Aspects of Multi-Class Nearest Hypersphere Classification

by

Frances Coetzer

Thesis presented in partial fulfilment of the requirements for the degree of Master of Commerce in the Faculty of Economic and Management Sciences at the University of Stellenbosch

Supervisor: Dr. M.M.C. Lamont

December 2017

Plagiarism Declaration

1. Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.

2. I agree that plagiarism is a punishable offence because it constitutes theft.

3. I also understand that direct translations are plagiarism.

4. Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5. I declare that the work contained in this assignment, except otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Signature	
Initials and surname	**Date**
F. Coetzer	December 2017

Abstract

Using hyperspheres in the analysis of multivariate data is not a common practice in Statistics. However, hyperspheres have some interesting properties which are useful for data analysis in the following areas: domain description (finding a support region), detecting outliers (novelty detection) and the classification of objects into known classes. This thesis demonstrates how a hypersphere is fitted around a single dataset to obtain a support region and an outlier detector. The all-enclosing and υ-soft hyperspheres are derived. The hyperspheres are then extended to multi-class classification, which is called nearest hypersphere classification (NHC).

Different aspects of multi-class NHC are investigated. To study the classification performance of NHC we compared it to three other classification techniques. These techniques are support vector machine classification, random forests and penalised linear discriminant analysis. Using NHC requires choosing a kernel function and in this thesis, the Gaussian kernel will be used. NHC also depends on selecting an appropriate kernel hyper-parameter γ and a tuning parameter C. The behaviour of the error rate and the fraction of support vectors for different values of γ and C will be investigated.

Two methods will be investigated to obtain the optimal γ value for NHC. The first method uses a differential evolution procedure to find this value. The R function `DEoptim()` is used to execute this. The second method uses the R function `sigest()`. The first method is dependent on the classification technique and the second method is executed independently of the classification technique.

Key words: Multi-class classification, nearest hypersphere classification (NHC), support vector machine classification (SVMC), random forests, penalised linear discriminant analysis, hyper-parameter, kernel function, similarity function.

Acknowledgements

I would like to thank my supervisor Dr Morné Lamont for all his help, support and encouragement. Thank you for always being available for any questions I had and thank you for always giving quick feedback on my thesis. I really appreciate everything you did to help me.

I would also like to acknowledge the National Research Foundation of South Africa that supported, in part, the research on which this work is based.

Table of contents

List of figures

List of tables

CHAPTER 1
INTRODUCTION

Using hyperspheres to classify objects into classes is not a common practice in Statistics. The usage of hyperspheres in a high-dimensional space to obtain a support region (similar to a confidence region) and an outlier detector was first introduced by Tax and Duin (1999) and Tax (2001). Several researchers have also introduced the use of spheres to solve classification problems (cf. Wang, Neskovic and Cooper (2006), Gu and Wu (2008), Hao, Chiang and Lin (2009), Song, Xiao, Jiang and Zhao (2015)). Van der Westhuizen (2014) used the method proposed by Tax and Duin (1999) to solve classification problems. Even though the idea of the hypersphere is to fit a sphere around a single dataset, it additionally results in a classifier for two or more classes. Van der Westhuizen (2014) studied the two-class case and showed that there are situations where the nearest hypersphere classification performs better than traditional methods like linear discriminant analysis. However, methods such as the support vector machine remains superior to the hypersphere classification.

This thesis is based on similar work by Van der Westhuizen and studies nearest hypersphere classification (NHC) in a multi-class context. The following are some interesting properties of NHC:

- It extends naturally to the multi-class case since you only have to fit a hypersphere around each class and classify cases to the nearest hypersphere.
- It is especially helpful in classification problems where the classes are separable using a non-linear classifier.
- It is possible to handle problems where $n \ll p$, even though no research has been done yet on whether NHC performs well in high-dimensional data settings.
- We can also derive posterior probabilities for NHC analogous to linear discriminant analysis with normal distributions.
- The hyperspheres used in NHC allow for sparsity in the number of objects used. This is a property similar to the support vector machine.
- Hyperspheres can be used to construct an outlier detector. Using this property in NHC allows us to remove outliers while deriving the NHC classifier.

In this thesis, NHC in a multi-class setting is explored. In Chapter 2 the classification framework for multi-class data is introduced and NHC for the multi-class cases is derived. The all-enclosing hypersphere and the υ-soft hypersphere are first reviewed. Then, NHC is derived for both hyperspheres in a multi-class context. Chapter 3 contains a summary of support vector machines, tree-based methods (Bagging, Boosting and Random Forests) and penalised linear discriminant analysis. The multi-class NHC will be compared to these methods in Chapter 5. Chapter 4 contains a limited simulation study where the behaviour of the error rate and the fraction of support vectors in NHC for different choices of the hyper-parameter for the Gaussian kernel is explored.

CHAPTER 2
HYPERSPHERE CLASSIFICATION

2.1 Introduction

The idea of fitting a hypersphere around a dataset was first introduced in Scholkopf, Burges and Vapnik (1995). Fitting a hypersphere in the Hilbert space results in a support region in input space. Tax and Duin (1999) and Tax (2001) refer to this application as data domain description, which is equivalent to a confidence region for the data. We will first look at the case where a dataset has only one class and draw a hypersphere around this dataset in Hilbert space. If we can draw a hypersphere around a dataset with one class, it is possible to draw hyperspheres around each class for a dataset with any number of classes. When each class is described by a hypersphere, we can extend this to a classification problem by classifying a new object to the closest hypersphere. Using hyperspheres easily extends to the multi-class classification setting since each class has its own hypersphere.

In this chapter, we will first discuss finding the hypersphere for one dataset using the all enclosing hypersphere in Section 2.2 (where all objects are included in the hypersphere) and, secondly, the υ-soft hypersphere in Section 2.3 (where not all objects have to be included in the hypersphere). Once we know how to describe a one-class dataset, we will discuss a dataset with more than one class and formulate multi-class classification using hyperspheres in Section 2.4. The last part of this chapter is dedicated to the implementation of hyperspheres in the R software and how we can perform the nearest hypersphere classification in R.

2.2 All enclosing hypersphere

The all enclosing hypersphere is also referred to as the smallest enclosing hypersphere, the minimum enclosing ball or the hard margin hypersphere in other literature. This section is adapted from Tax and Duin (1999) and will explain how the theory behind the all enclosing hypersphere classification technique was developed.

In this section, we will be working with a one-class dataset with n objects in p dimensions. Let $\boldsymbol{x}_i \in \mathbb{R}^p$, for $i = 1,2,\dots n$, be the training data in input space.

To draw a sphere around a set of objects, we need to find the center of the sphere which, will be denoted by $\boldsymbol{a}$, as well as the radius, which will be denoted by R. For the sphere, we want to minimize the radius subject to all the points. The radius is minimized under the constraints:

$$(\boldsymbol{x}_i - \boldsymbol{a})^{\mathrm{T}}(\boldsymbol{x}_i - \boldsymbol{a}) \leq R^2 \quad \text{for } i = 1,2,\dots,n. \tag{2.1}$$

We can now construct the Lagrangian by using equation (2.1) to solve the optimisation problem:

$$L(R, \boldsymbol{a}, \alpha_i) = R^2 - \sum_{i=1}^{n} \alpha_i \left\{R^2 - (\boldsymbol{x}_i^2 - 2\boldsymbol{a}\boldsymbol{x}_i + \boldsymbol{a}^2)\right\} \tag{2.2}$$

where $\alpha_i \geq 0$ are the Lagrange multipliers.

Setting the partial derivatives with respect to $\boldsymbol{a}$ and R equal to zero yield

$$\frac{\partial L}{\partial R} = 2R(1 - \sum_{i=1}^{n} \alpha_i) = 0, \text{ and}$$

$$\frac{\partial L}{\partial \boldsymbol{a}} = 2\sum_{i=1}^{n} \alpha_i(\boldsymbol{x}_i - \boldsymbol{a}) = \boldsymbol{0},$$

from which we obtain the new constraints:

$$\sum_{i=1}^{n} \alpha_i = 1, \text{ and} \tag{2.3}$$

$$\boldsymbol{a} = \frac{\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i}{\sum_{i=1}^{n} \alpha_i} = \sum_{i=1}^{n} \alpha_i \, \boldsymbol{x}_i. \tag{2.4}$$

The Lagrangian equation (2.2) can now be rewritten by resubstituting equations (2.3) and (2.4). The following equation is now found:

$$L = \sum_{i=1}^{n} \alpha_i \, (\boldsymbol{x}_i \cdot \boldsymbol{x}_i) - \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j), \tag{2.5}$$

with constraints $\alpha_i \geq 0 \; \forall_i$ and $\sum_{i=1}^{n} \alpha_i = 1$.

Note that the dot product $(\boldsymbol{x}_i \cdot \boldsymbol{x}_i) = \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_i$.

In equation (2.4) we see that $\boldsymbol{a} = \sum_{i=1}^{n} \alpha_i \, \boldsymbol{x}_i$. This states that the center of the sphere, $\boldsymbol{a}$, is a linear combination of data objects, with weight factors α_i. For equation (2.1), the equality will only be satisfied for objects lying on the boundary. The objects on the boundary are called the support vectors and their α_i will be non-zero. To be able to describe the sphere we only need the support vectors and not the whole dataset.

The objects and sphere are currently given for a p-dimensional input space. The objects can be mapped into an infinite dimensional space (feature space) which is also called a Hilbert space ($\mathcal{H}$). The equations used so far will not differ much when the objects are mapped to the Hilbert space. Only $(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$ and similar dot products will be mapped to the Hilbert space. The objects can be transformed from a p-dimensional vector to another m-dimensional vector $\Phi(\boldsymbol{x})$ in Hilbert space. The map can be expressed as

$$\Phi: \quad \begin{matrix} \mathbb{R}^p \rightarrow \mathcal{H}, \\ \boldsymbol{x} \rightarrow \Phi(\boldsymbol{x}). \end{matrix} \tag{2.6}$$

When we are in the Hilbert space, we can obtain a better and more tight description of the sphere. We will now let the center of the hypersphere in Hilbert space be $\boldsymbol{a} = \sum_i \alpha_i \Phi(\boldsymbol{x}_i)$. Consider the hypersphere representation in Hilbert space given in Figure 2.1. The solid red circle represents the all enclosing hypersphere while the dotted blue circle represents the υ-soft hypersphere which will be discussed in Section 2.3.

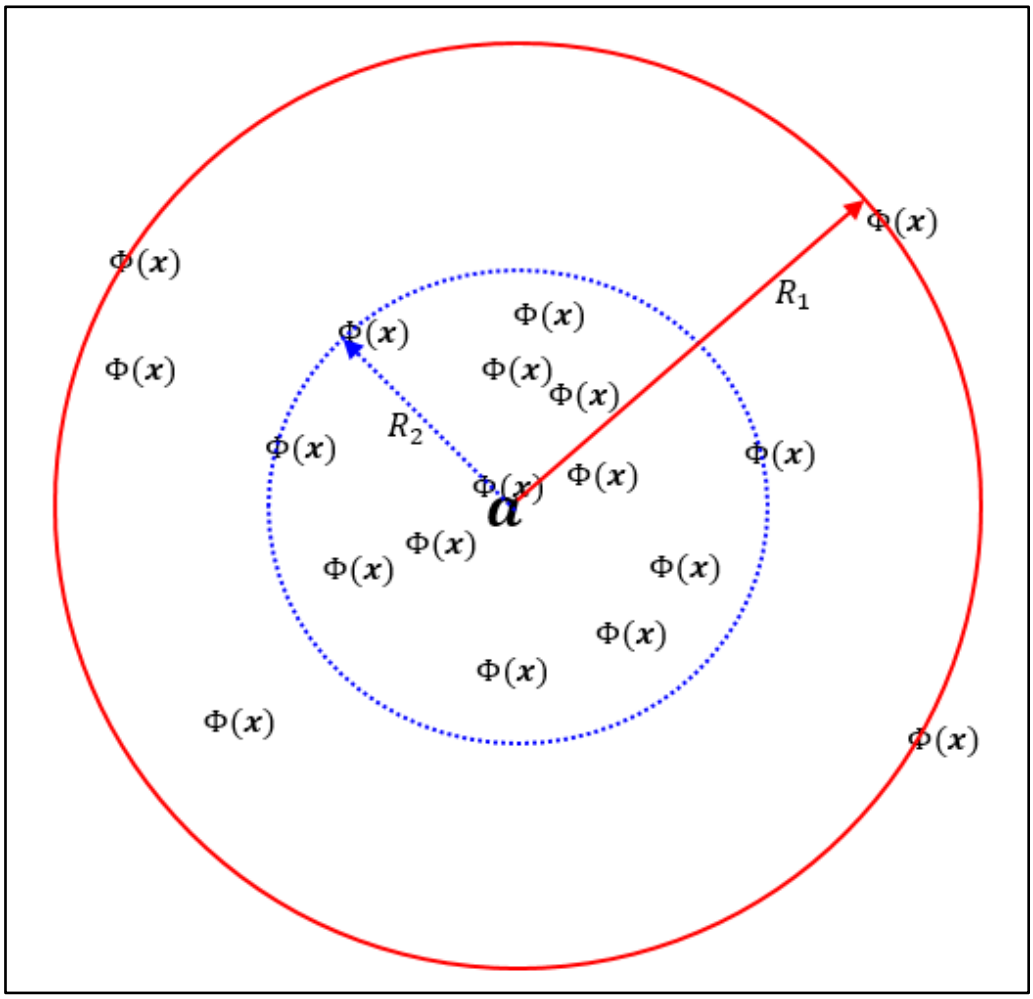

Figure 2.1: Hypersphere representation in Hilbert space.

The all enclosing hypersphere has center $\boldsymbol{a}$ and radius R_1. From Figure 2.1 it can be seen that there are three support vectors that lie on the hypersphere. The objects in the Hilbert space that lie on the surface of the hypersphere are the objects that lie the furthest from the center of the hypersphere. We can therefore choose any object that lie on the surface of the hypersphere to find the radius. The radius (R_1) is the distance from the centre of the hypersphere, $\boldsymbol{a}$**,** to any object in Hilbert space, $\Phi(\boldsymbol{x})$, on the surface of the hypersphere.

We do not know what the mapping function, Φ, is and the dot product in Hilbert space cannot be calculated since $\Phi(\boldsymbol{x}_i)$ could be an infinite dimensional vector. However, a kernel function can be used to replace the dot product between two objects mapped to the Hilbert space. The function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is defined to be a kernel if there exists a map Φ from the space $\mathbb{R}^p$ to the Hilbert space. When we map the objects to Hilbert space, we can use kernel functions, because

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_j)). \tag{2.7}$$

Examples of popular kernel functions are given in Table 2.1.

Table 2.1: Examples of kernel functions.

Kernel	**Kernel function**
Linear “vanilladot” kernel	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$
Gaussian kernel	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2)$
Polynomial kernel	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) + c)^d$
Hyperbolic tangent kernel	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\gamma(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) + c$
Laplace radial basis kernel	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|)$

Source: Karatzoglou, Smola, Hornik, and Zeileis. (2004: 4-5).

There are many different kernel functions as can be seen in Table 2.1, but for this thesis we will be using the Gaussian kernel function given by

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2). \tag{2.8}$$

The quantity γ in equation (2.8) is known as a hyper-parameter which also needs to be estimated using the data.

The Lagrangian in equation (2.5) can now be rewritten where the dot products are replaced by the kernel function:

$$L = \sum_{i=1}^{n} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_i) - \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{2.9}$$

with constraints $\alpha_i \geq 0 \; \forall_i$ and $\sum_{i=1}^{n} \alpha_i = 1$.

The next step is to find the optimal α_i values, which we will denote by α_i^*. These can be found by solving the following optimisation problem:

$$max_{\boldsymbol{\alpha}} [\sum_{i=1}^{n} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_i) - \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)] \, , \tag{2.10}$$

with constraints $\alpha_i \geq 0 \; \forall_i$ and $\sum_{i=1}^{n} \alpha_i = 1$.

Figure 2.2 represents the support region (see equation (2.25)) in input space which results from the hypersphere in Hilbert space. The solid red line is the support region for the all enclosing hypersphere while the dotted blue line is the support region for the υ-soft hypersphere which will be introduced in the next section.

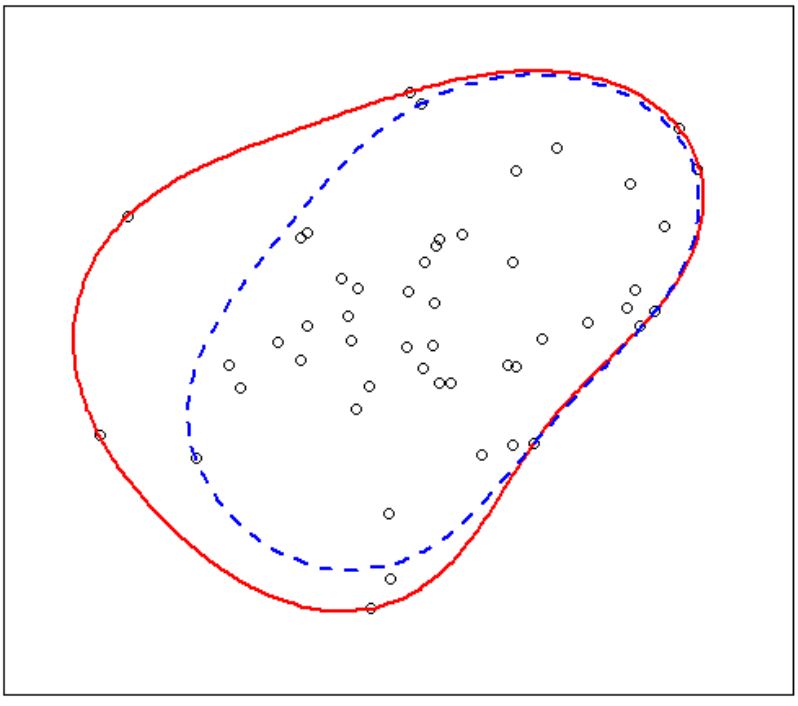

Figure 2.2: Support region representation in input space.

The solid red line in Figure 2.2 resulted from the all enclosing hypersphere in input space that we studied in this section. The hypersphere in Hilbert space is a round sphere (which we saw in Figure 2.1 where the solid red circle was the all enclosing hypersphere), but when it is transformed back to input space, it will no longer be a round sphere due to the kernel function used. We can clearly see from the figure the support vectors that determine the support region. They are the objects on the solid line.

2.3 The υ-soft hypersphere

The υ-soft hypersphere is also known as the soft margin hypersphere. This section is adapted from Tax and Duin (1999) and will explain how the theory behind the υ-soft hypersphere classification technique was developed. The previous section included all the objects of the dataset in the hypersphere. If the dataset contains outliers, the sphere will be larger than necessary if all the objects are included in the sphere. We will introduce slack variables which will be denoted by ξ_i to allow some of the objects to lie outside the sphere. We still want to minimize the radius and with the slack variables we will minimize the following equation:

$$F(R, \boldsymbol{a}, \xi_i) = R^2 + C \sum_{i=1}^{n} \xi_i \tag{2.11}$$

where C gives a trade-off between the volume of the sphere and the number of outliers.

Equation (2.11) is minimized under the constraints:

$$(\Phi(\boldsymbol{x}_i) - \boldsymbol{a})^{\mathrm{T}}(\Phi(\boldsymbol{x}_i) - \boldsymbol{a}) \leq R^2 + \xi_i, \quad \forall_i, \xi_i \geq 0. \tag{2.12}$$

We can now construct the Lagrangian again, but this time by incorporating the constraints in equation (2.12). We obtain

$$L(R, \boldsymbol{a}, \alpha_i, \xi_i) = R^2 + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \{R^2 + \xi_i - (\Phi(\boldsymbol{x}_i)^2 - 2\boldsymbol{a}\Phi(\boldsymbol{x}_i) + \boldsymbol{a}^2)\} - \sum_{i=1}^{n} \gamma_i \xi_i, \tag{2.13}$$

where $\alpha_i \geq 0$ and $\gamma_i \geq 0$ are the Lagrange multipliers.

