NTNU
Kunnskap for en bedre verden

Faculty of Information Technology and Electrical Engineering

Department of Electronic Systems

TTT4275 - Estimation, Detection and Classification

Classification Project 1 & 2

Authors:
Kristian Håland
Njål Svensen

Date: 30th April 2024

ABSTRACT

Machine learning is becoming an increasingly relevant topic, having several useful properties that benefits many different fields. This report is in accordance to the subject TTT4275 Estimation, detection and classification at NTNU. There has been implemented two different types of classifiers for two different tasks. For each task, a corresponding dataset has been provided and split into training and testing datasets. The performance of the classifiers have been evaluated using covariance matrices and error rates.

The first classifier aims to classify three different species of the Iris flower: Iris-Versicolor, Iris-Setosa, and Iris-Virginica [1]. These species are identified by examining four features, which are the length and width of the flowers' leaves. This is done using a linear discriminant classifier. The model produces good results with a total error rate as low as 2.7%.

The second classifier has the goal of classifying twelve different vowels present in human speech. This is done with a Plug-in Maximum A Posteori (MAP) classifier, using both a single Gaussian function for each vowel, as well as a Gaussian Mixture Model (GMM). All results for different scenarios yields an error rate in the range 40.6-58%, and therefore the classifier cannot be regarded as trustworthy. Inspection shows that the classifier may be subject to overfitting.

TABLE OF CONTENTS

1 INTRODUCTION

The use of machine learning is rapidly increasing in today's society, and is becoming a significant part of peoples lives. One example is the use of chatGPT, a chatbot which generates a conversation based on user input [2]. The essence of the problem with such a chatbot and machine learning in general includes classification.

A classifier is, simply put, something that takes in an input and assigns it to a predefined category based on its characteristics. This report will implement and explore two different classifiers solving two different tasks. Relevant theory will be explained followed by the implementation and results for each task separately.

2 THEORY

2.1 Linear Classification

A linear classifier is a type of classifier where two or more classes are separated by a line or a hyperplane [3]. This classifier may be too simple for practical problems, but can be useful in some cases due to its simplicity and computational efficiency [4]. The decision boundaries for a two and three dimensional case are shown in Figure 1.

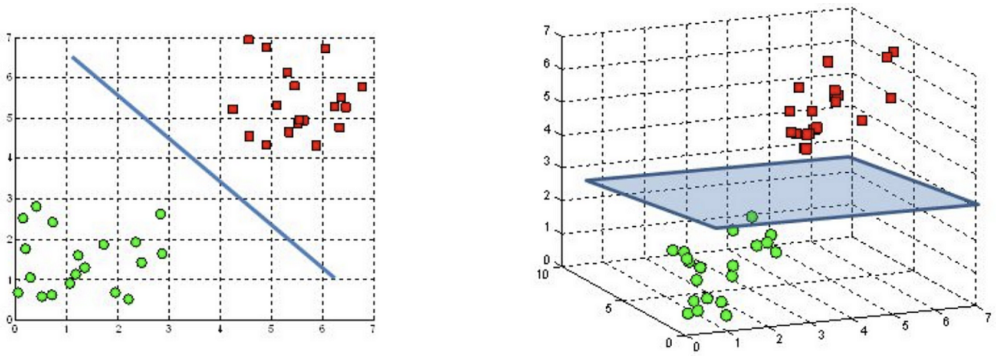


Figure 1: For the two dimensional case, we have two classes described by two features each. Therefore the decision boundary is a line. Likewise for the three dimensional case, each class is described by three features, and the decision boundary becomes a plane.

The following theory is taken from [4]. Each class in a linear classifier is described by a discriminant function $g_i(\mathbf{x})$, where the decision rule is

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

meaning the input sample $\mathbf{x}$ belongs to the class ω_j in which the function $g_i(\mathbf{x})$ yields the largest output value. Here $\mathbf{x}$ is a vector which holds the values of the features of the sample. For example for a class which is described by four features, $\mathbf{x}$ could be $\mathbf{x} = [1, 2.5, 3, 7.9]^T$. The function $g_i(\mathbf{x})$ is

$$g_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + w_{i0} \quad i = 1, \ldots, C, \tag{2}$$

where $\mathbf{w}_i$ is the weight vector, w_{i0} is offset for class ω_i and C is the number of classes. Combining the functions $g_i(\mathbf{x})$ for each class i into a vector $\mathbf{g} = [g_1, g_2, \ldots, g_C]^T$, and the weight vectors $\mathbf{w}_i$ into a weight matrix $\mathbf{W}$ gives the linear discriminant classifier on a compact matrix form

$$\mathbf{g} = \mathbf{W}\mathbf{x} + \mathbf{w}_0. \tag{3}$$

It is also possible to merge the $\mathbf{w}_0$-vector into the matrix-vector multiplication $\mathbf{W}\mathbf{x}$ simply by defining $\mathbf{w}_0$ as a new column vector in $\mathbf{W}$ and appending 1 to $\mathbf{x}$. Then $\begin{bmatrix}\mathbf{W} & \mathbf{w}_0\end{bmatrix} \rightarrow \mathbf{W}$ and $\begin{bmatrix}\mathbf{x}^T & 1\end{bmatrix}^T \rightarrow \mathbf{x}$, resulting in a single matrix-vector product to be computed.

$$\mathbf{g} = \mathbf{W}\mathbf{x} \tag{4}$$

For an input sample $\mathbf{x}$ which belongs to class i, $\mathbf{W}$ should weight each feature such that g_i in the $\mathbf{g}$-vector has the largest value.

An example of a neural network which describes Equation (4) is shown in Figure 2.

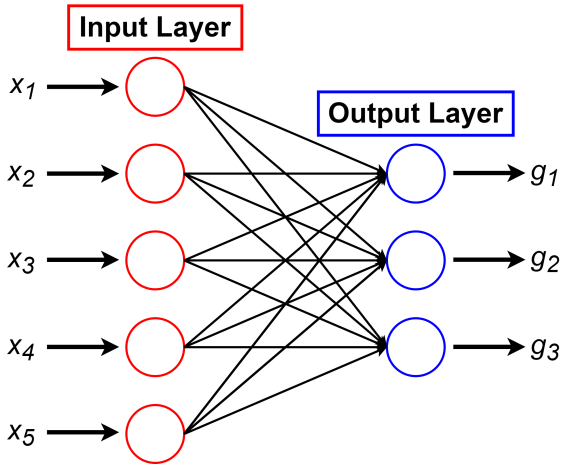


Figure 2: Example of a neural network with 5 inputs and 3 outputs.

Here there are 5 inputs which are the features in the $\mathbf{x}$-vector, and 3 outputs which are the elements in the $\mathbf{g}$-vector. The weights in the $\mathbf{W}$-matrix corresponds to the vectors going from the neurons in the input layer to the output layer.

2.2 MSE based training of a linear classifier

The following theory is taken from [4]. Training a linear classifier consists of updating the weight matrix $\mathbf{W}$ until the classifier reaches desired performance. In the case of a linear classifier, there is no optimization criteria of $\mathbf{W}$ since the model is not statistically based. Thus minimum-square-error (MSE) between the predicted and actual values is popular, given by

$$\text{MSE} = \frac{1}{2}\sum_{k=1}^{N}(\mathbf{g}_k - \mathbf{t}_k)^T(\mathbf{g}_k - \mathbf{t}_k). \tag{5}$$

Here $\mathbf{t}_k$ is the target vector for sample number k, meaning its true class. For example if a sample belongs to class 3, $\mathbf{t}_k = [0, 0, 1, ..., 0]^T$. Consequently $\mathbf{g}_k$ should hold values in the range $\{0, 1\}$ in order to match the output to the target vector. To achieve this a sigmoid function, which truncates all inputs to values in the range $\{0, 1\}$, may be used on each vector element i in each sample $\mathbf{g}_k$.

$$g_{ik} = \text{sigmoid}(\mathbf{z}_{ik}) = \text{sigmoid}(\mathbf{W}_{ik}\mathbf{x}_{ik}) \quad i = 1, \ldots, C \tag{6}$$

To optimize the linear classifier we want to have as small MSE as possible. This can be done with a gradient descent approach, meaning updating $\mathbf{W}$ in the opposite direction of the gradient. This is illustrated in Figure 3.

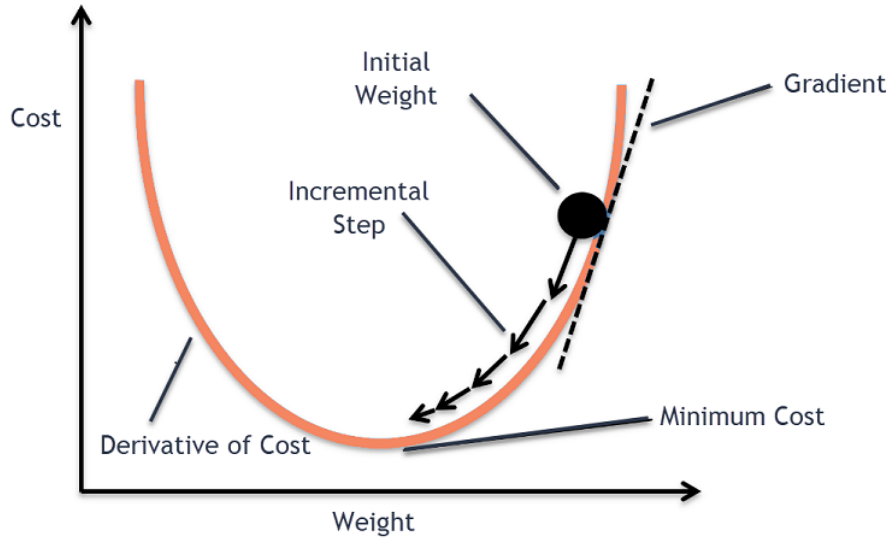


Figure 3: For a gradient descent, we start with an initial weight, which we update in the opposite direction of the gradient.

