TTT4275
EDC

Markeng, H., Jermstad, S. M.

Classification 2 - 03

April 29, 2021

Project Classification

Summary

This report describes the design and results of a linear and plug-in-MAP classifier, as part of our course TTT4275.

The linear classifier is designed to classify three different species of the Iris genus in task 3.1. The training set of the measured data is used to train a W matrix, such that the data is classified by multiplying itself with the W matrix. As the data is not completely linearly separable between the Iris Versicolor and Iris Virginica, there were some small error rates of these two classes. Iris Setosa, on the other hand, is completely separable from the other two classes and had a error rate of 0. When we switched the samples the test set consisted of, from the 20 last to the 20 first samples, the Iris Setosa still had an error rate of 0. The other two classes had varying error rates after switching, 0.050 for test and 0.022 for training, then 0.017 for test and 0.044 for training. This is because of individual samples being difficult to classify. When we classified based on only one feature, the classification only gave an error rate of 0.083 and 0.189 for the test and training set. Therefore, the classes are almost linearly separable.

To separate the vowels in task 3.2, a plug-in-MAP classifier have been used. The plug-in-MAP classifier uses a Gaussian distribution to model the data distribution, and then calculates the probability for the data. We have tested using both a single Gaussian model, with both a full and a diagonal covariance matrix, and a Gaussian Mixture Model of order $M = 2$ and $M = 3$. We found that using the full covariance matix is better than using the diagonal, with an error rate of respectively 0.257 and 0.395 for the single Gaussian model. We also observe that the GMM is a better model for our data, with an appr213rpiate order M. All these classifiers has an error rate above 25 %.

Contents

1 Introduction

Machine learning and classification is becoming increasingly important in an increasingly automated society. Speech recognition for handsfree communication, finding pictures of your friends on your phone by searching their name, and regocnizing red flags for diagnosing a disease are examples of tools made possible with machine learning. These are concepts that are not only improving the lives of consumers, but making oil industries, retail, transportation industries, government, and health care more efficient.

In machine learning, classification puts a label on different input data. This makes it possible to predict outcomes of future data, and finding trends.

The report is divided into four sections: Theory, Tasks, Results and implementation and Conclusion. These sections will describe both the linear and the plug-in-MAP classifier. Section 2 describes the theoretical background and equations needed to design the classifiers. Information and requirements about the two tasks we did are described in section 3, as well as the task we didn't choose. The first task, in section 3.1, is about designing and analyzing a linear classifier for classifying Iris species. The second and chosen task in section 3.2 is about designing and analyzing a plug-in-MAP classifier for classifying different vowels. Design, implementation, and results of the classifiers applied to training and test data are found in section 4. In section 5, one can see the final conclusion of the design and results of the classifiers.

2 Theory

This chapter explains the theory behind classification, by use of a linear classifier for the Iris task and plug-in-MAP classifier for the task of classification of pronounced vowels. Throughout this section the TTT4275 compendium on classification is used as source [1].

2.1 Training the classifier

To train the classifier we need N measurements X_N that have been classified. This is called the training set. The larger the N, the better the classifier performance for test data. X_N is a $C \times D$ matrix. C is the number of classes, and D is the total number of features that each measurement x_k consists of, where $k = 1, ..., N$.

The difference in performance between the classifier used on the training set and test set should not be too large, as that indicates that the classifier doesn't generalize.

2.2 Linear classifier

This section encompasses how to implement and train the linear classifier. The linear classifier is one of the simpler classifiers, and will often give worse result than a non-linear classifier. When it gives satisfactory results, though, it is often applied because of its incomplexity. For a linear classifier, a measurement x belongs in a class ω_j when the following equation (1) is true,

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

that is when the corresponding discriminant function $g_j(x)$ is the largest value in the vector g_i, with index j.

The array g and vector g_k is described by equation (2):

$$\begin{aligned} g &= Wx \\ g_k &= Wx_k \end{aligned} \tag{2}$$

where x_k is a $1 \times (D+1)$ vector and $W = [Ww_0]$ is a $C \times (D+1)$ matrix, and w_0 is the offset. The matrix $x = [x^T 1]^T$ has dimension $N \times (D+1)$, and the vector g_k has dimension C. Each g_k consists of C numbers g_{ik}. g_{ik} is a function of x_{ik}, and is shown in equation (3):

$$g_{ik} = sigmoid(x_{ik}) = \frac{1}{1+e^{-z_{ik}}} \quad i = 1, ..., C \tag{3}$$

Where $z_k = Wx_k$. A sigmoid function is an approximation of a Heaviside function, but smooth and differentiable. We have one sigmoid function for each number x_{ik}, that is, each feature that x_k measures.

Each measurement x_k corresponds to a target vector t_k with dimension $1 \times C$. The target vectors consists of C binary values, where only element $t_k(j)$ is 1. Then, $x_k \in \omega_j$. Each calculated g_{ik} is compared to each number t_{ik} in the corresponding t_k.

To train the linear classifier, we can use minimum square error (MSE). We can reduce the MSE by gradient descent: taking the gradient of the MSE, and updating the W matrix with the values that reduces the MSE. That is, updating with the values that move W down the gradient. We have an equation for the gradient of MSE with respect to W in equation (4):

$$\nabla_W MSE = \sum_{k=1}^{N} [(g_k - t_k) \circ g_k \circ (1 - g_k)] x_k^T \tag{4}$$

In the equation (4), the calculated g_k is compared to each t_k. Then, $(g_k - t_k)$ is elementwise multiplied with g_k, which is again elementwise multiplied with $(1 - g_k)$. Updating the matrix W is done with equation (5).

$$W(m) = W(m-1) - \alpha \nabla_W MSE \tag{5}$$

Here, m is the iteration number and α is a "step" number, the factor of each step closer to the minimum MSE. Making too small can demand several iterations before arriving at the minimum MSE, but a too large value can lead to the step "jump over" the minimum.

2.3 Plug-in-MAP classifier

This section is about the implementation of the plug-in-MAP (Maximum a Posteriori) classifier, and the Gaussian distribution used to model the data. The plug-in-MAP classifier uses the Bayes Decision Rule (BDR) to classify the set of training data. The BDR can be seen in equation (6).

$$x \in \omega_j \leftrightarrow p(x|\omega_j)P(\omega_j) = \max_i p(x|\omega_i)P(\omega_i) \tag{6}$$

That is, one calculates the density for each class for the features. If the calculated density that gives the largest probability for the sample x_k has index j, x_k belongs in class j.

If one has the density that describes the class distribution of the features perfectly, the BDR/MAP estimator is the optimal classifier. In the real world, the distribution does not fit perfectly. Therefore, one can approximate the class distribution with a chosen distribution with calculated estimated parameters.

In this project, we will use the Gaussian density to model the class distribution. From the training set, the mean and covariance must be estimated for a Gaussian distribution. If one estimates the distribution with a mix of M Gaussians, with the Gaussian Mixture Model (GMM), one must also estimate the weights for each Gaussian.

For a single Gaussian density we have the equation (7):

$$p(x|\omega_i) = N(\mu_i, \Sigma_i) = \frac{1}{\sqrt{(2\pi)^D|\Sigma_i|}} \exp\left(-\frac{1}{2}(x_k - \mu_i)^T \Sigma_i^{-1}(x_k - \mu_i)\right) \quad i = 1, ..., C \tag{7}$$

Here, x_k is the $1 \times D$ sample to be classified and μ_i is a $1 \times D$ vector for a single class. Σ_i is the covariance matrix for a single class with dimension $D \times D$. The covariance matrix models the relationship between the features of a class.

For the GMM one must also estimate and account for the weights c_{ik} of each Gaussian in a class ω_i. Therefore, one must sum over the number of Gaussians. The total sum of the weights must be one. The GMM density in equation (8) is:

$$\begin{aligned} p(x|\omega_i) &= \sum_{k=1}^{M_i} c_{ik} N(\mu_{ik}, \Sigma_{ik}) \\ &= \sum_{k=1}^{M_i} \frac{c_{ik}}{\sqrt{(2\pi)^D|\Sigma_{ik}|}} \exp\left(-\frac{1}{2}(x_k - \mu_{ik})^T \Sigma_i^{-1}(x_k - \mu_{ik})\right) \quad i = 1, ..., C \end{aligned} \tag{8}$$

By equation (6) we also need to know or estimate the prior probabilities $P(\omega_i), \quad i = 1, ..., C$. For simplicity we can assume that the probability for a class $P(\omega_i) = \frac{1}{C}$ where C is the number of classes.

To estimate the parameters $\Lambda_i = \{\mu_i, \Sigma_i\}, \quad i = 1, ..., C$ of the Gaussian distribution, we can apply a training based on the Maximum Likelihood (ML). This means that we find the parameters that finds the Gaussian under which the data from a training subset is more probable. The training subset $X_{N_i} = \{x_{i1}, ..., x_{iN_i}\}$ belongs to a single class ω_i.

Maximizing the likelihood for the subset is the same as taking the gradient of the logarithm of the likelihood LL, and setting it to zero. This is the same as curve fitting. Therefore, one can find the parameters that fulfil the following equation (9),

$$\nabla_{\Lambda_i} LL[X_{N_i,\Lambda_i}] = \sum_{k=1}^{N_i} \nabla_{\Lambda_i} \log p(x_{ik}|\Lambda_i) = 0 \tag{9}$$

for the different samples x_{ik} of the class ω_i.

When calculating the parameters for a single Gaussian, we can plug equation (7) into equation (9). The resulting estimators for the mean vector $\hat{\mu_i}$ and covariance matrix $\hat{\Sigma}_i$ for a single Gaussian corresponding to a class ω_i are formulated in equation (10) and (11), respectively:

$$\hat{\mu_i} = \frac{1}{N_i}\sum_{k=1}^{N_i} x_{ik} \tag{10}$$

$$\hat{\Sigma}_i = \frac{1}{N_i}\sum_{k=1}^{N_i} (x_{ik} - \hat{\mu_i})(x_{ik} - \hat{\mu_i})^T \tag{11}$$

For the case where we have M Gaussian distributions, we can find the parameters $\Lambda_i = \{\mu_i, \Sigma_i, c_i\}, \quad i = 1, ..., C$ suboptimally by use of the Expectation Maximation (EM) algorithm. The algorithm iteratively finds better estimates for the parameters from chosen initial values, up to a chosen number of iterations. From M Gaussian distributions, we get M weights c_i, $M \times D$ means and $M \times (D \times D)$ diagonal covariance matrices for each class.

3 Tasks

The following section describes the Iris task and the chosen classification of pronounced vowels task. Additionally, we will describe the classification of handwritten numbers 0-9 task, that we didn't choose.

3.1 The Iris task

The Iris genus has multiple different species. Three of them, Iris Setosa, Iris Versicolor, and Iris Virginica, can be differentiated depending on the petal length and width, as well as the sepal length and width, i.e. the features.

