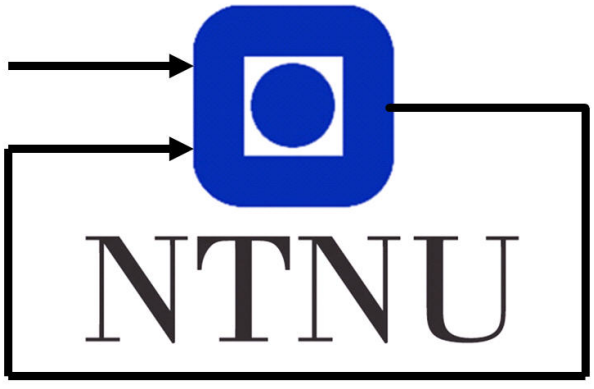
Classification speech lab rebort
TTT4275 Estimation, detection and classification

Saim Naveed Iqbal - 544916
Steve Carter Feujo Nomeny - 544963

April 30, 2023



Department of Engineering Cybernetics

Abstract

In this report, the results from using a linear classifier on the IRIS dataset will be presented and discussed. Furthermore, a dataset with vowels has been classified using a MAP-classifier, both when assuming a single Gaussian, and with Gaussian Mixture Models. These results will also be presented and discussed.

For the IRIS dataset, there was no observed performance difference when different samples for the training dataset were used. Moreover, selecting the features was also analyzed. This revealed that the elimination of the most overlapping features resulted in a higher required step length (learning rate), α, and required more iterations on W to attain comparable error rates. Thus, it appears that the removal of such features only results in suboptimal performance. However, it should be acknowledged that the dataset employed in the analysis was comparatively small. Hence, these findings cannot be extrapolated to broader populations.

For the vowels we observed that the error rate generally was better when using Gaussian Mixture Models, than when using a single Gaussian model. Using a full covariance matrix in the single Gaussian case performed better than using a diagonal covariance matrix. Since the GMMs used a diagonal covariance matrix, the improvement from a single Gaussian with a full covariance matrix was not very large, showing the significance of the covariances between the features.

Contents

1 Introduction

This project has goals of designing and being able to analyze the strength and weaknesses of two different classifiers. The first classifier aims to classify three types of Iris flowers based on the length and width of their petals and sepals, while the second classifier seeks to classify twelve vowels based on three extracted features called formants.

For the Iris part, the Iris flowers should be classified into 3 different classes called respectively Setosa, Versicolor, and Virginica. The flowers in question possess both large (sepal) and small (petal) leaves. The three aforementioned varieties can be distinguished based on the varying lengths and widths of the petals and sepals. Therefore, these four measurements constitute a reasonable choice for input features. The Iris task is among the few real-world problems that are nearly linearly separable [1]. The classification of these flowers is of general interest to society for many reasons. Understanding the distribution and characteristics of different species of plants is important for monitoring the health and diversity of ecosystems. Iris classification can help researchers and environmentalists in their efforts to protect and conserve natural habitats. Furthermore, the ability to accurately classify different varieties of Iris flowers can have applications in the development of new medicines and treatments. Certain compounds found in these flowers have been shown to have medicinal properties and could be used in the treatment of various diseases [2].

For the Vowels part, a set of twelve vowels should be classified based on three extracted features, so-called formants, taken from each sample. The formants correspond to three frequencies where the sound exhibits peaks in the frequency spectrum. These formant frequencies typically vary from one vowel to another, but there is also a significant degree of variation within each class. Therefore, we are dealing with a classical non-separable problem and a linear classifier will not work here [1]. This type of classification does also bring benefits to society, for instance in communication. The ability to accurately recognize and classify vowels is crucial for effective communication, particularly in fields such as telecommunications, broadcasting, and public speaking. The classification of vowels could also be of interest to researchers in fields such as acoustics, speech pathology, and phonetics, who are interested in understanding the mechanisms and properties of human speech.

All in all, classification algorithms are important as they can outperform humans in certain tasks, particularly those that require processing large amounts of data quickly and accurately. For example, in medical diagnosis, machine learning algorithms can analyze vast amounts of patient data to identify patterns that human doctors might miss, leading to earlier diagnoses and more effective treatments. In finance, classification algorithms can analyze market trends and make predictions about future performance, enabling traders to make more informed investment decisions.

The report is divided into two parts, each presenting the relevant theory, task description, results, and implementation of the classifier. The conclusion summarizes the project and highlights the key points of discussion.

2 Theory

2.1 Different types of classifiers

We can categorize classification problems into three main categories: linear-, nonlinear-, and nonseparable. Consider a two-dimensional feature space with two classes, denoted as ω_1 and ω_2. The simplest type of problem is one that is linearly separable, where the two classes can be perfectly separated without using a simple line. If the input dimension was three the decision boundary would be a plane, and in higher dimensions, a hyperplane. It's important to note that there are generally an infinite amount of lines/planes/hyperplanes that can achieve error-free decision. However, a common intuitive and robust approach is to choose the decision boundary with the same minimum distance to both classes. If there are more than two classes, a corresponding decision boundary is needed for each pair of classes. A classifier that utilizes linear decision boundaries is referred to as a linear classifier. Although linear classifiers may not always achieve the best performance, they are usually simple to design and apply [3].

If a problem is nonlinearly separable it is still possible to find an error-free decision boundary, but not one that is a hyperplane. In this case we use a nonlinear classifier. For nonseparable problems, there is no decision boundary that can separate the classes with zero error, as the classes overlap. In practical applications, most problems are nonseparable, and the choice of whether to use a linear or nonlinear classifier depends on the trade-off between complexity and performance. If the resulting error rate is too high for practical use, an alternative strategy is to search for better features or sensors that may result in a more separable problem.

2.2 Training a linear classifier

A linear classifier (also known as a discriminant classifier) involves describing each class with a discriminant function $g_i(x)$ and using a decision rule to make a classification based on the values of these (discriminant) functions. The decision is given by

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

and the linear discriminant classifier is defined by the function:

$$g_i(x) = \omega_i^T x + \omega_{io} \quad i = 1, ..., C \tag{2}$$

where ω_{io} is an offset for the class ω_i, and C is the number of classes [3].

A compact matrix form can be used for $C > 2$, $\boldsymbol{g} = W\boldsymbol{x} + \boldsymbol{\omega}_0$ with $\boldsymbol{g}$ and $\boldsymbol{\omega}_0$ being vectors of dimension $C \times 1$ and the weight matrix W has dimension $C \times D$ where D is the number of features. Alternatively, one can augment Equation 2 to $\boldsymbol{g} = W\boldsymbol{x}$ where $[\omega_0 \quad W] \rightarrow W$ and $[x_0 \quad \boldsymbol{x}] \rightarrow \boldsymbol{x}$ [3].

A criterion utilized to train the linear classifier is the Minimum Square Error (MSE) optimization. The MSE of the classifier is given by

$$\text{MSE} = \frac{1}{2} \sum_{k=0}^{N-1} (\boldsymbol{g}_k - \boldsymbol{t}_k)^T (\boldsymbol{g}_k - \boldsymbol{t}_k) \tag{3}$$

where $\boldsymbol{t}_k$ are the target values at the output of the filter. The elements in $\boldsymbol{t}_k$ are 1 when the class is correct and 0 otherwise. For instance, the target value for a sample in class 1 would be $\boldsymbol{t}_1 = [1 \ 0 \ ... \ 0]^T$.

The classifier and training technique are both named discriminative because they focus on directly discriminating between the classes based on the training data. In the case of the total matrix W, the training is performed on the entire training set for all classes simultaneously, without considering any underlying probabilistic model. This approach contrasts with generative

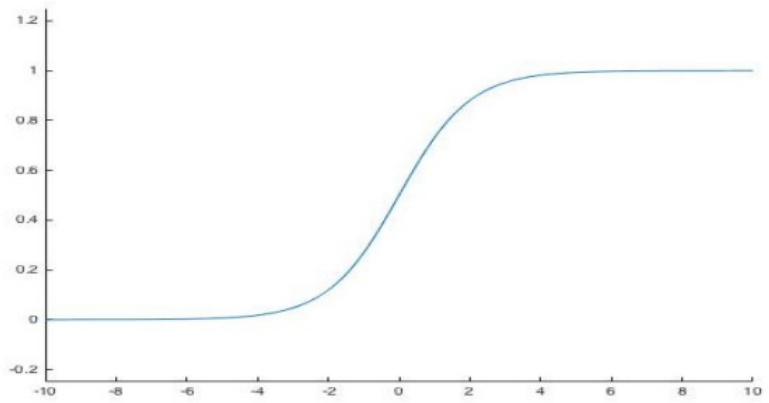


Figure 1: Sigmoid function

classifiers, which aim to model the underlying probability distribution of each class separately and then use these models to classify new data points [3].