Here we have an initial weight, which is then updated until it reaches the point where we have minimum cost, meaning as low MSE as possible. Notice that this minimum can be a local minimum or a global minimum, meaning we have no guarantee that we have the optimal MSE. Therefore the initial weights are important.

Taking the gradient of the MSE in Equation (5) with respect to $\mathbf{W}$ gives

$$\nabla_{\mathbf{W}}\text{MSE} = \sum_{k=1}^{N} \nabla_{\mathbf{g}_k}\text{MSE} \cdot \nabla_{\mathbf{z}_k} g_k \nabla_{\mathbf{W}} z_k = \sum_{k=1}^{N} \left[(\mathbf{g}_k - \mathbf{t}_k) \odot \mathbf{g}_k \odot (1 - \mathbf{g}_k)\right] \mathbf{x}_k^\top. \tag{7}$$

$\odot$ means element wise multiplication. The update of $\mathbf{W}$ then becomes

$$\mathbf{W}(m) = \mathbf{W}(m-1) - \alpha \nabla_{\mathbf{W}}\text{MSE}, \tag{8}$$

where the minus sign indicates an update in the negative direction of the gradient. m is the iteration number, meaning $\mathbf{W}(m-1)$ is the current $\mathbf{W}$-matrix we have, and $\mathbf{W}(m)$ is the updated one. α is called the step factor, meaning how large step we should take for each update of $\mathbf{W}$. An important part of training the linear classifier is tuning α. If α is too large we can potentially overshoot and never find the best MSE, and if α is too small, training could take an unnecessary long time to converge.

During training we initially start with a random $\mathbf{W}$-matrix. Then for the current $\mathbf{W}$-matrix Equation (7) is computed using our training dataset (remember $\mathbf{g} = \mathbf{W}\mathbf{x}$). Then $\mathbf{W}$ is updated according to Equation (8), before a new iteration can be executed, now with the updated $\mathbf{W}$-matrix. This process can for example be executed until we reach a predetermined threshold for the MSE.

2.3 Plug-in MAP classifier

The following theory is taken from [4]. A plug-in Maximum A posteriori (MAP) classifier is a classifier in which we estimate parameters of the distribution of some data, which we then use in a parametric form. One of these parametric forms is the Gaussian density function given as

$$p(\mathbf{x}|\omega_i) = \frac{1}{(\sqrt{2\pi})^D |\mathbf{\Sigma_i}|} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu_\mathbf{i})^T \mathbf{\Sigma_i}^{-1}(\mathbf{x}-\mu_\mathbf{i})\right], \quad i = 1, \ldots, C. \tag{9}$$

Here $\mathbf{x}$ is the input sample to be classified, D is the number of features in $\mathbf{x}$, $\mu_\mathbf{i}$ and $\mathbf{\Sigma_i}$ are the mean value vector and covariance matrix for class i, and C is the number of classes. During training $\mu_\mathbf{i}$ and $\mathbf{\Sigma_i}$ for each class i gets trained using our training dataset, which then yields a Gaussian density function $p(\mathbf{x}|\omega_i)$ for each class.

The decision rule for the classifier is the Bayes Decision Rule (BDR)

$$\mathbf{x} \in \omega_j \Leftrightarrow p(\mathbf{x}|\omega_j)P(\omega_j) = \max_i p(\mathbf{x}|\omega_i)P(\omega_i), \tag{10}$$

where $P(\omega_i)$ is the A Priori probability of class ω_i. During classification, we check the function value of the Gaussian density function $p(\mathbf{x}|\omega_i)$ for each class i. Classification is done with a MAP approach, where the class which yields the largest value of $p(\mathbf{x}|\omega_i)P(\omega_i)$ is the class which the input sample is classified as.

Figure 4 shows an example of classification using three classes, where the input sample $\mathbf{x}$ only has one feature, D =1.

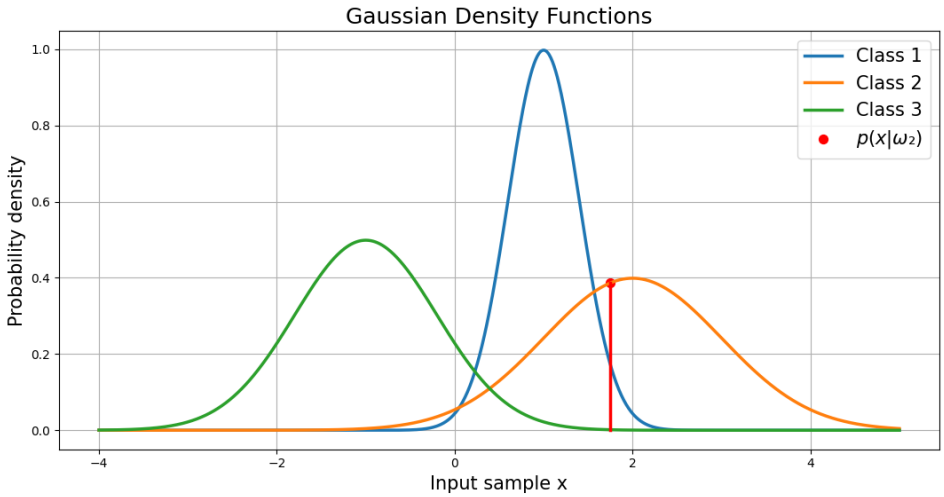


Figure 4: Three different Gaussian Density functions. The red line indicates where we find the Gaussian with the largest function value for the given input sample $\mathbf{x}$.

Assuming the A Priori probabilities $P(\omega_i)$ are equal for all three classes, the input sample $\mathbf{x}$ is here classified as class 2.

For input data with several features D, a single Gaussian function in Equation (9) may result in poor performance during classification. Combining several Gaussian functions for each class, where each Gaussian k has its own weight c_{ik}, may be a better choice when D is large. This model is called a Gaussian Mixture Model (GMM), given as

$$p(\mathbf{x}|\omega_i) = \sum_{k=1}^{M_i} c_{ik}\mathcal{N}(\mu_{\mathbf{ik}}, \mathbf{\Sigma_{ik}}) = \sum_{k=1}^{M_i} \frac{c_{ik}}{(2\pi)^D|\mathbf{\Sigma_{ik}}|} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu_{\mathbf{ik}})^T\mathbf{\Sigma_{ik}}^{-1}(\mathbf{x}-\mu_{\mathbf{ik}})\right] \quad i = 1, \ldots, C, \tag{11}$$

where $\sum_k c_{ik}$ =1. With a GMM model it is possible to a greater extent resemble the real distribution of a class. This is illustrated in Figure 5.

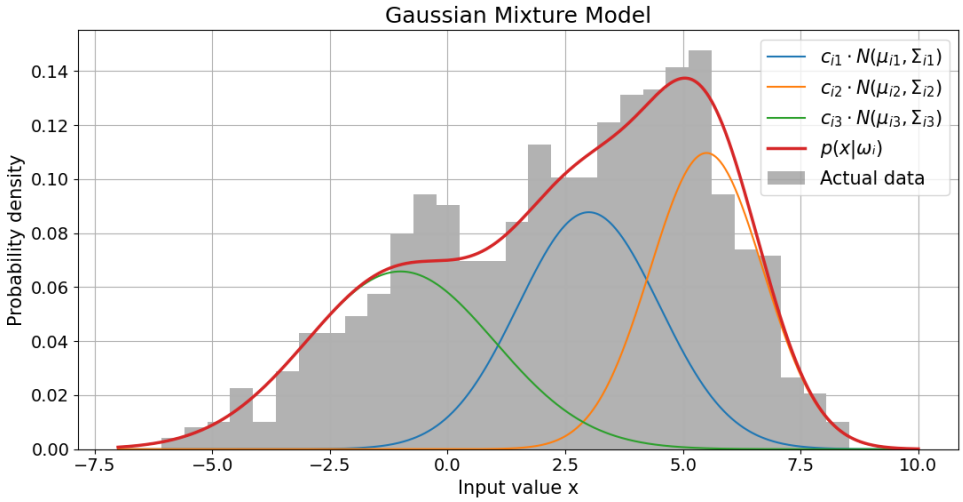


Figure 5: A Gaussian Mixture Model consisting of the mixture of three Gaussian distributions.