Taking partial derivatives with respect to $\boldsymbol{a}$, R and ξ_i and setting them equal to zero yield

$$\frac{\partial L}{\partial R} = 2R(1 - \sum_{i=1}^{n} \alpha_i) = 0,$$

$$\frac{\partial L}{\partial \boldsymbol{a}} = 2\sum_{i=1}^{n} \alpha_i(\boldsymbol{x}_i - \boldsymbol{a}) = \mathbf{0}, \text{ and}$$

$$\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \gamma_i = 0, \quad \forall_i,$$

which give the new constraints:

$$\sum_{i=1}^{n} \alpha_i = 1, \tag{2.14}$$

$$\boldsymbol{a} = \frac{\sum_{i=1}^{n} \alpha_i \Phi(\boldsymbol{x}_i)}{\sum_{i=1}^{n} \alpha_i} = \sum_{i=1}^{n} \alpha_i \, \Phi(\boldsymbol{x}_i), \text{ and} \tag{2.15}$$

$$C - \alpha_i - \gamma_i = 0, \quad \forall_i. \tag{2.16}$$

The first two constraints are the same as for the all enclosing hypersphere, but the third constraint is new. We can now say that since $\alpha_i \geq 0$ and $\gamma_i \geq 0$, we can remove the variables γ_i from equation (2.16) and use the constraint $0 \leq \alpha_i \leq C \; \forall_i$.

The Lagrangian equation (2.13) can now be rewritten by resubstituting equations (2.14), (2.15) and (2.16). The following equation is now found:

$$L = \sum_{i=1}^{n} \alpha_i(\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_i)) - \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j (\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_j)), \tag{2.17}$$

with constraints $0 \leq \alpha_i \leq C \; \forall_i$ and $\sum_{i=1}^{n} \alpha_i = 1$.

We can once again use a kernel and rewrite (2.17) as

$$L = \sum_{i=1}^{n} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_i) - \sum_{i=1}^{n} \sum_{j=}^{n} \alpha_i \alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{2.18}$$

with constraints $0 \leq \alpha_i \leq C \; \forall_i$ and $\sum_{i=1}^{n} \alpha_i = 1$.

The Lagrangian equation in (2.17) is the same as in equation (2.9), but with slightly different constraints. We now use $0 \leq \alpha_i \leq C \; \forall_i$. When we set $C = 1$, we will have an all enclosing hypersphere. The $\sum_{i=1}^{n} \alpha_i = 1$ constraint implies that if C is larger than 1 a solution for the α_i's can be found, but if $C < \frac{1}{n}$ no solution will be found, because the constraint $\sum_{i=1}^{n} \alpha_i = 1$ will never be met.

We need to find the radius of the sphere and it can be obtained by calculating the distance from a support object with a weight smaller than C to the center of the sphere. We once again want to find the α_i^* values. These can be found by optimising equation (2.18), i.e.

$$max_{\boldsymbol{\alpha}}[\sum_{i=1}^{n}\alpha_i K(\boldsymbol{x}_i,\boldsymbol{x}_i) - \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j)]\,, \tag{2.19}$$

with constraints $0 \leq \alpha_i \leq C\ \forall_i$ and $\sum_{i=1}^{n}\alpha_i = 1$.

The difference between the all enclosing hypersphere and the υ-soft hypersphere can be seen in Figure 2.1 for Hilbert space representation. The υ-soft hypersphere is represented by the dotted blue line with center $\boldsymbol{a}$ and radius R_2. When we use the υ-soft hypersphere we can see from the figure that not all objects are included in the hypersphere. In Figure 2.2 we see the dotted blue line which is the support region from the υ-soft hypersphere. The outliers are clearly visible from the plot and the support region is smaller. The support vectors are again lying on the boundary.

When the weight, α_i, is such that $\alpha_i = C$, the object has hit the upper bound in $0 \leq \alpha_i \leq C$ and the object therefore lies outside the sphere. This is a way of determining which objects are outliers. When we need to determine whether an object lies inside the sphere, the distance from the object to the center of the sphere is determined. If the distance is smaller than or equal to the radius, then it lies in the sphere. This can be written as the following equations where $\boldsymbol{x}$ is the object:

$$(\Phi(\boldsymbol{x}) - \boldsymbol{a})^{\mathrm{T}}(\Phi(\boldsymbol{x}) - \boldsymbol{a}) \leq R^2, \tag{2.20}$$

or

$$(\Phi(\boldsymbol{x})\cdot\Phi(\boldsymbol{x})) - 2\sum_{i=1}^{n}\alpha_i(\Phi(\boldsymbol{x})\cdot\Phi(\boldsymbol{x}_i)) + \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j(\Phi(\boldsymbol{x}_i)\cdot\Phi(\boldsymbol{x}_j)) \leq R^2. \tag{2.21}$$

Equation (2.21) only uses the support vectors to determine whether $\boldsymbol{x}$ is in the sphere, because when an object is not a support vector, its α_i will be zero and will not influence equation (2.21).

Once again, we can use kernels for the dot product between objects mapped into the Hilbert space. We can therefore rewrite the equation of when a test object is accepted as

$$K(\boldsymbol{x},\boldsymbol{x}) - 2\sum_{i=1}^{n}\alpha_i K(\boldsymbol{x},\boldsymbol{x}_i) + \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) \leq R^2, \tag{2.22}$$

or

$$K(\boldsymbol{x},\boldsymbol{x}) - 2\sum_{i=1}^{n}\alpha_i K(\boldsymbol{x},\boldsymbol{x}_i) + \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) - R^2 \leq 0, \tag{2.23}$$

with squared radius

$$\begin{aligned} R^2 &= (\Phi(\boldsymbol{x}_0) - \boldsymbol{a})^{\mathrm{T}}(\Phi(\boldsymbol{x}_0) - \boldsymbol{a}) \\ &= K(\boldsymbol{x}_0,\boldsymbol{x}_0) - 2\sum_{i=1}^{n}\alpha_i K(\boldsymbol{x}_0,\boldsymbol{x}_i) + \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j), \end{aligned} \tag{2.24}$$

and $\Phi(\boldsymbol{x}_0)$ a support vector.

We can now say that an object is inside a support region when $g(\boldsymbol{x}) \leq 0$ and, from (2.23), the support region can now be defined as

$$\{\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{p}} : g(\boldsymbol{x}) \leq 0\} \tag{2.25}$$

where

$$g(\boldsymbol{x}) = K(\boldsymbol{x},\boldsymbol{x}) - 2\sum_{i=1}^{n}\alpha_i K(\boldsymbol{x},\boldsymbol{x}_i) + \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) - R^2. \tag{2.26}$$

Now that we can determine whether an object lies in the hypersphere, we can define an outlier detector as

$$\varphi(\boldsymbol{x}) = I[\,K(\boldsymbol{x},\boldsymbol{x}) - 2\sum_{i=1}^{n}\alpha_i K(\boldsymbol{x},\boldsymbol{x}_i) + \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) > R^2]. \tag{2.27}$$

This outlier detector is an indicator function and will return 1 when the object is an outlier and 0 when the object is not an outlier.

2.4 Classifying objects using hyperspheres

We can now take a set of objects and describe it by using a hypersphere. If we have a dataset with more than one class, we can also describe each class using a hypersphere. Each hypersphere has a center and a radius. We will be able to classify a new object into one of the classes by using a dissimilarity or similarity measure to determine which class this new object belongs to. Let S_g be the hypersphere corresponding to the g^{th} class, $g = 1,2,\ldots,G$, with $\boldsymbol{a}_g$ as its center in Hilbert space and R_g the radius.

A similarity function determines how similar an object is to each class. An object will be classified into the class with the highest similarity value for that object. When using a dissimilarity measure, the object will be classified into the class with the smallest dissimilarity measure. Table 2.2 gives examples of different similarity functions that can be used.

Table 2.2: Examples of similarity functions.

Name	Similarity function
Distance-to-center-based similarity function	$sim(\boldsymbol{x}, S_g) = -\left\|\Phi(\mathbf{x}) - \boldsymbol{a}_g\right\|^2$
Zhu's similarity function	$sim(\boldsymbol{x}, S_g) = R_g^2 - \left\|\Phi(\mathbf{x}) - \boldsymbol{a}_g\right\|^2$
Gaussian-based similarity function	$sim(\boldsymbol{x}, S_g) = \frac{1}{R_g^2} - \exp\left(\frac{-\left\|\Phi(\boldsymbol{x}) - \boldsymbol{a}_g\right\|^2}{R_g^2}\right)$
Wu's similarity function for Case 3 (an object can be in more than one sphere)	$sim(\boldsymbol{x}, S_g) = -\frac{\left\|\Phi(\mathbf{x}) - \boldsymbol{a}_g\right\|}{R_g}$

Source: Hao, Chiang and Lin. (2009: 17-19).

For this thesis, we will use $sim(\boldsymbol{x}, S_g) = -\frac{\left\|\Phi(\mathbf{x}) - \boldsymbol{a}_g\right\|}{R_g}$. The object $\boldsymbol{x}$ will be classified into the class with the largest similarity measure. This can be rewritten as a dissimilarity measure

$$\delta_g = \frac{\left\|\Phi(\boldsymbol{x}) - \boldsymbol{a}_g\right\|}{R_g}, \quad g = 1, \dots G. \tag{2.28}$$

Now that we have the dissimilarity measure, we can build a classifier.

The two-class case:

Figure 2.3 shows how the distance from a new object, $\boldsymbol{z}$, is calculated for each of the two hyperspheres in Hilbert space where $\boldsymbol{z} \in \mathbb{R}^p$.

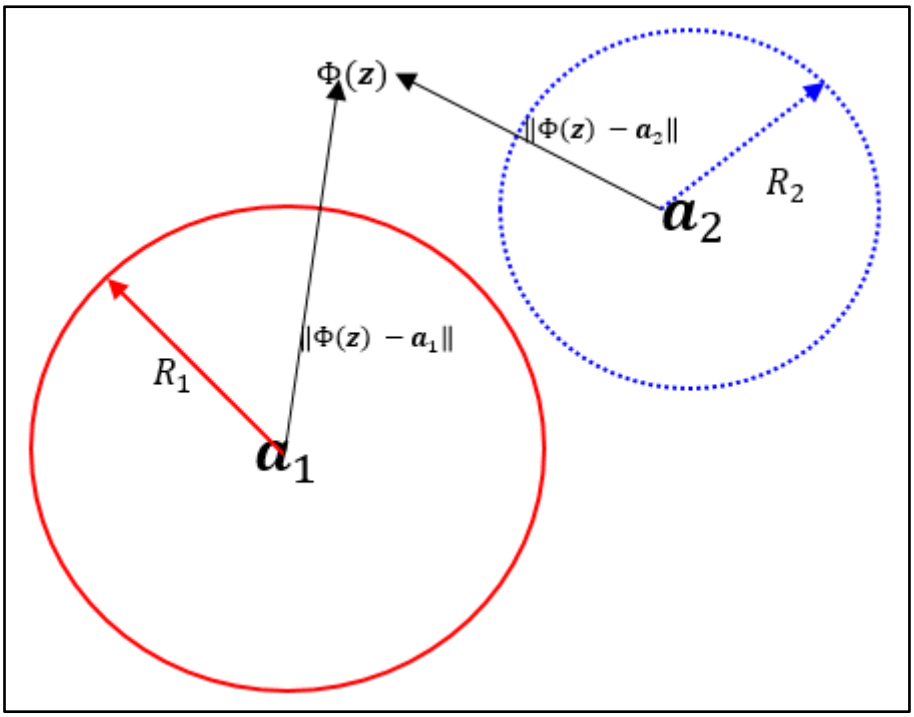

Figure 2.3: Two-class classification with hyperspheres in Hilbert space.

In Figure 2.3 the first class is the red solid line hypersphere with center $\boldsymbol{a}_1$ and radius R_1 while the second class is the blue dotted line hypersphere with center $\boldsymbol{a}_2$ and radius R_2**.** We want to classify $\boldsymbol{z}$ into one of the two classes. When we take the distance from $\Phi(\boldsymbol{z})$ to the center of each hypersphere in Hilbert space, it does not take into account the different variances of each class. We will therefore divide the distance to the center of each hypersphere by its radius to find the dissimilarity measure in equation (2.28). The new object will be classified into the class with the smallest dissimilarity measure.

Let the two classes be denoted by Π_1 and Π_2. Equation (2.28) can be rewritten by squaring the dissimilarity function and using equation (2.22) for the numerator:

$$\delta_g^2 = \frac{K(\boldsymbol{z},\boldsymbol{z}) - 2\sum_i \alpha_i K(\boldsymbol{z},\boldsymbol{x}_i) + \sum_{i,j} \alpha_i \alpha_j K(\boldsymbol{x}_i,\boldsymbol{x}_j)}{R_g^2}. \tag{2.29}$$

When we have two classes, we will find two dissimilarity measures for the new object. When the dissimilarity measure of Π_1 is less than the dissimilarity measure of Π_2, we will classify $\boldsymbol{z}$ into Π_1, and vice versa.

The two-class nearest hypersphere classifier can now be defined as:

Classify $\boldsymbol{z}$ into Π_1 if

$$\delta_1 < \delta_2, \tag{2.30}$$

otherwise $\boldsymbol{z}$ belongs to Π_2.

The multi-class case:

When there are more than two classes, similar reasoning will be applied. We will now have $G > 2$ classes and G dissimilarity measures for a new object. The new object will again be classified to the class which has the smallest dissimilarity measure. We will be using the following training dataset (T) to fit the hyperspheres:

$$T = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\} \tag{2.31}$$

where $\boldsymbol{x}_i \in \mathbb{R}^p$ and $y_i \in \{1, 2, \dots, G\},\ i = 1, \dots, n$.

Figure 2.4 shows how the distance from a new object, $\boldsymbol{z}$, is calculated for each of the three hyperspheres in Hilbert space where $\boldsymbol{z} \in \mathbb{R}^p$. Each class has center $\boldsymbol{a}_g$ and radius R_g for $g = 1, 2, 3$. We want to classify $\boldsymbol{z}$ into one of the three classes. We will use the same dissimilarity function as for the two-class case by dividing the distance to the center of each hypersphere by its radius to find the dissimilarity measure in equation (2.28) to take the variance of each class into account. The new object will be classified into the class with the smallest dissimilarity measure.

The multi-class nearest hypersphere classifier can now be defined as:

Classify $\boldsymbol{z}$ into Π_g if

$$\delta_g = min(\delta_1, \delta_2, \dots, \delta_G) \tag{2.32}$$

where δ_g is the dissimilarity measure for Π_g.

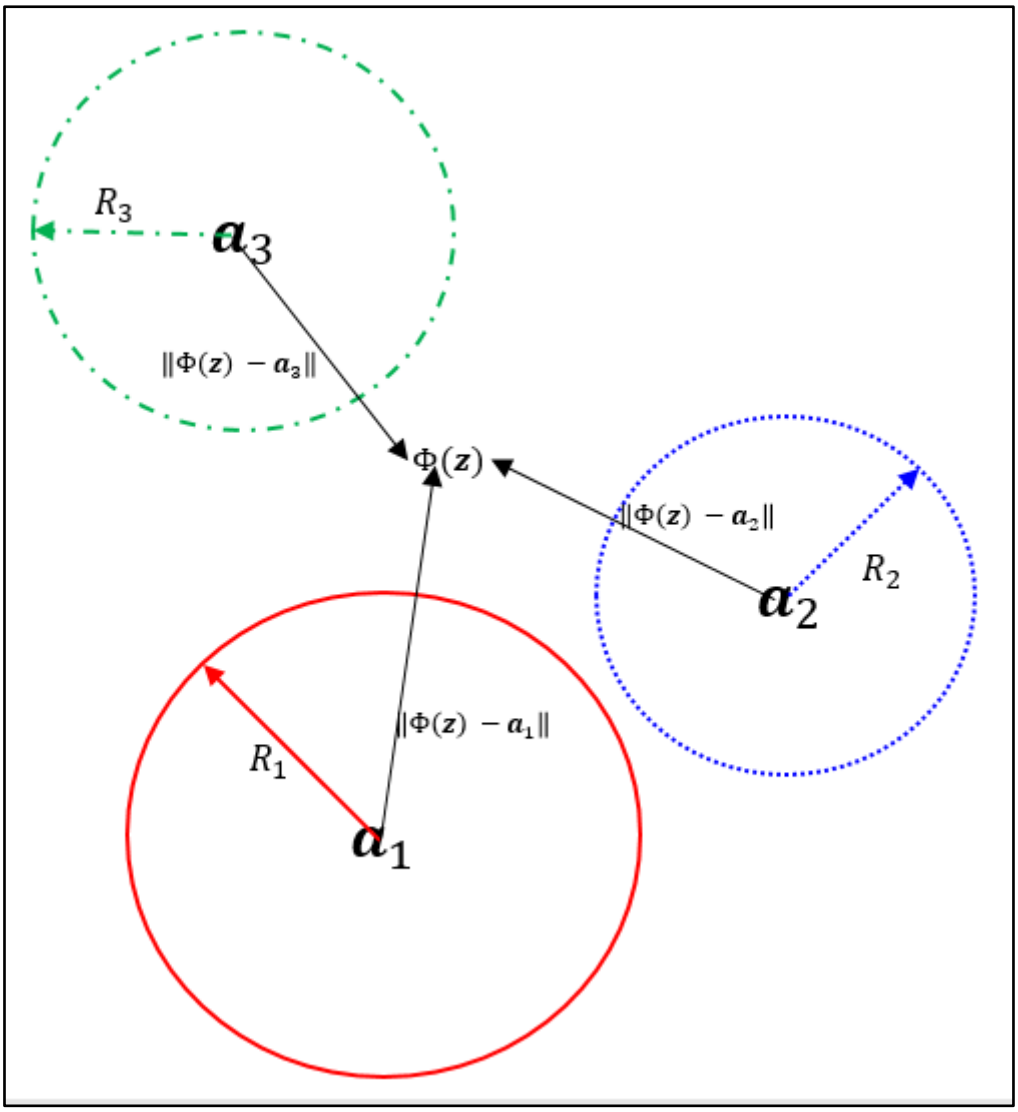

Figure 2.4: Multi-class classification with hyperspheres in Hilbert space.

In Figure 2.5 to Figure 2.12 we will look at the decision boundary and support regions for different values of the γ and C parameters using the Fisher (1936) Iris dataset. The Iris dataset has three classes (Setosa, Versicolor and Virginica) and four variables (Sepal.Length, Sepal.Width, Petal.Length and Petal.Width). In the examples, we will use two standardised variables (Sepal.Length and Sepal.Width). The Gaussian kernel in equation (2.8) was used with different γ values ($0.2, 0.5, 0.9$ and 5) and different C parameters (0.1 and 1) were also used. When $C = 1$, we have the all enclosing hypersphere and when $C = 0.1$, which is less than 1, we have the υ-soft hypersphere.

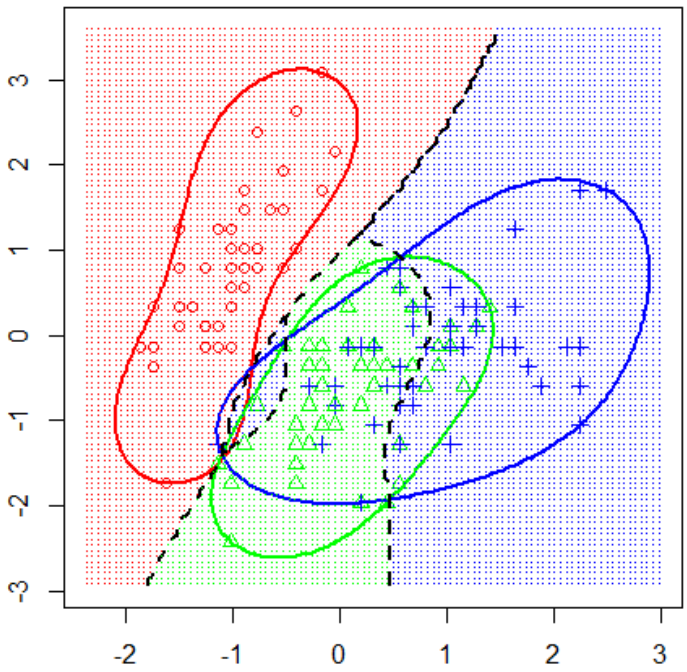

Figure 2.5: The NHC decision boundary and support regions when $\gamma = 0.2$ and $C = 1$.