To map the continuous-valued vector$\boldsymbol{g}_k = W\boldsymbol{x}_k$ to the target vector with binary values 0, 1, a smooth function with a derivative is needed instead of the ideal Heavyside function. A suitable approximation for the Heavyside function is the sigmoid function (as shown in Figure 1), which is also known as a squashing function [3]

$$s(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

Using this equation, we can smooth out the output of the linear classifier

$$z_k = Wx_k \tag{5}$$

by passing the output through the sigmoid. We then result with

$$g_k = \frac{1}{1 + e^{-Wx_k}} \tag{6}$$

An explicit solution to Equation 3 does not exist at we have to make use of a gradient-based technique, i.e. update the W matrix in the opposite direction of the gradient with a small step factor α

$$W(m) = W(m-1) - \alpha \nabla_W \text{MSE} \tag{7}$$

with the iteration number m [3].

Using the chain rule for gradients, the gradient of Equation 3 can be easily found to be

$$\nabla_W MSE = \sum_{k=0}^{N-1} \nabla_{\boldsymbol{g}_k} MSE \nabla_{\boldsymbol{z}_k} \boldsymbol{g}_k \nabla_W \boldsymbol{z}_k \tag{8}$$

where

$$\begin{aligned} \nabla_{\boldsymbol{g}_k} MSE &= \boldsymbol{g}_k - \boldsymbol{t}_k \\ \nabla_{\boldsymbol{z}_k} \boldsymbol{g} &= \boldsymbol{g}_k \circ (1 - \boldsymbol{g}_k) \\ \nabla_W \boldsymbol{z}_k &= \boldsymbol{x}_k^T \end{aligned}$$

and $\boldsymbol{g}_k \circ (\boldsymbol{1} - \boldsymbol{g}_k)$ means elementwise multiplication.

Careful selection of the step factor α is crucial, as a value that is too large can lead to large fluctuations in the mean squared error (MSE) and large changes in the weight matrix W, while a value that is too small may require a large number of iterations to reach the minimum MSE [3].

2.3 Maximum Likelihood (ML) based training for nonlinear classifier

ML-based training is used to train classifiers based on *Bayes Decision Rule (BDR)*. Bayes Decision Rule is written as

$$x \in \omega_j \Leftrightarrow p(x|\omega_j)P(\omega_j) = \max_i p(x|\omega_i)P(\omega_i) \tag{9}$$

It can be shown that this descision rule leads to a theoretically optimal classifier [3]. However, this is only in the case when the probabilities are known, which is rarely true in a real world scenario.

What one does instead is choose a parametric form, and then estimate the parameters. A common choice for the parametric form is the Gaussian, which makes estimating the parameters simple. The parametric form becomes:

$$p(x|\omega_i) = \frac{1}{\sqrt{(2\pi)^D|\Sigma_i|}} \exp\left[-\frac{1}{2}(x-\mu_i)^\top \Sigma_i^{-1}(x-\mu_i)\right], \quad i = 1, \ldots, C \tag{10}$$

Where we need to estimate μ_i and Σ_i. This is done my maximizing likelihood, which is where we get the name ML-based training. Given a training subset $X_{N_i} = \{x_{i1}, \ldots, x_{Ni}\}$, where all inputs are independent and belong to the same class ω_i. We then define the likelihood of the subset as the product of the class conditioned densities:

$$\mathcal{L}[X_{N_i}, \{\mu_i, \Sigma_i\}] = \prod_{k=1}^{N_i} p(x_{ik}|\{\mu_i, \Sigma_i\}) \tag{11}$$

Since the densities are chosen to be Gaussian, this simplifies if we take the logarithm. We then wish to maximize this log-likelihood function, which is done by taking the gradient.

$$\nabla_{\{\mu_i,\Sigma_i\}} \mathcal{LL}(X_{Ni}, \{\mu_i, \Sigma_i\}) = \sum_{k=1}^{N} \nabla_{\{\mu_i,\Sigma_i\}} \log[p(x_{ik}|\{\mu_i, \Sigma_i\})] = 0 \tag{12}$$

Inserting the equations for the Gaussian case finally gives us our estimators:

$$\hat{\mu}_i = \frac{1}{N_i} \sum_{k=1}^{N_i} x_{ik} \tag{13}$$

$$\hat{\Sigma}_i = \frac{1}{N_i} \sum_{k=1}^{N_i} (x_{ik} - \hat{\mu}_i)(x_{ik} - \hat{\mu}_i)^T \tag{14}$$

Which are the sample mean and sample covariance matrix.

We often can't reasonably assume that the densities can be modeled as a single Gaussian. The true densities are typically complex, and have multiple peaks. We can model this as a weighted sum of Gaussians. This model is called a *Gaussian Mixture Model (GMM)*, and looks as such:

$$p(x|\omega_i) = \sum_{k=1}^{M_i} c_{ik}\mathcal{N}(\mu_{ik}, \Sigma_{ik}) = \sum_{k=1}^{M_i} \frac{c_{ik}}{\sqrt{(2\pi)^D|\Sigma_{ik}|}} \exp\left[-\frac{1}{2}(x-\mu_{ik})^\top \Sigma_{ik}^{-1}(x-\mu_{ik})\right] \tag{15}$$

It is harder to use ML-based training in this instance, as we end up with the logarithm of a sum. This is solved by using an iterative algorithm called *Expectation Maximation*. In practice however, this problem is solved using built-in functions in MATLAB.

Finally, the prior probabilities $P(\omega_i)$ must be estimated as well, however this is easier, and not as critical. We will simply assume $P(\omega_i) = 1/C$ where C is the amount of classes.

3 IRIS dataset and linear classification

3.1 The Task

The objective of this task is to develop a linear classifier for the Iris dataset and investigate how modifications to the training set and features affect the classifier's performance. Initially, we will train the classifier using all available features and optimize the step factor, α. The performance will be evaluated in terms of error rate and confusion matrix. Next, we will select different samples for training and compare the performance of the two training sets. Finally, we will investigate how the classifier's performance is influenced by the linear separability of the features. Specifically, we will remove one least linearly separable feature from the dataset at a time and assess the resulting impact on performance.

3.2 Implementation and results

The linear classifier was implemented in Matlab, as it is a powerful tool for optimization algorithms such as gradient descent. Code is shown in appendix A.1.

Class name	Vector representation t_k
Iris Setosa	$[1\ 0\ 0]^\top$
Iris Versicolor	$[0\ 1\ 0]^\top$
Iris Virginica	$[0\ 0\ 1]^\top$

Figure 2: Target values for the different classes.

The different target values chosen are listed in Figure 2. The dataset is loaded from a CSV file where each column describes one of the features of the specific example. A histogram showing the distribution of the features of the Iris flowers is shown in Figure 3.