The three species, or classes, are almost linearly separable, and therefore, a linear classifier is enough for sufficient classification.

There are two parts to this task. In the first part, a linear classifier is to be designed, trained and evaluated. The training stops when the training converge, and after this, a confusion matrix and error rate must be found. Additionally, we must analyze if we get better results with the first 30 samples for training and the last 20 for testing, or the last 30 for training and first 20 for testing. In the second part, the importance of the different features related to the linear separability is analyzed, where we remove one by one feature and see how the confusion matrix and error rate changes.

3.2 The classification of pronounced vowels task

The classification of pronounced vowels task involves classifying 12 different vowel sounds as shown in equation (12):

$$\text{vowels} = \{\text{ae, ah, aw, eh, ei, er, ih, iy, oa, oo, uh, uw}\} \tag{12}$$

The vowels are surrounded by consonants, uttered in so-called CVCs. Each of these vowels have three features: the first three peaks for different frequencies in the frequency spectrum, also called formants. Therefore the vowels can be classified depending on where the frequencies lie. However, the three frequencies of a single class can also vary depending on the source of the sound, whether from a girl, boy, woman or man. As the frequency peaks can overlap between classes, the problem is not linearly separable.

We must therefore design a plug-in-MAP classifier. This will be done in two parts: firstly with a single Gaussian distribution, and finally for a mix of 2 and 3 Gaussians (GMM). To do this, we must find an estimate for the sample mean and sample covariance matrix (both full and diagonal) for the single Gaussian, and for the GMM, also the weights for the Gaussians. After this, the confusion matrix and error rate must be found for the test and training set.

In the end, we will compare the performances of the four model types for the classifiers: single Gaussian with full covariance matrix and diagonal covariance matrix, and a mix with both two and three Gaussians with diagonal covariance matrix.

We chose this task because we found it interesting how it is both applicable to and useful in real life. We wanted to test the classifier ourselves with our own voice.

3.3 The classification of handwritten numbers 0-9 task

For the classification of handwritten numbers task, one must design a classifier that recognize handwritten numbers between zero and nine. A large amount of training samples show pictures of the numbers with 28×28 pixels.

This task is divided into two parts. In the first part, the designed classifier should be based on the nearest neighbor (NN) method with Euclidian distance. Here, the whole training set will be used as templates for the classes. In the second part, clustering is used to make a smaller set of templates. Both the NN and K nearest neighbors (KNN) classifiers must be designed. Lastly, one should find the confusion matrices and error rates for the NN classifier in first part, and for both the NN and KNN classifier in the second part.

4 Implementation and results

Now we will look at the implementation, results and discussion of the results of the two tasks from section 3. The tasks are implemented using Python. The Iris task has been implemented using the linear classifier described in section 2.2, and the vocal recognition has been implemented using the plug-in-MAP classifier in section 2.3. The implementation, results and discussion for the Iris task and classification of pronounced vowels task are described in section 4.1 and section 4.2, respectively.

4.1 The Iris task

This subsection is about the implementation, results and discussion of the Iris task described in section 3.1, using the theory explained in section 2.2. See appendix 5.1 for the code used for the implementation.

We have $N = 50$ samples from each of the $C = 3$ classes. For simplicity, we will from now on call Iris Setosa class 1, Iris Versicolor is class 2 and Iris Virginica is class 3. The first 30 samples from each class are used for training and the last 20 for testing. We also have $D = 4$ features of each sample.

The g_k array is found using equation (2), and consist of 3 g_{ik}. Each g_{ik} is calculated using equation (3). To train the 3×5 W matrix used to calculate the g array, we start by setting all values in W to zero. All the samples x used for for training is placed in a 1D array, sorted after class. First the 30 from class 1, then the 30 from class 2 and last the 30 from class 3, a total of 90 training samples. Each sample x_k is a 1×5 vector, with 1 as the last element.

We also produce a T_n vector consisting of a total of 90 t_k. The three sets of 30 t_k corresponds to the three classes of the samples in the training array, as shown in equation (13).

$$T_n = [[1, 0, 0]] \cdot 30 + [[0, 1, 0]] \cdot 30 + [[0, 0, 1]] \cdot 30 \tag{13}$$

The first 30 values of T_n, $[1, 0, 0]$, correspond to class 1, the next 30 values, $[0, 1, 0]$, correspond to class 2, and the last 30 values, $[0, 0, 1]$, correspond to class 3.

To train the matrix W, we use the equations (4) og (5). To find $\nabla_W MSE$ we iterate through the entire training set, and from each sample x_k calculate a value d_MSE, as shown in figure 1. d_MSE is the elementwise multiplication between $g_k - t_k$, g_k, and $1 - g_k$. After we have summed all the d_MSE, we update W using equation (5). We do this until the change of one update is less than $\alpha \cdot 0.4$ for each of the values in the matrix.

In other words: when all the values in $\nabla_W MSE$ is below 0.4. We observe that W converges when α is below about 0.005, so to be on the safe side we choose $\alpha = 0.001$. This did not make a noticeable difference in the processing time.

```
def train(vec, alpha):
    """
    Training the W matrix.
    :param vec: the training data set.
    :param alpha: the step size
    """
    global W
    global training_threshold
    i = 0
    while True:
        W_last = W
        n_mse = 0
        for j in  range( len(vec)):
            xk = np.matrix(vec[j])
            tk = np.matrix(Tn[j])
            zk = np.dot(W, xk.T).T
            gk = sigmoid(zk)
            mid_term = np.multiply((gk-tk),gk)
            d_mse = np.multiply(mid_term,(1-gk))
            n_mse += np.dot(d_mse.T, xk)

        W = W_last - alpha*n_mse

        if np. all( abs(n_mse) <= training_threshold):
            print('Number of iterations:', i+1)
            print('nabla-nmse:',n_mse)
            print('W:', W)
            break
        i += 1
```

Figure 1: Function used to train the W matrix. It takes in a array *vec* with the training data, and updates the global matrix W until it converges.

With these limits, the training is done after 2082 iterations of the entire training set. W becomes as shown in figure 2.

```
[[ 0.30086157  1.12406013 -1.69223773 -0.77171527  0.20744435]
 [ 0.58260297 -1.29875635  0.11056942 -0.6411628   0.20685538]
 [-1.39101834 -1.08924302  2.0126079   1.52611551 -0.59139671]]
```

Figure 2: W with $\alpha = 0.001$.

With this W, we calculate a g_k for both the training and the test set. From equation (1), we find the index that gives the maximum value in g_k. This index corresponds to the class. The confusion matrix for the test set is shown in table 1 and for the training set in table 2. Here we have an error rate of 0.050 for the test set, and 0.022 for the training set.

Table 1: Confusion matrix of the test data, when the first 30 samples are used for training and the last 20 for testing.

Test set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	20	0	0
Versicolour	0	17	3
Virginica	0	0	20

Table 2: Confusion matrix of the training data, when the first 30 samples are used for training and the last 20 for testing.

Training set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	30	0	0
Versicolour	0	28	2
Virginica	0	0	30

If we change the samples used for training and testing, so that the 20 first samples are used for testing, we get the confusion matrices as shown in table 3 and 4 for the test and training data. Other than which samples are used for testing and training, the conditions are the same. We use 2275 iterations before $\nabla_W MSE$ becomes below the limit. This gives us an error rate of 0.017 for the test set, and 0.044 for the training set.

Table 3: Confusion matrix of the test data, when the last 30 samples are used for training and the first 20 for testing.

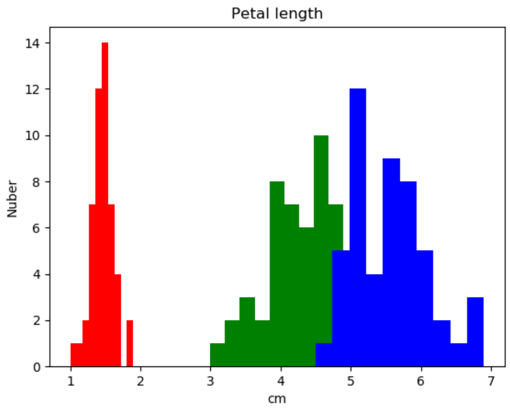
Test set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	20	0	0
Versicolour	0	19	1
Virginica	0	0	20

Table 4: Confusion matrix of the training data, when the last 30 samples are used for training and the first 20 for testing.

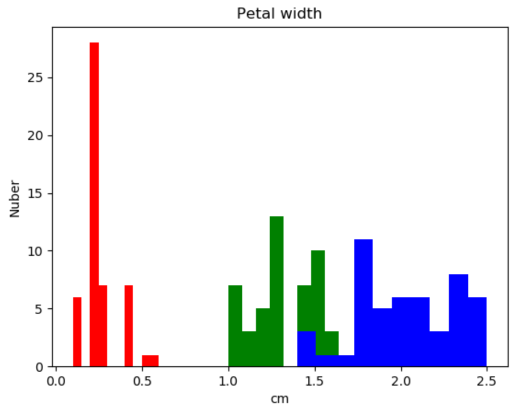
Training set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	30	0	0
Versicolour	0	26	4
Virginica	0	0	30

For the case where we use the last 20 samples for testing, we have a slightly better error rate for the training set than for the test set. When we have the first 20 samples for testing, the training set gives a slightly worse error rate. We can assume that this is because of some samples that are not linearly separable, as the total number of errors from both of these two cases is both five. From this we can say that there is almost no difference in performance whether we use the first or last 20 samples for the test set. However, the error rate for the Iris Setosa is 0 for both cases.

To test the linear separability of the system, we test the classifier while removing features one by one. From now on we use the first 30 samples of each class for training and the last 20 for testing. When plotting a histogram of each feature and class, shown i figure 3 for petal measurements and 4 for sepal measurements, we see that almost all features from class 2 and 3 are somewhat overlapping.

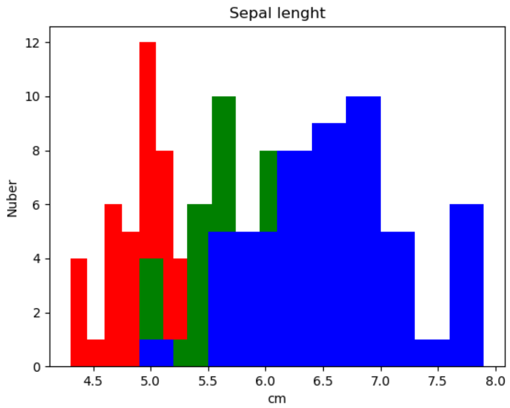


(a) Histogram of the petal length of each class. The red one is class 1, the green is class 2 and the blue is class 3.