2.4 Parameters for evaluating a machine learning model

There are several useful parameters for evaluating the performance of a machine learning model. Three of them are the confusion matrix, accuracy and error rate.

A confusion matrix is a visual way of evaluating how well a machine learning model classifies data. Figure 6 shows an example of the confusion matrix of a model which has four classes, where 50 samples are used for each class.

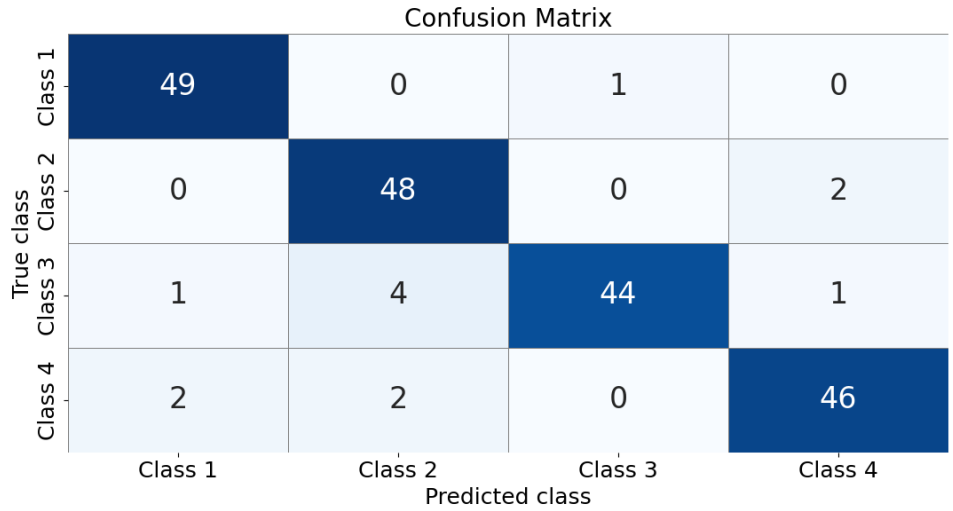


Figure 6: Example of a confusion matrix.

The elements on the diagonal marked in blue shows how many test samples have been classified correctly for each class. All the other elements shows the errors our machine learning model has done during classification. For example the row for true class 1 indicates one test sample were classified as class 3, when it in reality belonged to class 1. A confusion matrix can among other things be used to examine which classes our machine learning model struggles with.

Accuracy refers to a metric which quantifies the number of correct predictions made out of all the predictions made [5]. For example we could have an accuracy of 90%, meaning 90% of the test data were classified correctly. Accuracy is found by adding the correctly classified data on the diagonal of the confusion matrix (see Figure 6), and dividing by the total amount of test data used. Error rate is simply one minus the accuracy, which quantifies the number of incorrect predictions [5]. With the example above we would have an error rate of 10%.

2.5 Underfitting and Overfitting

A classifier should be able to classify unseen data correctly, meaning the classifier is able to generalize [6]. There are two problems which could arise during training: underfitting and overfitting. This is illustrated in Figure 7. Underfitting means our classifier is to simple, where it is not able to capture the complexity of the data [7]. The usual causes of underfitting are that the training dataset is to small, or that the features in the dataset is insufficient for representing each class [7]. Overfitting on the other hand means the classifier perform well on the data it is trained on, but is insufficient on the dataset for testing. Usual causes of overfitting are a classifier which is to complex, or a to small training dataset relative to the model complexity [8]. If a classifier performs poorly, these two cases should be taken into consideration in order to determine eventual improvements of the classifier.

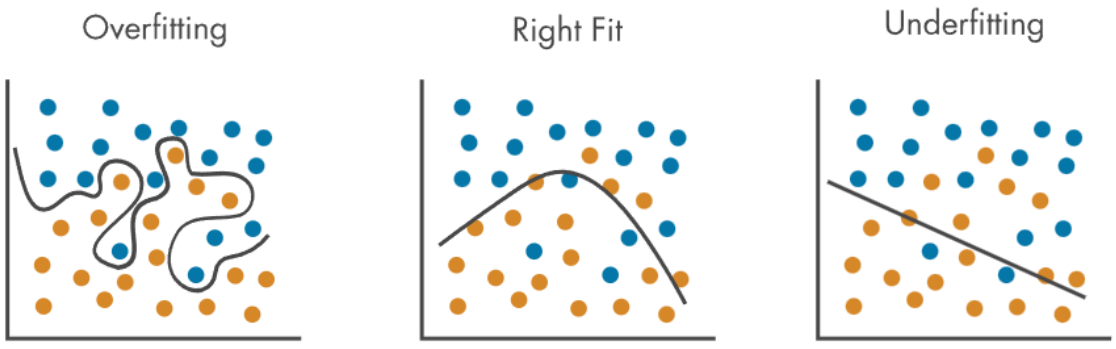


Figure 7: The classifier on the left is able to classify the training data, but would perform poorly on testing data. The classifier on the right is to simple for classification. The figure is taken from [8].

3 TASK

3.1 The Iris task

The Iris flower dataset is a well known problem within the field of machine learning. The iris flower comes in several variants including three called Setosa, Versicolor and Virginica, shown in Figure 8 [1].

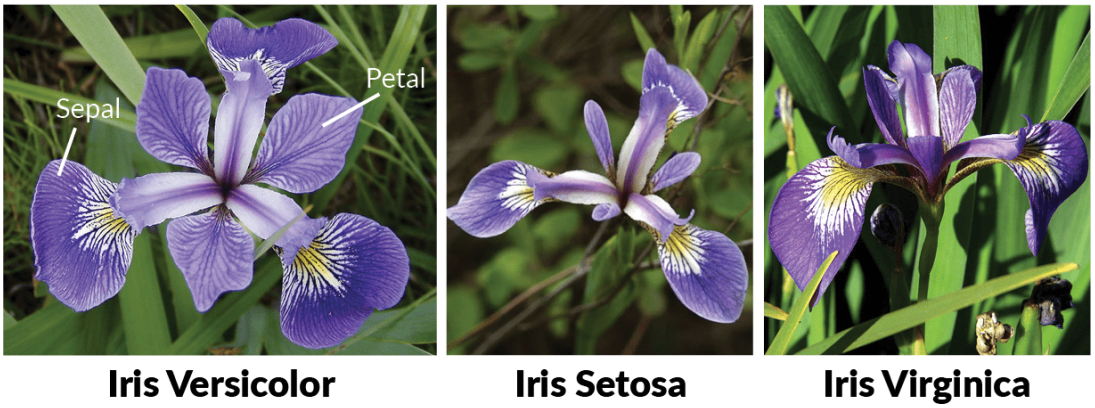


Figure 8: Three species of the iris flower [1].

Each species consists of two different leaves, where the large one is called Sepal, and the small one is called Petal. This is shown in Figure 9.

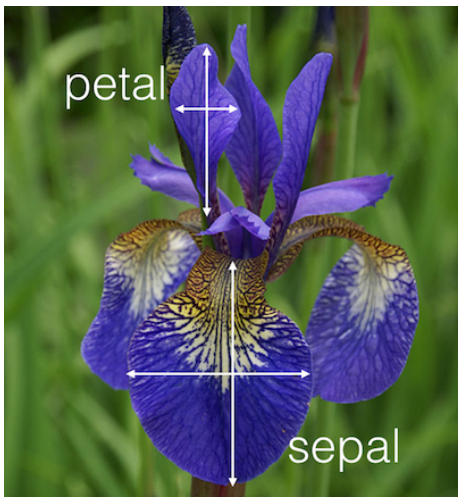


Figure 9: The Septal and Petal leaves for an iris flower [1].

The three iris flower species have different sizes of the Petal and Sepal leaves. If we use the width and length measurement for each leaf, we can classify every flower using these four measurements or "features" as is more commonly used when dealing with machine learning.

The Iris task is one of a few practical problems where each class (species) can be close to linearly separable [1]. For the Iris task there will be designed a linear classifier with the purpose of classifying the three Iris flower species shown in Figure 8. The classifier will then be inspected to evaluate its performance. Furthermore the importance of each of the features will be inspected with respect to linear separability and determine the importance of each feature.

The dataset provided to train and test the classifier consists of 150 samples of Iris flowers with 50 samples for each flower species. Each sample is a list with 4 values representing the measurements of the 4 features of each flower.

The following will be implemented for the Iris task:

1. Construct a linear classifier based on the theory presented in Subsections 2.1 and 2.2.
    - Split the dataset, using the first 30 samples for training and the remaining 20 samples for testing.
    - Repeat the process, this time training with the last 20 samples and testing on the first 30.
    - Analyze and compare the performance of the classifier in both scenarios using error rate and confusion matrix for both the training and test dataset.
2. Assessing Data Separability.
    - Generate histograms to visualize and compare the linear separability of individual features within the dataset.
    - Follow the training and testing from step 1, gradually reducing the feature count to three, two, and then one, while omitting the least conducive feature each time.

3.2 The Vowels task

There are several vowels present in human speech. Each class of vowel are made up of three or four so called formants, meaning peaks in frequency which makes up the sound of the vowel [1]. Where these formants are depends on the person speaking, since someones voice depends on age, sex as well as other factors. Figure 10 shows an example of the frequency spectrum of the "ae" and "ih" vowels.