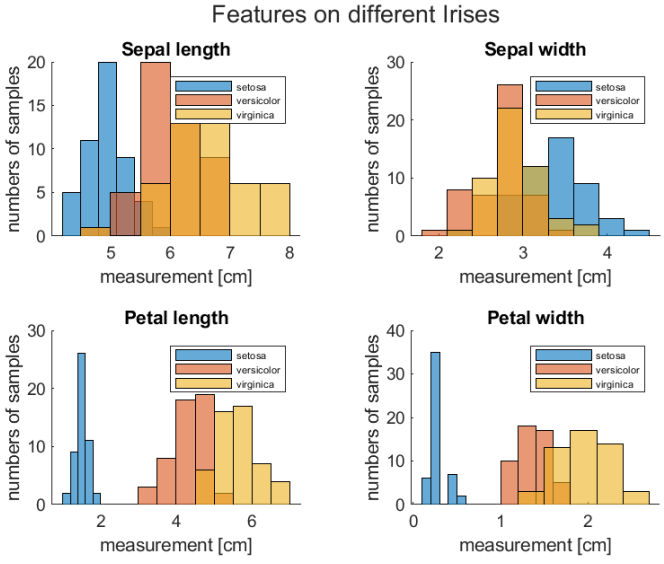


Figure 3: Histogram with the distribution of features in example set

Thereafter, it is divided into a training set and a test set, after which it is augmented as described in subsection 2.2. The augmented weight matrix W is now trained using the gradient on all the samples in the training set. Here different values for α and max iteration have been used.

For each alpha that is tested, the gradient of the MSE is computed together with the mean square error and error rate of that current iteration. The formulas and functions follows strictly from the equations in subsection 2.2

As mentioned in the theory, the choice of α has great importance for the performance of the classifier. The first 30 samples out of the 50 available were used for training, and the rest for testing. In this case, different alpha was tested out in order to evaluate the performance. The alpha values given by

$$\alpha \in (0.01,\ \ 0.0075,\ \ 0.0025,\ \ 0.005) \tag{16}$$

generated the MSE per iteration shown in Figure 4. From this figure, it is clear that for values of $\alpha \geq 0.0075$, the mean squared error (MSE) fluctuates as a function of the number of iterations. Oscillation is an indication that the minimum value is exceeded and not steadily approached. These αs' are too large. On contrary, $\alpha \leq 0.0025$ makes the MSE converge very slowly. using these αs' would require the number of iterations to go increase in order to reach a minimum. This yields $\alpha = 0.005$ as a good value for alpha as the MSE converges fast without oscillating.

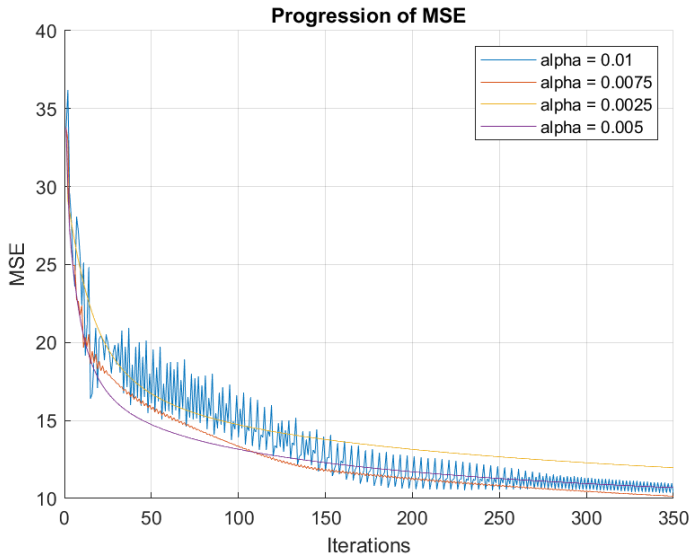


Figure 4: MSE per iteration with training set

After setting alpha, the number of iterations in the algorithm has to be decided. This is not intuitively given by Equation 7, but a possible solution is to calculate the error rate per iteration and note when it converges to a value. This is shown in Figure 5. We can observe that the error rate is constant after ≈ 350 iterations whereafter there are not many observed improvements. It is important to note that although the discrete error rate may not decrease further, the true error rate may still continue to decrease. This is because the discrete error rate is bounded by the number of samples and can only decrease by a discrete number of samples. However, for larger datasets, this may not be the case and may not necessarily lead to significantly better performance.

Having found good values for iteration number and α, the linear classifier was trained. The error rate was 5% for the test set and 2.2% for the training set. Furthermore, a confusion matrix was created to compare the true classes with the classification from the linear classifier. The classifier was tested on the training and test set, and the results are shown in Figure 6.

From the confusion matrix we see that the classifier manages to seperate all the Iris setosa, but fails to classify som of the samples in class 2 and 3 (Iris versicolor and virginica). This can be explained by the linear seperability in the petal features between class setosa and the two other classes which is clearly visible in Figure 3. One also observe from the figure that the

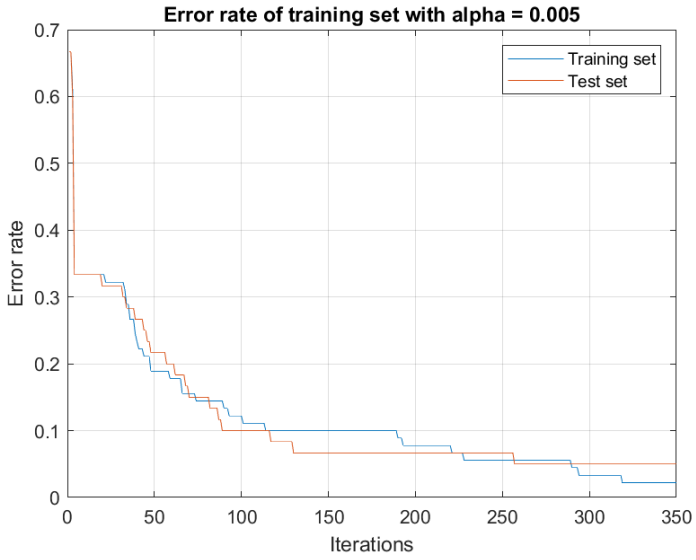
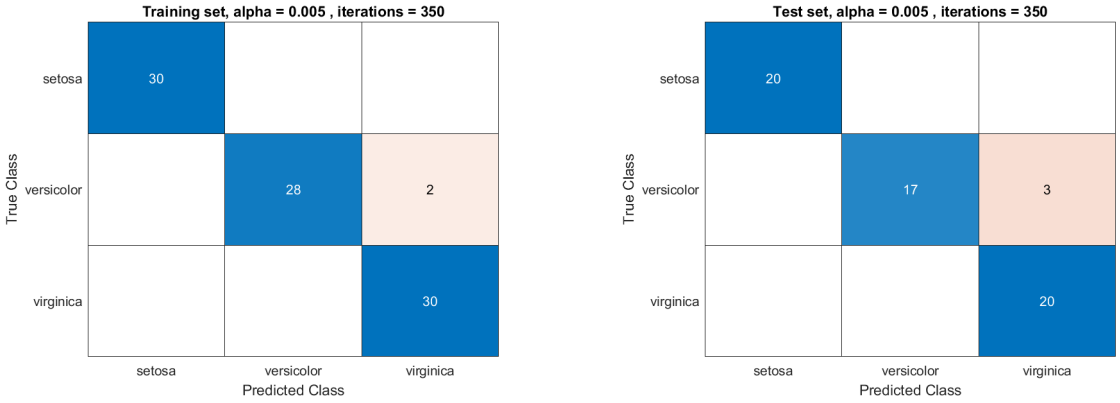


Figure 5: MSE per iteration with training set



(a) Training data set

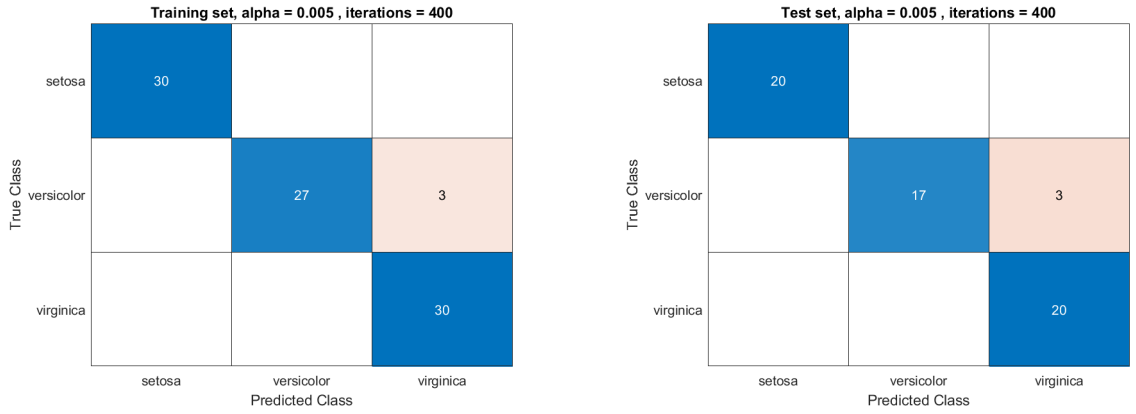
(b) Test data set

Figure 6: Confusion matrices for test and training set with first 30 samples

classes Iris Versicolor and Virginica are not linearly seperabe and thus it makes sense that the linear classifier misclassifies some of these samples in Versicoler as Virginica. This then yields the 5% error rate.

