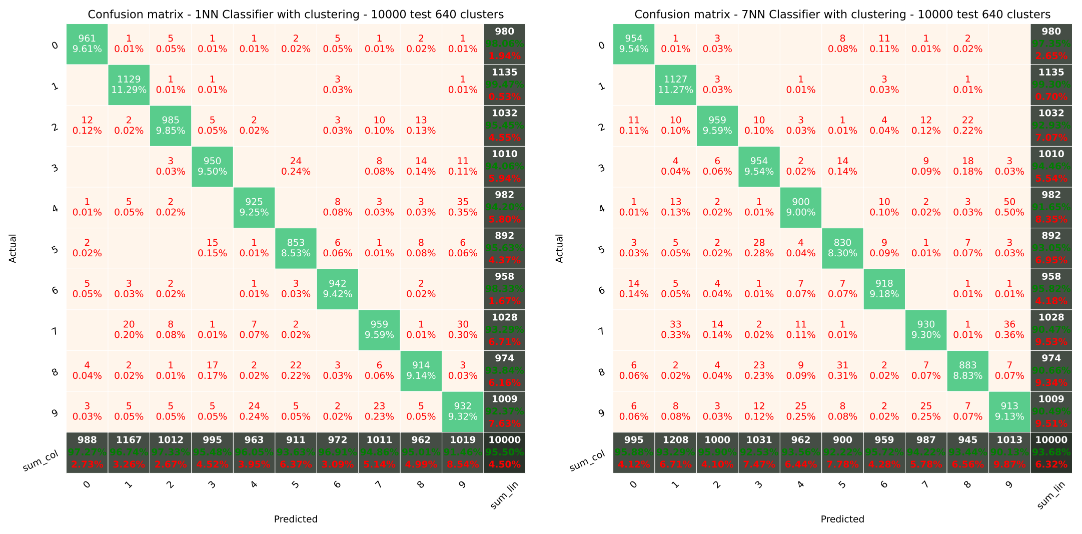
Figure 13: Correctly classified testing samples



(a) Classified 2 as 6 (b) Classified 2 as 7 (c) Classified 4 as 9 (d) Classified 8 as 2

Figure 14: Failed classification of testing samples.

For the second task, the 60000 training images were clustered into 64 clusters for each class, resulting in 640 templates. This was done using the clustering function found in Appendix B. For the first part, the clusters were used as input for the nearest neighbor classifier used in the previous part, and it resulted in an error rate of 0.045. A k-NN classifier, as explained in subsection 2.6, was then implemented (see Appendix B), and k was set to 7. This method resulted in an error rate of 0.063. The confusion matrices for both classifiers can be found in Figure 15. The running times were respectively 80 seconds for the 1-NN classifier and 75 seconds for the 7-NN classifier.



(a) 1NN Classifier with clustering, 95.5% success (b) 7NN Classifier with clustering, 93.7% success

Figure 15: Confusion matrices after clustering of training images

4.3 Discussion

This discussion will be split in three parts. The first part will consist of a discussion of human classifying against the results from the nearest neighbor classifier. In the second and third part, the discussion will cover why performance and processing times are affected with the introduction of clustering and the k-NN classifier.

Figure 13 shows a set of correctly classified digits. Any human would probably guess correctly for the first two images the majority of times. The third one, on the other hand, is not as clear as the others. While the classifier correctly classified it as a 7, a human could equally have guessed that it is either a 1 or a 4. On one hand, this shows that the classifier is able to classify correctly ambiguous data. On the other hand, if a human could not have classified a digit, then it is maybe a critique towards the hand-writing. Figure 14 shows instances where the classifier failed to classify digits correctly. For some of these cases, especially Figure 14a, the writing does not resemble any number, thereby making it equally difficult for a human to classify the image as for a computer. It is estimated that the human error rate for the MNIST dataset is around 2-2.5% [7]. This means that the error rate of 3.69% for the NN-classifier is actually pretty high. Looking at the confusion matrix for said classifier in Figure 12, it can be observed that the two most occurring errors happen when classifying 5's and 9's as respectively 8's and 4's. An intuitive thought is that a number 4 is more similar to a 9 than a 5 is to an 8. Thus, a human would more likely make the second mistake rather than the first. This indicates that the classifier works differently than the human mind.