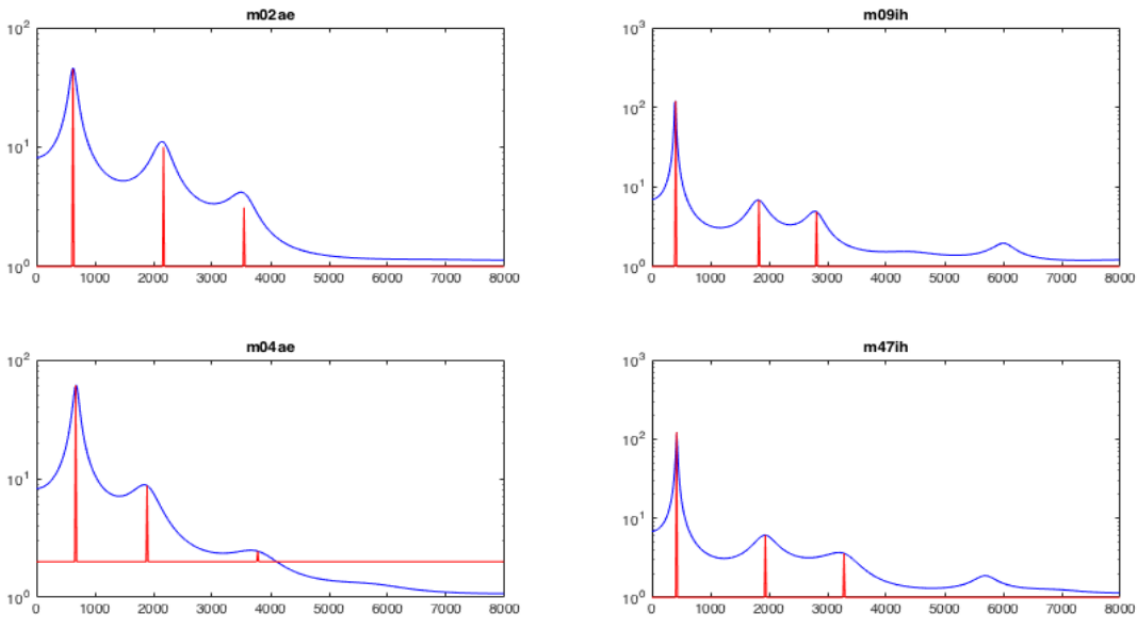


Figure 10: Formant examples of the "ae" and "ih" vowels. "m02ae" indicates that man number two is saying the vowel "ae". The figure is taken from [1].

The vowels dataset consist of 12 different classes of vowels, where the features of each vowel are the formants in its frequency spectrum. For the vowels task there will be designed a Plug-in Maximum

A posteori (MAP) classifier with the purpose of classifying each of the vowels. Same as described in Subsection 3.1, the classifier will be inspected to evaluate its performance. The classifier will be designed both with a single Gaussian density for each class, as well as with a Gaussian Mixture Model (GMM). The two approaches will then be compared to assess their performances. The use of a full and a diagonal covariance matrix will also be inspected.

The following will be implemented for the vowels task:

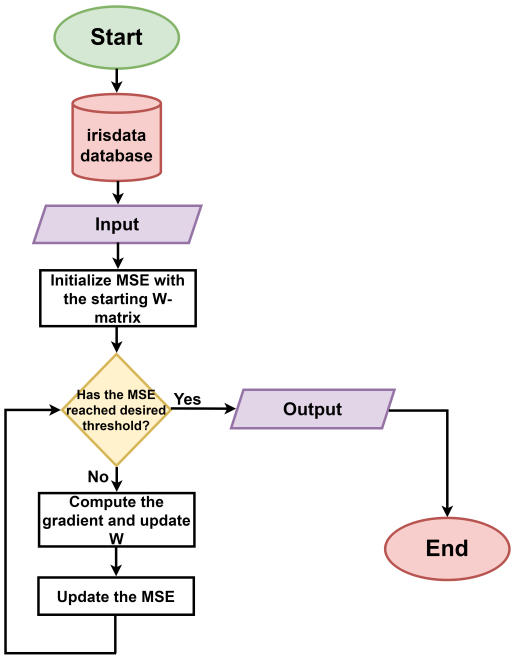
1. Design of a single Gaussian model for each class. This will be done both with a full and a diagonal covariance matrix.
    - The sample mean vector and covariance matrix will be found for each class.
    - The confusion matrix as well as the error rate will be found for the test dataset.
2. Design of a GMM for each class, both with 2 and 3 Gaussians for each class.
    - The GMM models are to be designed using a diagonal covariance matrix.
    - The confusion matrix and error rate will be found using both 2 and 3 Gaussians per class.

4 IMPLEMENTATION AND RESULTS - IRIS TASK

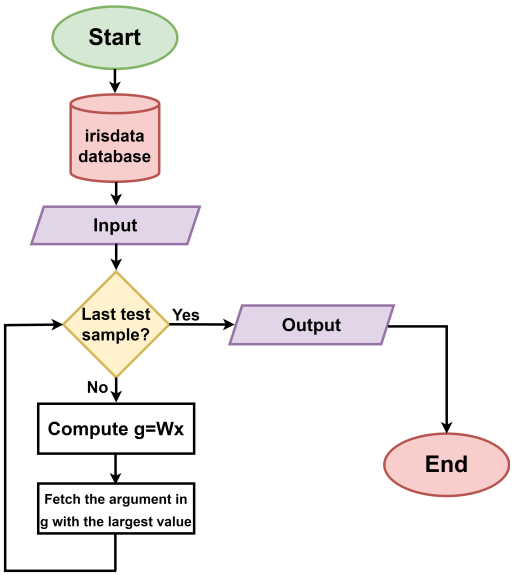
This section will cover the implementation and findings of the Iris task. The implementation is carried out using Python, with the code available in Appendix A through the provided GitHub link. The dataset used can also be accessed in Appendix A. Afterwards the results of the classifier will be showcased and discussed in accord with the theory described in Section 2.

4.1 Implementation of the Iris task

Instead of providing the entire code, Figure 11 shows a flow chart of the overall process of training and testing the classifier.



(a) Flowchart of the training of the iris classifier.



(b) Flowchart of the testing of the iris classifier.

Figure 11: Flowcharts of training and testing the classifier. Training is done until the MSE reaches desired threshold, while testing is done by computing Equation (4) and fetching the argument in $\mathbf{g}$ with the largest value.

The ultimate goal of the linear classifier is to minimize the cost function, which, in this case, is the Minimum-Square-Error (MSE), as discussed in Subsection 2.2. Selecting the appropriate α value is crucial for achieving this objective. Figure 12 illustrates how the MSE behaves for different α values. The analysis reveals that an α value of 0.001 delivers the optimal result. This value showcases quick convergence towards the smallest MSE within a reasonable sample size. In contrast, too large α values don't provide improvement in MSE over time, while too small α values converge too slowly. With further testing and tweaking, an α value of 0.008 provide the best results, and will be used throughout the implementation and testing.

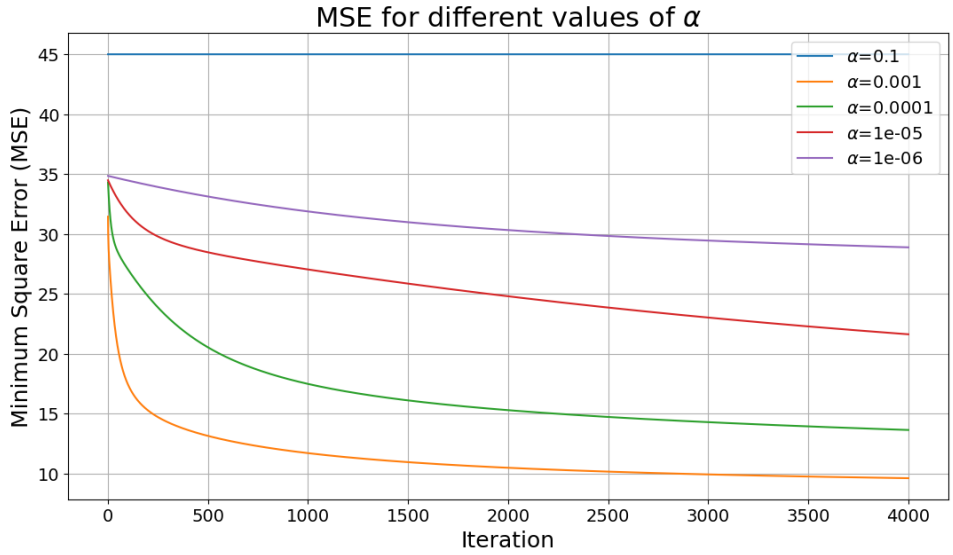


Figure 12: MSE for different values of α. To large α provides no improvement of the MSE, while to small α means the MSE converge to slowly.