Moving onward, the selection of training set is changed to evaluate how that affects the performance of the classifier. The last 30 samples where used for training, and the first 20 where used for testing. Using the same method as described earlier to find appropriate α and number of iteration, the appropriate values found were $\alpha = 0.005$ and 400 for the number of iterations (this in order for the error rate to approach its minimum). Testing the trained linear classifier yielded an error rate of 1.7% on the test set and 5.6% on the training set. This seems to be a huge improvement from before, however, looking at the confusion matrix in Figure 7, we observe that the classifier still misclassifies 3 samples in Versicoler as Virginica in the training set. This is also the case in the training set. The performance of this trained classifier is thus similar to the previous one. Therefore, the choice of training samples does not have any significant impact on the overall performance of the classifier in this case.

By removing the least linearly separable feature, we may expect the classifier performance to improve slightly, as the feature was contributing to the overlap between the classes. However, it



(a) Training data set

(b) Test data set

Figure 7: Confusion matrices for test and training set with last 30 samples

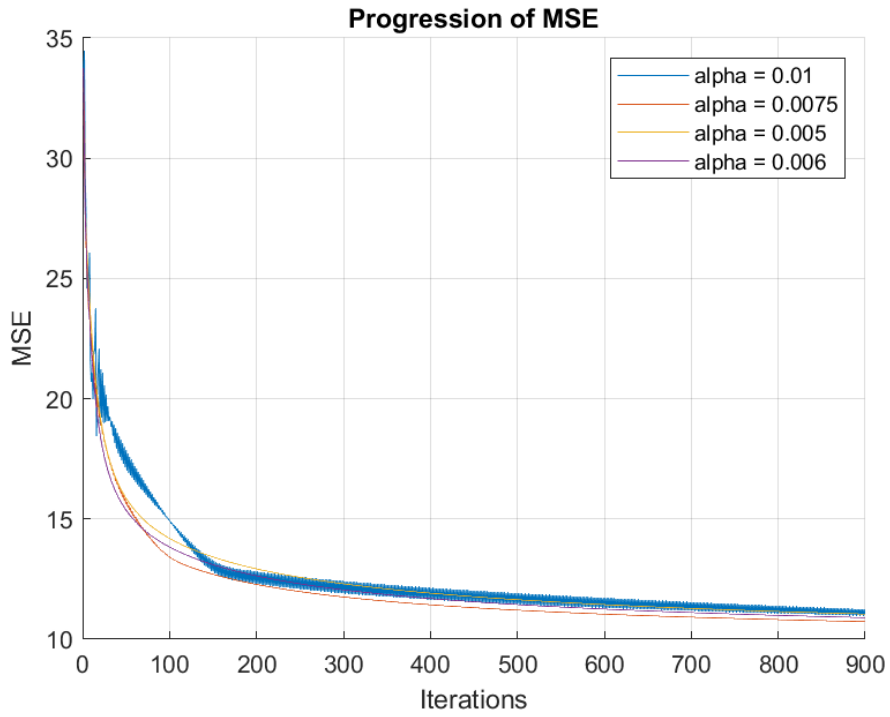
is also possible that removing a feature could cause the classifier to miss important information and result in worse performance. By evaluating the features in Figure 3, one can observe that the Sepal width and length had features overlapped the most and therefore had the least linear separability. However the Sepal width had the most overlap and was hence chosen to be removed.

The same training process as before was repeated, but this time excluding the Sepal width feature from the feature set. For this, and the coming, tests, the first 30 samples were used. From Figure 8, one can observe that a slightly higher step factor ($\alpha = 0.006$) is required for the training process. With this higher α one would think that the error rate would converge faster. However, the error rate now converges more slowly, requiring $\approx$ 900 iterations to converge. This is because the Sepal width feature has a small variance and contributes to a larger partial derivative, making the MSE converge faster when included in the training.

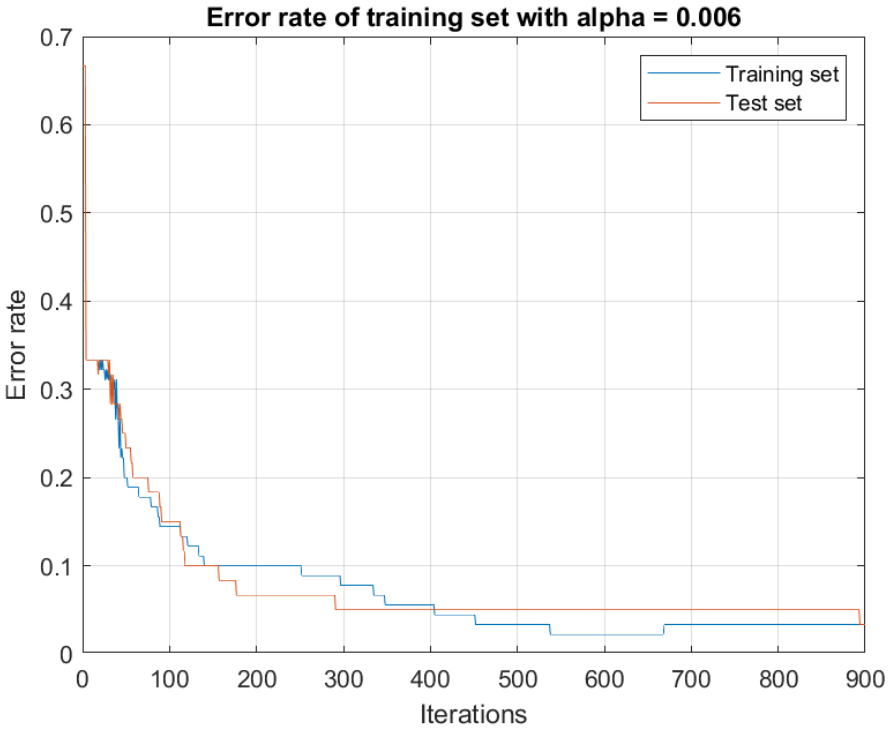
The resulting classifier can then be evaluated on the same test set and compared to the performance of the classifier trained on all features. The confusion matrix visualized in Figure 9 shows the classifier's results. Although the total number of misclassification is 5, as in the case when all features were considered (see Figure 6), we observe that the number of misclassification for the test set has reduced with one. Furthermore, the error rate for both the test set and the training set was found to be 3.33%. By removing a feature that was not linearly separable, the classifier becomes more efficient and accurate. The removed feature was causing more errors and confusion, so its absence has resulted in better performance, as expected.

Removing the second most overlapping feature, the Sepal length, left us with the two features Petal width and length. Figure 10 shows the results of analyzing the MSE and error rate for the modified dataset, following the same procedure as before. α is chosen to be 0.15 as this provides the fastest convergence without oscillation. However, the convergence of the error rate is far slower. One can observe from the figure that it takes $\approx$ 6500 iterations before it actually converges. This suggests that the error rate is likely to decrease after removing the least linearly separable features and increasing the number of iterations, as both of these changes should result in better classification performance. However, this is not necessarily the case. From Figure 11 we can see that the total number of misclassification is 6, but the error rate for the test set is 3.33% and 4.44%. This is a bit worse performance than for the linear classifier where only the Sepal width was removed.

The absence of a trend in decreasing error rates clearly proves that removing the nonlinear features does not lead to lower error rates. Instead, removing the features only increases the number of computations required by a considerable amount. To achieve comparable results, we now have to perform a significantly larger number of iterations, around 6500, compared to the 300-400 iterations needed when using the full feature set.