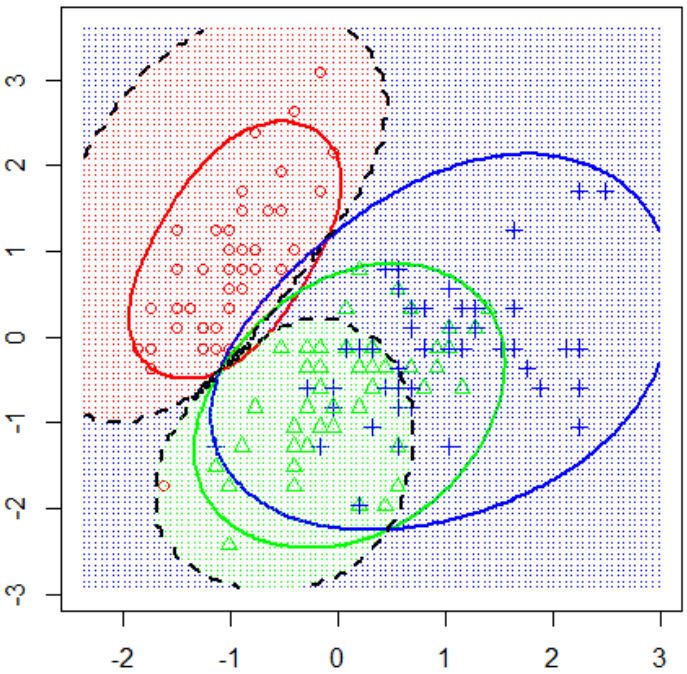

Figure 2.6: The NHC decision boundary and support regions when $\gamma = 0.2$ and $C = 0.1$.

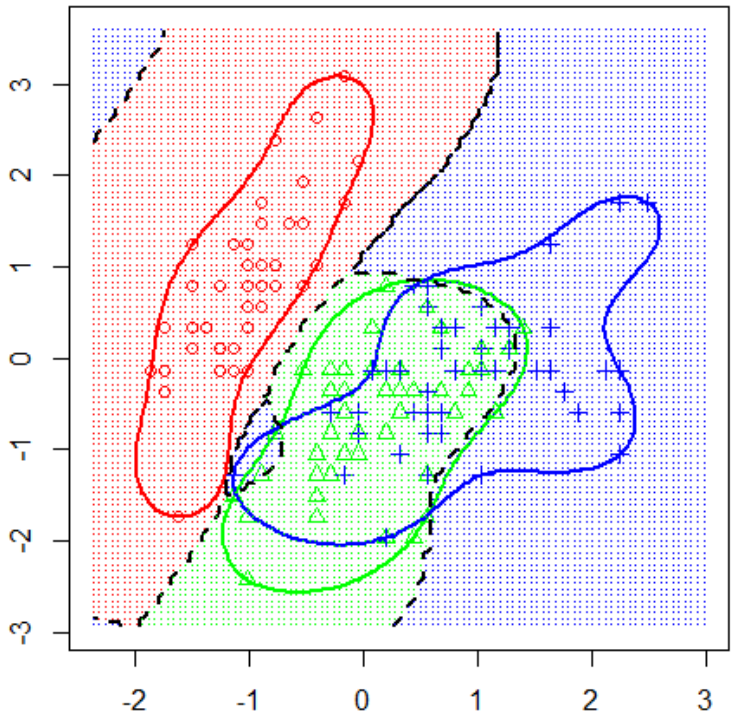

Figure 2.7: The NHC decision boundary and support regions when $\gamma = 0.5$ and $C = 1$.

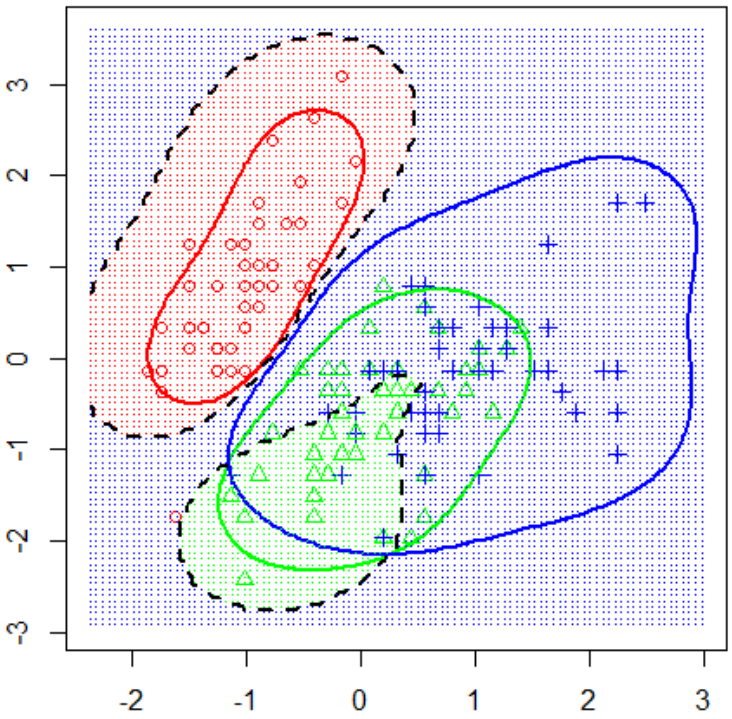

Figure 2.8: The NHC decision boundary and support regions when $\gamma = 0.5$ and $C = 0.1$.

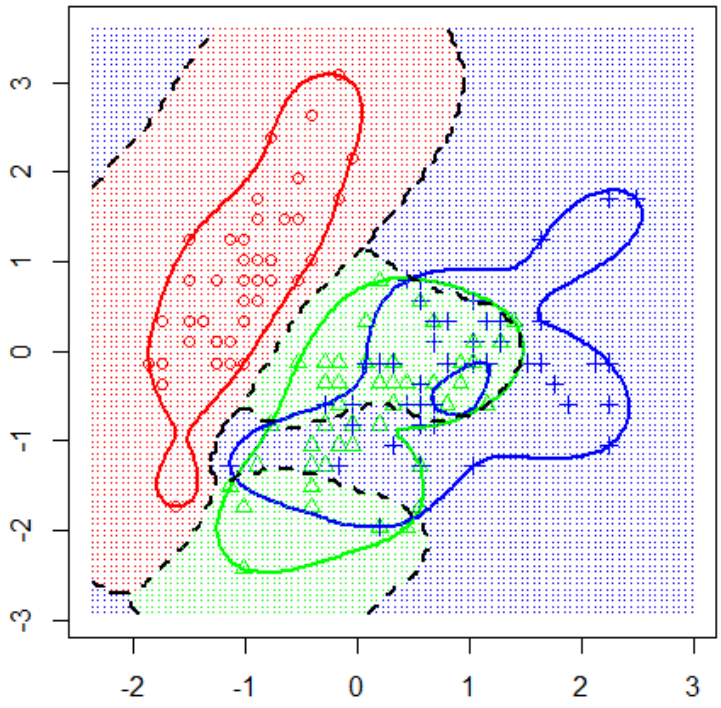

Figure 2.9: The NHC decision boundary and support regions when $\gamma = 0.9$ and $C = 1$.

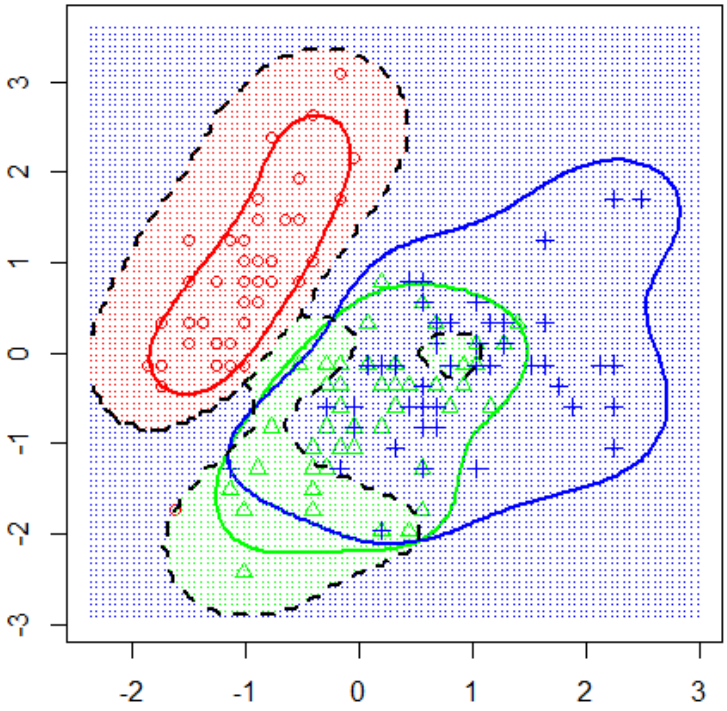

Figure 2.10: The NHC decision boundary and support regions when $\gamma = 0.9$ and $C = 0.1$.

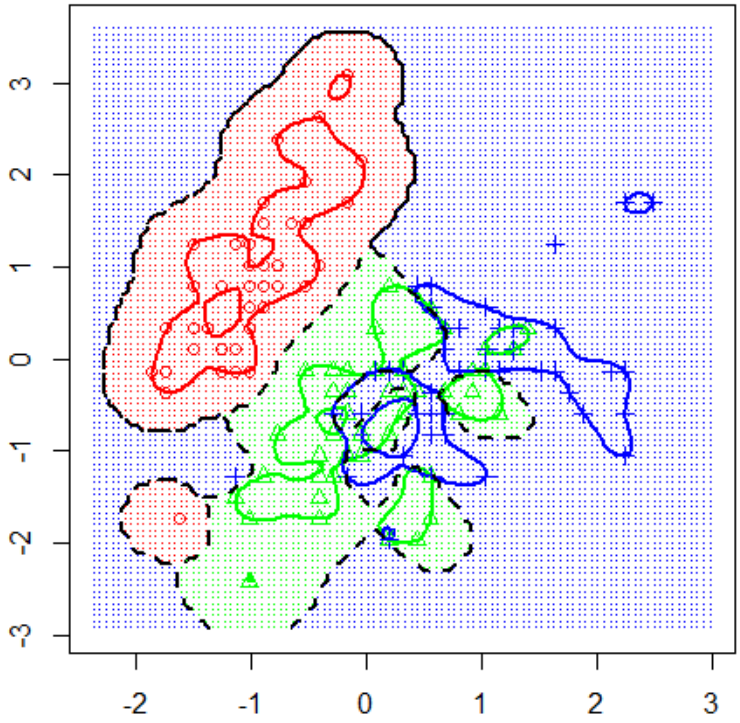

Figure 2.11: The NHC decision boundary and support regions when $\gamma = 5$ and $C = 1$.

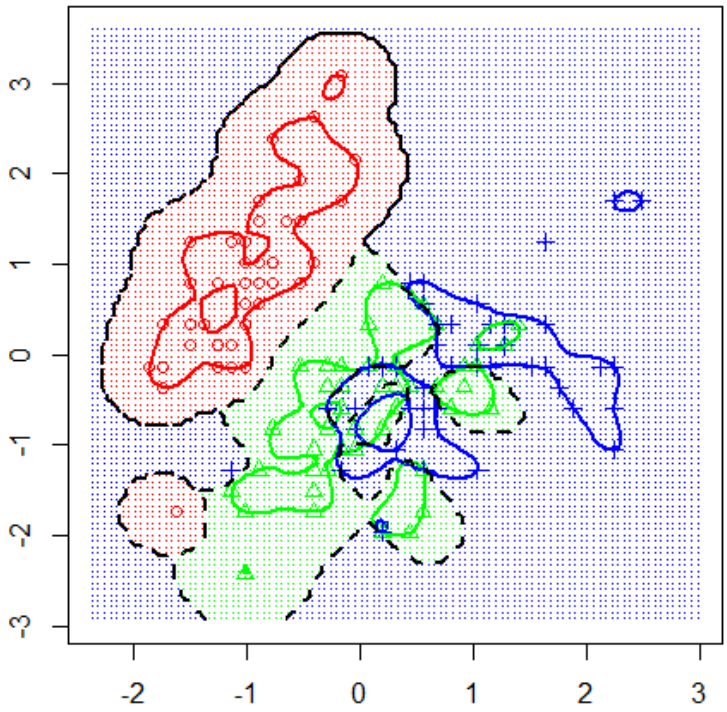

Figure 2.12: The NHC decision boundary and support regions when $\gamma = 5$ and $C = 0.1$.

We can see from Figure 2.5 to Figure 2.12 that for small γ values, the support regions look spherical, and for large γ values, the support regions become more flexible. This is because as γ increases, the number of support vectors also increases. This affects the shape of the support region dramatically. When $C = 1$ the all enclosing hypersphere is used and we can clearly see that all the objects in the class are included in the support region. When $C = 0.1 < 1$, the υ-soft hypersphere is used and we can see that the outliers are not included in the support region. The support region is more flexible for $C = 1$ and more spherical for $C = 0.1$ when not all objects are included in the support region. We can see that when $\gamma = 5$, the model is overfitting.

The NHC is a non-parametric classification technique that can be used for datasets with any number of classes. It is a non-linear classifier, because of the non-linear kernel. The NHC is not restricted to datasets with $p < n$. We can apply this technique to datasets with any number of variables and NHC will therefore also work when $p \gg n$. Each class only uses the support vectors to determine the support region, and not all objects, which is a computational advantage.

We can also calculate posterior probabilities in the NHC framework. We can estimate the posterior probabilities (Wang et al., 2006), by analogy to the linear discriminant analysis, as:

$$P(\Pi_g|\boldsymbol{x}) = \frac{p_g}{(n\pi R_g^2)^{p/2}} \exp\left\{-\frac{1}{2}\left(\frac{\|\boldsymbol{x}-\boldsymbol{a}_g\|}{R_g}\right)^2\right\} / \sum_{l=1}^{G} \frac{p_l}{(n\pi R_l^2)^{p/2}} \exp\left\{-\frac{1}{2}\left(\frac{\|\boldsymbol{x}-\boldsymbol{a}_l\|}{R_l}\right)^2\right\}$$
$$= \frac{p_g}{(n\pi R_g^2)^{p/2}} \delta_g^2 / \sum_{l=1}^{G} \frac{p_l}{(n\pi R_l^2)^{p/2}} \delta_l^2$$

with p_g the prior probabilities. If we assume equal radii for the hyperspheres then we obtain

$$P(\Pi_g|\boldsymbol{x}) = \frac{p_g e^{-\frac{1}{2}\delta_g^2}}{\sum_{l=1}^{G} p_l e^{-\frac{1}{2}\delta_l^2}}$$

where δ_g^2 is given in equation (2.29).

2.5 Implementation in R

Now that we have covered all the theory of the NHC, we need to implement the theory using R software. Our first problem is to solve the optimisation problem in equation (2.19) and find the optimal α_i values. We will use the `ipop()` function in R which can be found in the `kernlab` package.

The `ipop()` usage is as follows:

```
ipop(c, H, A, b, l, u, r, sigf = 7, maxiter = 40, margin = 0.05,
     bound = 10, verb = 0).
```

The `ipop()` function solves the following quadratic programming problem:

$$min(\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \tfrac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{x}) \tag{2.33}$$

where

$$\boldsymbol{b} \leq \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} + \boldsymbol{r} \tag{2.34}$$

and $$\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}. \tag{2.35}$$

Before we carry on, we need to define the Gram matrix which is denoted by $\boldsymbol{K}$. When we have a kernel function, $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, the i^{th} row and j^{th} column element of the Gram matrix is $K_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

When using the Gaussian kernel, we know that $K(\boldsymbol{x}_i, \boldsymbol{x}_i) = 1$ for all i. Let $\mathbf{1}^{\mathrm{T}} = (1, \dots, 1)$ be a vector of size n. We can rewrite equation (2.19) as follows:

$$max_{\boldsymbol{\alpha}}[\mathbf{1}^{\mathrm{T}}\boldsymbol{\alpha} - \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha}]\,, \tag{2.36}$$

with constraints $0 \leq \alpha_i \leq C\ \forall_i$ and $\sum_i \alpha_i = 1$.

Table 2.3 compares the Lagrangian with the arguments of the `ipop()` function.

Table 2.3: A comparison between the arguments in the `ipop()` function and the terms in the Lagrangian.

Term in `ipop()` function	Term in Lagrangian
$\boldsymbol{x}$	$\boldsymbol{\alpha}$
$\boldsymbol{c}$	$diag(K(\boldsymbol{x}_i, \boldsymbol{x}_i))$
$\boldsymbol{H}$	$\boldsymbol{K} = Gram\ matrix$
$\boldsymbol{l}$	$\mathbf{0}$
$\boldsymbol{u}$	$(C, C, \dots, C)$
b	1
$\boldsymbol{A}$	$\mathbf{1}^{\mathrm{T}}$
r	0

- The r is equal to zero and b is equal to 1 so that $\boldsymbol{Ax}$ in the `ipop()` function, which is $\mathbf{1}^{\mathrm{T}}\boldsymbol{\alpha} = \sum_i \alpha_i = 1$, satisfies the constraint.
- The α_i can be between 0 and C. We will therefore set $\boldsymbol{u}$ equal to a vector $(C, C, \dots, C)$ of size n and $\boldsymbol{l}$, which is the lower limit, equal to the zero vector.
- To be able to calculate the kernel values we need the `rbfdot` function which is the Gaussian kernel function and can be found in the `kernlab` package.

The following R function will be used to find the values for $\boldsymbol{\alpha}^*$.

```
optimal.alpha.values<-function(data,gamma,C)

{

      # data is the matrix for one class

      # gamma value is given in gamma

      # data has n objects

      # C is the C parameter NHC


      ### Find values for ipop function to find the alpha values ###

      require(kernlab)

      n<-nrow(data)

      rownames(data)<-NULL

      data.mat<-as.matrix(data)

      Gram.mat<-kernelMatrix(rbfdot(gamma), data.mat)

      c.vec<-diag(Gram.mat)

      A.vec<-matrix(1, nrow=1, ncol=n)

      b.unit<-1

      l.vec<-rep(0, n)

      u.vec<-rep(C, n)

      r.unit<-0


      ### Calculate alpha values using the ipop function ###

      alpha.vec<-primal(ipop(c=c.vec, H=Gram.mat, A=A.vec, b=b.unit,
                        l=l.vec, u=u.vec, r=r.unit))

      return(alpha.vec)

}
```

This function returns the optimal alpha values $\boldsymbol{\alpha}^*$. We need to give the function a γ value to use in determining the α_i^* values. We will discuss the choice of γ in Chapter 4 and Chapter 5. A value for $C \geq 1$ will find the all enclosing hypersphere while $\frac{1}{n} \leq C < 1$ will find the υ-soft hypersphere. For any $C > 1$, the constraint $\sum_i \alpha_i = 1$ will cause the output for the function to be the same as when $C = 1$.

The R function `MultiClass.NHC()`, which will be introduced next, was written to perform NHC on a training dataset to find the α_i^* values using a given γ value. Once we have the α_i^* values, we can determine the center of the hypersphere and the radius. This function will determine the α_i^* values for each class and then find the center of the hypersphere as well as the radius for each class. When we have the center and radius of each class, we can then use the dissimilarity function for each object that has to be classified, and the function will return the class each object was classified to. The function can also return the radius and number of support vectors used per class. We will need the number of support vectors used in Chapter 4. The arguments of the `MultiClass.NHC()` function are explained in Table 2.4.

Table 2.4: The arguments of the `MultiClass.NHC()` function.

Arguments	Explanation
`data`	Training data matrix
`class.vector`	Vector with the class of the training data
`points.to.classify`	Test data matrix
`kernel.type`	Default is the Gaussian kernel (`rbfdot`)
`kernel.parameters`	Value of the hyper-parameter of the kernel
`C.val`	C parameter for NHC
`return.classification.only`	TRUE returns only the class of the test data FALSE returns the class of the test data, number of support vectors used and the radii per hypersphere.