The focus will now shift onto clustering. When introducing clustering, a performance loss of 0.8% was experienced. When clustering, the number of templates are reduced drastically, resulting in a general loss of information. Before clustering, an image would match similarity against 60000 images compared to 640 clusters. This means a weirdly written digit would more likely match a similar template rather than a cluster since the clusters are average based and thus more general. The runtime after introducing clusters however, is greatly reduced. Before clustering, the algorithm took around 118 minutes, compared to only about 80 seconds after clustering. Both of these runtimes are much higher than what is possible, and could be reduced greatly by writing faster code. However, since this is not an algorithms course the focus will be on the runtime difference instead. The dominant reason for using clustering in a 1-NN classifier is to reduce runtime, in this case by a large magnitude.

Lastly, the discussion will shift towards the introduction of the k-NN classifier. When changing to this decision rule a performace loss of 1.8% was experienced compared to running a 1-NN classifier with clustering. This is quite dramatic, and not intentional. Introducing a more complex decision rule should ideally increase performance and reduce the error rate, not the opposite. A reason why a k-NN perform worse than for the 1-NN could be that the majority vote is working against its purpose. If an image is closer to a cluster of the same class, but more clusters of another class are in the area, then the other class would be selected. Reducing k could possible lead to a better performance. The runtime difference between the 1-NN classifier with clustering and the 7-NN classifier with clustering is close to negligible, resulting in the 1-NN classifier being the better choice of the two.

5 Conclusion

In the first task the Linear Discriminant Classifier was looked at; how well it classfied the Iris set, how different dividing of training and test set impacted the performance, and how the features related to performance. Firstly, the linear classifier managed a best success rate of 98,3%. Further, the different partitions of training and test set proved to have a small impact on the performance most likely due to a small bias between the first and last part of the data set. Lastly, reducing the most inseparable features resulted in a performance loss such that, for the case at hand, more features resulted in an overall better performance. In the second task the Nearest Neighbour Classifier was implemented and studied. It was looked at how it performed on the MNIST data set, and how introducing clustering and a k-NN classifier affected performance and runtime. Firstly, it was found that the 1-NN classifier performed best with a success rate of 96.31%. Secondly, it was found that the clustering reduced computation time drastically, while also introducing a small performance loss. The k-NN classifier with k=7 performed worse than the 1-NN and did not improve computation time.

This project served as a general introduction to classifiers and basics of machine learning. Many important concepts concerning classification in general were covered such as different classifiers, supervised learning, overfitting and clustering. Furthermore, the project showed how some of these classifiers work on a couple of data sets, and why they work as they do on these sets.

Bibliography

[1] Wikipedia contributors. *Gradient descent.* [Online; accessed April 29, 2021]. 2021. URL: https://en.wikipedia.org/wiki/Gradient_descent.

[2] IBM Cloud Education. *Supervised Learning.* [Online; accessed April 29, 2021]. 2020. URL: https://www.ibm.com/cloud/learn/supervised-learning.

[3] M.H. Johnsen. *Classification.* 2017.

[4] *Linear separability figure.* [Online; accessed April 29, 2021]. URL: http://www.cs.uccs.edu/%20jkalita/work/cs587/2014/03SimpleNets.pdf.

[5] *Photo of Iris flowers and features.* [Online; accessed April 28, 2021]. URL: http://www.lac.inpe.br/%20rafael.santos/Docs/CAP394/WholeStory-Iris.html.

[6] *Photo of MNIST examples.* [Online; accessed April 29, 2021]. URL: https://morioh.com/p/e9539c98d631.

[7] P. Simard, Y. Le Cun and J. Denker. *Efficient Pattern Recognition Using a New Transformation Distance.* 1993. URL: https://papers.nips.cc/paper/1992/file/26408ffa703a72e8ac0117e74ad46f33-Paper.pdf.

[8] *Simple example of clustering.* [Online; accessed April 29, 2021]. URL: https://www.geeksforgeeks.org/clustering-in-machine-learning/.