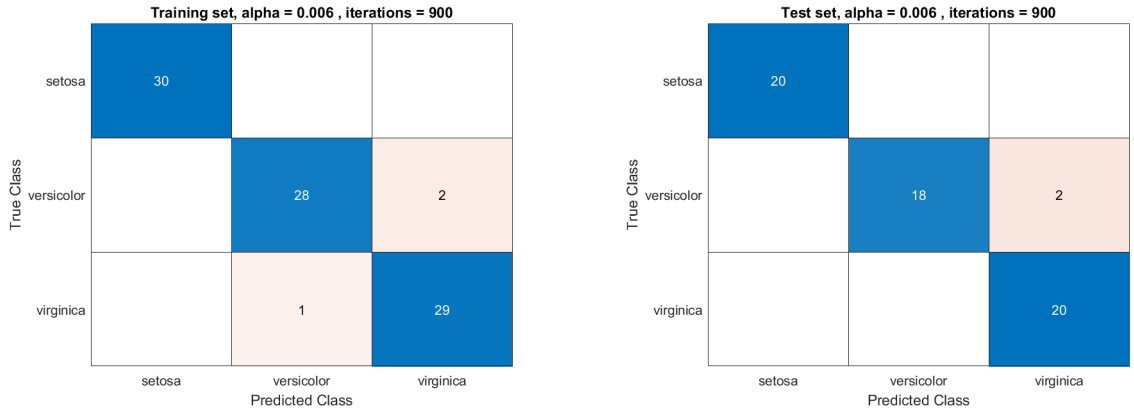
(a) Training data set



(b) Test data set

Figure 8: The variation of Mean Square Error (MSE) and error rate with the number of iterations on W for different values of α, when the dataset is without the Sepal width feature.

Taking it one step further, the Petal width was removed as the feature that was most overlapping out of Petal width and length. Removing this feature left us with a choice of $\alpha = 0.1$ and an iteration number of 5000 for convergence of the error rate. However, as the confusion

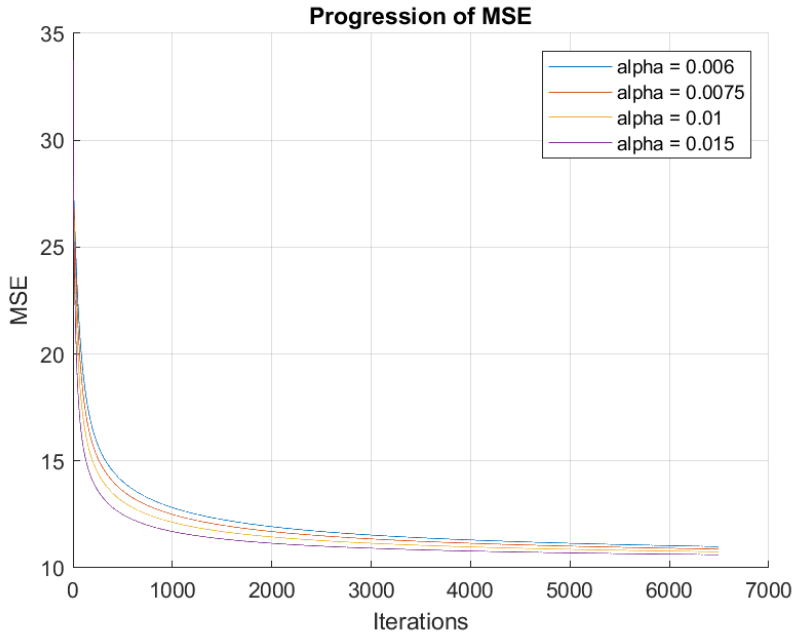


(a) Training data set

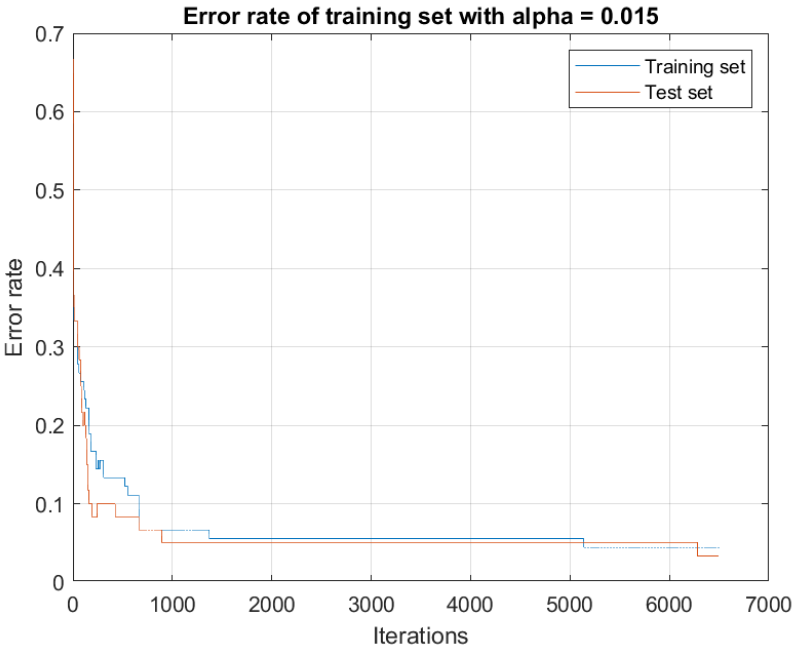
(b) Test data set

Figure 9: Confusion matrices for test and training set with first 30 samples and Sepal width feature removed.

matrix in Figure 12 shows, the total number of misclassification was high. The error rate for the test set was 1.67% and 6.67% for the training set. This might seem like an improvement, however, the high number of misclassifications in the training set implies otherwise. With only one sample, important and distinguishing information about the classes is lost, degrading the performance of the linear classifier.

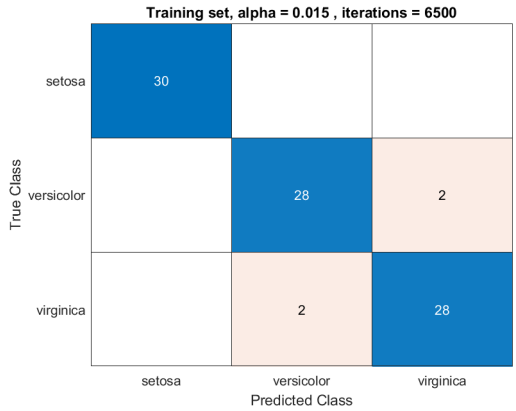


(a) Training data set

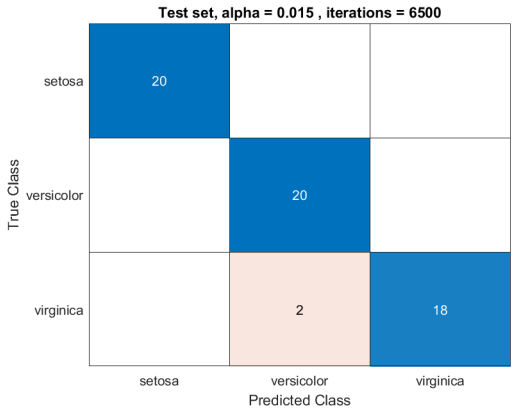


(b) Test data set

Figure 10: The variation of Mean Square Error (MSE) and error rate with the number of iterations on W for different values of α, when the dataset is without the Sepal width and length feature.