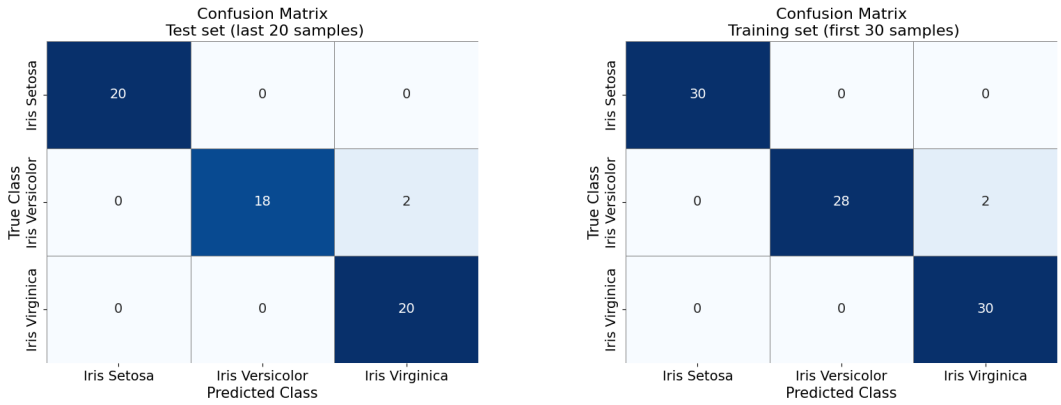
In the implementation, instead of specifying the amount of iterations, an MSE threshold is established. The classifier ends its training as the MSE value becomes less than 8.5 for the first implementation part, see Appendix A. With an α value of 0.008, this MSE threshold is reached in around 4000 iterations. The MSE threshold will vary depending on the nature of the dataset, therefore the numbers of iterations will also vary. The exact MSE value and number of iterations will be stated for each data set and corresponding confusion matrix. The sigmoid function used in the implementation is the logistic sigmoid function [9].

4.2 Results of the Iris task

Training the classifier on the first 30 samples, until the MSE dips below 8.5, produces the following $\mathbf{W}$-matrix:

$$\mathbf{W} = \begin{bmatrix} 0.451 & 1.810 & -2.684 & -1.238 & 0.332 \\ 1.468 & -3.103 & -0.286 & -0.999 & 2.320 \\ -3.318 & -2.877 & 4.877 & 4.472 & -2.461 \end{bmatrix}. \tag{12}$$

Now that the classifier has been trained according to Figure 11a, testing can be done according to Figure 11b. By feeding new inputs into the weight matrix $\mathbf{W}$, we obtain a new $\mathbf{g}$-vector, as explained in Subsection 2.1. To represent the performance of the classifier, confusion matrices are generated as explained in Subsection 2.4. Using the first 30 samples as training data and the last 20 as testing data, results in a classifier that produces the confusion matrices shown in Figure 13a and 13b.



(a) α: 0.008, iterations: 4137, error rate: 3.33%.

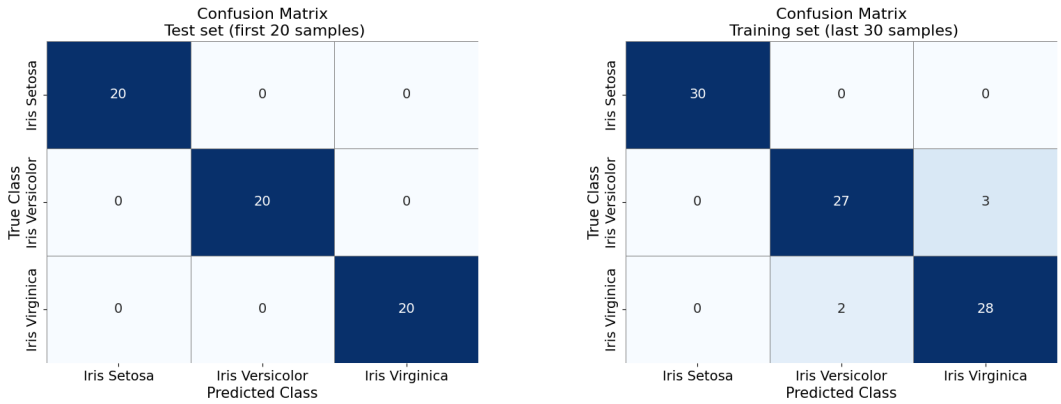
(b) α: 0.008, iterations: 4135, error rate: 2.2%.

Figure 13: Confusion matrices for the test and training datasets. The first set consists of the first 30 samples as the training set, while the second set is the last 20 samples as the testing set. All features are included.

Now we try to switch what part of the data is used for training an which part is used for testing. Using the first 20 samples as testing data and the last 30 samples as training results in the following weight matrix:

$$\mathbf{W} = \begin{bmatrix} 0.4875 & 1.6694 & -2.6193 & -1.2161 & 0.3069 \\ 0.2344 & -2.1930 & 1.4675 & -3.3011 & 3.0312 \\ -2.1173 & -2.8412 & 3.4696 & 4.1809 & -2.3912 \end{bmatrix}. \tag{13}$$

This matrix is similar, but not quite the same as the previous weight matrix. This training/testing split results in the confusion matrices shown in Figure 14a and 14b.



(a) α: 0.008, iterations: 2362, error rate: 0%.

(b) α: 0.008, iterations: 2363, error rate: 5.6%.

Figure 14: Confusion matrices for the test and training datasets. The first set consists of the first 20 samples as the test set, while the second set is the last 30 samples as the training set. All features are included.

The linear separability of the data at hand is very important when making a linear classifier. To visualize the linear separability of each feature, their histograms are shown in Figure 15. The linear classifier will get more use out of the features where each species is more distinct from the other, and the trained weight matrix will probably already have compensated for this. When tasked with removing a feature from the dataset, the Sepal width is the obvious choice due to its significant overlap across the iris species.

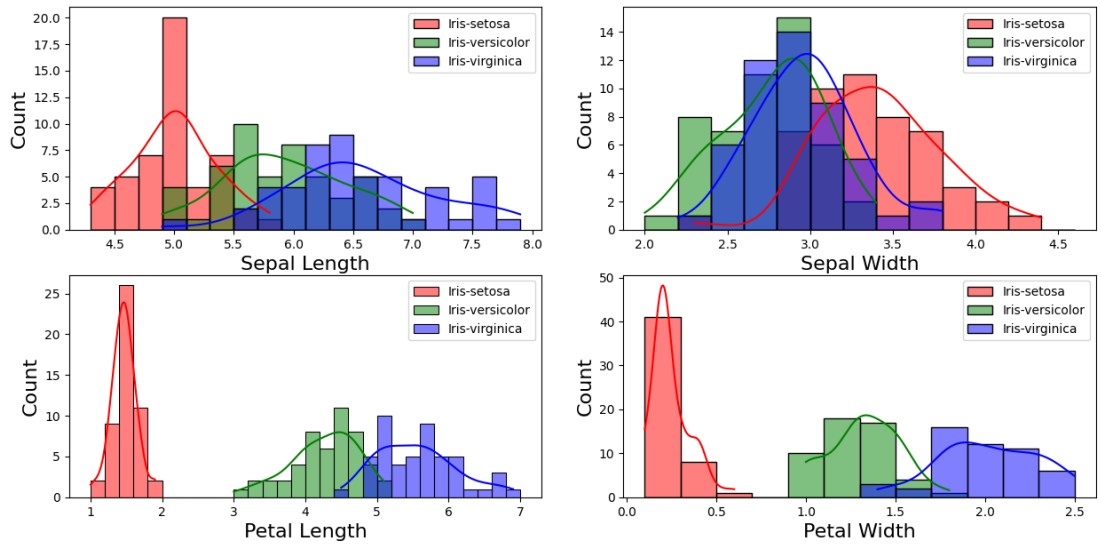
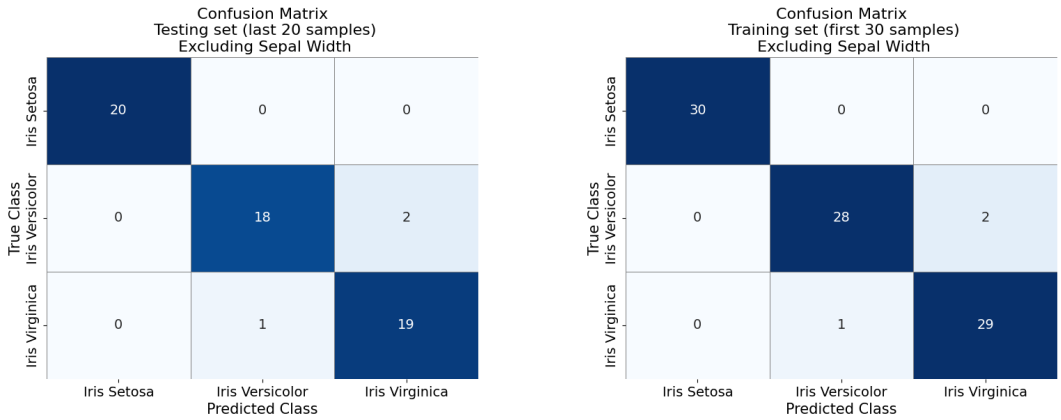


Figure 15: Histogram of the Iris leaves widths and lengths.