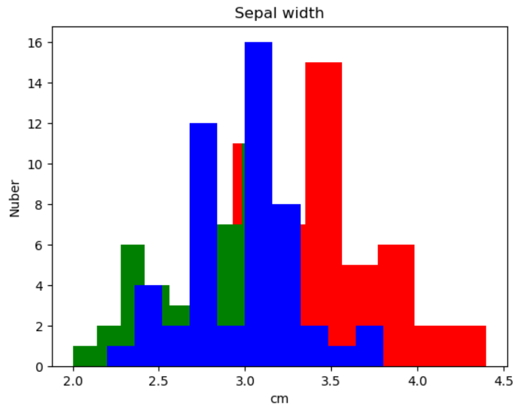


(b) Histogram of the petal width of each class. The red one is class 1, the green is class 2 and the blue is class 3.

Figure 3: Histograms for petal length in a) and width in b), for all the three classes.



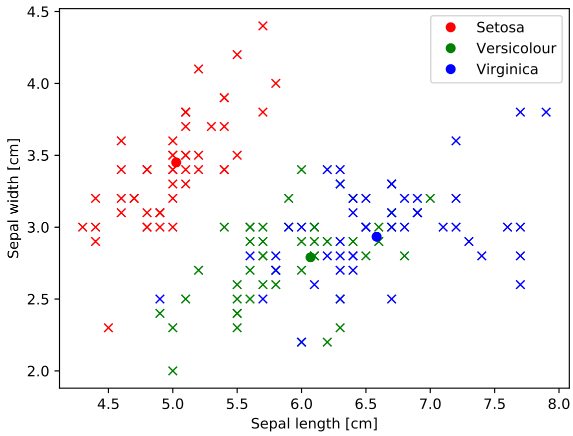
(a) Histogram of the sepal length of each class. The red one is class 1, the green is class 2 and the blue is class 3.



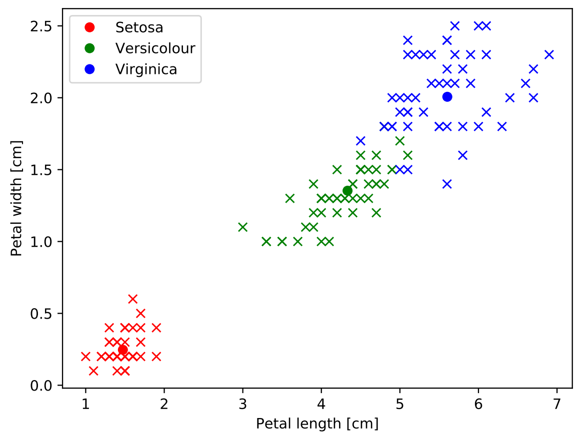
(b) Histogram of the sepal width of each class. The red one is class 1, the green is class 2 and the blue is class 3.

Figure 4: Histograms for sepal length in a) and width in b), for all the three classes.

We can also observe this in figure 5, where sepal length and width are plotted against each other in a scatter plot, and same for petal length and width.



(a) Scatter plot of the sepal length and width of each class. The red one is class 1, the green is class 2 and the blue is class 3.



(b) Scatter plot of the petal length and width of each class. The red one is class 1, the green is class 2 and the blue is class 3.

Figure 5: The scatter plots of the petal and sepal lengths and widths in a) and b) of the three classes show the samples' linear separability.

We start by removing the most overlapping feature. From the histograms we can see that this is sepal width. We train the classifier again, and get the confusion matrix shown in table 5 and 6 for the testing and training data, respectively. The error rate is here calculated to 0.050 and 0.067, whith is a bit worse than before.

Table 5: Confusion matrix of the test data, with sepal width removed.

Test set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	20	0	0
Versicolour	0	17	3
Virginica	0	0	20

Table 6: Confusion matrix of the training data, with sepal width removed.

Training set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	30	0	0
Versicolour	0	24	6
Virginica	0	0	30

Furthermore we remove sepal length. This gives us confusion matrices as shown in table 5 and 8 for the testing and training data, respectively. This gives us error rates of 0.100 and 0.133.

Table 7: Confusion matrix of the test data, with sepal width and length removed.

Test set confusion matrix			
Classified: → True/label: ↓	Setosa	Versicolour	Virginica
Setosa	20	0	0
Versicolour	0	16	4
Virginica	0	2	18

Table 8: Confusion matrix of the training data, with sepal width and length removed.

Training set confusion matrix			
Classified: → True/label: ↓	Setosa	Versicolour	Virginica
Setosa	30	0	0
Versicolour	0	20	10
Virginica	0	2	28

Lastly we remove petal length, and get the confusion matrices shown in table 9 and 10. This gives error rates of 0.083 for the test set and 0.189 for the training set.

Table 9: Confusion matrix of the test data, with sepal width and length, and petal length removed.

Test set confusion matrix			
Classified: → True/label: ↓	Setosa	Versicolour	Virginica
Setosa	20	0	0
Versicolour	0	15	5
Virginica	0	0	20

Table 10: Confusion matrix of the training data, with sepal width and length, and petal length removed.

Training set confusion matrix			
Classified: $\rightarrow$ True/label: $\downarrow$	Setosa	Versicolour	Virginica
Setosa	30	0	0
Versicolour	0	13	17
Virginica	0	0	30

When we compare all the confusion matrices we see that when we remove features, the error rates increases. The more features we remove, the worse results. This can be explained from that we when we remove features, we get less data to work with, both when training and classifying. Even though some samples of the classes are overlapping, there are many that don't, and they contribute to better classification. Although this is the case, we see that with only one feature, the classifier is still able to get most of the samples in the right class. The classes are therefore almost linearly separable. We also see that we get all the samples from class 1, Iris setosa, correct, with an error rate of 0. From this, and the histograms, it is a fair assumption to say that Iris Setosa is linearly separable from the other two classes.

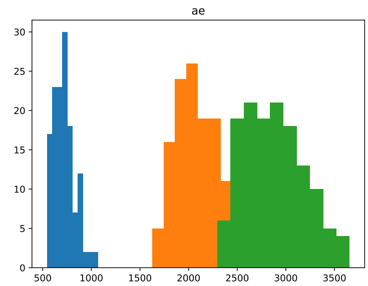
We also tried training with a lower limit for $\nabla_W MSE$, and therfore more iterations. However, no matter what, there was always a couple of mistakes in the differentiation of class 2 and 3. From this we can assume that class 2 and 3 are not linearly separable, but still somewhat separable with a linear classifier.

4.2 Classification of pronounced vowels

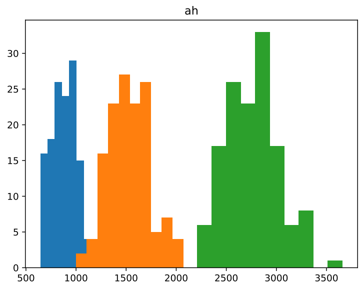
This subsection is about the implementation, results and discussion of the classification of pronounced vowels task described in section 3.2, using the theory about the plug-in-MAP classifier explained in section 2.3. See appendix 5.2 for the code used for the implementation of the single Gaussian of the first part of the task. The code for the GMM in the second part of the task is shown in appendix 5.3.

The implemented classifier from this task classify vowels in $C = 12$ different classes, shown in equation (12), where 'ae' is class 1, 'ah' is class 2 and so on.

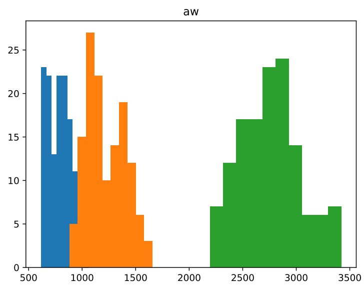
We use the first three formants in each vowels frequency spectrum as features, $D = 3$. The formants for each class are plotted in histograms in figure 6,7, 8 and 9. For each vowel, i.e. each class, we have $N = 139$ samples, and these are from both men, women, boys and girls. When we separate the data in test and training set, we make sure to use split the samples from men, women, boys and girls for each set. The first 70 samples are used for training and the last 69 for testing.



(a) Ae.

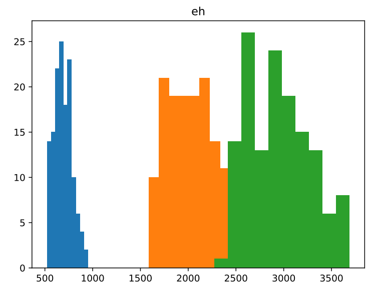


(b) Ah.

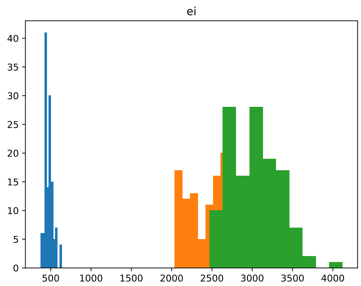


(c) Aw.

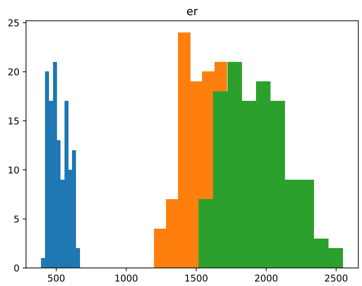
Figure 6: Histograms of the formants for classes 'ae', 'ah' and 'aw'.



(a) Eh.

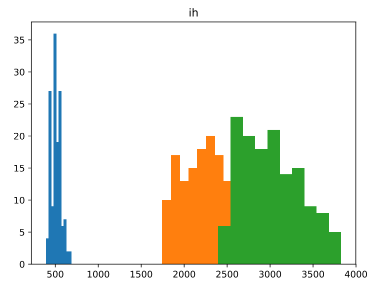


(b) Second image

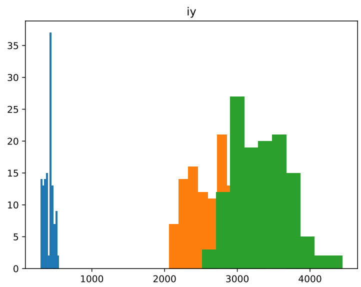


(c) Er.

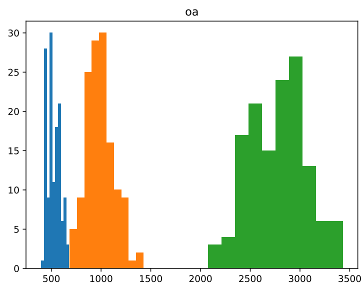
Figure 7: Histograms of the formants for classes 'eh', 'ei' and 'er'.



(a) Ih.