Appendix

A IRIS Code

```
from logging import logProcesses
import numpy as np
from numpy.core.fromnumeric import argmax, reshape, transpose
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from pandas import DataFrame
import matplotlib.pyplot as plt
from confusion_matrix_pretty_print import _test_cm, pretty_plot_confusion_matrix
# Constants
N_CLASS = 3  # Number of classes
N_FEAT = 4  # Number of features

# Parameters for linear classifier
alpha = 0.01
N_ITER = 10000   # Number of interations
train_size = 30  # Number of training samples for each class
test_size = 20      # Number of test samples for each class

### Load data ###
irisdata = datasets.load_iris()['data']
class1 = irisdata[0:50]
class2 = irisdata[50:100]
class3 = irisdata[100:150]

# Splitting the three data sets into training and test sets
train1, test1 = train_test_split(class1,test_size=test_size,train_size=train_size,shuffle=False)
train2, test2 = train_test_split(class2,test_size=test_size,train_size=train_size,shuffle=False)
train3, test3 = train_test_split(class3,test_size=test_size,train_size=train_size,shuffle=False)
train = np.concatenate((train1,train2,train3),axis=None)
train = np.reshape(train,[train_size*N_CLASS,N_FEAT])
test = np.concatenate((test1,test2,test3),axis=None)
test = np.reshape(test,[test_size*N_CLASS,N_FEAT])

# Creating targets for each class
target1 = np.tile([1,0,0],train_size)
target2 = np.tile([0,1,0],train_size)
target3 = np.tile([0,0,1],train_size)
target = np.concatenate((target1,target2,target3),axis=None)
target = np.reshape(target,[train_size*N_CLASS,N_CLASS])

# Sigmoid function
def sigmoid(x):
    return np.array(1/(1+np.exp(-x)))

# Get confidence vectors for each observation
def forward_propagate(x_vec, W,n_batch):
    g_vec = np.zeros([n_batch*N_CLASS,N_CLASS])
    for i,x in enumerate(x_vec):
        x = np.append([x],[1])
        z = W @ x
        g_vec[i] = sigmoid(z)
    return g_vec
```

```python
# get update matrix
def get_mse_derivative(train,n_feat,g_vec,target):
    dmse = np.zeros([N_CLASS,n_feat+1])
    for xk,gk,tk in zip(train,g_vec,target):
        xk = np.append([xk],[1])
        xk = xk.reshape(n_feat+1,1)
        dmse += (((gk-tk)*gk).reshape(N_CLASS,1) *
        (np.ones((N_CLASS,1))-gk.reshape(N_CLASS,1))) @ xk.reshape(1,n_feat+1)
    return dmse

def train_linear_classifier(data,W,iterations,step_size,train_size):
    loss = np.zeros([iterations,1],dtype=float)
    for i in range(iterations):
        g_vec = forward_propagate(data, W,train_size)
        dmse = get_mse_derivative(data,len(data[0]),g_vec,target)
        loss[i] = mean_squared_error(g_vec,target)
        W = W - step_size*dmse
        print(f"Training iteration: {i}")
    return W, loss

def get_confusion_matrix(W,data_set,N_CLASS,N_BATCH):
    confusion_mat = np.zeros([N_CLASS,N_CLASS])

    predicted = forward_propagate(data_set, W,N_BATCH)
    row = -1
    for i,gk in enumerate(predicted):
        if i%N_BATCH == 0:
            row += 1
        col = np.argmax(gk)
        print(row)
        confusion_mat[row][col] += 1
    return confusion_mat

def plot_alphas():
    alphas = [0.05,0.01,0.005]
    for a in alphas:
        W = np.zeros([N_CLASS, N_FEAT+1],dtype=float)