The `MultiClass.NHC()` function can be found below and will be explained by referring to the line numbers:

- Lines 003 to 007 are explained in Table 2.4.
- Line 009 loads the `kernlab` package so that we can use the Gaussian kernel and the `ipop()` function.
- Lines 011 to 022 find the number of classes and separates each class into its own matrix and stores it in a list called `data.list`.
- The `hypersphere.info()` function that is written in lines 024 to 058 finds the radius, alpha values, Gram matrix and number of support vectors for one class in the data matrix.
- Line 061 runs the `hypersphere.info()` function for each matrix (for each class) in `data.list`. This is stored in a list called `hypersphere.output`.
- An empty list for the $\boldsymbol{\alpha}^*$ vectors is created in lines 062 and 063.
- An empty vector for the radii is created in line 064.
- An empty list for the Gram matrices is created in lines 065 and 066.
- An empty vector for the number of support vectors is created in line 067.
- The empty vectors and lists created in lines 062 to 067 are all the size of the number of classes.
- Lines 068 to 074 separate the information in the `hypersphere.output` list to the vectors and lists created in lines 062 to 067. The g^{th} item in each list or vector always corresponds to the g^{th} class.
- In line 077 to 096 we use equation (2.22) to find the distance from each object to be classified to the center of each hypersphere.
- The squared dissimilarity function in equation (2.29) is used to find the squared dissimilarity measure per class in line 98.
- Now that we have the dissimilarity measure per class, we find the name of the class that each object was classified into in lines 100 to 103.
- Lines 104 to 109 return the class of the test objects and if `return.classification.only` is set as `FALSE`, it returns the radius and number of support vectors used as well.

The `MultiClass.NHC()` function given below was developed as part of this thesis.

```
001MultiClass.NHC<-function(data, class.vector, points.to.classify,
                      kernel.type=rbfdot, kernel.parameters=0.2,
                           C.val=1, return.classification.only=TRUE)
002{
003   # data is the datamatrix without the class parameter (usually
           training data)
004   # class.vector is the vector containing the class name or
           number for each item in data
005   # points.to.classify is the test data
006   # return.classification.only if TRUE returns the classes of
           points.to.classify only,
007   # otherwise it returns classes, number of support vectors and
           radius vector
008
009   require(kernlab)
010
011   data<-as.matrix(data)
012   class.names<-unique(class.vector)
013   p<-ncol(data)
014   n.class<-length(class.names)
015
016   ### Create a list with each group as a separate matrix ###
017
018   data.list<-list()
019   length(data.list)<-length(class.names)
020
021   for (i in 1:n.class)
022         data.list[[i]]<-data[class.vector==class.names[i],]
```

```
023  ##########################################################
024  ### hypersphere.info is a function that the radius,
          alpha values, Gram matrix and number of support vectors
               for each class in the data matrix
025
026  hypersphere.info<-function(data.mat=data,
          kernelType=kernel.type, kernel.par=kernel.parameters,
               C=C.val)
027  {
028       # data.mat is the matrix for one class
029       # gamma value is given in kernel.par
030       # data has p dimension
031       # data has n objects
032
033       ### Find values for ipop function to find the alpha
               values ###
034       n<-nrow(data.mat)
035       rownames(data.mat)<-NULL
036       data.mat<-as.matrix(data.mat)
037       Gram.mat<-kernelMatrix(kernelType(kernel.par),data.mat)
038       c.vec<-diag(Gram.mat)
039       A.vec<-matrix(1,nrow=1,ncol=n)
040       b.unit<-1
041       l.vec<-rep(0,n)
042       u.vec<-rep(C,n)
043       r.unit<-0
044       alpha.vec<-primal(ipop(c=c.vec, H=Gram.mat, A=A.vec,
               b=b.unit, l=l.vec, u=u.vec, r=r.unit))
```

```
045         ##### Find the radius ###
046         support.vectors<-data.mat[alpha.vec>0.00001,]
047
048         #Use (2.24) to caluculate radius squared where z is any
                 support vector
049         z<-support.vectors[1,]
050         kzz<-(kernelType(kernel.par))(z,z)#First term in (2.23)
051         kzx<-
               Gram.mat[min(which(apply(t(data.mat)==z,2,prod)!=0)),]
052
053         #equation (2.24)
054         radius.sq<-kzz - 2*t(alpha.vec)%*%kzx +
                 t(alpha.vec)%*%Gram.mat%*%alpha.vec
055         radius<-sqrt(radius.sq)
056
057         return(list(radius=radius, alpha.vec=alpha.vec,
                 Gram.mat=Gram.mat, n.sv=nrow(support.vectors)))
058   }
059   ############################################################
060
061   hypersphere.output<-lapply(data.list,hypersphere.info)
062   alpha.list<-list()
063   length(alpha.list)<-n.class
064   radius.vec<-rep(0,n.class)
065   Gram.list<-list()
066   length(Gram.list)<-n.class
067   n.sv<-rep(0,n.class)
```

```
for (i in 1:n.class)
{
      alpha.list[[i]]<-hypersphere.output[[i]]$alpha.vec
      radius.vec[i]<-hypersphere.output[[i]]$radius
      Gram.list[[i]]<-hypersphere.output[[i]]$Gram.mat
      n.sv[i]<-hypersphere.output[[i]]$n.sv
}

############ Classify the given points ###########
kernelrbf<-(kernel.type(kernel.parameters))
kzz<-matrix(apply(points.to.classify, 1,
           function(z) kernelrbf(z,z)), ncol=1)

t.alpha.kzx.mat<-NULL
kzx.func<-function(one.object)
{
      t(alpha.vec) %*%
           matrix(apply(data[(1:(nrow(data)))[class.vector==
                class.names[i]],], 1, function(x)
                     kernelrbf(one.object,x)),ncol=1)
}
for (i in 1:n.class)
{
      alpha.vec<-alpha.list[[i]]
      t.alpha.kzx.mat<-cbind(t.alpha.kzx.mat,
           apply(points.to.classify, 1, kzx.func))
}
```

```
090   rownames(t.alpha.kzx.mat)<-1:nrow(t.alpha.kzx.mat)
091
092   points.to.classify.radii<-NULL
093
094   #equation (2.22)
095   for (i in 1:n.class)
096         points.to.classify.radii<-cbind(points.to.classify.radii,
                kzz- 2*t.alpha.kzx.mat[,i,drop=FALSE]+
                  rep(t(alpha.list[[i]])%*% Gram.list[[i]]%*%
                     alpha.list[[i]], nrow(points.to.classify)))
097
098   dissimilarities<-t(t(points.to.classify.radii)/(radius.vec^2))
099
100   minimum.dist<-as.matrix(apply(dissimilarities, 1, min))
101   position.of.minimum<-matrix(apply(dissimilarities,2,
         function(x) minimum.dist==x),ncol=n.class)
102   class.of.points<-apply(position.of.minimum, 1, function(x)
         which(x==TRUE))
103   class.names.of.points<-
         matrix(class.names[class.of.points],ncol=1)
104   if (return.classification.only)
105   {
106         return(list(class.names.of.points=class.names.of.points))
107   }else{
108         return(list(class.names.of.points=class.names.of.points,
                num.support.vec=n.sv, radii=radius.vec))
109   }
110}
```

2.6 Conclusion

In Chapter 2 we studied the all enclosing hypersphere in Section 2.2 and the υ-soft hypersphere in Section 2.3. Once we knew how to find the hypersphere for a one-class dataset, we could find the hypersphere for each class in a dataset with more than one class. An object can now be classified into the class with the smallest dissimilarity measure. The classification of objects was discussed in Section 2.4 and we then discussed the implementation of NHC in the R software. The `MultiClass.NHC()` function can be used for multi-class NHC and will be used in Chapter 4 and Chapter 5.

CHAPTER 3
CLASSIFICATION TECHNIQUES

3.1 Introduction

In Chapter 2 we discussed NHC. To study the classification performance of NHC, we will compare it to other classification techniques in Chapter 5. In Chapter 3 we will look at these other classification techniques. We review two techniques that are known to be good classifiers and can be used for multi-class classification. These techniques are support vector machine classification in Section 3.2 and random forests in Section 3.3. We introduce the Penalised LDA technique which was designed for when we have more variables than objects in Section 3.4. We will now discuss these techniques and specifically look at the multi-class classifiers for each.

3.2 Support Vector Machine

The Support Vector Machine (SVM) was first introduced by Boser, Guyon and Vapnik (1992) and Vapnik (1998). The SVM is a very popular classifier and has been used in research by many authors. This section will discuss the SVM classification technique and how it can be used in multi-class classification. We will first look at the linear SVM in Section 3.2.1 and then the non-linear SVM in Section 3.2.2. A convex loss function is optimized under certain constraints for both versions of SVMs. Section 3.2.3 will look at the extension of two-class classification to multi-class classification.

3.2.1 Linear SVM

The linear SVM in this section is adapted from Deng, Tian & Zhang (2013:41-61). We will first look at the case where there are only two classes. We will use a training dataset as was defined in equation (2.31), but we will only have two classes so that $y_i \in \{-1, 1\}, i = 1, \dots, n$. For the SVM we need to find a real function $g(\boldsymbol{x})$ in $\mathbb{R}^p$ to predict the value of y for any $\boldsymbol{x}$ by the decision function,

$$f(\boldsymbol{x}) = sign(g(\boldsymbol{x})),$$

where $sign(a) = \begin{cases} -1, & a < 0 \\ 1, & a \geq 0 \end{cases}$.

The training set is used to separate the $\mathbb{R}^p$ space into two regions so that we can classify new objects into one of the two classes. When objects of two different classes are linearly separable, we can draw a straight line between the objects of the two classes. The one class is the positive class and the other class is the negative class. This can be seen in Figure 3.1 below.

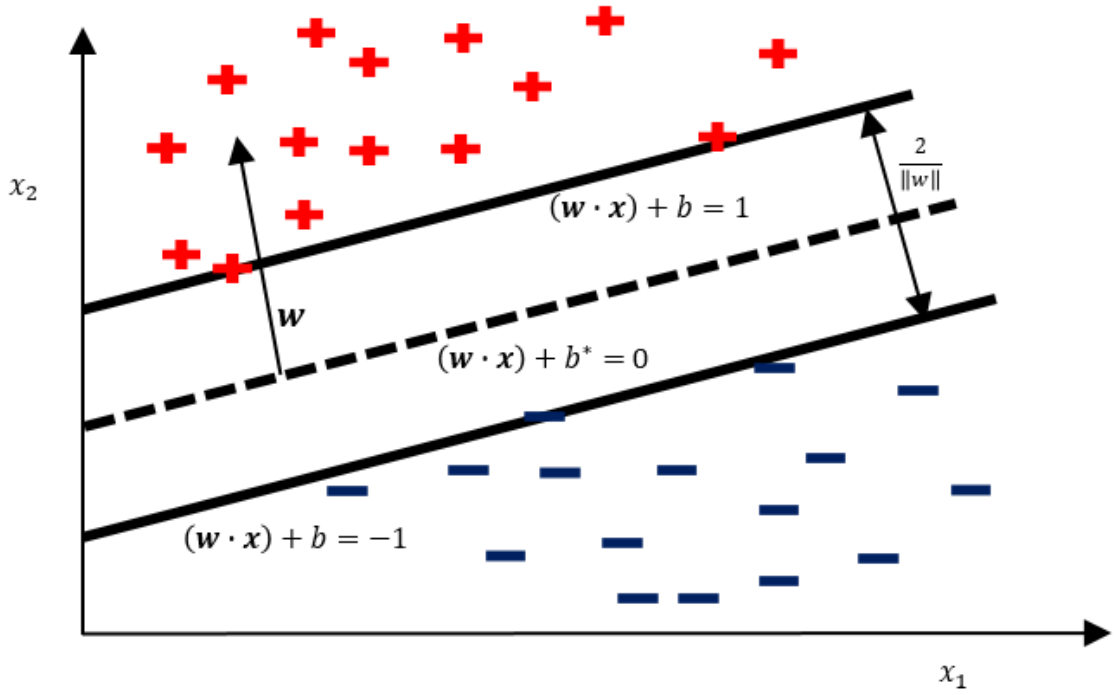

Figure 3.1: Representation of the maximal margin method of the SVM.

The hyperplane that separates the two classes can be defined as $(\boldsymbol{w}\cdot\boldsymbol{x})+b=0$, where $\boldsymbol{w}=(w_1,w_2)^T$ if the objects are in two dimensions and $\boldsymbol{x}=(x_1,x_2)^T$. For ease of explanation, we will be working in two dimensions, but it can easily be extended to more dimensions. Everything that holds for two dimensions will also hold for any p dimensions. We know that when we have two classes, a new object can be classified by determining on which side of the separating line it falls. The straight line will separate the plane into two regions: $(\boldsymbol{w}\cdot\boldsymbol{x})+b\geq 0$ and $(\boldsymbol{w}\cdot\boldsymbol{x})+b<0$. We can therefore determine the class of any object by finding $y=sign((\boldsymbol{w}\cdot\boldsymbol{x})+b)$. We call this method linear SVM classification, because the hyperplane that is used to separate the $\mathbb{R}^p$ space into two regions is linear. We can define this hyperplane as $\{\boldsymbol{x}\colon g(\boldsymbol{x})=(\boldsymbol{w}\cdot\boldsymbol{x})+b=0\}$.

There are many different straight lines that can be drawn, but we want to find the line that will optimally separate the two classes. We will be using the maximal margin method. Figure 3.1 describes the maximal margin method. This method is used by drawing two parallel lines with maximum distance between them where each line touches at least one object in a different class. These two parallel lines are called the support hyperplanes. The line that is drawn exactly in the middle of these two lines is the best separating hyperplane. The two support hyperplanes have a given normal direction $\boldsymbol{w}$. The normal direction that maximises the margin, is selected. The vectors that lie on the support hyperplanes are called the support vectors. The separating hyperplane is $\{\boldsymbol{x}: (\boldsymbol{w} \cdot \boldsymbol{x}) + b = 0\}$, since the two support hyperplanes can be written as $\{\boldsymbol{x}: (\boldsymbol{w} \cdot \boldsymbol{x}) + b = 1\}$ and $\{\boldsymbol{x}: (\boldsymbol{w} \cdot \boldsymbol{x}) + b = -1\}$.

We want to maximise the margin which is defined as $\frac{2}{\|\boldsymbol{w}\|}$. This leads to the following optimisation problem for $\boldsymbol{w}$ and b:

$$max_{\boldsymbol{w},b}\left[\frac{2}{\|\boldsymbol{w}\|}\right],$$

such that

$$(\boldsymbol{w} \cdot \boldsymbol{x}_i) + b \geq 1, \forall i: y_i = 1, \text{ and}$$

$$(\boldsymbol{w} \cdot \boldsymbol{x}_i) + b \leq 1, \forall i: y_i = -1.$$

This is equivalent to the primal problem

$$min_{\boldsymbol{w},b}\left[\frac{1}{2}\|\boldsymbol{w}\|^2\right], \tag{3.1}$$

subject to

$$y_i\big((\boldsymbol{w} \cdot \boldsymbol{x}_i) + b\big) \geq 1, i = 1, \dots, n. \tag{3.2}$$

Another way to find the maximal margin hyperplane is to solve its dual problem. This is done by directly solving the optimisation problem in (3.1) and (3.2). We derive the dual problem by using the Lagrange function. The Lagrange function is defined as follows:

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i\big(y_i\big((\boldsymbol{w} \cdot \boldsymbol{x}_i) + b\big) - 1\big), \tag{3.3}$$

where $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_n)^T$ is the Lagrange multiplier vector.

The following optimisation problem (dual problem) can be formulated from this (Deng et al., 2013: 50)

$$max_{\boldsymbol{\alpha}}[-\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)\alpha_i \alpha_j + \sum_{j=1}^{n} \alpha_j], \tag{3.4}$$

subject to

$$\sum_{i=1}^{n} y_i \alpha_i = 0, \text{ and} \tag{3.5}$$

$$\alpha_i \geq 0, i = 1, \dots, n. \tag{3.6}$$

When optimising, finding the maximum is the same as finding the minimum of the negative of the same function. This is applied to equation (3.4) and a convex quadratic problem is derived. We now have the optimisation problem

$$min_{\boldsymbol{\alpha}}\ [\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)\alpha_i \alpha_j - \sum_{j=1}^{n} \alpha_j], \tag{3.7}$$

subject to

$$\sum_{i=1}^{n} y_i \alpha_i = 0, \text{ and} \tag{3.8}$$

$$\alpha_i \geq 0, i = 1, \dots, n. \tag{3.9}$$

We are still considering the linearly separable problem and by solving the dual problem in (3.7), (3.8) and (3.9), we obtain the solutions $\boldsymbol{\alpha}^* = (\alpha_1^*, \dots, \alpha_n^*)^T$ where there must be a nonzero component α_j^*. We can obtain the unique solution to the primal problem in (3.1) and (3.2) for any nonzero component α_j^* of $\boldsymbol{\alpha}^*$ in the following way (Deng et al., 2013: 52):

$$\boldsymbol{w}^* = \sum_{i=1}^{n} \alpha_i^* y_i \boldsymbol{x}_i, \text{ and}$$

$$b^* = y_j - \sum_{i=1}^{n} \alpha_i^* y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}_j).$$

We only need $\boldsymbol{w}^*$ and b^* to find the optimal separating hyperplane. Before we go further, we want to define support vectors. By solving the optimisation problem in (3.7), we find the optimal α_i values as $\boldsymbol{\alpha}^* = (\alpha_1^*, \dots, \alpha_n^*)^T$. When we have input $\boldsymbol{x}_i$, which is associated with the training object $(\boldsymbol{x}_i, y_i)$, it is said to be a support vector if the corresponding component α_i^* of $\boldsymbol{\alpha}^*$ is nonzero. By looking at Figure 3.1, we can see that the support vectors are the objects lying on the $(\boldsymbol{w} \cdot \boldsymbol{x}) + b = 1$ support line or the $(\boldsymbol{w} \cdot \boldsymbol{x}) + b = -1$ support line. These support vectors are used to determine the optimal separating hyperplane.

Not all datasets with two classes will be completely linearly separable. Some of the objects in the positive class may lie between the objects in the negative class and some of the objects in the negative class may lie between the objects in the positive class. The hyperplane can therefore not completely separate the two classes. We will introduce slack variables, $\xi_i \geq 0$ for $i = 1, \dots, n$, to relax the requirement in order to separate the objects correctly.

We must allow the existence of training objects that violate the constraints $y_i g(\boldsymbol{x}_i) = y_i\big((\boldsymbol{w}^* \cdot \boldsymbol{x}_i) + b^*\big) \geq 1$ by introducing slack variables. We will now rewrite equation (3.2) as

$$y_i\big((\boldsymbol{w} \cdot \boldsymbol{x}_i) + b\big) \geq 1 - \xi_i\,, \qquad i = 1, \dots, n.$$

We want to make the above violation as little as possible. This can be done by superimposing a penalty upon the ξ_i in the objective function. The primal problem in (3.1) and (3.2) will be changed by adding a term $\sum_{i=1}^{n} \xi_i$ to the objective function:

$$min_{\boldsymbol{w},b,\boldsymbol{\xi}}\left[\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i\right], \tag{3.10}$$

subject to

$$y_i\big((\boldsymbol{w} \cdot \boldsymbol{x}_i) + b\big) \geq 1 - \xi_i, \quad i = 1, \dots, n, \text{ and} \tag{3.11}$$

$$\xi_i \geq 0, \quad i = 1, \dots, n, \tag{3.12}$$

where $\boldsymbol{\xi} = (\xi_1, \dots, \xi_n)^T$, and $C > 0$ is a penalty parameter. The parameter C is referred to as a cost parameter. The objective function (3.10) will minimise $\|\boldsymbol{w}\|^2$, which maximises the margin. It will also minimise $\sum_{i=1}^{n} \xi_i$ which is what we wanted, because $\sum_{i=1}^{n} \xi_i$ is a measurement of violation of the constraints $y_i\big((\boldsymbol{w} \cdot \boldsymbol{x}_i) + b\big) \geq 1, i = 1, \dots, n$. The parameter C determines the weighting between the two terms in the objective function (3.10).