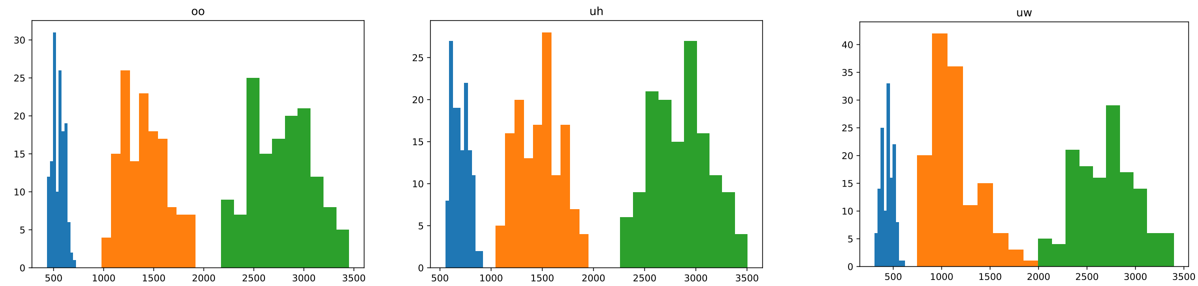


(b) Iy.



(c) Oa.

Figure 8: Histograms of the formants for classes 'ih', 'iy' and 'oa'.



(a) Oo. **(b)** Uh. **(c)** Uw.

Figure 9: Histograms of the formants for classes 'oo', 'uh' and 'uw'.

From the task in section 3.2, we use four different methods to classify the data. First a single gaussian class model, using both the full covariance matrix and the diagonal covariance matrix. Later we implement the GMM with a diagonal covariance matrix, with both 2 and 3 mixtures for each class. To find the first three formants, our features, we choose to use 50 % of vowel duration.

First we look at the single Gaussian class model. We must estimate the mean $\hat{\mu_i}$ and the covariance matrix $\hat{\Sigma}_i$ of the Gaussian distribution in equation (7). The mean for each feature for each sample x_k of a class ω_i is calculated by equation (10) and illustrated in the 2D array in figure 10. The result $\hat{\mu_i}$ of a single class ω_i is a 1×3 array containing the mean of each feature.

```
[[ 662.5323741  2257.43884892 2839.11510791]
 [ 891.25179856 1474.30935252 2770.64028777]
 [ 766.89928058 1145.5323741  2746.87769784]
 [ 686.21582734 2050.05035971 2953.83453237]
 [ 526.03597122 2425.50359712 2869.37410072]
 [ 528.71223022 1564.48920863 1729.56834532]
 [ 476.07194245 2322.58273381 3053.58273381]
 [ 411.56834532 2725.35971223 3058.98561151]
 [ 551.15107914 1013.15827338 2767.48920863]
 [ 519.88489209 1286.00719424 2783.23021583]
 [ 707.92086331 1369.35971223 2860.99280576]
 [ 444.69064748 1157.50359712 2675.79136691]]
```

Figure 10: A 2D array containing the mean for all of the features for all of the classes. If we look at the first line, we see the feature means for class 1, 'ae'.

Further we calculate the covariance matrix. This is shown in the code in figure 11. The covariance matrix becomes for each class a 3×3 matrix because we have $D = 3$ features. Thus, the total covariance matrix becomes $(3 \times 3) \times 12$, because we have 12 classes. In figure 12 we see the covariance matrix for class 1.

```
#Finds the covariance array for each class. Returns a (3x3)x12 array
def covariance(peaks, mean_vec):
    sigma = []
    for i in range(C):
        s = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        for xk in peaks[i]:
            xk = np.array(xk)
            mu = np.array(mean_vec[i])
            diff = xk - mu
            #print(diff)
            for j in range( len(diff)):
                for k in range( len(diff)):
                    s[j][k] += diff[j]*diff[k]/ len(peaks[i])
        sigma.append(s)
    return sigma
```

Figure 11: Code to find the covariance matrix.

```
[[  9236.66075255   9902.74421392  12091.16216256]
 [  9902.74421392  68105.82216094  74028.21049857]
 [ 12091.16216256  74028.21049857 108730.51442768]]
```

Figure 12: A 2D array containing the covariances for all of the features in class 1, 'ae'.

To classify the vowels we use the formula given in equation (7), for each of the classes. This is implemented in the code in figure 13.

```
#Classifies one sample, and returns the index of
def find_class(xk, mu, sigma):
    prob = []
    xk = np.array(xk)
    for i in  range(C):
        mu0 = np.array(mu[i])
        sigma0 = np.array(sigma[i])
        det_sigma0 = np.linalg.det(sigma0)
        inv_sigma0 = np.linalg.inv(sigma0)
        #print('s',inv_sigma0)
        diff = xk - mu0
        #print('d', diff)
        exp1 = np.dot(diff.T, inv_sigma0)
        eksponent = (-1/2)*np.dot(exp1, diff)
        p = np.exp(eksponent)/np.sqrt(((2*np.pi)**3)*det_sigma0)
        prob.append(p)
    #print(np.argmax(prob))
    return np.argmax(prob)
```

Figure 13: Code for classifying one sample, using a single Gaussian distribution.

Here we take in one sample, and calculate the probability of the sample being in each class. To classify the sample we use equation (6), where we assume $P(\omega_i)$ is the same for each class and therefore need not be taken into account. Ergo, the class that gives the highest probability given by the Gaussian density is the class the sample is placed in. The probabilities calculated for each class is stored in a vector *prob*. The index that gives the maximum value in this vector, is the index that corresponds to the correct class. We do this for every sample in the test dataset, and plot it in a counfusion matrix, shown in figure 14. Here we get an error rate of 0.257.

	ae	ah	aw	eh	ei	er	ih	iy	oa	oo	uh	uw
ae	35	2	0	12	0	5	1	0	0	0	0	0
ah	1	47	9	0	0	0	0	0	0	0	4	0
aw	0	11	49	0	0	0	0	0	1	0	5	0
eh	27	0	0	51	0	0	0	0	0	1	2	0
ei	1	0	0	0	40	0	3	12	0	0	0	0
er	4	0	1	3	0	64	0	0	0	2	0	1
ih	1	0	0	2	19	0	63	2	0	0	0	2
iy	0	0	0	0	10	0	2	55	0	0	0	0
oa	0	0	0	0	0	0	0	0	59	4	0	13
oo	0	0	0	0	0	0	0	0	0	51	7	3
uh	0	9	10	1	0	0	0	0	4	5	51	0
uw	0	0	0	0	0	0	0	0	5	6	0	50

Figure 14: Confusion matrix for the test data, using a single Gaussian classifier with full covariance matrix. The true classes are seen on the rows, and the classified classes are on the columns.

Furthermore we use the diagonal covariance matrix, as shown in figure 15. We see that the diagonal values is the same as in figure 12, but everything else is zero.

```
[[  9236.66075255       0.                  0.        ]
 [      0.          68105.82216094        0.        ]
 [      0.              0.           108730.51442768]]
```

Figure 15: Diagonal covariance matrix for class 1, 'ae'.

To classify the test data with the diagonal matrix, we use the same technique and code in figure 13, but with the diagonal covariance matrix. This gives us the confusion matrix shown in figure 16, and an error rate of 0.395.

	ae	ah	aw	eh	ei	er	ih	iy	oa	oo	uh	uw
ae	31	1	0	16	0	3	1	0	0	0	0	0
ah	1	42	17	0	0	0	0	0	0	0	7	0
aw	0	15	42	0	0	0	0	0	0	0	5	0
eh	23	0	0	37	0	1	0	0	0	7	10	0
ei	1	0	0	0	29	0	7	10	0	0	0	0
er	5	0	1	0	1	64	3	0	0	1	0	1
ih	3	0	0	1	30	0	57	3	0	1	0	3
iy	0	0	0	0	9	0	1	56	0	0	0	0
oa	0	0	8	0	0	0	0	0	43	6	2	13
oo	4	0	0	13	0	0	0	0	4	32	18	11
uh	1	11	1	2	0	1	0	0	4	5	27	0
uw	0	0	0	0	0	0	0	0	18	17	0	41

Figure 16: Confusion matrix for the test data, using a single Gaussian classifier with diagonal covariance matrix. The true classes are seen on the rows, and the classified classes are on the columns.

Now we will look at the implementation of a GMM, with a diagonal covariance matrix. We use a built in function called sklearn.mixture.GaussianMixture.fit() from the sklearn libary [2]. This function takes in a matrix containing all the training samples from one class, and the number of mixed Gaussians wanted. It returns information of the Gaussian distributions which will fit the class distribution, including a mean array and a covariance array. It also contains the weights multiplied with each Gaussian to make the mixing of them model the observations in the class. This is done by using the EM algorithm. The algorithm iteration will stop when the average gain is below the threshold of 1e-3, by default. Therefore, the function contains all the estimated parameters needed for calculating equation (8).

In the code in figure 17, we use this function for the training data for all of the classes, and make a list containing the information about the Gaussian distributions: the mean, covariance and weight arrays, called *gm_vec*.

```python
#Takes all the training data and gets the GMM information
def train(vec, n):
    global C
    gm_vec = []
    for i in  range(C):
        F = np.array(vec[i])
        gm = GaussianMixture(n_components=n, random_state=0).fit(F)
        gm_vec.append(gm)
    return gm_vec
```

Figure 17: Code for training the plug-in-MAP classifier by finding the information about the Gaussian distributions *gm* used to model the observation data in the vector *vec*.

When we classify the test data, we use this *gm_vec*. Just like for the single Gaussian, we use equation (6) to classify the data, by finding the argument that gives the highest probability in the vector *prob*. This is shown in the code in figure 18.

```python
#Uses the gauss function to calculate the probability of one sample being in each class
def find_class(xk, gm_vec):
    xk = np.array(xk)
    prob =[]
    for gm in gm_vec:
        mu = gm.means_
        c = gm.weights_
        sigma = find_dig_cov(gm.covariances_)
        p = 0
        for i in  range( len(mu)):
            mu0 = np.array(mu[i])
            sigma0 = np.array(sigma[i])
            p += gauss(xk, sigma0, mu0)*c[i]
        prob.append(p)
    return np.argmax(prob)
```

Figure 18: Code for the final classification, using equation (6).

Here we take in one sample, and calculate the probability for this sample being in each class. To calculate this we use equation (8). The implementation of this equation is shown in the code in figure 19.

```python
#Implementation of the function calculating the probability of one sample being in one class.
def gauss(xk, sigma0, mu0):
    det_sigma0 = np.linalg.det(sigma0)
    inv_sigma0 = np.linalg.inv(sigma0)
    diff = xk - mu0
    exp1 = np.dot(diff[0].T, inv_sigma0)
    eksponent = (-1/2)*np.dot(exp1, diff[0])
    p = np.exp(eksponent)/np.sqrt(((2*np.pi)**3)*det_sigma0)
    return p
```

Figure 19: Code for calculating the equation (8), used for the final classification in figure 18.

In this function, we take in a Gaussian distribution and a sample, and calculate the probability for this sample being in this distribution. In the code in figure 18, we sum up the all the probabilities for one class, and multiply them by their weights. When we have done this for all the classes, we return the class index for the class with highest probability for this one sample.