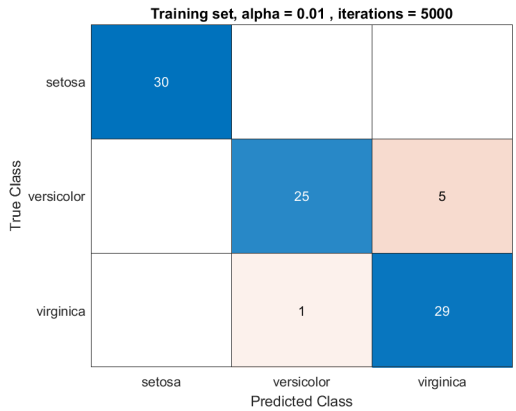


(a) Training data set

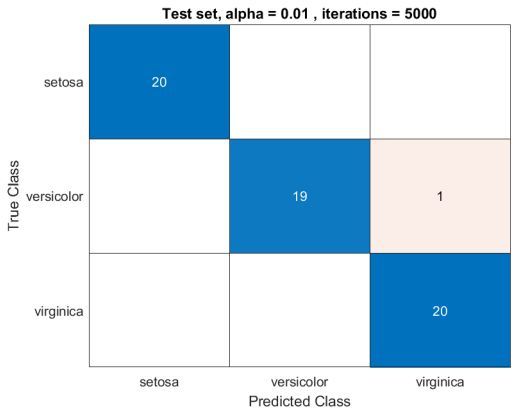


(b) Test data set

Figure 11: Confusion matrices for test and training set with first 30 samples and Sepal width and length features removed.



(a) Training data set



(b) Test data set

Figure 12: Confusion matrices for test and training set with first 30 samples only Petal length feature.

4 Classification of pronounced vowels using a MAP-classifier

4.1 The Task

The objective of the task is to analyze and compare different Gaussian models for classification. The first part assumes a single Gaussian model for the classes. We calculate the sample mean and covariance matrices, design a classifier, and compare confusion matrices and error rates for both diagonal and full covariance matrices.

In the second part, a mixture of 2-3 Gaussians is used for each class and Gaussian mixture models (GMM) with diagonal covariance matrices are found. A modified classifier from the previous part is used to deal with mixtures of Gaussians, and confusion matrices and error rates are found for GMM classifiers with M=2 and M=3 Gaussians per class. The performance of all four model types is compared, and the difference in performance is examined for each class.

4.2 Implementation

The vowel dataset contains the formants for each soundclip at 20%, 50% and 80% of the duration of the clip, as well as the fundamental and steady-state formants. Most soundclips have three clear formants in the frequency spectrum. In this task we use the steady-state formants *F1*, *F2*, and *F3* as features. In figure 13 is a histogram of these formants for the vowel *ei*. As we can see, the distribution of these formants are complex, and while we can parameterise them with a single gaussian, it won't be accurate. We observe that the histograms have 2 to 3 peaks, which is why GMMs will also be tested. The dataset contains vowels from four classes of speakers, men, women, boys and girls. This contributes to the complex distribution of the formants.

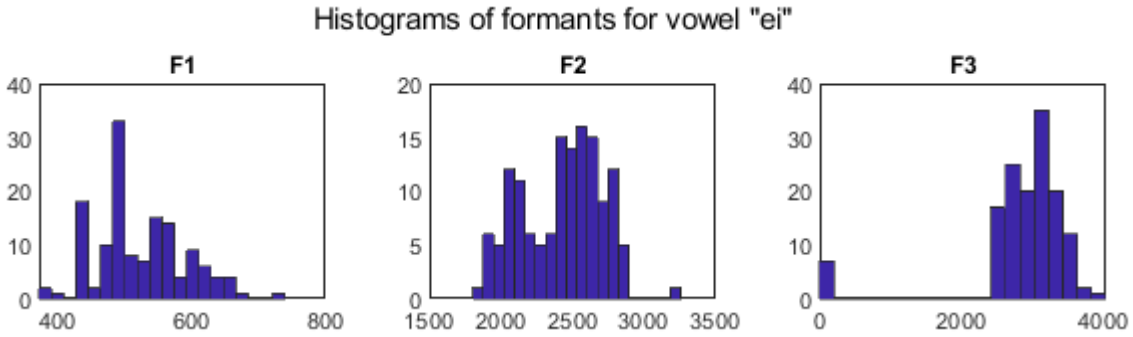


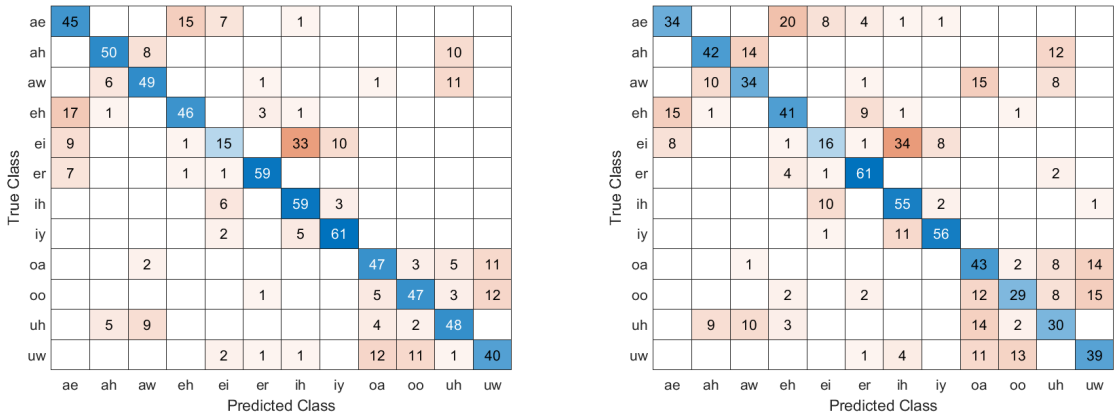
Figure 13: Formants F1, F2, F3 for all speakers

We begin by assuming a single Gaussian model. Similarily to the iris task, split the dataset in two, such that half of each class is used for testing and the other half for training. The training dataset is then used to calculate class means, and class covariances. We also create diagonal covariance matrices by removing non-diagonal values. Code for this is shown in appendix A.2. The implementation of the classifier itself is very simple. For each sample, we calculate the likelihood of it beloninging to each class. Then the class with the highest A posteriori likelihood is chosen.

Next, we assume 2-GMM and 3-GMM respectively. To do this, we use the MATLAB function `GMM = fitgmdist(v, M);` to create a GMM with M mixtures, from the training vector v. We do this for each class. We then do the same as in the single Gaussian case, by extracting the probability density function from the GMM, and calculating the likelihood of each sample belonging in each class. Then the class with the highest A posteriori likelihood is chosen. This is done first with a mixture of 2 Gaussians, and then 3 Gaussians.

4.3 Results

In the case of the single Gaussian, we plot the confusion chart for both full and diagonal covariance matrices. These are shown in figure 14. We see that the with a full covariance matrix, the prediction is generally more accurate. This is reflected in the error rates. For the full matrix case it was 30.64%, compared to the diagonal case, where it was 41.18%. When using a diagonal covariance matrix, the classifier loses information about the covariances of the features, which is equivalent to assuming that the features are independent of each other. This is a simplification, and as we can see it results in a poor fit.



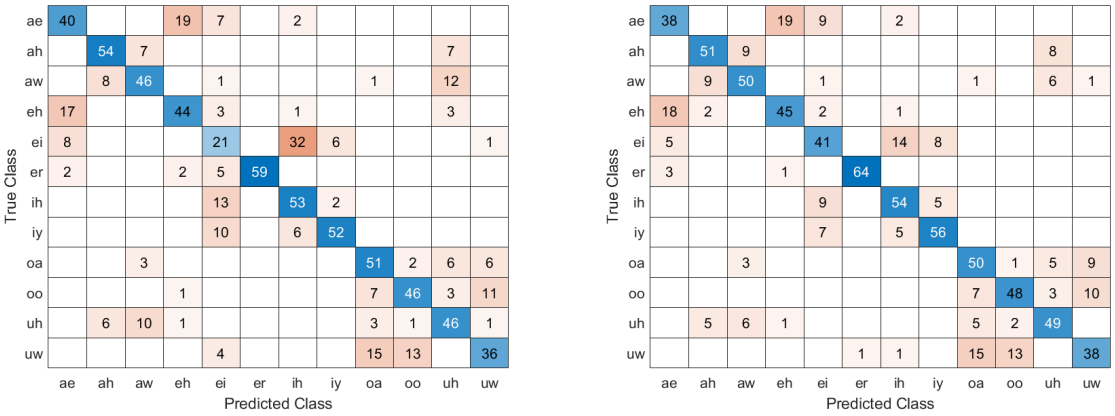
(a) Full covariance matrix

(b) Diagonal covariance matrix

Figure 14: Confusion matrices for the single Gaussian case

We do the same for the GMM case, shown in figure 15. The error rate for the 2-GMM case was 32.84%, and in the 3-GMM case it was 28.43%. In this case we also used diagonal covariance matrices, however, the performance is improved compared to the single Gaussian case. This could be because a GMM model captures more complex patterns in the data. As we saw initially, the vowel formant distributions were not perfect Gaussians, but rather had multiple peaks. We see that assuming a mixture of 3 Gaussians seems reasonable, as it resulted in the lowest error rate. From the confusion matrices we see that the vowels *ih* and *ei* were often confused. This however is solved with the 3-GMM, where the performance is significantly better.