To find the solution to the primal problem in (3.10) to (3.12) we solve its dual problem. The Lagrange function corresponding to the primal problem in (3.10) to (3.12) is:

$$L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i\big(y_i\big((\boldsymbol{w} \cdot \boldsymbol{x}_i) + b\big) - 1 + \xi_i\big) - \sum_{i=1}^{n}\beta_i\xi_i\,,$$

where $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_n)^T$ and $\boldsymbol{\beta} = (\beta_1, \dots, \beta_n)^T$ are Lagrange multiplier vectors.

The optimisation problem is the same as before, but with extra constraints and minimisation of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. The extra constraints are (Deng et al., 2013: 59)

$$C - \alpha_i - \beta_i = 0, \quad i = 1, \dots, n,$$

$$\alpha_i \geq 0, \quad i = 1, \dots, n, \text{ and}$$

$$\beta_i \geq 0, \quad i = 1, \dots, n,$$

which is equivalent to

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, n.$$

This implies that only $\boldsymbol{\alpha}$ has to be minimised, because $\boldsymbol{\beta}$ is no longer a constraint. The threshold b is solved in exactly the same way as for the linearly separable problem without slack variables. The following algorithm can now be formulated for support vector machine classification (SCMC).

Algorithm 3.1 (Linear Support Vector Machine Classification)

(1) Input the training set $T = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$, where $\boldsymbol{x}_i \in \mathbb{R}^p, y_i \in \mathcal{Y} = \{-1, 1\}$, $i = 1, \dots, n$.

(2) Choose an appropriate penalty parameter $C > 0$.

(3) Construct and solve the convex quadratic program

$$min_{\boldsymbol{\alpha}}[\tfrac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)\alpha_i \alpha_j - \sum_{i=1}^{n} \alpha_i]$$

subject to

$$\sum_{i=1}^{n} y_i \alpha_i = 0,$$

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, n\ ,$$

obtaining a solution $\boldsymbol{\alpha}^* = (\alpha_1^*, \dots, \alpha_n^*)^T$.

(4) Compute b^*: choose a component of $\boldsymbol{\alpha}^*, \alpha_j^* \in (0, C)$ with corresponding support vector $\boldsymbol{x}_j$ and compute

$$b^* = y_j - \sum_{i=1}^{n} \alpha_i^* y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)\ ;$$

(5) Construct the linear classifier (decision function) as:

$$f(\boldsymbol{x}) = sign(g(\boldsymbol{x})), \text{ where}$$

$$g(\boldsymbol{x}) = \sum_{i=1}^{n} y_i \alpha_i^* (\boldsymbol{x}_i \cdot \boldsymbol{x}) + b^*.$$

3.2.2 Non-linear SVM

The linear SVM is not always the best choice, because classes are sometimes not linearly separable at all. We will now look at the non-linear SVM, which is more complicated than the linear SVM, but we can derive it by adjusting and extending the linear SVM that was discussed in Section 3.2.1. This section is adapted from Deng et al. (2013:81-92).

The objects in the dataset are in the $\mathbb{R}^p$ input space, but non-linear SVM classification cannot be done in $\mathbb{R}^p$. We will therefore transform the objects to the Hilbert space ($\mathcal{H}$). As was discussed in Section 2.2, the map Φ transforms a p-dimensional vector $\boldsymbol{x}$ into another m-dimensional vector $\Phi(\boldsymbol{x})$ in Hilbert space.

Once the objects have been mapped to the Hilbert space, we can now find the linear separating hyperplane $\{\boldsymbol{x}: (\boldsymbol{w}^* \cdot \Phi(\boldsymbol{x})) + b^* = 0\}$ in the Hilbert space. The decision function $f(\boldsymbol{x}) = sign((\boldsymbol{w}^* \cdot \Phi(\boldsymbol{x}) + b^*)$ is used in the Hilbert space.

The distance between the two support hyperplanes in the Hilbert space can still be represented by $\frac{2}{\|\boldsymbol{w}\|}$. The two support hyperplanes can now be expressed as

$$(\boldsymbol{w} \cdot \Phi(\boldsymbol{x})) + b = 1 \text{ and } (\boldsymbol{w} \cdot \Phi(\boldsymbol{x})) + b = -1.$$

We can construct the primal problem similar to the problem in Section 3.2.1, but the objects will now be mapped to the Hilbert space. The optimisation problem is defined as in Section 3.2.1, but the object $\boldsymbol{x}$ is replaced by $\Phi(\boldsymbol{x})$**.** We know from Section 2.2 that the dot product between two objects mapped to the Hilbert space can be replaced by a kernel function. We will use the Gaussian kernel in SVMC for this thesis. The $\boldsymbol{\alpha}$ and b are solved in the same way as in Algorithm 3.1. The following algorithm can now be constructed for non-linear SVMC.

Algorithm 3.2 (Non-linear Support Vector Machine Classification)

(1) Input the training set $T = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$,

where $\boldsymbol{x}_i \in \mathbb{R}^p, y_i \in \mathcal{Y} = \{-1,1\}, \ i = 1, \dots, n$.

(2) Choose an appropriate kernel $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and a penalty parameter $C > 0$.

(3) Construct and solve the convex quadratic program

$$min_{\boldsymbol{\alpha}}[\tfrac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)\alpha_i\alpha_j - \sum_{i=1}^{n}\alpha_i]$$

subject to

$$\sum_{i=1}^{n} y_i\alpha_i = 0 \ ,$$

$$0 \le \alpha_i \le C, \quad i = 1, \dots, n$$

obtaining a solution $\boldsymbol{\alpha}^* = (\alpha_1^*, \dots, \alpha_n^*)^T$.

(4) Compute b^*: choose a component of $\boldsymbol{\alpha}^*, \alpha_j^* \in (0, C)$ with corresponding support vector $\Phi(\boldsymbol{x}_j)$ and compute

$$b^* = y_j - \sum_{i=1}^{n}\alpha_i^* y_i K(\boldsymbol{x}_i, \boldsymbol{x}_j) \ .$$

(5) Construct the non-linear classifier (decision function) as:

$$f(\boldsymbol{x}) = sign(g(\boldsymbol{x})),$$

where

$$g(\boldsymbol{x}) = \sum_{i=1}^{n} y_i\alpha_i^* K(\boldsymbol{x}_i, \boldsymbol{x}_j) + b^*.$$

3.2.3 Multi-class SVM

In Section 3.2.1 and Section 3.2.2 we worked with a dataset that has only two classes. We will now extend SVM classification to the multi-class setting. This section is adapted from Deng et al. (2013:232-234).

Since we are now looking at the multi-class SVM, we will have to redefine our training dataset so that there are G classes. As before we need to find a decision function $f(\boldsymbol{x})$ in $\mathbb{R}^p$, such that the class number y for any $\boldsymbol{x}$ can be predicted by $y = f(\boldsymbol{x})$. We will now be separating the $\mathbb{R}^p$ space into G regions according to the training set and this can be used to classify new objects.

We discuss three methods for multi-class classification. The first method is called one-versus-one. This method is performed by finding all the possible pairs of the G classes and finding a decision function for each pair. Each pair will be $(i,j) \in \{(i,j)|i<j;\ i,j=1,\dots,G\}$. This will result in $\frac{G(G-1)}{2}$ decision functions. Each pair will have a binary classification problem to separate the i^{th} class from the j^{th} class, which will be $g^{i-j}(\boldsymbol{x})$, and the corresponding decision function can be stated as follows:

$$f^{i-j}(\boldsymbol{x}) = \begin{cases} i, & g^{i-j}(\boldsymbol{x}) > 0; \\ j, & otherwise. \end{cases}$$

Each time an object is classified into a class, that class gets a point. The object is classified into the class with the highest number of points. If two or more classes have the same number of points, the object is unclassified. This is therefore not a great approach.

The second method for multi-class classification is called the one-versus-the-rest method. This method has G binary problems compared to the $\frac{G(G-1)}{2}$ binary problems in the previous method. The j^{th} binary problem separates the j^{th} class from the rest of the classes. This gives the decision function $f^j(\boldsymbol{x}) = sign(g^j(\boldsymbol{x}))$ for $j = 1,2,\dots,G$. We will now perform the multi-class classification according to $g^1(\boldsymbol{x}), g^2(\boldsymbol{x}), \dots, g^G(\boldsymbol{x})$. There may be cases where the test object is not classified into any class or into more than one class. This is solved by predicting the test object into the class where $g^j(\boldsymbol{x})$ is the largest for $j = 1,2,\dots,G$. This leads to the following algorithm.

Algorithm 3.3 (One-versus-the-rest algorithm)

(1) Input the training set

$$T = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$$

where $\boldsymbol{x}_i \in \mathbb{R}^p, y_i \in \mathcal{Y} = \{1,2,\dots,G\},\ i = 1,\dots,n.$

(2) For $j = 1,2,\dots,G$, construct the training set of the j^{th} binary problem with the training set

$$T^j = \{(\boldsymbol{x}_1, y_1^j), \dots, (\boldsymbol{x}_n, y_n^j)\}$$

where $y_i^j = \begin{cases} 1, & if\ y_i = j; \\ 0, & otherwise. \end{cases}$

Find the corresponding function

$$g^j(\boldsymbol{x}).$$

(3) Construct the multi-class classifier (decision function) as:

$$f(\boldsymbol{x}) = argmax_{j=1,\ldots,G} g^j(\boldsymbol{x}).$$

The third method is Weston and Watkins' multi-class SVM (Weston and Watkins (1998)) where all classes are considered at once for a decision function. We now use the decision function:

$$f(\boldsymbol{x}) = argmax_{g=1\ldots G}(\sum_{i:y_i=g} \alpha_i^* K(\boldsymbol{x}, \boldsymbol{x}_j) + b_g^*).$$

Figure 3.2 shows the multi-class decision boundary for Weston and Watkins' method. We used the first two standardised variables of the Iris dataset as in Chapter 2. The Gaussian kernel with its hyper-parameter (γ) equal to 1 and $C = 1$ was used for the multi-class classifier.

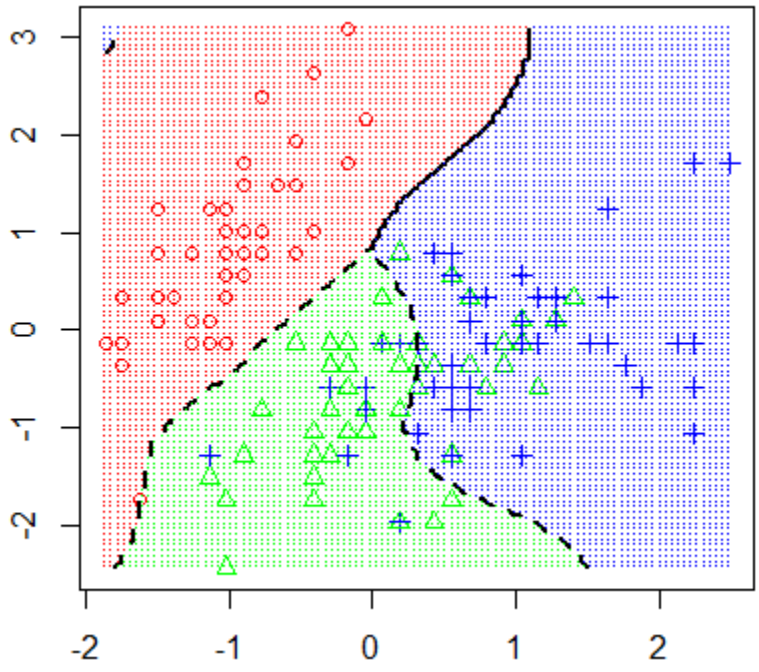

Figure 3.2: Representation of the Weston and Watkins SVM decision boundary when $\gamma = 1$ and $C = 1$.

3.3 Random Forests

3.3.1 Single tree

To be able to understand random forests, we first need to know what trees and nodes are. The tree discussed here was introduced by Breiman, Friedman, Olshen, and Stone (1984). We can use trees for regression or classification, but for this thesis, we will only be discussing classification trees. This section is adapted mostly from Izenman (2008: 282-283).

The following diagram in Figure 3.3 explains a classification tree. The root node contains the entire training set. We will now have a condition that can be true or false. This is called a Boolean condition. There will therefore be a binary split into two nodes from the root node. Each node can be terminal or nonterminal. A terminal node will not split further and will be assigned a class. If a node is nonterminal, there will be another Boolean condition and this node will split into two new nodes that can be terminal or nonterminal. This will continue until all the data are classified into a class. More than one terminal node may contain the same class. We will again use the training set in equation (2.31) as in the previous chapters.

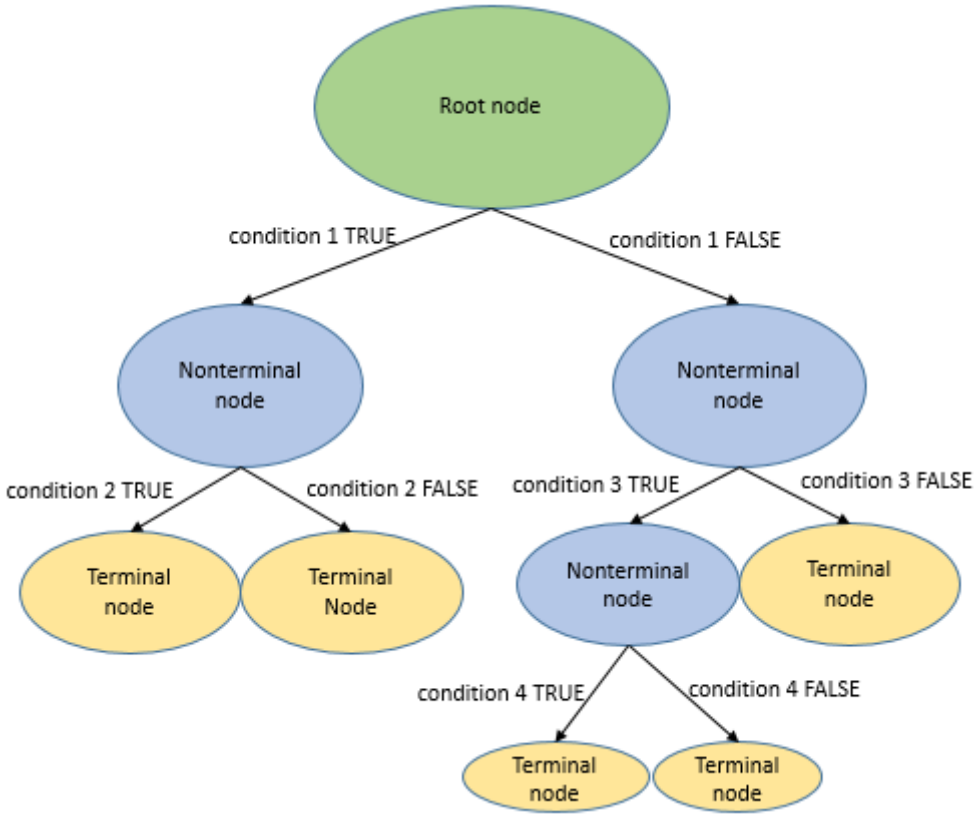

Figure 3.3: Graphical representation of a tree and nodes.

At each node, the algorithm has to decide which variable will be best to split. A few strategies that can be used for splitting is discussed in Izenman (2008: 285-292).

A better practical explanation might be to consider the Iris dataset to show how to classify an object. We will use the Iris dataset that was introduced in Chapter 2. We will only be using Petal Width and Sepal Width variables for the explanation of a classification tree in Figure 3.4.

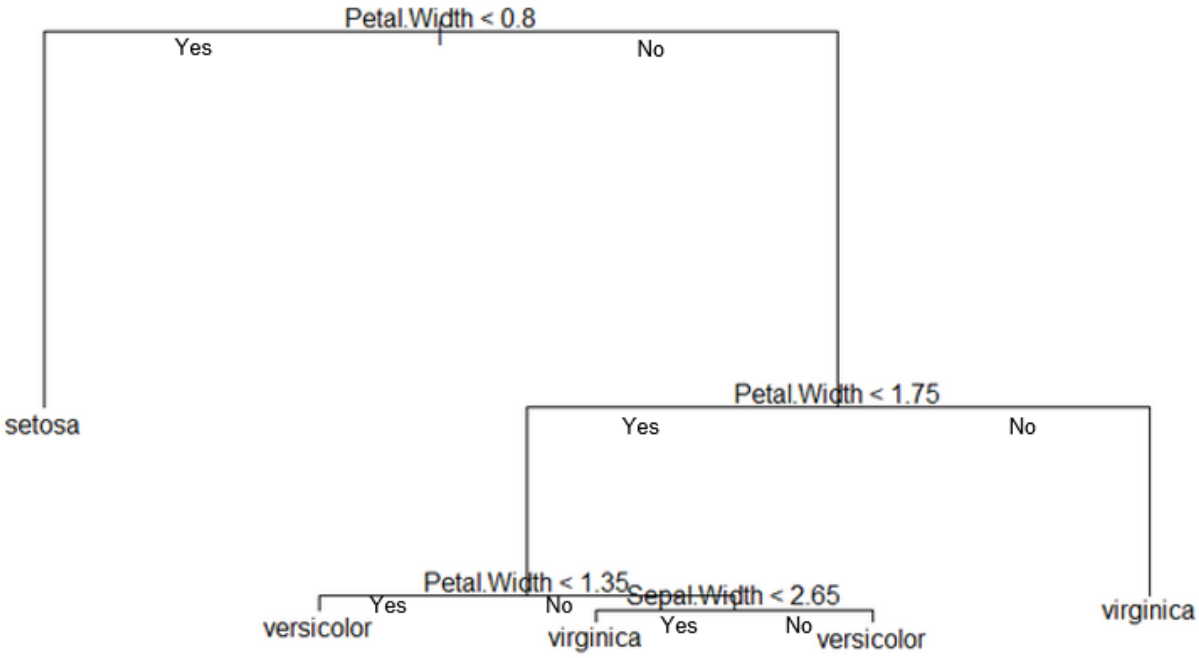

Figure 3.4: Classification tree for Iris dataset.

Source: Tang (2013).

At the top of Figure 3.4, we will have the entire dataset. Looking at all the objects, but only the Petal Width variable, we classify the objects with a Petal Width less than 0.8 into the Setosa class. Now we are left with all objects that have a Petal Width greater than or equal to 0.8. Looking at only these objects now, we will classify the objects with a Petal Width greater than or equal to 1.75 into the Virginica class. The objects that have not been classified yet will be classified into the Versicolor class if the object has a Petal Width less than 1.35. Next, all objects that have not been classified will be classified into the Virginica class if the object has a Sepal Width less than 2.65. All objects that have not been classified are now classified into the Versicolor class.

Now that we have an idea of what trees and nodes are, we will look at some extensions of tree methodology. For this thesis, we want to work with random forests, but to understand random forests, we need to briefly discuss bagging in Section 3.3.2 and boosting in Section 3.3.3. We will then discuss random forests in Section 3.3.4.

3.3.2 Bagging

Bagging was introduced by Breiman (1996). This section is adapted from Izenman (2008: 506-507). The phrase "bootstrap aggregation" is where the acronym "bagging" comes from. Bagging uses B different training sets where each training set is found by taking bootstrap samples of n objects with replacement from the original training set. Each object in the training set has an equal probability of being selected (which is $\frac{1}{n}$) when there are n objects in the training set. We can denote each of the training sets as $T^{*b} = \{(\boldsymbol{x}_1^{*b}, y_1^{*b}), \dots, (\boldsymbol{x}_n^{*b}, y_n^{*b})\}$ for $b = 1, \dots, B$.

For each of the B training sets, a classification tree is grown. We will be working with multi-class classification and will therefore have G different classes as in previous chapters. Now that we have B trees, we can use a test object called $\boldsymbol{z}$ and drop it down each of the trees. When $\boldsymbol{z}$ reaches a terminal node, it will be classified into the class of that node. We will have B classifications of $\boldsymbol{z}$ and the class that has the majority of the counts will be the class of $\boldsymbol{z}$.