To test the classifier, now without taking into account the Sepal width, the first 30 samples are used for training and the last 20 for testing. Change in the dataset causes the MSE value to converge differently to the previous dataset. An MSE threshold of 8.5 will take too many iterations to reach, therefore the threshold is changed to 10 for the new dataset. Training the classifier with an MSE threshold of 10 and an α of 0.008 yields the following weight matrix and confusion matrices:

$$\mathbf{W} = \begin{bmatrix} 1.9079 & -3.5958 & -1.7314 & 0.7589 \\ -0.7893 & 1.7052 & -2.8569 & 0.9191 \\ -4.6443 & 5.4676 & 4.0397 & -4.4609 \end{bmatrix}. \tag{14}$$



(a) α: 0.008, iterations: 6962, error rate: 5%.

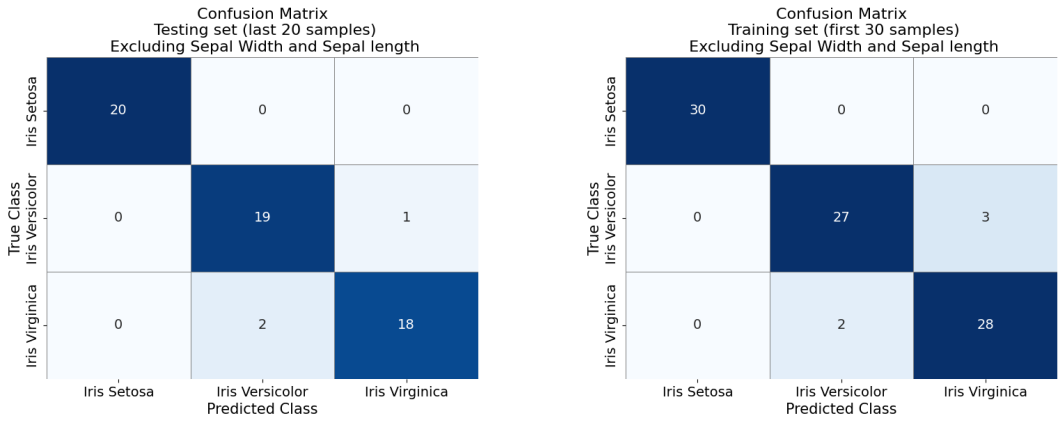
(b) α: 0.008, iterations: 6960, error rate: 3.3%.

Figure 16: Confusion matrices for the test and training datasets without Sepal width.

Based on Figure 16a and 16b, it is clear that there's minimal change in the confusion matrices, with only one additional misclassification observed for the Iris Virginica species, both for the training and testing datasets.

As seen from Figure 15 the next feature to be removed should be Sepal length as it has the most overlap of the remaining features. Again, when changing the dataset, a new MSE threshold is chosen. A new threshold of 11 results in the following weight matrix and confusion matrices.

$$\mathbf{W} = \begin{bmatrix} -1.7554 & -2.6416 & 6.4281 \\ 1.2001 & -2.4183 & -2.2938 \\ 0.5727 & 4.7851 & -10.7392 \end{bmatrix}. \tag{15}$$



(a) α: 0.008, iterations: 4809, error rate: 5%.

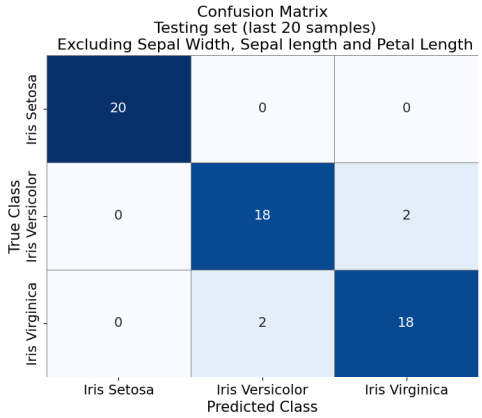
(b) α: 0.008, iterations: 4809, error rate: 5.5%.

Figure 17: Confusion matrices for the test and training sets without Sepal width and Sepal length.

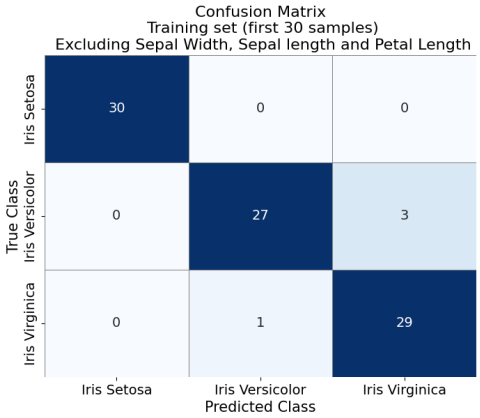
Figures 17b show that when removing both Sepal width and Sepal length, we achieve similar results as to removing only Petal width for the training set. It is worth noting that the misclassifications are not exactly the same for the two.

It is more difficult to determine the next feature to omit as Petal length and Petal width have similar histograms in terms of overlap between the classes. Therefore both options will be investigated. $\mathbf{W}_{PL}$ is the weight matrix for the dataset omitting Petal length while $\mathbf{W}_{PW}$ is without Petal width. These weight matrices as well as confusion matrices of the two options are shown below.

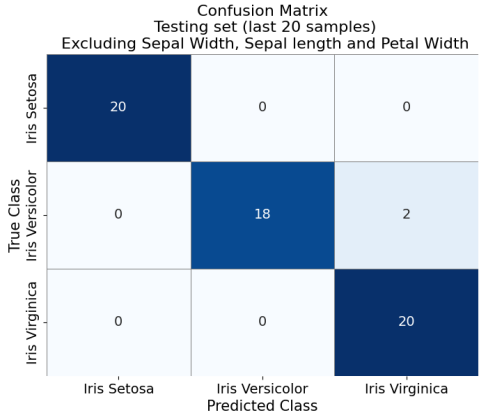
$$\mathbf{W}_{PL} = \begin{bmatrix} -6.3656 & 4.5576 \\ 0.2829 & -1.0245 \\ 5.4704 & -9.0395 \end{bmatrix} \quad (16) \qquad \mathbf{W}_{PW} = \begin{bmatrix} -2.6844 & 7.0496 \\ 0.1833 & -1.3724 \\ 2.2948 & -11.2615 \end{bmatrix} \quad (17)$$



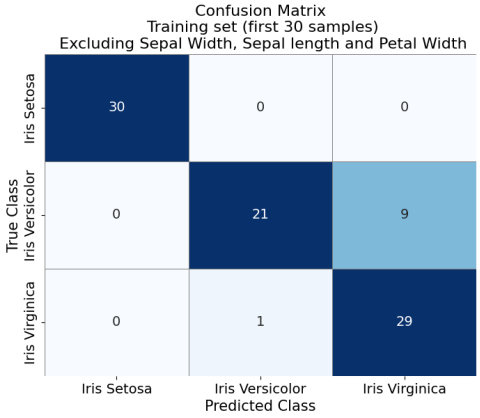
(a) α: 0.008, iterations: 3786, error rate: 6.7%.



(b) α: 0.008, iterations: 3786, error rate: 4.4%.



(c) α: 0.008, iterations: 4747, error rate: 3.3%.



(d) α: 0.008, iterations: 4747, error rate: 11.1%.

Figure 18: Confusion matrices for training and testing datasets when excluding Petal length and Petal width respectively as well as excluding Sepal length and width.

When changing the datasets to omitting Sepal length and Sepal width, the MSE threshold is changed to 12 and 12.5 respectively. Figure 18 illustrates that removing Petal length leads to a decrease in classification performance of the testing dataset, while enhancing performance for the training dataset, compared to when Petal width is removed.

To compare the performance of the various datasets, it would be beneficial to calculate and analyze the total error using each dataset. This is shown in Table 1.

Table 1: Total error rate for each dataset. SW: Sepal Width, SL: Sepal Length, PW: Petal width, PL: Petal Length. (30/20): The first 30 samples used for training and the 20 last used for testing. (20/30): The first 20 samples used for testing and the 30 last used for training.

Dataset	**Total error**
SW, SL, PW, PL (30/20)	2.7%
SW, SL, PW, PL (20/30)	3.3%
SL, PW, PL	4%
PW, PL	4.7%
PW	5.3%
PL	8%

Overall, the removal of features has a negative effect on the performance of the classifier. Removal of the least linearly separable features has a smaller negative effect than removal of the more easily linearly separable ones. A trend to note is that as one removes more and more features, the MSE threshold has to be adjusted to a bigger and bigger value to limit the number of iterations needed. In other words, allow for more and more error. This is reflected in Table 1.

Every time the model makes an error, it is between the Iris-Versicolor and Iris-Virginica. By examining the histograms in Figure 15, this is expected because of the minimal overlap between the Iris-Setosa and the other two species. An interesting observation is that the test sets often outperform the training sets in terms of error rate. This would suggest that there is no overfitting as described in Subsection 2.5.