Additionally, we have calculated error rates when using the classifier on the training set, to see whether or not the classifier was overfitted. For the single Gaussian case the error rates were 26.06%, and 39.2%. For the GMM-case the error rates were 26.53% and 22.89%. While there is a difference, it is not very large. Considering how inaccurate the MAP classifier is, we count this difference as small enough. From this we can conclude that the classifier generalizes well.



(a) 2-GMM

(b) 3-GMM

Figure 15: Confusion matrices for the GMM case

5 Conclusion

In the Iris task impact of the choice of samples was analyzed. It was observed that the performance of the linear classifier was not affected when using the last 30 samples instead of the first 30. This is however only possible if both training and testing samples are selected such that they are representative of the entire population, ensuring that the linear classifier gives accurate and reliable results.

Continuing with the Iris task the effect of eliminating features with notable overlap on the performance of the classifier was examined. The initial expectation was that the classifier's performance would be enhanced as a linear classifier works best on linearly separable data. Nevertheless, the outcomes demonstrated otherwise. To attain similar error rates, the weighing matrix W needed significantly more iterations, and more iterations did not boost the classifier's performance.

It should be emphasized that the dataset used in our study was relatively small, comprising only 50 samples of each feature. Hence, it is challenging to establish whether the classifier's slightly better performance was statistically significant or merely a result of chance. In this task, removing nonlinear features did not improve the performance of the linear classifier, but this cannot be generalized as it might be applicable only to datasets with similar characteristics. To obtain a comprehensive insight into this matter, further research involving larger datasets with varying distributions is imperative.

In the Vowels task, the impact of different parametrizations are analyzed. Firstly, when using a single Gaussian, it was observed that a full covariance matrix performed better than a diagonal one. This shows that the features are likely not independant of eachother, and the covariances are needed. When using a mixture of Gaussians, the case with 3 mixtures performed better than the one with 2. This implies that the distribution of the features are most accurately estimated as a mixture of 3 Gaussians. Comparing all tasks, it was shown that the mixture models perform better than the single Gaussian model, however the full covariance single Gaussian performs slightly better than the 2-GMM. Since the GMMs use a diagonal covariance matrix, even though the improvement when using a single Gaussian with a full covariance matrix is not very large, this shows the significance of the covariances between the features.

While we do not know the true error rate, we can conclude that the classifier generalizes well from the error rates on the testing set compared to the error rates on the training set. We can also conclude that the true error rates are similar to the estimated error rates, as the testing set is quite large, with about 70 samples for each class.

References

[1] Magne H. Johnsen. Project descriptions and tasks in classification, February 2018. Accessed: 2023-04-14.

[2] Latifa Bouissane Sohaib Khatib, Cecilia Faraloni. Exploring the use of *Iris* species: Antioxidant properties, phytochemistry, medicinal and industrial applications. March 2022. Accessed: 2023-04-22.

[3] Magne H. Johnsen. Classification, December 2017. Accessed: 2023-04-14.

A Code

A.1 Iris

Listing 1: Code for plug-in MAP with single Gaussian

```
clear
% Load and define all examples
x1all = load('class_1','-ascii');
x2all = load('class_2','-ascii');
x3all = load('class_3','-ascii');

xAll = [x1all; x2all; x3all];


% Define constants for training and saving
nSamples = length(x1all); %50
nTraining = 30;
nTests = nSamples - nTraining;
nClasses = 3;
nFeatures = 4;

iters = 350; %num of iteration when training W
alpha_test_array = [0.01 0.0075 0.0025 0.005];

% Define the true classification for xAll
targets = [kron(ones(nTraining,1), [1 0 0]);
           kron(ones(nTraining,1), [0 1 0]);
           kron(ones(nTraining,1), [0 0 1])];

% Chose case to run
casenr = 5;
saveAll = 1;

% Save names for plots
IrisHistogram_filename = "./Figures/Iris_Histogram.png";
MSE_progression_filename = "./Figures/MSE_progression_case" +
   num2str(casenr) + ".png";
ErrorRate_progression_filename = "./Figures/
   ErrorRate_progression_case" + num2str(casenr) + ".png";
CCTraining_filename = "./Figures/ConfusionChartTraining_case" +
   num2str(casenr) + ".png";
CCTest_filename = "./Figures/ConfusionChartTest_case" + num2str(
   casenr) + ".png";

switch casenr
    case 1 %first 30 samples for training
        xTraining = [x1all(1:nTraining, :);
                     x2all(1:nTraining, :);
                     x3all(1:nTraining, :)];

        xTests  = [x1all(nTraining+1:end, :);
```

```
                x2all(nTraining+1:end, :);
                x3all(nTraining+1:end, :)];

    case 2 % last 30 samples for training
        xTraining = [x1all(nTests+1:end, :);
                     x2all(nTests+1:end, :);
                     x3all(nTests+1:end, :)];

        xTests  = [x1all(1:nTests, :);
                   x2all(1:nTests, :);
                   x3all(1:nTests, :)];
        iters = 400;
    case 3 % Sepal witdh removed from features (indx 2)
        xAll(:,2) = [];

        xTraining = [x1all(1:nTraining, :);
                     x2all(1:nTraining, :);
                     x3all(1:nTraining, :)];

        xTests  = [x1all(nTraining+1:end, :);
                   x2all(nTraining+1:end, :);
                   x3all(nTraining+1:end, :)];

        xTraining(:,2) = [];
        xTests(:,2)  = [];
        nFeatures = 3;
        alpha_test_array = [0.01 0.0075 0.005 0.006];
        iters = 900;
    case 4 % Sepal witdh and length removed from features (indx
       1, 2)
        xAll(:,1:2) = [];

        xTraining = [x1all(1:nTraining, :);
                     x2all(1:nTraining, :);
                     x3all(1:nTraining, :)];

        xTests  = [x1all(nTraining+1:end, :);
                   x2all(nTraining+1:end, :);
                   x3all(nTraining+1:end, :)];

        xTraining(:,1:2) = [];
        xTests(:,1:2)  = [];
        nFeatures = 2;
        alpha_test_array = [0.006 0.0075 0.01 0.015];
        iters = 6500;
    case 5 % Sepal witdh and length and petal width removed from
       features (indx 1, 2, 4)
        xAll = xAll(:,3);

        xTraining = [x1all(1:nTraining, 3);
                     x2all(1:nTraining, 3);
```

```
                    x3all(1:nTraining, 3)];

        xTests  = [x1all(nTraining+1:end, 3);
                   x2all(nTraining+1:end, 3);
                   x3all(nTraining+1:end, 3)];

        nFeatures = 1;
        alpha_test_array = [0.15 0.0075 0.01];
        iters = 5000;
end

xTraining_aug = [ ones(nTraining*nClasses,1) xTraining];

% To be used when creating the confusion chart
TrueClasses_training = [ones(1,nTraining), 2*ones(1,nTraining),
   3*ones(1,nTraining)]; % 'setosa','versicolor','virginica'
TrueClasses_test =  [ones(1,nTests), 2*ones(1,nTests), 3*ones(1,
   nTests)];

Error_rate_training_vals = zeros(1, iters);
Error_rate_test_vals = zeros(1, iters);

MSE_vals = zeros(1,iters);
MSE_indx = 1;
tic
for alpha = alpha_test_array

    W_k = zeros(nClasses,nFeatures+1);
    for i = 1:iters
        [MSE_k, grad_W_MSE] = grad_MSE(W_k, xTraining_aug,targets
           );

        MSE_vals(MSE_indx,i) = MSE_k;

        % Calculate error rate
        Error_rate_training_vals(1,i) = computeErrorRate(W_k,
           xTraining, TrueClasses_training, nTraining*nClasses);
        Error_rate_test_vals(1,i) = computeErrorRate(W_k, xTests,
            TrueClasses_test, nTests*nClasses);

        W_k = W_k - alpha * grad_W_MSE;

    end

    MSE_indx = MSE_indx + 1;

end
toc

nexttile;
hold on;
```

```
grid on;
xlabel("Iterations");
ylabel("MSE");
title("Progression of MSE");
for r = 1:MSE_indx-1
    plot(1:iters, MSE_vals(r,:));
    Legend{r} = sprintf("alpha = %g", alpha_test_array(r));
end
legend(Legend);
hold off

if saveAll == 1
    saveas(gca, MSE_progression_filename)
end

figure(2);
plot(1:iters, Error_rate_training_vals, 1:iters,
   Error_rate_test_vals);
xlabel("Iterations");
ylabel("Error rate");
legend("Training set", "Test set")
title( sprintf("Error rate of training set with alpha = %g",
   alpha));
grid on;

if saveAll == 1
    saveas(gca, ErrorRate_progression_filename)
end

W_trained = W_k;

confmTraining = zeros(nClasses);
confmTest = zeros(nClasses);

% groupClasses = zeros(1,nTraining*nClasses); % 'setosa','
   versicolor','virginica'
% groupChat = zeros(1,nTraining*nClasses);

for i = 1:3 %True class

    xClass = xAll((i-1)*nSamples+1:i*nSamples ,:);
    for k = 1:nSamples
        x = xClass(k,:).';
        j = LinClassify(W_trained, x); % Result from classifier

        % Save in correct confusion matrix
        if k <= nTraining
            confmTraining(i,j) = confmTraining(i,j) + 1;
            % groupClasses(1,(i-1)*nTraining+k) = i;
            % groupChat(1,(i-1)*nTraining+k) = j;