3.3.3 Boosting

Boosting was originally designed for classification and later extended to regression, but for this thesis we will only look at classification. The name "boosting" comes from the fact that this technique improves or boosts the performance of a weak classifier. A weak classifier correctly classifies slightly more than 50% of the time. In this section, we will look at the AdaBoost.M1 boosting algorithm that was introduced by Freund and Schapire (1997). AdaBoost is an acronym for "adaptive boosting" and is an algorithm that is used for the binary classification problem. AdaBoost can be generalised to more than two classes and is called "AdaBoost.M1".

This section is adapted from Hastie et al. (2009: 337-340). We first need to define the training error rate. Let $F(\boldsymbol{x}_i)$ be the predicted class for an object. Then, when we work with the training dataset, we have training error

$$err = \frac{1}{n}\sum_{i=1}^{n} I(y_i \neq F(\boldsymbol{x}_i)).$$

Each object in the training set is assigned a weight which gives a weight vector $\boldsymbol{w} = (w_1, \dots, w_n)^{\mathrm{T}}$. The initial weights will all be equal to $\frac{1}{n}$. When using boosting, we select a weak classifier and repeatedly apply this to the training dataset with modified weights to train the classifier. We will perform M iterations that will result in M classifiers. Each classifier will be denoted as $F_m(\boldsymbol{x}_i)$, for $m = 1, \dots, M$. The final classifier will be calculated by assigning a weight, α_m, to each of the M classifiers and the weight is determined by how well the classifier performed.

The AdaBoost.M1 algorithm is given in Algorithm 3.4.

Algorithm 3.4 (AdaBoost.M1 algorithm)

(1) Initialise the object weights $w_i = \frac{1}{n}, i = 1, \dots, n.$

(2) For $m = 1,2, \dots, M$:

a) Fit a classifier (for example a tree) $F_m(\boldsymbol{x})$ to the training dataset using weights w_i.

b) Compute

$$err_m = \frac{\sum_{i=1}^{n} w_i I(y_i \neq F_m(\boldsymbol{x}_i))}{\sum_{i=1}^{n} w_i}.$$

c) Compute $\alpha_m = \log(\frac{1-err_m}{err_m})$.

d) Set $w_i = w_i \exp\{\alpha_m I(y_i \neq F_m(\boldsymbol{x}_i))\},\ i = 1, \dots, n.$

(3) Output $F(\boldsymbol{x}) = sign\{\sum_{m=1}^{M} \alpha_m F_m(\boldsymbol{x})\}$.

3.3.4 Random Forests

Breiman (2001) introduced random forests. This section is adapted from Hastie et al. (2009: 587-588). Random forests are similar to bagging. We will use the multi-class training set and we will now have G classes. As with bagging we draw B bootstrap samples from the training set. We will construct the B trees different from the way bagging constructed the trees. We will now use randomisation to construct trees. This is done by splitting each node in a random manner. The algorithm for random forest classification is given below.

Algorithm 3.5 (Random forest classification algorithm using random input selection at each tree node)

(1) Input the training set

$$T = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$$

where $\boldsymbol{x}_i \in \mathbb{R}^p, y_i \in \mathcal{Y} = \{1,2,\dots,G\},\ i = 1,\dots,n.$

Let $m =$ number of variables to be chosen at each node $(m << p)$ and $B =$ number of bootstrap samples.

(2) For $b = 1,2,\dots,B$:

Draw a bootstrap sample T^{*b} from the training set T.

From T^{*b}, grow a tree classifier using random input selection: at each node, randomly select a subset m of the p input variables and, using only the m selected variables, determine the best split at that node.

Using an input vector $\boldsymbol{z}$, define a classifier $F_b(\boldsymbol{z})$ having a single vote for the class of $\boldsymbol{z}$.

(3) The B randomised tree-structured classifiers $\{F_b(\boldsymbol{z})\}$ are collectively called a random forest.

(4) The object $\boldsymbol{z}$ is assigned to a class using majority votes as determined by the random forest.

Figure 3.5 shows the decision boundary of random forests when the model is fitted using the first two columns of the standardised Iris dataset. We used $m = 1$, so the single variable to split on is randomly selected from the two available ones. The decision boundary is very flexible which implies that the model is probably overfitting.

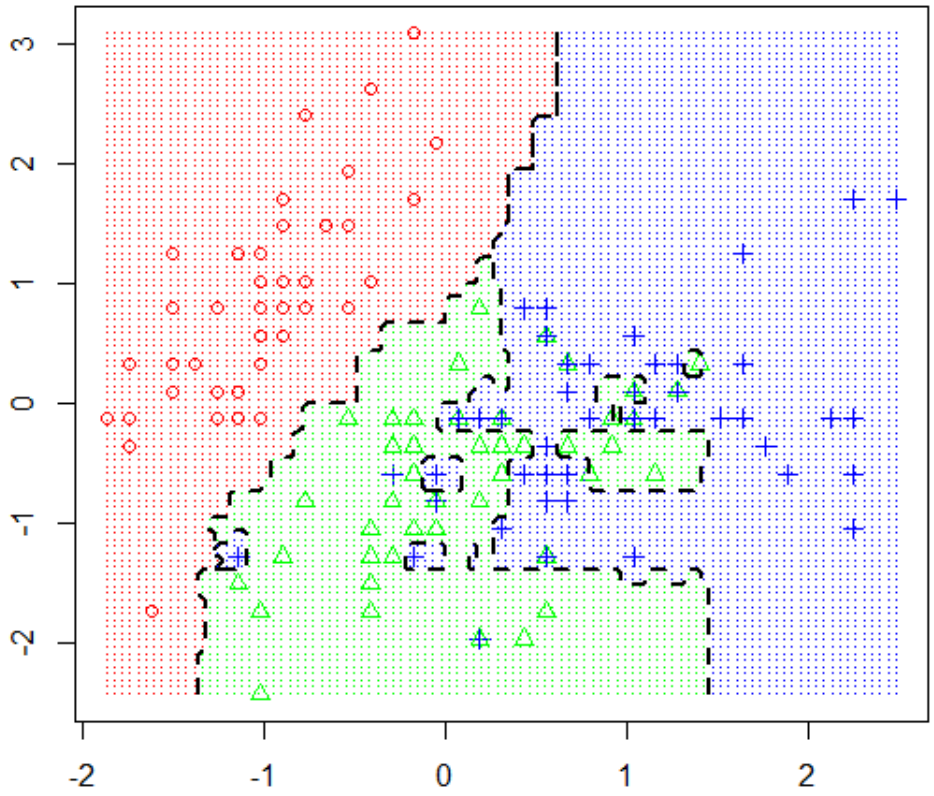

Figure 3.5: Representation of the decision boundary for random forest with $m = 1$.

3.4 Penalised LDA

3.4.1 Background

The penalised linear discriminant analysis (penalised LDA) technique was developed for datasets where $p \gg n$. This technique is discussed in Witten and Tibshirani (2011) and this section was adapted from their article. This section will briefly describe penalised LDA which is basically an extension of Fisher's LDA with a lasso penalty.

3.4.2 Fisher's LDA classifier

Before we can look at penalised LDA we must know how LDA works. For this section, we will look at Fisher's discriminant problem to derive the LDA classifier. We need to define a few terms before continuing:

- $\boldsymbol{X}$ is the data matrix where the features are centred to have mean zero,
- G is the number of classes,
- $\boldsymbol{y}$ is the class vector where $y \in \{1,2,\dots,G\}$,
- Π_g is class g,
- $\boldsymbol{Y}$ is an $n \times G$ matrix with Y_{ig} and indicator of whether object i is in Π_g,
- n is the number of rows in $\boldsymbol{X}$,
- p is the number of columns in $\boldsymbol{X}$,
- $\boldsymbol{x}_i$ is the i^{th} row in $\boldsymbol{X}$,
- $\boldsymbol{c}_i$ is the j^{th} column in $\boldsymbol{X}$,
- n_g is the number of objects in Π_g with $n = \sum_{g=1}^{G} n_g$, and
- $\widehat{\boldsymbol{\mu}}_g$ is the sample mean vector for class g.

We need to define the standard estimate of the within-class covariance matrix ($\widehat{\boldsymbol{\Sigma}}_w$) as well as the standard estimate of the between-class covariance matrix ($\widehat{\boldsymbol{\Sigma}}_b$), because we want to find a low dimensional projection of the objects such that the between-class variance is large relative to the within class variance.

The standard estimate of the within-class covariance matrix is given by

$$\widehat{\boldsymbol{\Sigma}}_w = \frac{1}{n}\sum_{g=1}^{G}\sum_{i\in\Pi_g}(\boldsymbol{x}_i - \widehat{\boldsymbol{\mu}}_g)(\boldsymbol{x}_i - \widehat{\boldsymbol{\mu}}_g)^{\mathrm{T}}, \tag{3.13}$$

and the standard estimate of the between-class covariance matrix is given by

$$\begin{aligned}\widehat{\boldsymbol{\Sigma}}_b &= \frac{1}{n}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} - \widehat{\boldsymbol{\Sigma}}_w \\ &= \frac{1}{n}\sum_{g=1}^{G} n_g\, \widehat{\boldsymbol{\mu}}_g \widehat{\boldsymbol{\mu}}_g^{\mathrm{T}} \\ &= \frac{1}{n}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}(\boldsymbol{Y}\boldsymbol{Y})^{-1}\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{X}.\end{aligned} \tag{3.14}$$

Now that we have defined $\widehat{\boldsymbol{\Sigma}}_w$ and $\widehat{\boldsymbol{\Sigma}}_b$ we can define the following problem that has to be solved:

$$max_{\boldsymbol{\beta}_g \in \mathbb{R}^p}\{\boldsymbol{\beta}_g^{\mathrm{T}}\widehat{\boldsymbol{\Sigma}}_b\boldsymbol{\beta}_g\}, \tag{3.15}$$

subject to

$$\boldsymbol{\beta}_g^{\mathrm{T}}\widehat{\boldsymbol{\Sigma}}_w\boldsymbol{\beta}_g \leq 1 \text{ and } \boldsymbol{\beta}_g^{\mathrm{T}}\widehat{\boldsymbol{\Sigma}}_w\boldsymbol{\beta}_i = 0,\ \forall i < g,$$

where the solution to the problem will be referred to as the g^{th} discriminant vector and denoted as $\widehat{\boldsymbol{\beta}}_g$. When $\widehat{\boldsymbol{\Sigma}}_w$ has full rank, the inequality constraint in (3.15) is replaced with an equality constraint (Witten and Tibshirani, 2011). There are $G-1$ non-trivial discriminant vectors in general which means that we can obtain the classification rule by computing projections $\boldsymbol{X}\widehat{\boldsymbol{\beta}}_1, \boldsymbol{X}\widehat{\boldsymbol{\beta}}_2, \ldots, \boldsymbol{X}\widehat{\boldsymbol{\beta}}_{G-1}$ and assigning each object to its nearest centroid in this transformed space. We can alternatively transform the objects by only using the first $g < G-1$ discriminant vectors. This will then result in reduced rank classification.

3.4.3 Penalised linear discriminant analysis

LDA performs well when we have $n > p$, but problems arise when we have $n < p$. The classification rule for LDA involves a linear combination of all p features. It will therefore become very difficult to interpret the classifier when p becomes large. When we have datasets in high dimensions, (3.15) does not result in a good classifier, because $\widehat{\boldsymbol{\Sigma}}_w$ is singular. We will now modify (3.15) by replacing $\widehat{\boldsymbol{\Sigma}}_w$ with $\widetilde{\boldsymbol{\Sigma}}_w$, where $\widetilde{\boldsymbol{\Sigma}}_w$ is a positive definite estimate of $\boldsymbol{\Sigma}_w$. We will use $\widetilde{\boldsymbol{\Sigma}}_w$ as the diagonal estimate $\mathrm{diag}(\hat{\sigma}_1^2, \hat{\sigma}_2^2, \ldots \hat{\sigma}_p^2)$ where $\hat{\sigma}_j^2$ is the j^{th} diagonal element of $\widehat{\boldsymbol{\Sigma}}_w$.

Equation (3.15) will now become

$$max_{\boldsymbol{\beta}_g \in \mathbb{R}^p}\{\boldsymbol{\beta}_g^{\mathrm{T}} \widehat{\boldsymbol{\Sigma}}_b \boldsymbol{\beta}_g\}, \tag{3.16}$$

subject to

$$\boldsymbol{\beta}_g^{\mathrm{T}} \widetilde{\boldsymbol{\Sigma}}_w \boldsymbol{\beta}_g \leq 1 \text{ and } \boldsymbol{\beta}_g^{\mathrm{T}} \widetilde{\boldsymbol{\Sigma}}_w \boldsymbol{\beta}_i = 0, \ \forall i < g,$$

which will be modified even further to make the results more interpretable.

The following problem in equation (3.16) is solved by finding the solution to $\widehat{\boldsymbol{\beta}}_g$:

$$max_{\boldsymbol{\beta}_g}\{\boldsymbol{\beta}_g^{\mathrm{T}} \widehat{\boldsymbol{\Sigma}}_b^g \boldsymbol{\beta}_g\}, \tag{3.17}$$

subject to

$$\boldsymbol{\beta}_g^{\mathrm{T}} \widetilde{\boldsymbol{\Sigma}}_w \boldsymbol{\beta}_g \leq 1,$$

where

$$\widehat{\boldsymbol{\Sigma}}_b^g = \frac{1}{n} \boldsymbol{X}^{\mathrm{T}} \boldsymbol{Y} (\boldsymbol{Y}\boldsymbol{Y})^{-1/2} \boldsymbol{P}_g^{\perp} (\boldsymbol{Y}\boldsymbol{Y})^{-1/2} \boldsymbol{Y}^{\mathrm{T}} \boldsymbol{X}. \tag{3.18}$$

Here $\boldsymbol{P}_1^{\perp} = \boldsymbol{I}$ and $\boldsymbol{P}_g^{\perp}$, where $g > 1$, is an orthogonal projection matrix into the space that is orthogonal to $(\boldsymbol{Y}\boldsymbol{Y})^{-1/2} \boldsymbol{Y}^{\mathrm{T}} \boldsymbol{X} \widehat{\boldsymbol{\beta}}_i$ for all $i < g$.

Penalty functions are now imposed on the discriminant vectors. Setting $g = 1$ we can find the first penalised discriminant vector $\widehat{\boldsymbol{\beta}}_1$ by solving the following problem:

$$max_{\boldsymbol{\beta}_1}\{\boldsymbol{\beta}_1^{\mathrm{T}} \widehat{\boldsymbol{\Sigma}}_b \boldsymbol{\beta}_1 - P_1(\boldsymbol{\beta}_1)\}, \tag{3.19}$$

subject to

$$\boldsymbol{\beta}_1^{\mathrm{T}} \widetilde{\boldsymbol{\Sigma}}_w \boldsymbol{\beta}_1 \leq 1,$$

where P_1 is a convex penalty function.

Using what we found in (3.17) and (3.18), we can now define the g^{th} penalised discriminant vector $\widehat{\boldsymbol{\beta}}_g$ to be the solution to

$$max_{\boldsymbol{\beta}_g}\{\boldsymbol{\beta}_g^{\mathrm{T}}\widehat{\boldsymbol{\Sigma}}_b^g\boldsymbol{\beta}_g - P_g(\boldsymbol{\beta}_g)\} \tag{3.20}$$

subject to

$$\boldsymbol{\beta}_g^{\mathrm{T}}\widetilde{\boldsymbol{\Sigma}}_w\boldsymbol{\beta}_g \leq 1.$$

Here P_g is a convex penalty function on the g^{th} discriminant vector. A minorisation algorithm is used to solve the optimisation problem in equation (3.20). The details of how (3.20) is solved will not be discussed in this thesis, but the reader is referred to Witten and Tibshirani (2011) for further information.

The following algorithm is used to obtain the g^{th} penalised discriminant vector.

Algorithm 3.6 (Obtaining the g^{th} penalised discriminant vector)

(1) If $g > 1$, define an orthogonal projection matrix $\boldsymbol{P}_g^{\perp}$ that projects onto the space that is orthogonal to $(\boldsymbol{YY})^{-1/2}\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{X}\widehat{\boldsymbol{\beta}}_i$ for all $i < g$.
Let $\boldsymbol{P}_1^{\perp} = \boldsymbol{I}$.

(2) Let $\widehat{\boldsymbol{\Sigma}}_b^g = \frac{1}{n}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}(\boldsymbol{YY})^{-1/2}\boldsymbol{P}_g^{\perp}(\boldsymbol{YY})^{-1/2}\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{X}$.
Note that $\widehat{\boldsymbol{\Sigma}}_b^1 = \widehat{\boldsymbol{\Sigma}}_b$.

(3) Let $\boldsymbol{\beta}_g^{(0)}$ be the first eigenvector of $\widetilde{\boldsymbol{\Sigma}}_w^{-1}\widehat{\boldsymbol{\Sigma}}_b^g$.

(4) For $m = 1,2,\ldots$ until convergence: let $\boldsymbol{\beta}_g^{(m)}$ be the solution to

$$max_{\boldsymbol{\beta}_g}\{2\boldsymbol{\beta}_g^{\mathrm{T}}\widehat{\boldsymbol{\Sigma}}_b^g\boldsymbol{\beta}_g^{(m-1)} - P_g(\boldsymbol{\beta}_g)\}$$

subject to

$$\boldsymbol{\beta}_g^{\mathrm{T}}\widetilde{\boldsymbol{\Sigma}}_w\boldsymbol{\beta}_g \leq 1.$$

Let $\widehat{\boldsymbol{\beta}}_g$ denote the solution at convergence.

Now that we have the algorithm, we need to know how to define P_g. Witten and Tibshirani (2011) give two specific forms for P_g namely the L_1 penalty and the fused lasso penalty. We will use the penalised LDA-L_1 method in this thesis and equation (3.20) can therefore be rewritten as

$$max_{\boldsymbol{\beta}_g}\{\boldsymbol{\beta}_g^{\mathrm{T}}\widehat{\boldsymbol{\Sigma}}_b^g\boldsymbol{\beta}_g - \lambda_g \sum_{j=1}^{p}|\hat{\sigma}_j\beta_{gj}|\}, \tag{3.21}$$

subject to

$$\boldsymbol{\beta}_g^{\mathrm{T}}\widetilde{\boldsymbol{\Sigma}}_w\boldsymbol{\beta}_g \leq 1.$$

To solve the problem of selecting the tuning parameter λ_g, we will fix a non-negative constant value for λ and then take

$$\lambda_g = \lambda \left\| \widetilde{\boldsymbol{\Sigma}}_w^{-1/2} \widehat{\boldsymbol{\Sigma}}_b^g \widetilde{\boldsymbol{\Sigma}}_w^{-1/2} \right\|, \tag{3.22}$$

where $\|\cdot\|$ indicates the largest eigenvalue. When $p \gg n$, this largest eigenvalue can be quickly computed by using the fact that $\widehat{\boldsymbol{\Sigma}}_b^g$ has low rank. The value of λ can be obtained by cross-validation.

3.5 Conclusion

Now that we have discussed three different classification techniques in the multi-class setting, we want to compare these classification techniques to the NHC which we discussed in Chapter 2. SVMC depends on a hyper-parameter, support vectors and a value of C for classification which is very similar to NHC. Before we compare all the techniques to NHC, we first want to examine the behaviour of the error rates and fraction of support vectors used for NHC and SVMC as well as the similarities between NHC and SVMC in Chapter 4. After the empirical study in Chapter 4 we will compare the performance of the classification techniques in Chapter 5.

CHAPTER 4
EMPIRICAL STUDY

4.1 Introduction

In Chapter 2 and Chapter 3 we studied classification techniques and showed that SVMC and NHC both use parameters C and γ (from the Gaussian kernel) as well as support vectors. This chapter will only study NHC and SVMC with the aim to investigate the behaviour of the error rates and number of support vectors for different values of C and γ. In Section 4.2 we will describe v-fold cross validation which will be used to estimate the error rates in Section 4.3 and Section 4.4. Section 4.3 is a short simulation study, while Section 4.4 will repeat what was done in Section 4.3 using real-world data.