5 IMPLEMENTATION AND RESULTS - VOWELS TASK

5.1 Implementation of the Vowels task

As mentioned in Subsection 5.1, the purpose of the vowels task is classifying a set of 12 different vowels. This will be done with a Plug-in MAP classifier approach described in Subsection 2.3, with both a single Gaussian and a GMM for each class. This subsection will explain the implementation of the classifier. The classifier is implemented using Python, and the full implementation is in Appendix A. The dataset used can also be accessed in Appendix A.

The vowels dataset consists of 12 different vowels, with 139 data samples for each class. The first 70 data samples for each class is used for training, while the last 69 is used for testing. For the single Gaussian case, training is done by finding the mean value vector and covariance matrix for each class. Since the same amount of test data is used for each class, the A Priori probabilities $P(\omega_i)$ will all be the same and equal to $1/12$. Training in the GMM case is done by fitting the amount of Gaussians used per class according to the training data at hand. The initial weights for each Gaussian are set to fixed values, ensuring reproducability in between runs of the code.

Testing the classifier is done according to the flowchart in Figure 19.

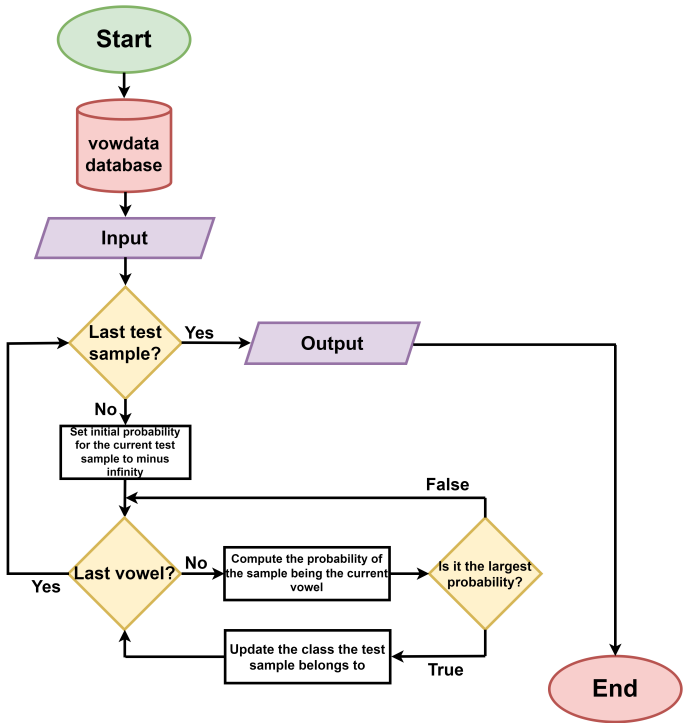


Figure 19: Code flowchart of the testing of the vowels classifier.

Testing the vowels classifier is done by iterating over all the data used for testing. For each test sample, the classifier computes the value provided by the Gaussian of each vowel iteratively. Meanwhile the class which provides the largest value according to the decision rule in Equation (10) gets updated. Thus, this is the vowel the test sample is classified as.

5.2 Results of the Vowels task

Figure 20 shows the confusion matrix using a single Gaussian and full covariance matrix for each class.

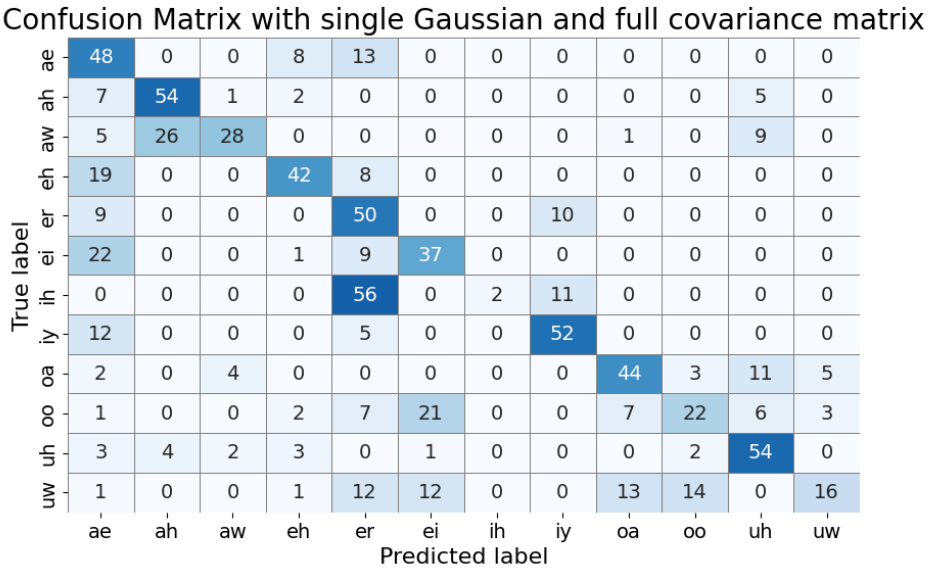


Figure 20: Confusion matrix using a single Gaussian as well as a full covariance matrix for each class.

The classifier has an error rate of 45.8%, meaning a little less than half of the classified data is classified incorrectly. Therefore the classifier cannot be regarded as trustworthy. Figure 21 shows the confusion matrix still using a single Gaussian for each class, but with a diagonal covariance matrix.

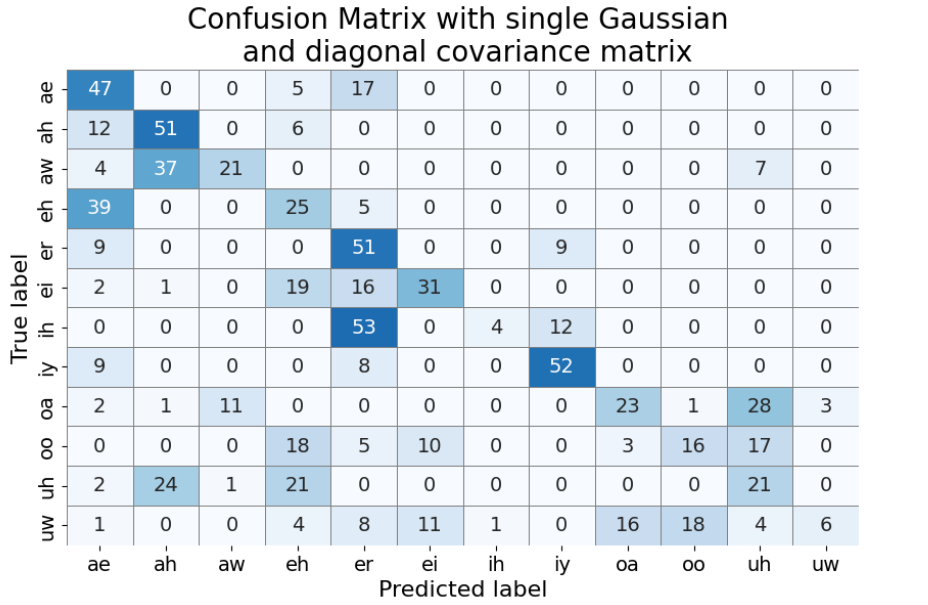


Figure 21: Confusion matrix using a single Gaussian as well as a diagonal covariance matrix for each class.

Switching to a diagonal covariance matrix shows a reduction in classification performance, where the classifier now has an error rate of 58%. The increase in error rate indicate a covariance between the different vowels, which for this case should be taken into consideration during classification. In both cases the ”ih” vowel has the most misclassifications, where almost all test samples are misclassified as the ”er” vowel. Analyzing the histogram of the training data (which the classifier

is trained on) for each formant in the ”ih” and ”er” vowels reveals that they exhibit overlapping features, making it harder to distinguish them from one another. The histograms are shown in Figure 22.

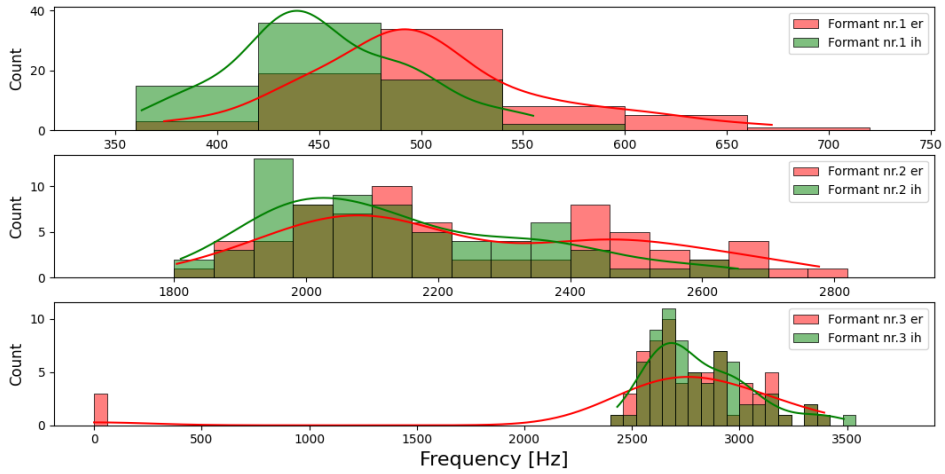


Figure 22: Histogram of the three formants for the ”ih” and ”er” vowels. For formant nr.3, some training samples belonging to the ”er” vowel has a formant at frequency 0 Hz since some training samples do not include a third formant.