```

```
        else
            confmTest(i,j) = confmTest(i,j) + 1;
        end
    end
end

% c = confusionmat(groupClasses, groupChat )
training_title = sprintf("Training set, alpha = %g , iterations = %d", alpha, iters);
confusionchart(confmTraining, {'setosa','versicolor','virginica'}, 'Title',training_title)
if saveAll == 1
    saveas(gca, CCTraining_filename)
end

test_title = sprintf("Test set, alpha = %g , iterations = %d", alpha, iters);
confusionchart(confmTest, {'setosa','versicolor','virginica' }, 'Title',test_title)
if saveAll == 1
    saveas(gca, CCTest_filename)
end

error_rate_training = computeErrorRate(W_trained, xTraining, TrueClasses_training, nTraining*nClasses); %Accuracy of training set
error_rate_test = computeErrorRate(W_trained, xTests, TrueClasses_test, nTests*nClasses); %Accuracy of test set

confmTraining
disp("Error rate for traingin set: "); error_rate_training

confmTest
disp("Error rate for test set: "); error_rate_test
```

A.2 Vowels

Listing 2: Code for plug-in MAP with single Gaussian

```
[training_set, testing_set, vowel_code, talker_group_code, talker_number, vowel_classes] = prepare_data();
%% (a) Class means and covariance matrices.
class_means = get_mean(training_set, vowel_code);
class_covariances = get_cov(training_set, vowel_code);
%% (b) Classification with full covariance matrix
targets = get_targets(testing_set, vowel_classes)
predicted_classes = map_classifier(testing_set, class_means, class_covariances)';
cm = confusionmat(targets, predicted_classes);
confusionchart(cm, string(vowel_classes))
%% (c) Classification with diagonal covariance matrix
```

```
for i = 1:size(class_covariances, 3)
    class_covariances(:,:,i) = class_covariances(:,:,i).*eye(3);
end
targets = get_targets(testing_set, vowel_classes)
predicted_classes = map_classifier(testing_set, class_means,
   class_covariances)';
cm = confusionmat(targets, predicted_classes);
confusionchart(cm, string(vowel_classes))
```

Listing 3: Functions for calculating sample mean and covariances

```
function class_means = get_mean(dataset, vowel_code)
    vowel_classes = ["ae" "ah" "aw" "eh" "er" "ei" "ih" "iy" "oa"
        "oo" "uh" "uw"];
    F1 = NaN(12, 1); F2 = F1; F3 = F1;
    class_means = table(F1, F2, F3, RowNames=vowel_classes);
    for vowel = 1:12
        vowel_set = dataset(vowel_code==vowel, :);
        class_means.F1(vowel) = mean(vowel_set.F1_ss);
        class_means.F2(vowel) = mean(vowel_set.F2_ss);
        class_means.F3(vowel) = mean(vowel_set.F3_ss);
    end
end

function class_covariances = get_cov(dataset, vowel_code)
    vowel_classes = ["ae" "ah" "aw" "eh" "er" "ei" "ih" "iy" "oa"
        "oo" "uh" "uw"];
    num_vowels = length(vowel_classes);
    F1 = NaN(num_vowels, 1); F2 = F1; F3 = F1;
    class_covariances = NaN(3, 3, num_vowels);
    cov_mat = NaN(3,3);
    for vowel = 1:num_vowels
        vowel_set = dataset(vowel_code==vowel, :);
        features = [vowel_set.F1_ss vowel_set.F2_ss vowel_set.
           F3_ss];
        cov_mat = cov(features);
        class_covariances(:,:,vowel) = cov_mat;
    end
end
```

Listing 4: Code for classification function

```
function predicted_classes = map_classifier(data, class_means,
   class_covariances)
    predicted_classes = NaN(1, size(data,1));
    for sample = 1:length(data.file)
        likelihoods = NaN(1, height(class_means));
        data_sample = data(sample, :);
        data_sample = [data_sample.F1_ss data_sample.F2_ss
           data_sample.F3_ss];
        for vowel = 1:height(class_means)
            mean_estimate = table2array(class_means(vowel, :));
            cov_estimate = class_covariances(:, :, vowel);
```

```
            p = mvnpdf(data_sample, mean_estimate, cov_estimate);
            likelihoods(:, vowel) = p * 1/12;
        end
        [~, max_likelihood_index] = max(likelihoods, [], 2);
        predicted_classes(:, sample) = max_likelihood_index;
    end
end
```

Listing 5: Code for plug-in MAP with a Gaussian Mixture Model

```
GMM2 = GMM_model(training_set, vowel_code, 2);
GMM3 = GMM_model(training_set, vowel_code, 3);
targets = get_targets(testing_set, vowel_classes);
predicted_classes_gmm2 = GMM_classifier(testing_set, GMM2);
predicted_classes_gmm3 = GMM_classifier(testing_set, GMM3);
cm = confusionmat(targets, predicted_classes_gmm2);
confusionchart(cm, string(vowel_classes))
cm = confusionmat(targets, predicted_classes_gmm3);
confusionchart(cm, string(vowel_classes))
```

Listing 6: Code for generating GMM

```
function GMMs = GMM_model(dataset, vowel_code, order)
    GMMs = cell(1, 12);
    for vowel = 1:12
        vowel_set = dataset(vowel_code==vowel, :);
        vowel_set = table2array(vowel_set(:, ["F1_ss" "F2_ss" "
            F3_ss"]));
        GMMs{vowel} = fitgmdist(vowel_set, order, '
            RegularizationValue', 0.0001, 'CovarianceType', '
            diagonal');
    end
    GMMs = GMMs;
end
```

Listing 7: Modified classification function

```
function predicted_classes = GMM_classifier(data, GMMs)
    predicted_classes = NaN(1, size(data,1));
    for sample = 1:length(data.file)
        data_sample = data(sample, :);
        data_sample = [data_sample.F1_ss data_sample.F2_ss
            data_sample.F3_ss];
        for vowel = 1:12
            GMM = GMMs{vowel};
            likelihoods(:, vowel) = pdf(GMM, data_sample) * 1/12;
        end
        [max_likelihood, max_likelihood_index] = max(likelihoods,
            [], 2);
        predicted_classes(sample, :) = max_likelihood_index;
    end
end
```