4.2 v-fold cross-validation

There are many ways to estimate the error rate of a classifier. V-fold cross-validation is commonly used for this purpose. Mathematically, v-fold cross-validation error can be defined as:

$$CV_{error} = \sum_{i=1}^{v} \frac{n_i}{n} Err_i$$

where n is the sample size, n_i is the size of the i^{th} part, and $Err_i = \frac{1}{n_i}\sum_{k \in C_i} I(y_k \neq \hat{y}_k)$ (Gareth, Witten, Hastie, & Tibshirani, 2013: 184).

We will be working with 5-fold cross-validation. The dataset is divided into five roughly equal parts ($C_i, for\ i = 1,2,3,4,5$). We fit the classification model on four parts and the remaining part is used to test the fitted model. This is repeated five times and the average error is taken as the estimated error.

4.3 Simulation Study

In this section, we will use a simulation study to investigate the behaviour of the error rates and the number of support vectors used for different values of C and γ. We will also be comparing SVMC and NHC. For the simulated data, we will simulate datasets with small samples and datasets with large samples. We will consider cases with no correlation and cases with a positive correlation. The purpose of the study is to see whether sample size and

correlation has an influence on the behaviour of the error rates and the fraction of support vectors used.

4.3.1 Simulation setup

The datasets for the study in this section will be generated from the four different configurations given in Table 4.1.

- There will be three classes per dataset which will be generated as follows:

$$\boldsymbol{X}_1 \sim N(\boldsymbol{\mu}_1, \sum_j)$$

$$\boldsymbol{X}_2 \sim N(\boldsymbol{\mu}_2, \sum_j)$$

$$\boldsymbol{X}_3 \sim N(\boldsymbol{\mu}_3, \sum_j) \quad j = 1,2$$

where the mean vector for each class is

$$\boldsymbol{\mu}_1 = (0,0,0,0,0)$$

$$\boldsymbol{\mu}_2 = (1,1,1,1,1)$$

$$\boldsymbol{\mu}_3 = (0,0,1,1,1)$$

and the covariance matrices are (correlated no/yes)

$$\sum_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } \sum_2 = \begin{bmatrix} 1 & 0.7 & 0.7 & 0.7 & 0.7 \\ 0.7 & 1 & 0.7 & 0.7 & 0.7 \\ 0.7 & 0.7 & 1 & 0.7 & 0.7 \\ 0.7 & 0.7 & 0.7 & 1 & 0.7 \\ 0.7 & 0.7 & 0.7 & 0.7 & 1 \end{bmatrix}.$$

- All classes will have the same number of objects for each dataset.
  The sizes will be (small versus large):

$$n_1 = n_2 = n_3 = 100 \text{ and } n_1 = n_2 = n_3 = 400.$$

- The datasets can be summarised by the following configurations:

Table 4.1: Description of simulated datasets.

Configuration:	ρ (correlation)	n (sample size)
1	0	100
2	0.7	100
3	0	400
4	0.7	400

- We used 5-fold cross-validation to estimate the error rate.

- The techniques under investigation are SVMC and NHC. For each of the techniques, we used the same γ values as well as the same value for the C parameter. We used 100 equally spaced values between 0 and 5 for γ. The values used for the C parameter are $0.1, 1$ and 5.

- We estimated the cross-validation error and fraction of support vectors used for each technique for all configurations and values for the C parameter.

- The simulation steps will be repeated 20 times and the mean value for each γ value will be calculated. The results are displayed graphically for the cross-validation error and fraction of support vectors used over the different γ values.

4.3.2 Simulation results

We will now look at the behaviour of the error rates and the fraction of support vectors used over the values for γ. Figure 4.1 to Figure 4.12 show the behaviour of the four configurations for different values of C. The blue dotted line is the fraction of support vectors used for each technique. The solid red line is the 5-fold cross-validation error rate. SVMC is displayed on the left side and NHC on the right side. Figures 4.1 to 4.6 are the small samples and Figures 4.7 to 4.12 are the large samples. Figures 4.1 to 4.3 and Figures 4.7 to 4.9 are the uncorrelated cases. Figures 4.4 to 4.6 and Figures 4.10 to 4.12 are the correlated cases.

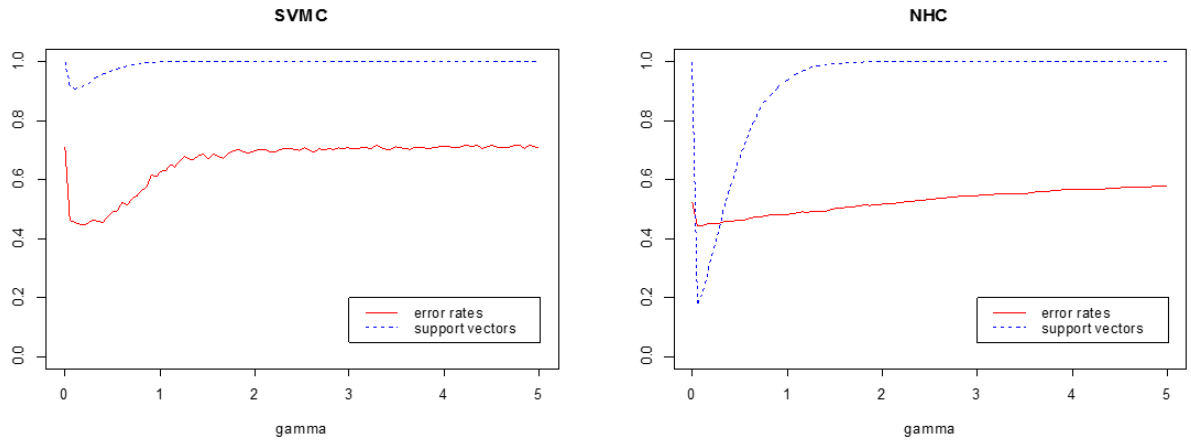

Figure 4.1: Behaviour of fraction of support vectors and error rates with $n = 100$, $\rho = 0$ and $C = 0.1$.

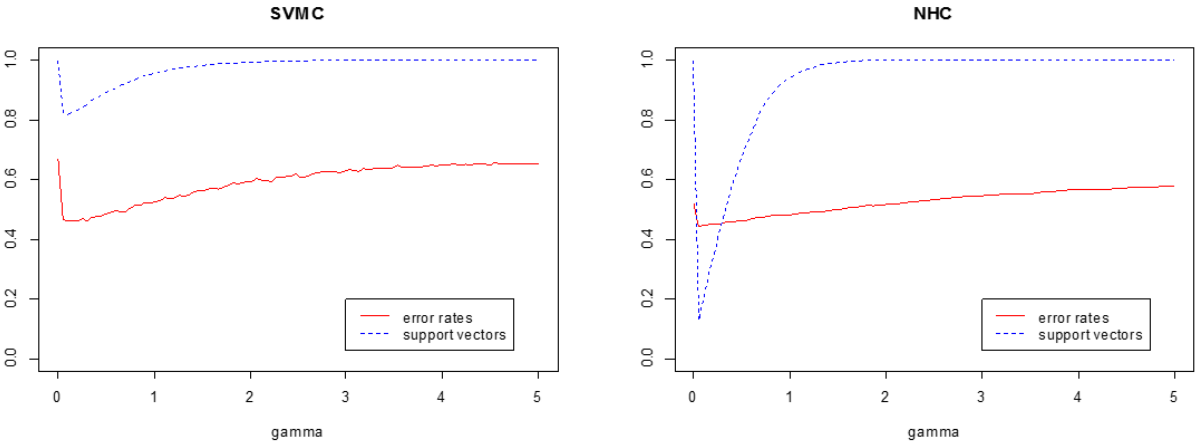

Figure 4.2: Behaviour of fraction of support vectors and error rates with $n = 100$, $\rho = 0$ and $C = 1$.

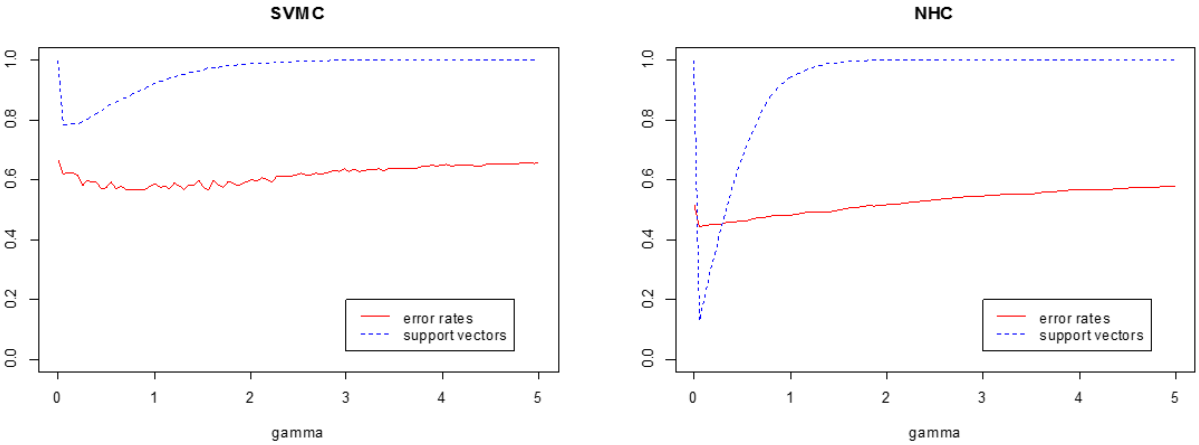

Figure 4.3: Behaviour of fraction of support vectors and error rates with $n = 100$, $\rho = 0$ and $C = 5$.

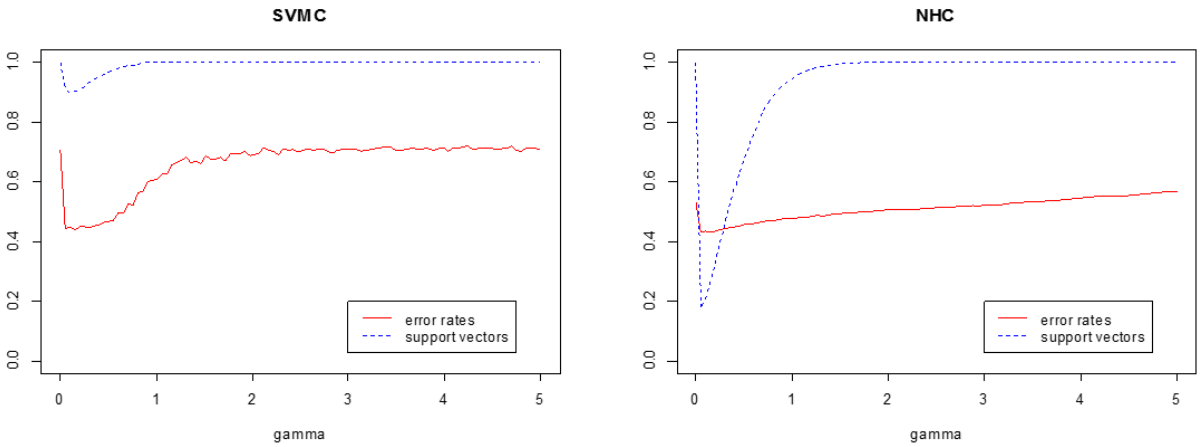

Figure 4.4: Behaviour of fraction of support vectors and error rates with $n = 100$, $\rho = 0.7$ and $C = 0.1$.

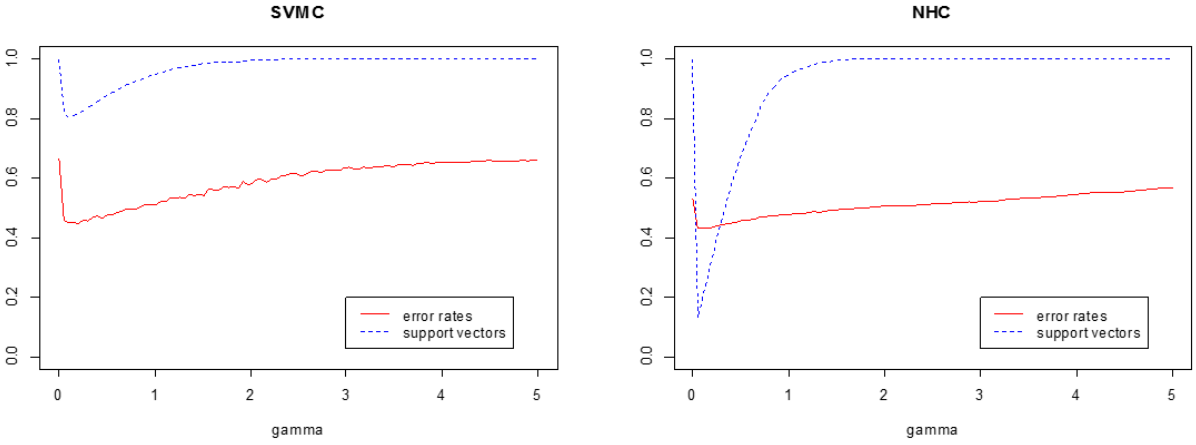

Figure 4.5: Behaviour of fraction of support vectors and error rates with $n = 100$, $\rho = 0.7$ and $C = 1$.

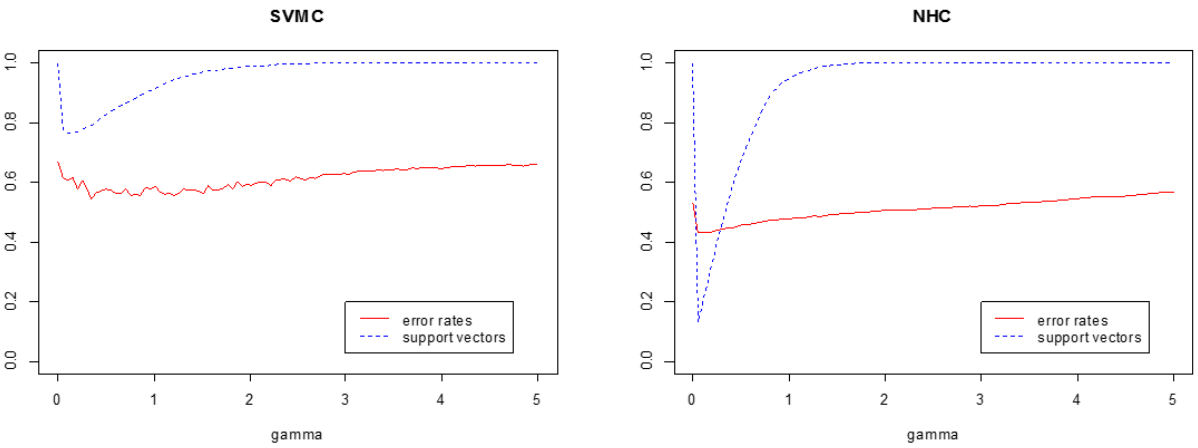

Figure 4.6: Behaviour of fraction of support vectors and error rates with $n = 100$, $\rho = 0.7$ and $C = 5$.

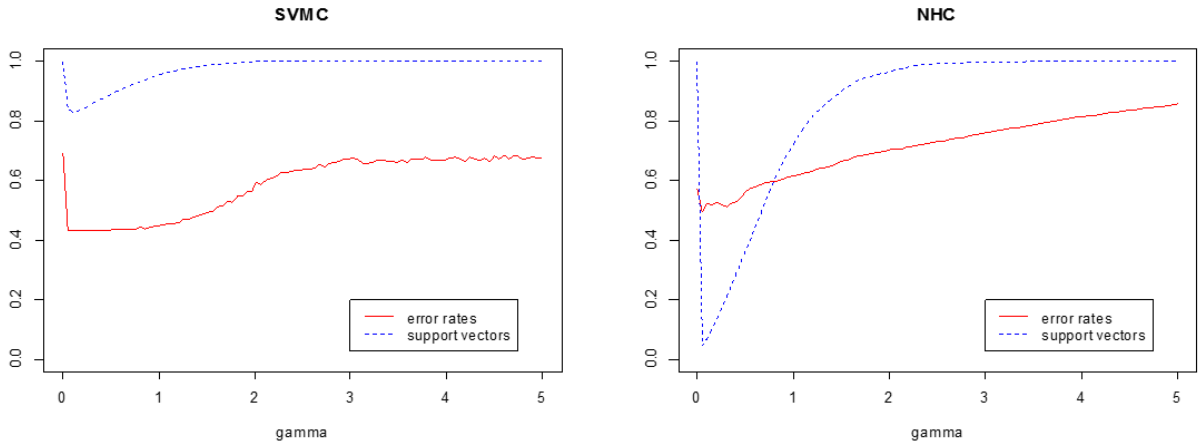

Figure 4.7: Behaviour of fraction of support vectors and error rates with $n = 400$, $\rho = 0$ and $C = 0.1$.

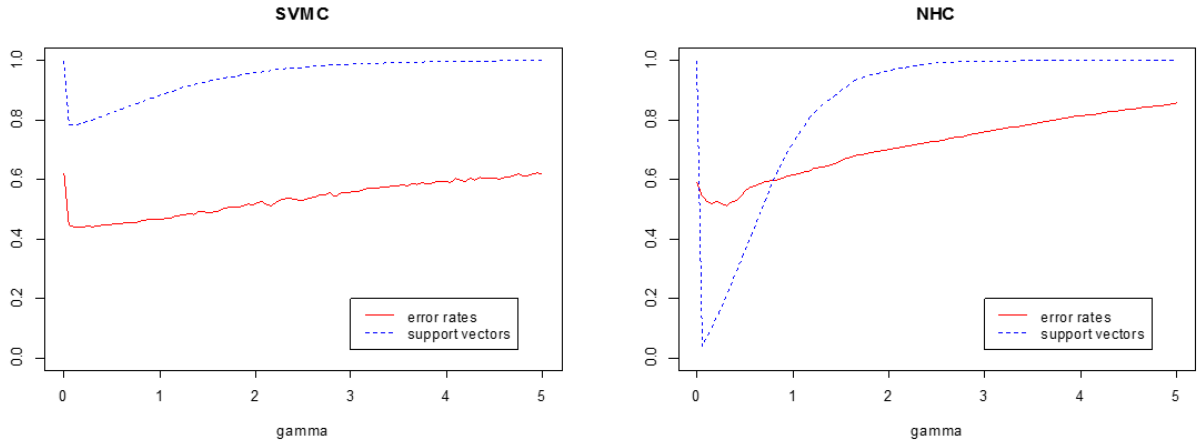

Figure 4.8: Behaviour of fraction of support vectors and error rates with $n = 400$, $\rho = 0$ and $C = 1$.

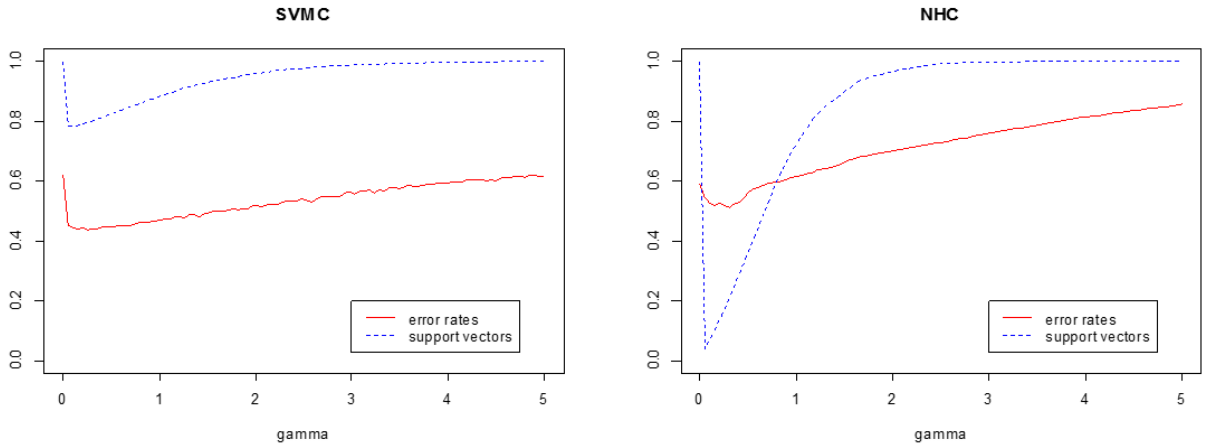

Figure 4.9: Behaviour of fraction of support vectors and error rates with $n = 400$, $\rho = 0$ and $C = 5$.