The task in section 3.2 specifies that we should use a mixture of both $M = 2$ and $M = 3$ Gaussians. Firstly, we use a mixture of $M = 2$ Gaussian distributions for each class. The confusion matirx for the test data is illustrated in figure 20. This gives us an error rate of 0.320.

	ae	ah	aw	eh	ei	er	ih	iy	oa	oo	uh	uw
ae	41	1	0	24	0	3	1	0	0	0	0	0
ah	2	47	12	0	0	0	0	0	0	0	5	0
aw	0	19	41	0	0	0	0	0	1	0	6	0
eh	21	0	0	38	0	2	0	0	0	3	2	0
ei	1	0	0	0	34	0	17	12	0	0	0	0
er	1	0	1	1	0	63	0	0	0	1	0	1
ih	1	0	0	3	20	0	49	2	0	0	0	3
iy	1	0	0	0	15	0	2	55	0	0	0	0
oa	0	0	0	0	0	0	0	0	54	6	5	16
oo	0	0	0	1	0	0	0	0	3	49	1	7
uh	1	2	15	2	0	1	0	0	2	4	50	0
uw	0	0	0	0	0	0	0	0	9	6	0	42

Figure 20: Confusion matrix for the test data, using a GMM of order 2, with diagonal covariance matrix.

Next we implement this using $M = 3$ Gaussians for each class. The result when classifying the test data is now illustrated in figure 21. The error rate is now 0.252.

	ae	ah	aw	eh	ei	er	ih	iy	oa	oo	uh	uw
ae	38	0	0	18	0	3	1	0	0	0	0	0
ah	1	53	5	0	0	0	0	0	0	0	4	0
aw	0	10	50	0	0	0	0	0	2	0	5	0
eh	23	1	0	43	0	0	0	0	0	2	1	0
ei	1	0	0	0	53	0	11	15	0	0	0	0
er	4	0	1	0	0	66	0	0	0	0	0	1
ih	1	0	0	3	5	0	56	1	0	0	0	0
iy	0	0	0	0	11	0	1	53	0	0	0	0
oa	1	1	1	0	0	0	0	0	57	3	2	17
oo	0	0	0	1	0	0	0	0	0	54	5	7
uh	0	4	12	3	0	0	0	0	4	5	52	0
uw	0	0	0	1	0	0	0	0	6	5	0	44

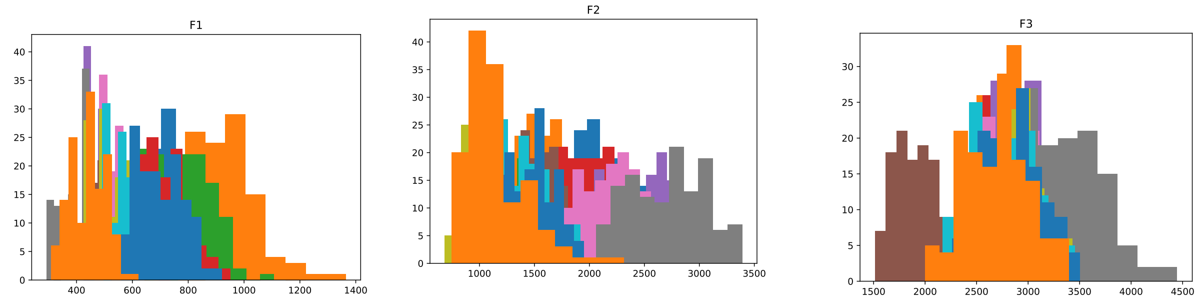
Figure 21: Confusion matrix for the test data, using a gaussian mixed model of order 3, with diagonal covariance matrix.

We see in this task, from the error rate and confusion matrices in figure 14 and 16, that using the full covariance for a single Gaussian gives better results than using only the diagonal. The diagonal confusion matrix will only give the variances of the different features, but not how they relate to each other. When using a GMM, we see that the error rate is lower when $M = 3$, than when $M = 2$. For the single Gaussian we got an error rate of 0.257 using the full covariance matrix. For a GMM with $M = 2$ mixtures, we get a larger error rate of 0.320. This is because for the GMM, we only use diagonal covariance matrices. However, we got the best error rate and covariance matrix using the GMM with $M = 3$ mixtures. We can therfore assume that this is the distribution that models our data distribution the best.

The difference in performance also varies from class to class. We see that 'er' performes quite well for alle the classifiers. 'uh' on the other hand has for GMM with $M = 3$ been classified correctly 52 times, but for the single Gaussian with diagonal covariance matrix, only 27 times. Same goes for the 'ei'.

Of course one also have to have the runtime and complexity in mind when choosing a classifier. That is, when choosing full covariance matrix instead of diagonal and GMM with $M = 3$ instead of single Gaussian. In our case the data set is quite small, so the difference in runtime is minimal.

The error rate is always above 25% which is quite high for a classifier. From the histograms in figure 22, we see that the formants are overlapping. Some of the classes seems to be more separable than others, as we see in the confusion matrices, especially 'er', 'oa' and 'ih'. Additionally, the approximated Gaussian distributions may not fit the real data distribution.



(a) Formant 1 for all the classes. **(b)** Formant 2 for all the classes. **(c)** Formant 3 for all the classes.

Figure 22: Histograms of the formants for all the classes. We see that the formans are somewhat overlapping.

To improve the classifier, one idea is to get more samples for the training set, as described in section **??**. This will be at the expense of runtime, so this is a tradeoff.

To test the classifier with our own voices, we made a script which records a vowel for one second and classifies it using the GMM with $M = 3$. This is included in appendix 5.4. After visual observation, the error rate seems to correspond to the error rate found earlier.

5 Conclusion

We have in this report described the design and results of a linear and plug-in-MAP classifier, as part of our course TTT4275.

The linear classifier was used to classify three different species of the Iris genus. As the data is not completely linearly separable between the Iris Versicolor and Iris Virginica, there were some small error rates of these two classes. We also saw this when we switched the samples the test set consisted of, from the 20 last to the 20 first samples, and saw the error rate wasn't changing much. This is because of the same individual samples being difficult to classify. Iris Setosa, on the other hand, is completely separable from the other two classes and had a error rate of 0. This also held true when we removed the features but one, even though this worsened the cassification overall. This is because we also removed data that were not overlapping. However, even then the classification only gave an error rate of 0.083 and 0.189 for the test and training set. Therefore, we can conclude that the classes are almost linearly separable.

To separate the vowels in task 3.2, a plug-in-MAP classifier have been used. We have tested using both a single Gaussian model, with both a full and a diagonal covariance matrix, and a Gaussian Mixture Model of order $M = 2$ and $M = 3$. We see that using the full covariance matix is better than using the diagonal, with error rates of respectivly 0.257 and 0.395. We also observe that the GMM is a better model for our data, as long as the order M is correct. For $M = 2$ we get an error rate of 0.320, and for $M = 3$ an error rate of 0.252. The different implementations perform different for the different classes. Especially for 'ei' and 'uh' the difference in performance is big. Even though none of theese classifiers have an error below 25%, we see that there is a clear trend along the diagonal in the confusion matrices. This shows that the classifiers work fairly ok.

To get a smaller error rate, it would be an idea to have more samples. The more samples we use for training, the better the result. But the more samples, the longer runtime for the traning, so we would have to find a balance.

References

[1] Johnsen, Magne H. “Classification”. TTT4275. 2017.

[2] *scikit-learn*. `https://scikit-learn.org/stable/`. Jan. 2021.