        W,loss = train_linear_classifier(data,W,N_ITER,a,N_BATCH)
        #plot loss
        plt.plot(np.arange(N_ITER),loss,label="alpha= "+str(a))
        plt.legend()
        plt.xlabel("iterations")
        plt.ylabel("MSE")
    plt.show()

def task2_remove_feature(train,test,N_CLASS,N_FEAT,alpha,N_ITER,train_size,test_size):

    # Remove feature "sepal width"
    W = np.zeros([N_CLASS, N_FEAT],dtype=float)
    train = np.delete(train,1,1)
    test = np.delete(test,1,1)
    W, _ = train_linear_classifier(train,W,N_ITER,alpha,train_size)
    conf1_train = get_confusion_matrix(W,train,N_CLASS,train_size)
    conf1_test = get_confusion_matrix(W,test,N_CLASS,test_size)

    # Remove feature "sepal length"
    W = np.zeros([N_CLASS, N_FEAT-1],dtype=float)
    train = np.delete(train,0,1)
    test = np.delete(test,0,1)
    W, _ = train_linear_classifier(train,W,N_ITER,alpha,train_size)
    conf2_train = get_confusion_matrix(W,train,N_CLASS,train_size)
    conf2_test = get_confusion_matrix(W,test,N_CLASS,test_size)

    # Remove feature "petal length"
    W = np.zeros([N_CLASS, N_FEAT-2],dtype=float)
    train = np.delete(train,0,1)
    test = np.delete(test,0,1)
    W, _ = train_linear_classifier(train,W,N_ITER,alpha,train_size)
    conf3_train = get_confusion_matrix(W,train,N_CLASS,train_size)
    conf3_test = get_confusion_matrix(W,test,N_CLASS,test_size)

    # Plotting confusion matrices
    conf_arr = [[conf1_train,conf1_test],[conf2_train,conf2_test],[conf3_train,conf3_test]]
    labels=["sepal width","sepal width, sepal length","sepal width, sepal length, petal length"]
    for i, (train, test) in enumerate(conf_arr):

        df_cm = DataFrame(train, index=["setosa","versicolor","virginica"],
        columns=["setosa","versicolor","virginica"])
        pretty_plot_confusion_matrix(df_cm,title='Training set - Removed '
        +str(labels[i]),cmap="Oranges",pred_val_axis='x')

        df_cm = DataFrame(test, index=["setosa","versicolor","virginica"],
        columns=["setosa","versicolor","virginica"])
        pretty_plot_confusion_matrix(df_cm,title='Test set - Removed '
        +str(labels[i]),cmap="Oranges",pred_val_axis='x')
    return


def main():
    ### Task 1 ###
    #Generate empty weights
    W = np.zeros([N_CLASS, N_FEAT+1],dtype=float)

    # Train linear classifier
    W, loss = train_linear_classifier(train,W,N_ITER,alpha,train_size)

    # Compute confusion matrices
    confusion_train = get_confusion_matrix(W,train,N_CLASS,train_size)
    confusion_test = get_confusion_matrix(W,test,N_CLASS,test_size)

    # Plot confusion matrix for training set
    df_cm = DataFrame(confusion_train, index=["setosa","versicolor","virginica"],
    columns=["setosa","versicolor","virginica"])
    pretty_plot_confusion_matrix(df_cm,title='Training set',cmap="Oranges",pred_val_axis='x')
    # Plot confusion matrix for test set
    df_cm = DataFrame(confusion_test, index=["setosa","versicolor","virginica"],
    columns=["setosa","versicolor","virginica"])
    pretty_plot_confusion_matrix(df_cm,title='Test set',cmap="Oranges",pred_val_axis='x')

    ### Task 2 ###
    task2_remove_feature(train,test,N_CLASS,N_FEAT,alpha,N_ITER,train_size,test_size)
    return

if __name__ == '__main__':
    main()
```