For the GMM case, testing is first done using a mixture of two Gaussians and a diagonal covariance matrix for each class. The confusion matrix is shown in Figure 23.

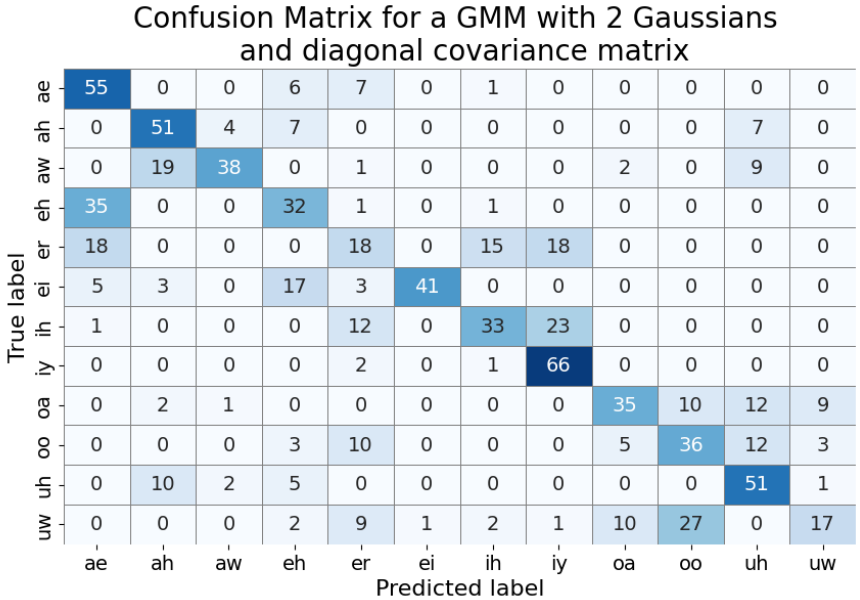


Figure 23: Confusion matrix using a GMM with two Gaussians, as well as a diagonal covariance matrix for each class.

The classifier now has an error rate of 42.9%, which is a decrease of 2.9% compared to Figure 20. The classifier has a more even distribution of correct classifications for the different classes, however some classes show worsened performance. The ”er” and ”ih” vowels now has a more even distribution of classifications between them, meaning less ”ih” vowels are interpreted as ”er”, while more ”er” vowels are interpreted as ”ih”.

Figure 24 shows the confusion matrix using a mixture of three Gaussians in the GMM, still with a diagonal covariance matrix.

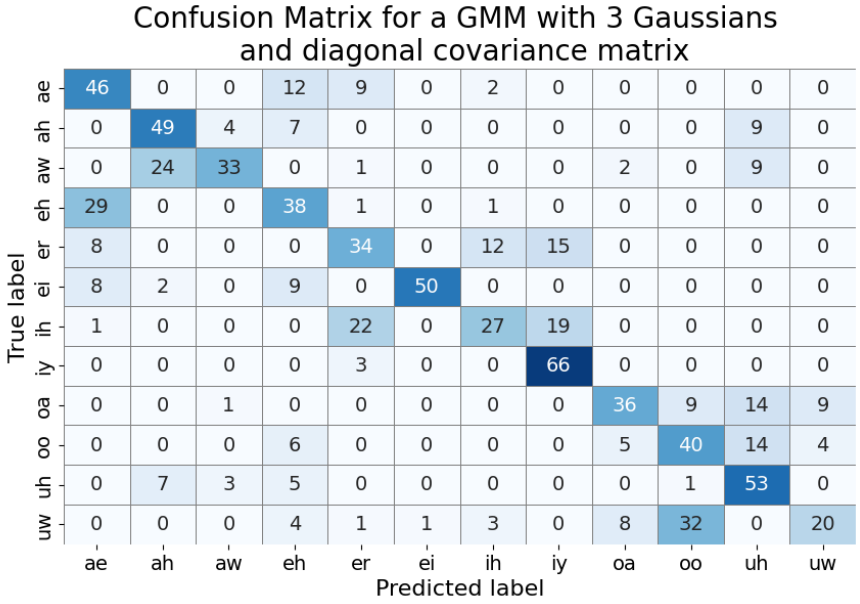


Figure 24: Confusion matrix using a GMM with three Gaussians, as well as a diagonal covariance matrix for each class.

The classifier has an error rate of 40.6%, which is the best error rate achieved for all implementations. However, a classifier with an error rate this high cannot be regarded as a good classifier, and could suggest we are underfitting or overfitting our data.

To investigate whether the classifier is subject to overfitting, the classifier is tested using the training data. This is done using a GMM with three Gaussians and a diagonal covariance matrix, and the confusion matrix is shown in Figure 25.

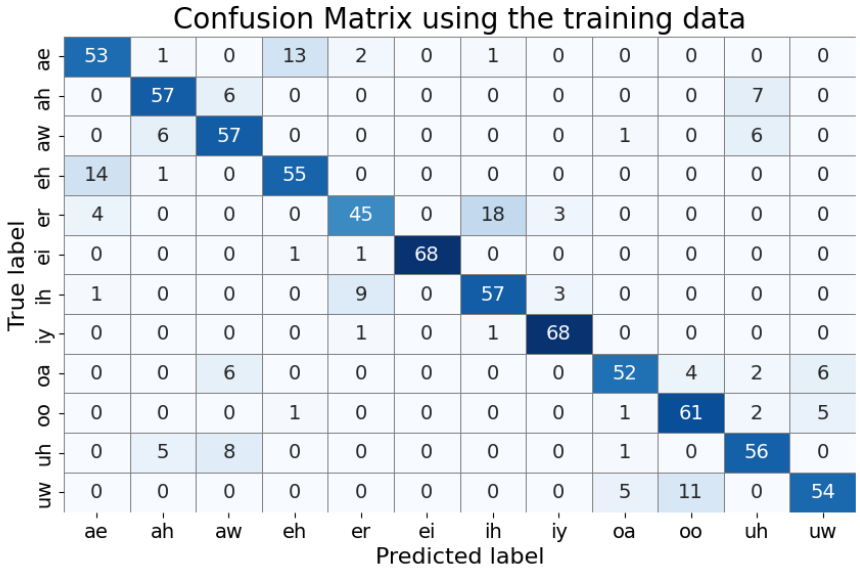


Figure 25: Confusion matrix using the training data.

Using the training data the classifier has an error rate of 18.6%, which is a significant improvement

compared to using the test data. This indicates our classifier struggles with generalization, and is subject to overfitting.

The performance of the different classifiers are summarized in Table 2.

Table 2: Error rates for the different classifiers.

Classifier	**Error rate**
Single Gaussian, full covariance matrix	45.8%
Single Gaussian, diagonal covariance matrix	58%
GMM, 2 Gaussians	42.9%
GMM, 3 Gaussians	40.6%

6 CONCLUSION

In conclusion, two different types of classifiers have been implemented using Python, shown in Appendix A. The Iris task produces results as low as 2.7% total error rate using all features and 5.3% using only one feature (Petal width). These error rates make a quite reliable classifier.

The Vowels task produces error rates in the range 40.6-58%, and can therefore not be regarded as a reliable classifier. General results shows that a GMM yields better results than single Gaussians, however inspections show that the classifier may be subject to overfitting.

This project has in large parts thought us the idea and essence behind how classifiers work, and this is reflected as to how the tasks have been implemented in Python. Is is worth mentioning that there exists several useful libraries and techniques for making machine learning models, which may result in better performances than those achived in this project.

BIBLIOGRAPHY

[1] Magne H. Johnsen. *Project descriptions and tasks in classification*. https://ntnu.blackboard.com/ultra/courses/_43833_1/cl/outline. 2018.

[2] Wikipedia. *ChatPGT*. URL: https://en.wikipedia.org/wiki/ChatGPT (visited on 29th Apr. 2024).

[3] ScienceDirect. *Linear Classifier*. URL: https://www.sciencedirect.com/topics/computer-science/linear-classifier (visited on 28th Apr. 2024).

[4] Magne H. Johnsen. *Classification*. https://ntnu.blackboard.com/ultra/courses/_43833_1/cl/outline. 2017.

[5] DeepAI. *Accuracy (error rate)*. URL: https://deepai.org/machine-learning-glossary-and-terms/accuracy-error-rate (visited on 22nd Apr. 2024).

[6] Google. *Generalization*. URL: https://developers.google.com/machine-learning/crash-course/generalization/video-lecture (visited on 26th Apr. 2024).

[7] GeeksforGeeks. *Underfitting and Overfitting*. URL: https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/ (visited on 26th Apr. 2024).

[8] MathWorks. *What is Overfitting?* URL: https://se.mathworks.com/discovery/overfitting.html (visited on 28th Apr. 2024).

[9] Wikipedia. *WSigmoid function*. URL: https://en.wikipedia.org/wiki/Sigmoid_function (visited on 30th Apr. 2024).

APPENDIX

A Github

URL: https://github.com/KristianHaaland/Estimering-Prosjekt