5.1 Iris script

```python
import matplotlib.pyplot as plt
import numpy as np
import csv
import random
import pandas as pd


"""
This script is for classifying samples of three classes of Irises, by training and testing a linear classifier.
    1. sepal length in cm
    2. sepal width in cm
    3. petal length in cm
    4. petal width in cm
"""

C = 3
classes = ['Setosa', 'Versicolour', 'Virginica']
features = ['Sepal length', 'Sepal width', 'Petal length', 'Petal width']

training_data = []
testing_data = []
tot_vec = []
Tn = [[1, 0, 0]]*30 + [[0, 1, 0]]*30 + [[0, 0, 1]]*30
W = np.zeros((3, 5))
training_threshold = 0.4


len_train_class = 30
len_test_class = 20


def get_data():
    """
    Updates the training and testing data sets with data from text file.
    """
    training_data0 = []
    testing_data0 = []
    tot_vec0 = []
```

```
    for i in  range(C):
        filename = ".\Iris_TTT4275\class_"+ str(i+1)+".txt"
        with  open(filename,'r') as my_file:
            data = csv.reader(my_file, delimiter= ',')
            i = 0
            for line in data:
                tot_vec0.append(line)
                if i < len_train_class:
                    training_data0.append(line)
                else:
                    testing_data0.append(line)
                i += 1

    for line in training_data0:
        line_new = []
        for value in line:
            line_new.append( float(value))
        line_new.append(1)
        training_data.append(line_new)

    for line in tot_vec0:
        line_new = []
        for value in line:
            line_new.append( float(value))
        line_new.append(1)
        tot_vec.append(line_new)

    for line in testing_data0:
        line_new = []
        for value in line:
            line_new.append( float(value))
        line_new.append(1)
        testing_data.append(line_new)
    return


def plot_petal_data(vec):
    """
    Plots of the length and width of the petal against each
       other for illustrative purposes.
    :param vec: data set.
    """
    length = []
```

```
    width = []
    l =  int( len(vec)/3)
    for line in vec:
        length.append( float(line[2]))
        width.append( float(line[3]))
    plt.plot(length[0:l], width[0:l], 'r' 'x')
    plt.plot(length[l:2*l], width[l:2*l], 'g' 'x')
    plt.plot(length[2*l:3*l], width[2*l:3*l], 'b' 'x')
    return


def plot_sepal_data(vec):
    """
    Plots of the length and width of the sepal against each
       other for illustrative purposes.
    :param vec: data set.
    """
    length = []
    width = []
    l =  int( len(vec)/3)
    for line in vec:
        length.append( float(line[0]))
        width.append( float(line[1]))
    plt.plot(length[0:l], width[0:l], 'r' 'x')
    plt.plot(length[l:2*l], width[l:2*l], 'g' 'x')
    plt.plot(length[2*l:3*l], width[2*l:3*l], 'b' 'x')
    return


def average(data):
    """
    Finds the average point of each class.
    :param data: training data set.
    :return: the average points of the classes
    """
    class_1 = [0, 0, 0, 0]
    class_2 = [0, 0, 0, 0]
    class_3 = [0, 0, 0, 0]
    l =  int( len(data)/3)
    i = 0
    for i in  range(l):
        class_1[0] += data[i][0]/l
        class_1[1] += data[i][1]/l
```

```
        class_1[2] += data[i][2]/l
        class_1[3] += data[i][3]/l

        class_2[0] += data[i+l][0]/l
        class_2[1] += data[i+l][1]/l
        class_2[2] += data[i+l][2]/l
        class_2[3] += data[i+l][3]/l

        class_3[0] += data[i+l*2][0]/l
        class_3[1] += data[i+l*2][1]/l
        class_3[2] += data[i+l*2][2]/l
        class_3[3] += data[i+l*2][3]/l
    return class_1, class_2, class_3


def sigmoid(vec):
    """
    Calculates the sigmoid function.
    :param vec: training or testing data set.
    :return: the sigmoid function
    """
    return 1/(1+np.exp(-vec))


def train(vec, alpha):
    """
    Training the W matrix.
    :param vec: the training data set.
    :param alpha: the step size
    """
    global W
    global training_threshold
    i = 0
    while True:
        W_last = W
        n_mse = 0
        for j in range(len(vec)):
            xk = np.matrix(vec[j])
            tk = np.matrix(Tn[j])
            zk = np.dot(W, xk.T).T
            gk = sigmoid(zk)
            mid_term = np.multiply((gk-tk),gk)
            d_mse = np.multiply(mid_term,(1-gk))
```

```
                n_mse += np.dot(d_mse.T, xk)

            W = W_last - alpha*n_mse

            if np.all(abs(n_mse) <= training_threshold):
                print('Number of iterations:', i+1)
                print('nabla-nmse:',n_mse)
                print('W:', W)
                break
            i += 1


def test(vec, number_per_class):
    """
    Testing a data set.
    :param vec: the training or test data set.
    :param number_per_class: number of samples in each class
    :return: classified data
    """
    class_vec = [[], [], []]
    for i in range(len(vec)):
        xk = np.matrix(vec[i])
        zk = np.dot(W, xk.T).T
        gk = sigmoid(zk[0])
        classified = np.argmax(gk)+1
        if i < number_per_class:
            class_vec[0].append(classified)
        elif number_per_class <= i < number_per_class*2:
            class_vec[1].append(classified)
        elif number_per_class*2 <= i:
            class_vec[2].append(classified)
    return class_vec


def error_rate(vec):
    """
    :param vec: the tested data set
    :return: error rate of the classified data set
    """
    sum_error = 0
    len_class = len(vec[0])
    for i in range(len(vec)):
        sum_error += len_class - vec[i].count(i+1)
```

```
    err_t = sum_error/( len(vec)*len_class)
    return err_t


def plot_confusion(vec):
    """
    Plotting the confusion matrix.
    :param vec: the classified data set.
    """
    global classes
    data = {
        classes[0]: [vec[0].count(1), vec[1].count(1), vec[2].count(1)],
        classes[1]: [vec[0].count(2), vec[1].count(2), vec[2].count(2)],
        classes[2]: [vec[0].count(3), vec[1].count(3), vec[2].count(3)]
        }
    df = pd.DataFrame(data, index=classes)
    print(df)


def plot_feature(vec, index, len_class):
    """
    Plotting the histograms of the different features for all of the classes.
    :param vec: data set
    :param index: index of the feature to be plotted
    :param len_class: number of samples in each class
    """
    global C
    class1 = []
    class2 = []
    class3 = []
    for i in  range(len_class):
        class1.append(vec[i][index])
        class2.append(vec[i+len_class][index])
        class3.append(vec[i+len_class*2][index])
    plt.hist(class1, color='r', rwidth=1)
    plt.hist(class2, color='g', rwidth=1)
    plt.hist(class3, color='b', rwidth=1)
    plt.title(features[index])
    plt.xlabel('cm')
```

```python
    plt.ylabel('Number')
    plt.show()


def remove_feature(vec, index):
    """
    This is used on exercise 2 a and b, where we remove a
       feature one by one.
    :param vec: the data set
    :param index: index of the feature to be removed
    """
    new_vec = []
    for xk in vec:
        xk.pop(index)
        new_vec.append(xk)
    return new_vec


if __name__ == "__main__":
    get_data()

    alpha = 0.001
    train(training_data, alpha)

    print('\n\n', 'Testing!')

    print('confusion matrix for the test data')
    test_classified = test(testing_data, len_test_class)
    plot_confusion(test_classified)
    err_t_test = error_rate(test_classified)
    print(err_t_test)

    print('\n')
    print('confusion matrix for the train data')
    train_classified = test(training_data, len_train_class)
    plot_confusion(train_classified)
    err_t_train = error_rate(train_classified)
    print(err_t_train)

    # This is used on exercise 2 a and b, where we remove a
       feature one by one.
    # The dimensions of W need to be changed as well.
    '''
```

```
    training_data_1 = remove_feature(training_data, 1)
    testing_data_1 = remove_feature(testing_data, 1)

    training_data_1 = remove_feature(training_data_1, 0)
    testing_data_1 = remove_feature(testing_data_1, 0)

    training_data_1 = remove_feature(training_data_1, 0)
    testing_data_1 = remove_feature(testing_data_1, 0)

    train(training_data_1, alpha)
    test_classified = test(testing_data_1, len_test_class)
    train_classified = test(training_data_1, len_train_class)

    plot_confusion(test_classified)
    plot_confusion(train_classified)

    err_t_test = error_rate(test_classified)
    print(err_t_test)
    err_t_train = error_rate(train_classified)
    print(err_t_train)
    '''

    # Here we plot the histograms of the different features
       for all of the classes.
    # tot_vec = remove_feature(tot_vec, 1)
    # plot_feature(tot_vec, 0, 50)
    # plot_feature(tot_vec, 1, 50)
    # plot_feature(tot_vec, 2, 50)
    # plot_feature(tot_vec, 3, 50)

    '''
    # Plotting of the different features against each other
       for illustrative purposes.
    f = plt.figure()
    mu_1, mu_2, mu_3 = average(training_data)
    plot_sepal_data(training_data)
    plot_sepal_data(testing_data)
    class1 = plt.plot(mu_1[0], mu_1[1], 'ro', label = classes
       [0])
    class2 = plt.plot(mu_2[0], mu_2[1], 'go', label = classes
       [1])
    class3 = plt.plot(mu_3[0], mu_3[1], 'bo', label = classes
       [2])
```

```
    plt.legend()
    plt.xlabel('Sepal length [cm]')
    plt.ylabel('Sepal width [cm]')
    f.savefig("Sepalvssepal.pdf", bbox_inches='tight')
    plt.show()

    f = plt.figure()
    plot_petal_data(testing_data)
    plot_petal_data(training_data)
    class1 = plt.plot(mu_1[2], mu_1[3], 'ro', label = classes
       [0])
    class2 = plt.plot(mu_2[2], mu_2[3], 'go', label = classes
       [1])
    class3 = plt.plot(mu_3[2], mu_3[3], 'bo', label = classes
       [2])
    plt.legend()
    plt.xlabel('Petal length [cm]')
    plt.ylabel('Petal width [cm]')
    f.savefig("Petalvspetal.pdf", bbox_inches='tight')
    plt.show()
    '''
```

5.2 Vowels script, task 1

```
import numpy as np
import csv
import matplotlib.pyplot as plt
import pandas as pd


people = ['m', 'w', 'b', 'g']
count = np.zeros((4,12))
classes = ['ae', 'ah', 'aw', 'eh', 'ei', 'er', 'ih', 'iy','oa'
   , 'oo', 'uh', 'uw']
C = 12

data_list = [[], [], [],  []]

for j in  range(4):
    for i in  range(12):
        data_list[j].append([])
```

```python


train_data = []
test_data = []
tot = []
#To get the right dimentions
for i in  range(12):
    train_data.append([])
    test_data.append([])
    tot.append([])

#Finds the class of one samples, and returns the index in the
   claases array
def find_class_index(name):
    vowel = name[3:5]
    index = classes.index(vowel)
    return index

#Finds out weather it is a man, woman boy or girl, and return
   the corresponding index people
def find_person_index(name):
    person = name[0]
    index = people.index(person)
    return index

#Gets data from the file "vowdata_nohead.dat" and sorts it
   into test_data and train_data
def get_data():
    global data_list
    filnavn = "vowdata_nohead.dat"
    with  open(filnavn,'r') as myfile:
        data = csv.reader(myfile, delimiter= ' ')
        for line in data:
            while('' in line):
                line.remove('')
            person = find_person_index(line[0])
            index = find_class_index(line[0])
            xk = [ float(line[10]),  float(line[11]),
               float(line[12])]
            data_list[person][index].append(xk)

    for i in  range(4):
        for j in  range(C):
```

```python
                l =  len(data_list[i][j])
                train_data[j]+=data_list[i][j][: round(l/2)]
                test_data[j]+=data_list[i][j][ round(l/2):]
                tot[j]+=data_list[i][j]


#plots histograms of the features in each class
def plot_hist(peaks, l = None):
    global C
    if l == None:
        l = C
    for i in  range(l):
        F1 = []
        F2 = []
        F3 = []
        for value in peaks[i]:
            if(0 not in value):
                F1.append(value[0])
                F2.append(value[1])
                F3.append(value[2])
        f = plt.figure()
        plt.hist(F1)
        plt.hist(F2)
        plt.hist(F3)
        plt.title(classes[i])
        f.savefig(classes[i]+".pdf", bbox_inches='tight')

    return

#Finds the means for all features for each class. Returns a 3
   x12 array
def find_mean_vec(peaks):
    mean_vec = np.zeros((12, 3))
    l =  len(peaks[0])
    for i in  range(C):
        for j in  range(l):
            mean_vec[i] += peaks[i][j]
    mean_vec /= l
    return mean_vec

#Finds the covariance array for each class. Returns a (3x3)x12
    array
def covariance(peaks, mean_vec):
```

```
    sigma = []
    for i in range(C):
        s = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        for xk in peaks[i]:
            xk = np.array(xk)
            mu = np.array(mean_vec[i])
            diff = xk - mu
            #print(diff)
            for j in range(len(diff)):
                for k in range(len(diff)):
                    s[j][k] += diff[j]*diff[k]/len(peaks[i])
        sigma.append(s)
    return sigma

#Classifies one sample, and returns the index of
def find_class(xk, mu, sigma):
    prob = []
    xk = np.array(xk)
    for i in range(C):
        mu0 = np.array(mu[i])
        sigma0 = np.array(sigma[i])
        det_sigma0 = np.linalg.det(sigma0)
        inv_sigma0 = np.linalg.inv(sigma0)
        #print('s',inv_sigma0)
        diff = xk - mu0
        #print('d', diff)
        exp1 = np.dot(diff.T, inv_sigma0)
        eksponent = (-1/2)*np.dot(exp1, diff)
        p = np.exp(eksponent)/np.sqrt(((2*np.pi)**3)*
           det_sigma0)
        prob.append(p)
    #print(np.argmax(prob))
    return np.argmax(prob)

#Plots the confusion matrix
def plot_confusion(vec):
    global classes
    data = {}
    for i in range(C):
        data.update({classes[i]:vec[i]})
    df = pd.DataFrame(data, index = classes)
    print(df)
```

```
#Finds the error rate
def error_rate(vec):
    sum_not_error = 0
    tot = 0
    len_class =  len(vec[0])
    for i in  range( len(vec)):
        sum_not_error += vec[i][i]
        tot +=  sum(vec[i])

    err_t = (tot-sum_not_error)/tot
    return err_t

#Takes in a full covariance matrix, and returns a diagonal
   covariance matrix
def find_dig_cov(vec):
    ide = np.identity(3)
    mat = []
    for i in  range(C):
        mat.append(np.multiply(vec[i], ide))
    return np.array(mat)

#Main
if __name__ == "__main__":
    get_data()
    #plot_hist(train_data, l = 6)
    #plot_hist(test_data, l = 3)
    plot_hist(tot)
    mean_vec = find_mean_vec(tot)
    print(np.array(mean_vec), '\n')
    sigma = covariance(train_data, mean_vec)
    #print(np.array(sigma))

    prob = []
    for j in  range(C):
        p = []
        for i in  range(69):
            p.append(find_class(test_data[j][i], mean_vec,
                sigma))
            #print(find_class(test_data[j][i], mean_vec, sigma
                ))
        prob.append([p.count(0), p.count(1), p.count(2), p.
           count(3), p.count(4), p.count(5), p.count(6), p.
           count(7), p.count(8),p.count(9), p.count(10), p.
```

```
            count(11)])
    plot_confusion(prob)
    err_t = error_rate(prob)
    print(err_t)


    print('Det')
    dig_sigma = find_dig_cov(sigma)
    #print(dig_sigma)
    prob = []
    for j in  range(C):
        p = []
        for i in  range(69):
            p.append(find_class(test_data[j][i], mean_vec,
               dig_sigma))
            #print(find_class(test_data[j][i], mean_vec, sigma
               ))
        prob.append([p.count(0), p.count(1), p.count(2), p.
            count(3), p.count(4), p.count(5), p.count(6), p.
            count(7), p.count(8),p.count(9), p.count(10), p.
            count(11)])
    plot_confusion(prob)
    err_t = error_rate(prob)
    print(err_t)
```