B MNIST Code

```python
from logging import logProcesses
from os import PRIO_PGRP
import numpy as np
from numpy.core.fromnumeric import argmax, reshape, transpose
from sklearn import cluster
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.cluster import KMeans
from scipy.spatial import distance
from pandas import DataFrame
import matplotlib.pyplot as plt
from PIL import Image
from confusion_matrix_pretty_print import pretty_plot_confusion_matrix
from scipy.spatial import distance
import time

def load_data(N_TRAIN, N_TEST):
    with open('Data/train_images.bin','rb') as binaryFile:
        train_img = binaryFile.read()
    with open('Data/train_labels.bin','rb') as binaryFile:
        train_labels = binaryFile.read()
    with open('Data/test_images.bin','rb') as binaryFile:
        test_img = binaryFile.read()
    with open('Data/test_labels.bin','rb') as binaryFile:
        test_labels = binaryFile.read()
    train_img=np.reshape(np.frombuffer(train_img[16:16+784*N_TRAIN],dtype=np.uint8),(N_TRAIN,784))
    train_labels=np.frombuffer(train_labels[8:N_TRAIN+8],dtype=np.uint8)
    test_img=np.reshape(np.frombuffer(test_img[16:16+784*N_TEST],dtype=np.uint8),(N_TEST,784))
    test_labels=np.frombuffer(test_labels[8:N_TEST+8],dtype=np.uint8)
    return train_img, train_labels, test_img, test_labels

def euclid_distance(img1, img2):
    a = img1-img2
    b = np.uint8(img1<img2) * 254 + 1
    return np.sum(a * b)

def nn_classifier(train_img, train_labels, test_img, test_labels, N_TRAIN, N_TEST):
    correct_pred = [] # Array of set of indeces of correct predictions
    failed_pred = [] # Array of set of indeces of incorrect predictions
    confusion_matrix = np.zeros((10,10),dtype=int)
    start = time.time()
    # Iterating through every test image
    for i in range(N_TEST):
        prediction = 0
        pred_img_index = 0
        min = float('inf')
        # Comparing distance of test image to every training image
        for j in range(N_TRAIN):
            d = distance.euclidean(test_img[i], train_img[j])
            #d = euclid_distance(test_img[i], train_img[j])
            if d < min:
                min = d
                pred_img_index = j
                prediction = train_labels[j]
        if prediction == test_labels[i]:
            correct_pred.append([i, pred_img_index])
            confusion_matrix[test_labels[i]][test_labels[i]] += 1
        else:
            confusion_matrix[test_labels[i]][prediction] += 1
            failed_pred.append([i, pred_img_index])
    end = time.time()
    print(f"Runtime of program is {(end-start)/60} minutes.")
    df_cm = DataFrame(confusion_matrix, index=["0","1","2","3","4","5","6","7","8","9"],
    columns=["0","1","2","3","4","5","6","7","8","9"])
    pretty_plot_confusion_matrix(df_cm,title='Confusion matrix - 1NN Classifier without
    clustering - 10000 test 60000 train',cmap="Oranges",pred_val_axis='x')
    return

def knn_classifier(train_img, train_labels, test_img, test_labels, N_TRAIN, N_TEST,K):
    confusion_matrix = np.zeros((10,10),dtype=int)
    start = time.time()

    # Iterating through every test image
    for i in range(N_TEST):
        distance_vec = []
        for j in range(N_TRAIN):
            d = distance.euclidean(test_img[i], train_img[j])
            distance_vec.append(d)

        #Using k nearest clusters to get prediction
        prediction = get_majority_vote(distance_vec,train_labels,K)
        confusion_matrix[test_labels[i]][prediction] += 1
    end = time.time()
    print(f"Runtime of program is {(end-start)/60} minutes.")
    df_cm = DataFrame(confusion_matrix, index=["0","1","2","3","4","5","6","7","8","9"], columns=
    pretty_plot_confusion_matrix(df_cm,title='Confusion matrix - 7NN Classifier with clustering -
    return

def get_majority_vote(distances,labels,K):

    # Gathering votes for the K-nearest candidates
    sorted_index_by_distance = np.argsort(np.array(distances),axis=0)
    majority_vote = [0]*10
    majority_index_value = [0]*10
    for k in range(K):
        majority_vote[int(labels[sorted_index_by_distance[k]])] += 1
        majority_index_value[int(labels[sorted_index_by_distance[k]])] += k

    # Selecting the candidate with highest number of votes
    cand_inx = majority_vote.index(max(majority_vote))
    cand_count = majority_vote[cand_inx]
    cand_ival = majority_index_value[cand_inx]

    #Handeling case where there is a tie in votes
    for i in range(10):
        inx = majority_vote[i:10].index(max(majority_vote[i:10]))
        count = majority_vote[inx]
        ival = majority_index_value[inx]
        if ((count==cand_count) and (ival < cand_ival)):
            cand_inx = inx
            count_ival = ival

    return cand_inx
```

```python
def clustering(data,data_lab, n_clusters):
    N = 10
    # Sorting out classes from dataset
    templates_by_class = []
    for n in range(N):
        t = []
        for i,label in enumerate(data_lab):
            if n == label:
                t.append(data[i])
        templates_by_class.append(t)

    clusters = []
    labels = []
    for i in range(N):
        part = templates_by_class[i]
        kmeans = KMeans(n_clusters=n_clusters).fit(part)
        clusters.append(kmeans.cluster_centers_)
        labels.append([i]*n_clusters)

    cluster_img = np.array(clusters).flatten().reshape((n_clusters*N,data.shape[1]))
    cluster_lab = np.array(labels,dtype=int).flatten().reshape(n_clusters*N,1)
    np.save('cluster_img.npy',cluster_img)
    np.save('cluster_lab.npy',cluster_lab)
    return cluster_img, cluster_lab


def main():
    N_TRAIN = 60000
    N_TEST = 10000
    K = 7

    # Load data
    template_img, template_labels, test_img, test_labels = load_data(N_TRAIN, N_TEST)

    ### TASK 1 ###
    nearest_neighbor_classifier(template_img,template_labels,test_img,test_labels,N_TRAIN,N_TEST)

    ### TASK 2 ###
    # Run the clustering once, saves data into files
    cluster_img, cluster_labels = clustering(train_img,train_labels,64)
    # If already clustered, load the files instead of running the clustering
    cluster_img = np.load('cluster_img.npy')
    cluster_labels = np.load('cluster_lab.npy')

    # Run nearest neighbor classifier and k-nearest neighbor classifier
    nn_classifier(cluster_img,cluster_labels,test_img,test_labels,cluster_img.shape[0],N_TEST)
    knn_classifier(cluster_img,cluster_labels,test_img,test_labels,cluster_img.shape[0],N_TEST,K)
    return

if __name__=='__main__':
    main()
```