5.3 Vowels script, task 2

```
import numpy as np
import csv
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.mixture import GaussianMixture


people = ['m', 'w', 'b', 'g']
count = np.zeros((4,12))
classes = ['ae', 'ah', 'aw', 'eh', 'ei', 'er', 'ih', 'iy','oa'
    , 'oo', 'uh', 'uw']
C = 12


```

```
#List used as a middleman to sort the data from the file
data_list = [[], [], [],  []]

#To get the right dimentions on data_list
for j in  range(4):
    for i in  range(12):
        data_list[j].append([])

#Sorts data lists
train_data = []
test_data = []
tot = []
for i in  range(12):
    train_data.append([])
    test_data.append([])
    tot.append([])

#Finds the index in "classes" for a goven vowel
def find_class_index(name):
    vowel = name[3:5]
    index = classes.index(vowel)
    return index

#Finds the index in "persons" for a givem person
def find_person_index(name):
    person = name[0]
    index = people.index(person)
    return index

#Gets and sorts the data form the datafile
def get_data():
    global data_list
    filnavn = "vowdata_nohead.dat"
    with  open(filnavn,'r') as myfile:
        data = csv.reader(myfile, delimiter= ' ')
        for line in data:
            while('' in line):
                line.remove('')
            person = find_person_index(line[0])
            index = find_class_index(line[0])
            xk = [ float(line[10]),  float(line[11]),
               float(line[12])]
            data_list[person][index].append(xk)
```

```python

    for i in  range(4):
        for j in  range(C):
            l =  len(data_list[i][j])
            train_data[j]+=data_list[i][j][: round(l/2)]
            test_data[j]+=data_list[i][j][ round(l/2):]
            tot[j]+=data_list[i][j]


#Takes all the training data and gets the GMM information
def train(vec, n):
    global C
    gm_vec = []
    for i in  range(C):
        F = np.array(vec[i])
        gm = GaussianMixture(n_components=n, random_state=0).
           fit(F)
        gm_vec.append(gm)
    return gm_vec

#Implementation of the function calculating the probability of
     one sample being in one class.
def gauss(xk, sigma0, mu0):
    det_sigma0 = np.linalg.det(sigma0)
    inv_sigma0 = np.linalg.inv(sigma0)
    diff = xk - mu0
    exp1 = np.dot(diff[0].T, inv_sigma0)
    eksponent = (-1/2)*np.dot(exp1, diff[0])
    p = np.exp(eksponent)/np.sqrt(((2*np.pi)**3)*det_sigma0)
    return p

#Plots the confusion matrix
def plot_confusion(vec):
    global classes
    data = {}
    for i in  range(C):
        data.update({classes[i]:vec[i]})
    df = pd.DataFrame(data, index = classes)
    print(df)

#Uses the gauss function to calculate the probability of one
    sample being in each class
def find_class(xk, gm_vec):
```

```
    xk = np.array(xk)
    prob =[]
    for gm in gm_vec:
        mu = gm.means_
        c = gm.weights_
        sigma = find_dig_cov(gm.covariances_)
        p = 0
        for i in range(len(mu)):
            mu0 = np.array(mu[i])
            sigma0 = np.array(sigma[i])
            p += gauss(xk, sigma0, mu0)*c[i]
        prob.append(p)
    return np.argmax(prob)

#Finds the error rate
def error_rate(vec):
    sum_not_error = 0
    tot = 0
    len_class = len(vec[0])
    for i in range(len(vec)):
        sum_not_error += vec[i][i]
        tot += sum(vec[i])
    err_t = (tot-sum_not_error)/tot
    return err_t

#Takes in a 3x3 covariance matrix and returns the diagonal
   version of it
def find_dig_cov(vec):
    ide = np.identity(3)
    return np.multiply(vec, ide)

#Main
if __name__ == "__main__":
    get_data()
    gm_vec = train(train_data, 2)
    prob = []
    for j in range(C):
        p = []
        for i in range(69):
            p.append(find_class([test_data[j][i]], gm_vec))
        prob.append([p.count(0), p.count(1), p.count(2), p.
           count(3), p.count(4), p.count(5), p.count(6), p.
           count(7), p.count(8),p.count(9), p.count(10), p.
```

```
            count(11)])
    plot_confusion(prob)
    err_t = error_rate(prob)
    print(err_t)
```

5.4 Vocal recognition

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from scipy import signal
import sounddevice as sd
import time
import pysptk



people = ['m', 'w', 'b', 'g']
classes = ['ae', 'ah', 'aw', 'eh', 'ei', 'er', 'ih', 'iy','oa', 'oo', 'uh', 'uw']
avslag = ['Nei', 'nei', 'No', 'Nope', 'no', 'nope', 'n']
C = 12
Fs = 16000
GMM_number = 3


mu =  [np.array([[ 591.49349236, 1912.50992355, 2591.22584184],
       [ 624.2       , 2541.         ,    0.        ],
       [ 697.1793304 , 2400.10853874, 3107.23046549]]), np.array([[ 963.97358086, 1612.0936153 , 2863.39038548],
       [1069.66666667,    0.         , 2923.        ],
       [ 766.62305769, 1341.10015312, 2615.95765414]]), np.array([[ 950.         ,    0.         , 2955.         ],
       [ 635.        , 1163.         ,    0.        ],
       [ 763.80740741, 1170.85925926, 2762.6        ]]), np.array([[ 705.14898422, 2116.68165268, 2998.0314528 ],
       [ 581.6684733 , 1793.84804368, 2590.0328176 ],
```

```
        [ 807.5191198 , 2285.3407006 , 3442.36534737]]), np.array([[ 548.38263345, 2578.24992699, 3184.38602757],
        [ 526.85714286, 2648.71428571,    0.        ],
        [ 481.06997669, 2083.45613523, 2694.80857687]]), np.array([[ 496.8       , 1553.86666667,    0.        ],
        [ 483.9431512 , 1462.6526521 , 1800.33508871],
        [ 590.7074655 , 1689.05260762, 2104.30900481]]), np.array([[ 478.96627823, 2353.89386487, 3020.4665573 ],
        [ 513.09307592, 2565.21247738, 3437.53969635],
        [ 432.10994604, 2020.81943858, 2667.435509  ]]), np.array([[ 374.43172946, 2456.68709207, 3092.60836627],
        [ 441.16666667, 2957.66666667,    0.        ],
        [ 448.19566491, 2986.65949245, 3641.25356398]]), np.array([[ 546.62341621, 1000.70430885, 2690.09535887],
        [ 603.11245855,  373.98225894, 2641.74624817],
        [ 562.481108  , 1149.61783498, 3097.66771732]]), np.array([[ 511.8478501 , 1239.75662051, 2889.69393915],
        [ 586.28780308, 1545.00995072, 3081.50249188],
        [ 476.07609921, 1124.84492869, 2493.52916447]]), np.array([[ 713.36135877, 1336.1527646 , 2780.35643643],
        [ 709.76985346, 1553.70149569, 3122.23158743],
        [ 632.23925517,  976.05738302, 2720.82617118]]), np.array([[ 379.0520588 ,  982.42051405, 2363.53226861],
        [ 490.9098739 , 1534.3602661 , 2981.25959873],
        [ 469.41403641, 1107.7490993 , 2754.35242883]]])]


sigma =  [np.array([[[ 1.71578555e+03, -1.17350318e+03, -1.10269586e+03],
         [-1.17350318e+03,  1.11798147e+04,  7.84279371e+03],
         [-1.10269586e+03,  7.84279371e+03,  1.32390614e+04]],

        [[ 5.86776000e+03, -3.69980000e+03,  0.00000000e+00],
         [-3.69980000e+03,  1.04168000e+04,  0.00000000e+00],
         [ 0.00000000e+00,  0.00000000e+00,  1.00000000e-06]],
```

```

        [[ 6.66029558e+03, -2.13585632e+02,  4.86829532e+03],
         [-2.13585632e+02,  3.23305944e+04,  3.89569907e+04],
         [ 4.86829532e+03,  3.89569907e+04,  7.93043961e+04]]]), np.array([[[1.27874414e+04, 6.75506542e+03, 9.81062940e+03],
         [6.75506542e+03, 2.93753351e+04, 2.18410232e+04],
         [9.81062940e+03, 2.18410232e+04, 5.74561645e+04]],

        [[2.27902222e+04, 0.00000000e+00, 8.47466667e+03],
         [0.00000000e+00, 1.00000000e-06, 0.00000000e+00],
         [8.47466667e+03, 0.00000000e+00, 6.19940000e+04]],

        [[3.76783321e+03, 3.11006445e+03, 6.70636489e+03],
         [3.11006445e+03, 1.57157837e+04, 2.61371053e+03],
         [6.70636489e+03, 2.61371053e+03, 4.53717105e+04]]]), np.array([[[8.91800000e+03, 0.00000000e+00, 2.60680000e+04],
         [0.00000000e+00, 1.00000000e-06, 0.00000000e+00],
         [2.60680000e+04, 0.00000000e+00, 8.03086667e+04]],

        [[1.00000000e-06, 3.41212003e-24, 0.00000000e+00],
         [3.41212003e-24, 1.00000000e-06, 0.00000000e+00],
         [0.00000000e+00, 0.00000000e+00, 1.00000000e-06]],

        [[1.11852666e+04, 1.39986618e+04, 1.67815081e+04],
         [1.39986618e+04, 2.83790395e+04, 2.30101289e+04],
         [1.67815081e+04, 2.30101289e+04, 7.89310844e+04]]]), np.array([[[ 4330.31264138, -2933.80227284, -1551.05268062],
         [-2933.80227284, 33960.31158972, 23129.78682287],
         [-1551.05268062, 23129.78682287, 39235.18735666]],

        [[  958.2700418 ,   640.47297142,  1053.36417373],
         [  640.47297142,  8529.44456428,  5158.46958355],
         [ 1053.36417373,  5158.46958355, 13782.40832027]],

        [[ 8750.62728906,  3384.26935016,  1890.87584391],
         [ 3384.26935016, 24705.29679433, 14178.58389369],
         [ 1890.87584391, 14178.58389369, 20135.08181487]]]), np.array([[[ 4.14870120e+03, -1.10908710e+02, 2.58637908e+03],
         [-1.10908710e+02,  2.32179306e+04,  2.67643706e+04],
```

```
       [ 2.58637908e+03,  2.67643706e+04,  6.38027338e+04]],

      [[ 8.70669388e+03,  1.68571020e+04,  0.00000000e+00],
       [ 1.68571020e+04,  1.11098204e+05,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  1.00000000e-06]],

      [[ 1.15574513e+03, -1.02499934e+03,  9.59208469e+02],
       [-1.02499934e+03,  1.52986274e+04,  1.10222643e+04],
       [ 9.59208469e+02,  1.10222643e+04,  2.38529713e+04]]]), np.array([[[1.22496000e+03, 5.01640000e+02, 0.00000000e+00],
       [5.01640000e+02, 2.61147822e+04, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 1.00000000e-06]],

      [[7.98239947e+02, 1.60575276e+03, 2.36221169e+03],
       [1.60575276e+03, 1.86437211e+04, 1.52715198e+04],
       [2.36221169e+03, 1.52715198e+04, 2.17513952e+04]],

      [[2.05263735e+03, 3.04792969e+02, 2.52678022e+03],
       [3.04792969e+02, 2.20748894e+04, 2.32231566e+04],
       [2.52678022e+03, 2.32231566e+04, 4.44530652e+04]]]), np.array([[[ 1.53725767e+03, -6.73823898e+02, -3.84864447e+00],
       [-6.73823898e+02,  2.09714226e+04,  1.10376072e+04],
       [-3.84864447e+00,  1.10376072e+04,  1.47132403e+04]],

      [[ 1.41208731e+03, -9.76946653e+02,  1.94470675e+03],
       [-9.76946653e+02,  2.20246127e+04,  1.04716517e+04],
       [ 1.94470675e+03,  1.04716517e+04,  2.87723156e+04]],

      [[ 8.93764488e+02,  3.64355976e+02,  1.02738933e+03],
       [ 3.64355976e+02,  8.63314171e+03,  3.88389672e+03],
       [ 1.02738933e+03,  3.88389672e+03,  8.96035799e+03]]]), np.array([[[2.91447988e+03, 8.89922099e+03, 5.34961856e+03],
       [8.89922099e+03, 5.25825019e+04, 2.87321152e+04],
       [5.34961856e+03, 2.87321152e+04, 5.20069649e+04]],

      [[1.57047222e+03, 3.88322222e+03, 0.00000000e+00],
       [3.88322222e+03, 7.20100556e+04, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 1.00000000e-06]],

      [[2.09386665e+03, 7.79028994e+02, 1.28743945e+03],
```

```
       [7.79028994e+02, 4.50524123e+04, 3.48349093e+04],
       [1.28743945e+03, 3.48349093e+04, 7.39619233e+04]]]), np.array([[[  5519.23899179,   7683.02550953,  10993.98106071],
       [  7683.02550953,  16957.39356566,  17878.1789828 ],
       [ 10993.98106071,  17878.1789828 ,  65006.31758315]],

      [[  6247.6701149 , -29037.7625754 ,  12945.09785632],
       [-29037.7625754 , 175114.09283479, -63273.03051344],
       [ 12945.09785632, -63273.03051344,  27062.55374311]],

      [[  6506.82470648,  13233.10981731,   1307.32410564],
       [ 13233.10981731,  37360.29214959,  -4138.92225431],
       [  1307.32410564,  -4138.92225431,  25383.85300847]]]), np.array([[[ 1.13723679e+03,  2.54386802e+02,  3.58921784e+02],
       [ 2.54386802e+02,  1.76711084e+04, -6.31352675e+02],
       [ 3.58921784e+02, -6.31352675e+02,  2.53861268e+04]],

      [[ 1.96153454e+03,  5.15342663e+01,  2.77208070e+02],
       [ 5.15342663e+01,  1.95771649e+04,  5.33820480e+03],
       [ 2.77208070e+02,  5.33820480e+03,  5.20162678e+04]],

      [[ 8.25548318e+02,  5.61140812e+02,  1.45284318e+03],
       [ 5.61140812e+02,  7.70925581e+03, -6.54115542e+02],
       [ 1.45284318e+03, -6.54115542e+02,  2.63291774e+04]]]), np.array([[[   9329.63043804,   12056.71749279,   21241.83210559],
       [  12056.71749279,   24811.39969442,   31088.84942766],
       [  21241.83210559,   31088.84942766,   73997.57242723]],

      [[   3149.81967455,    2455.50609417,    3324.43968185],
       [   2455.50609417,   17107.44823439,    6241.90226669],
       [   3324.43968185,    6241.90226669,   30036.48284129]],

      [[    184.109271  ,   -3592.09946463,    2790.4613278 ],
```

```
         [  -3592.09946463,  146080.31323881, -111595.51392397],
         [   2790.4613278 , -111595.51392397,  86372.58932112]]]), np.array([[[  1062.03278302,  1039.60646605,   -323.71420675],
         [  1039.60646605,   8200.51092494,  -7218.79681607],
         [  -323.71420675,  -7218.79681607, 181832.24024718]],

        [[  2578.85449378,   3399.58429965,  -2905.86536366],
         [  3399.58429965, 101117.15308826,  17086.96228468],
         [ -2905.86536366,  17086.96228468,  36170.58048122]],

        [[  1493.04288264,  -1389.4011042 ,   5200.85014983],
         [ -1389.4011042 ,  59428.59553534,   2438.17739549],
         [  5200.85014983,   2438.17739549,  51151.56459485]]])]

c =  [np.array([0.30299046, 0.03597122, 0.66103832]), np.array([0.59836797, 0.02158273, 0.3800493 ]), np.array([0.02158273, 0.00719424, 0.97122302]), np.array([0.53010816, 0.29681583, 0.173076  ]), np.array([0.63376144, 0.05035971, 0.31587885]), np.array([0.10791367, 0.48575534, 0.40633099]), np.array([0.3447646 , 0.34337512, 0.31186028]), np.array([0.48832128, 0.08633094, 0.42534778]), np.array([0.78104841, 0.02597984, 0.19297175]), np.array([0.27265903, 0.30899896, 0.41834201]), np.array([0.69816084, 0.24564924, 0.05618992]), np.array([0.32399005, 0.21180813, 0.46420182])]

def find_class_index(name):
    vowel = name[3:5]
    index = classes.index(vowel)
    return index


def find_person_index(name):
    person = name[0]
    index = people.index(person)
    return index


def gauss(xk, sigma0, mu0):
```

```python
    det_sigma0 = np.linalg.det(sigma0)
    inv_sigma0 = np.linalg.inv(sigma0)
    #print(inv_sigma0)
    diff = xk - mu0
    #print(xk, mu0, diff)
    exp1 = np.dot(diff.T, inv_sigma0)
    #print(exp1)
    eksponent = (-1/2)*np.dot(exp1, diff)
    #print(eksponent)
    p = np.exp(eksponent)/np.sqrt(((2*np.pi)**3)*det_sigma0)
    #print(p, '\n')
    return p


def find_class(xk):
    global mu
    global sigma
    global c
    global C
    xk = np.array(xk)
    prob =[]
    for i in range(C):
        mui = mu[i]
        ci = c[i]
        sigmai = sigma[i]
        p = 0
        for i in range(GMM_number):
            mu0 = np.array(mui[i])
            sigma0 = np.array(sigmai[i])
            p += gauss(xk, sigma0, mu0)*ci[i]
        prob.append(p)
    return np.argmax(prob)


def find_dig_cov(vec):
    ide = np.identity(3)
    return np.multiply(vec, ide)

def normalize(vec):
    vec_new = []
    for i in range(len(vec)):
        vec_new.append(vec[i]/np.max(vec))
    return np.array(vec_new)
```

```

def opptak(duration):
    global Fs
    print('Spiller inn lyd..')
    rec = sd.rec( int(duration*Fs),samplerate=Fs, channels=2)
    time.sleep(duration/4)
    print('.')
    time.sleep(duration/4)
    print('.')
    time.sleep(duration/4)
    print('.')
    time.sleep(duration/4)
    print('Prosessere...')
    rec_ny = []
    for i in  range( len(rec)):
        rec_ny.append(rec[i][0])
    return normalize(rec_ny)

def fourier(vector):
    sp = np.fft.fft(vector)
    freq = np.fft.fftfreq(vector.shape[-1])
    return  abs(sp.real[0: int( len(sp)/2)]), freq[0: int( len(sp)/2)]

def find_peaks(vec):
    ak = pysptk.sptk.lpc((vec), order=16)
    ak[0]  = 1
    w, h = signal.freqz([1], ak,  fs = Fs)
    h = normalize( abs(h.real))
    peaks = signal.find_peaks(h, height = 0.0065, distance = 20 )
    freq_peaks =  list(w[peaks[0]])
    if  len(freq_peaks)==3:
        return freq_peaks
    elif  len(freq_peaks)>3:
        return freq_peaks[0:3]
    else:
        while( len(freq_peaks)<3):
            freq_peaks.append(0)
        return freq_peaks

if __name__ == "__main__":
    true =  input('Trykk hvasomhelst for   komme igang')
```

```
    while(true not in avslag):
        rec = opptak(1)
        xk = find_peaks(np.array(rec))
        class_rec = find_class(xk)
        print('Vokal detektert:', classes[class_rec])
        true =  input('Vil du pr ve igjen?')
```