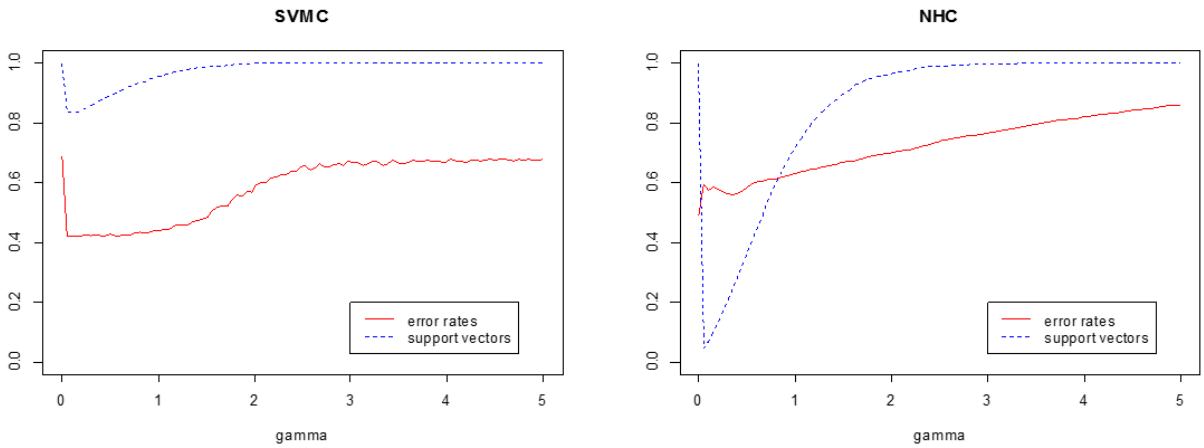

Figure 4.10: Behaviour of fraction of support vectors and error rates with $n = 400$, $\rho = 0.7$ and $C = 0.1$.

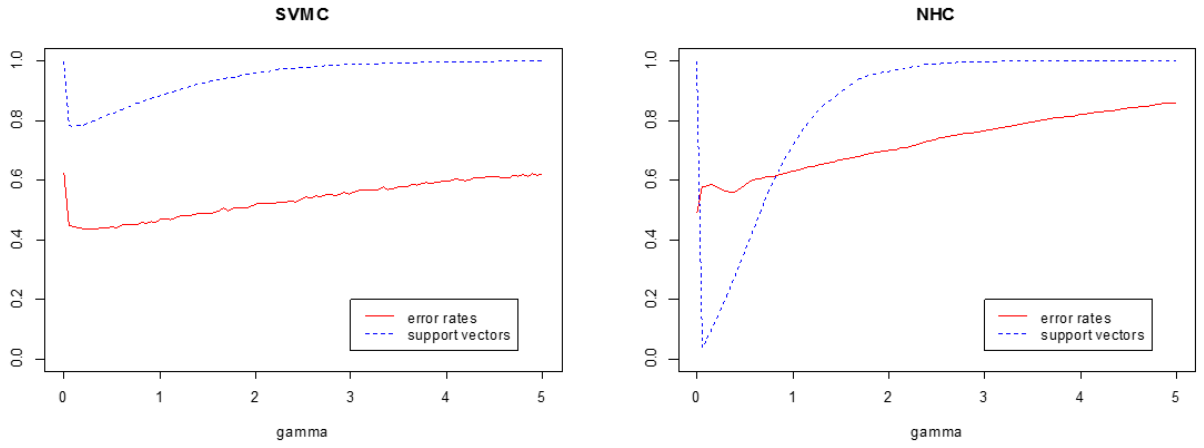

Figure 4.11: Behaviour of fraction of support vectors and error rates with $n = 400$, $\rho = 0.7$ and $C = 1$.

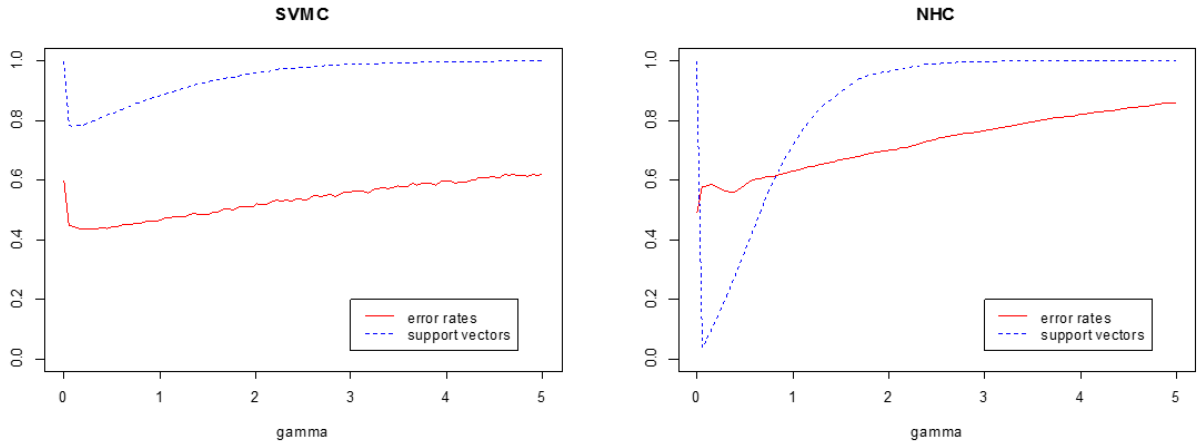

Figure 4.12: Behaviour of fraction of support vectors and error rates with $n = 400$, $\rho = 0.7$ and $C = 5$.

4.3.3 Discussion of simulation results

Looking at the graphs for SVMC, it is clear that the error rate is high for very small γ values and then decreases quickly until it reaches a minimum value and increases as the value of γ increases. The behaviour of the error rates are also clearly different for the different values of the C parameter for each dataset. The fraction of support vectors follow similar patterns to that of the error rates. For most of the figures it appears as if the minimum error rate and minimum fraction of support vectors used lie near the same γ value. Correlation does not seem to have a noticeable effect for small datasets when $C = 0.1$ or $C = 1$, but for $C = 5$, the error rates are slightly lower for datasets with correlation. For large datasets, it seems like the correlation does not really have an influence on the error rates and fraction of support vectors.

If we now consider the graphs for NHC, the error rate once again starts at a high value when γ is very close to zero. The error rate then decreases very quickly until it reaches a minimum value and increases as the γ value increases. The behaviour of the error rates are very similar for the different values of the C parameter for each dataset. The different values for C do not seem to have a noticeable effect. We know that for $C > 1$, the graphs will be exactly the same as when $C = 1$. This can be clearly seen in the figures. The fraction of support vectors used starts very high for a γ value very close to zero and then immediately decreases to a very small value from where it then increases to a value of 1 and remains there for large values of gamma. The correlation does not seem to have a noticeable effect for small datasets. For large datasets it seems like the correlation causes the error rates to behave differently for γ almost equal to zero. The error rate starts very low, increases and then decreases before it continues to increase again.

We will now compare the behaviour of the error rates and fraction of support vectors used for SVMC versus NHC. The error rates and fraction of support vectors both follow the same pattern of reaching a minimum and then increasing as γ increases for both classification techniques. For the smaller datasets, the minimum error rates seem to be very similar for both techniques, but for the larger datasets, the minimum error rate under SVMC seems to be slightly less than the minimum error rate under NHC. The minimum fraction of support vectors used is always less for NHC than for SVMC and when the minimum error rate is used to determine the value of γ, the fraction of support vectors used for NHC will be less than that of SVMC.

This section only looked at simulated data, but it would be interesting to know whether the behaviour of the error rates and the fraction of support vectors used will be the same for real-world datasets. In the next section, we will look at three real-world datasets.

4.4 Application to real-world data

In this section, we will be using the Iris dataset, Glass dataset and Vehicle dataset. We will be doing exactly the same study as in the previous section, because we want to see whether the behaviour for these real-world datasets will be similar to the behaviour for simulated datasets.

4.4.1 Datasets

The following table gives a quick summary of the three datasets that we will be using.

Table 4.2: Summary of real-world datasets.

Dataset	Classes	Objects	Variables
Iris	3	150	4
Glass	6	214	9
Vehicle	4	846	18

Where the datasets can be found:

- The Iris dataset can be found in the base package of R.
- The Glass dataset can be found in the `mlbench` R package.
- The Vehicle dataset comes from the Turing Institute, Glasgow, Scotland and can be found in the `mlbench` R package.

4.4.2 Real-world data results

We will now look at the behaviour of the error rates and the fraction of support vectors used over the values of γ. Figure 4.13 to Figure 4.21 show the behaviour for the three datasets for different values of C. Each dataset was standardised before we applied the techniques.

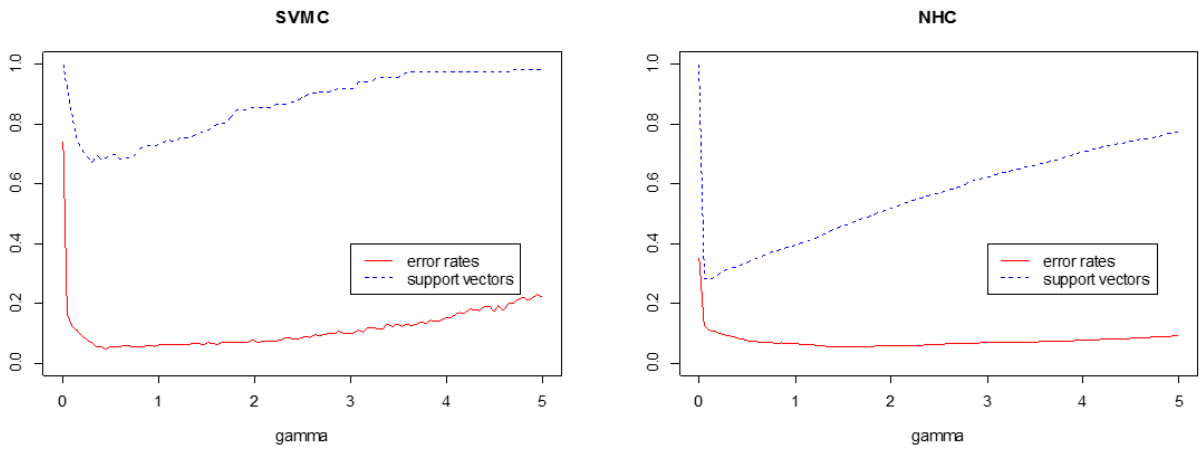

Figure 4.13: Behaviour of fraction of support vectors and error rates for Iris dataset with $C = 0.1$.

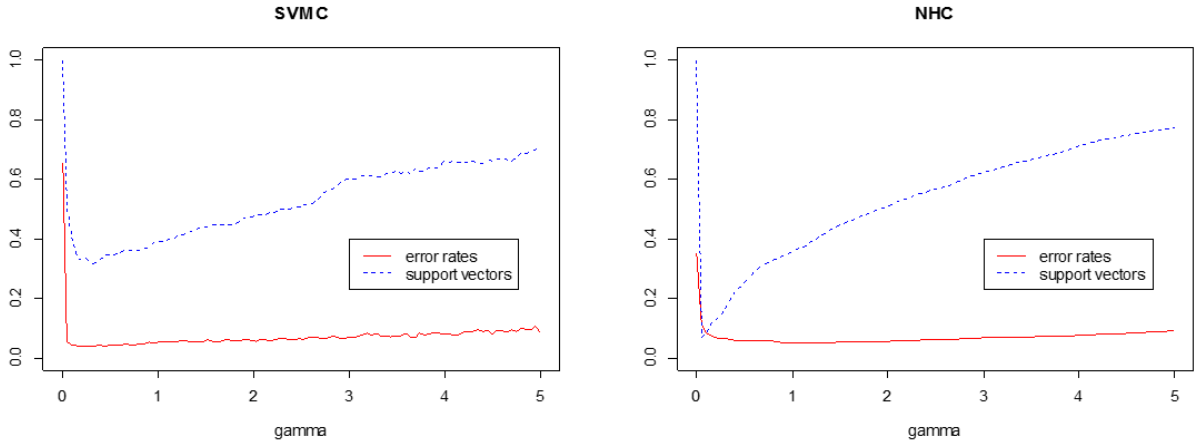

Figure 4.14: Behaviour of fraction of support vectors and error rates for Iris dataset with $C = 1$.

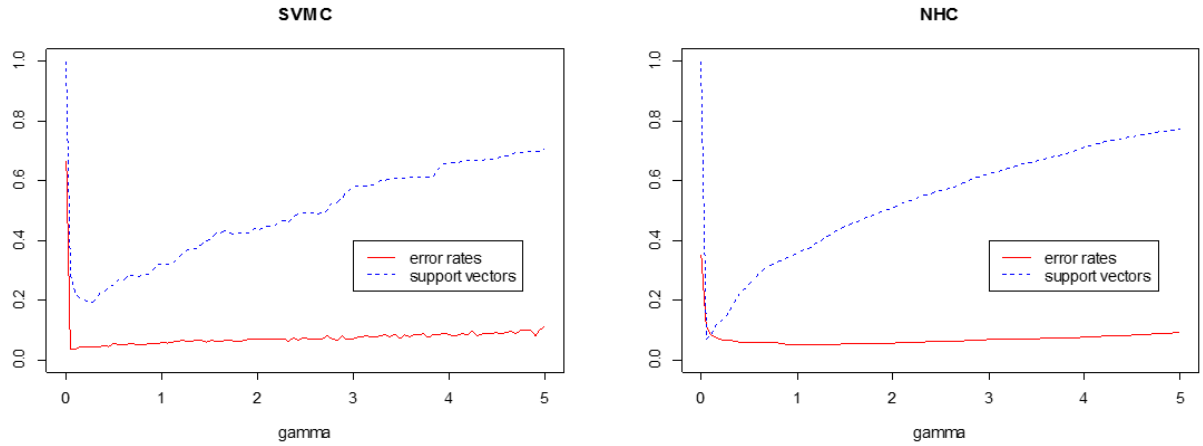

Figure 4.15: Behaviour of fraction of support vectors and error rates for Iris dataset with $C = 5$.

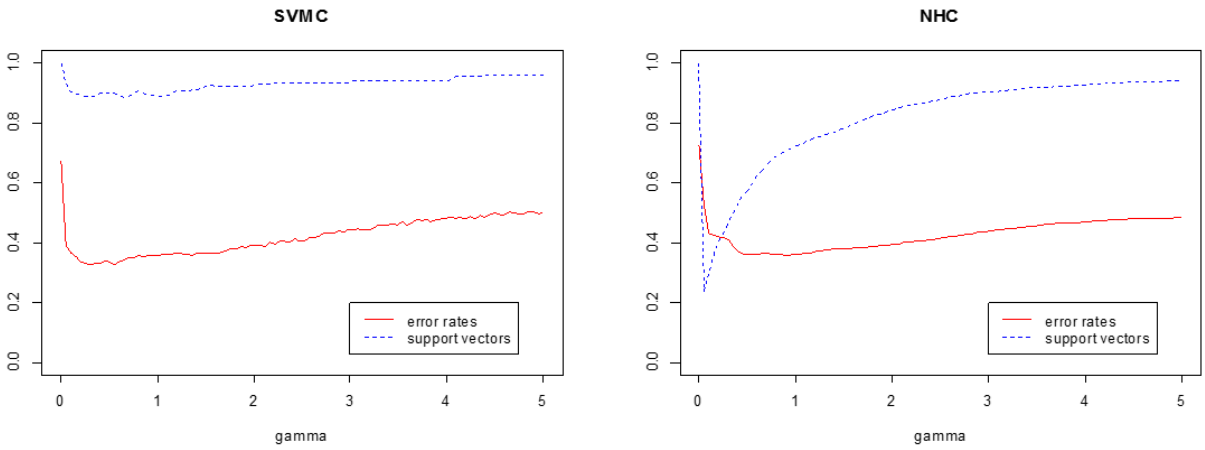

Figure 4.16: Behaviour of fraction of support vectors and error rates for Glass dataset with $C = 0.3$.

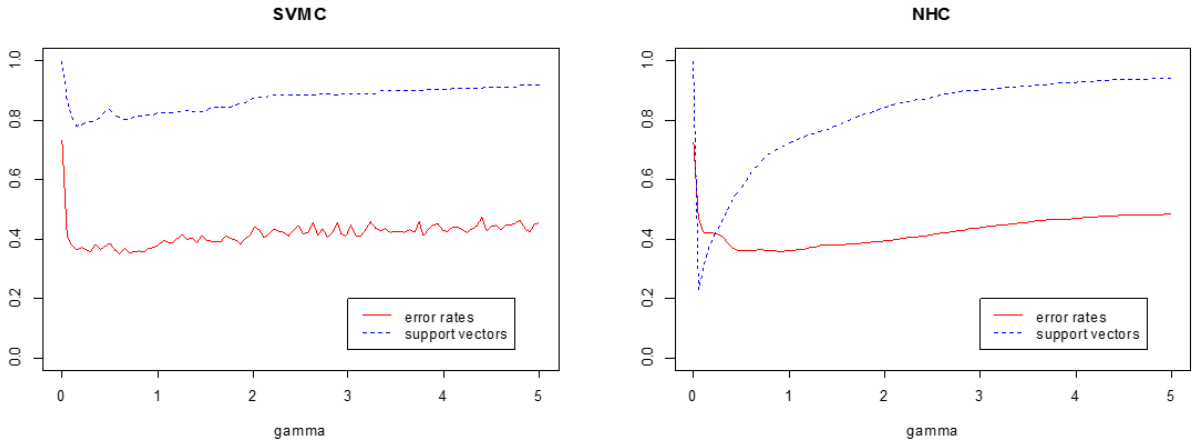

Figure 4.17: Behaviour of fraction of support vectors and error rates for Glass dataset with $C = 1$.

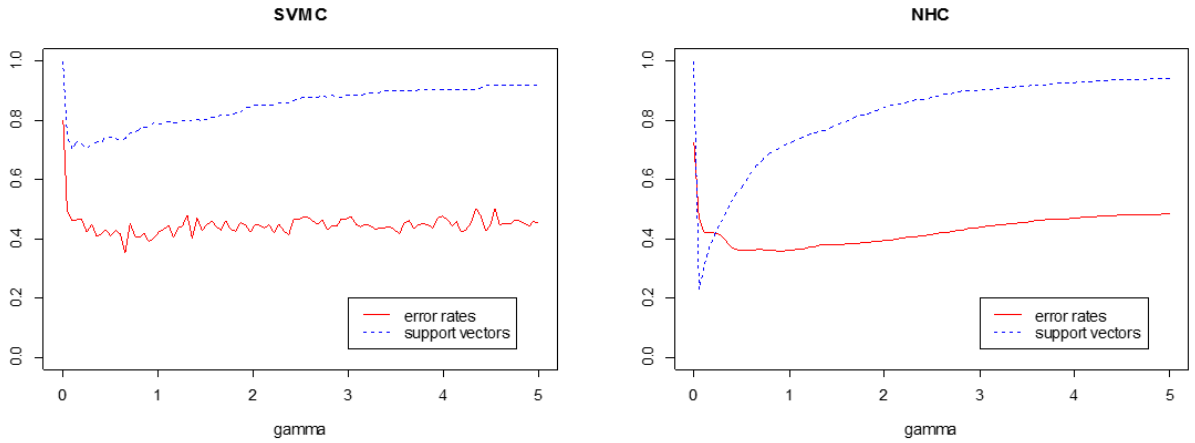

Figure 4.18: Behaviour of fraction of support vectors and error rates for Glass dataset with $C = 5$.

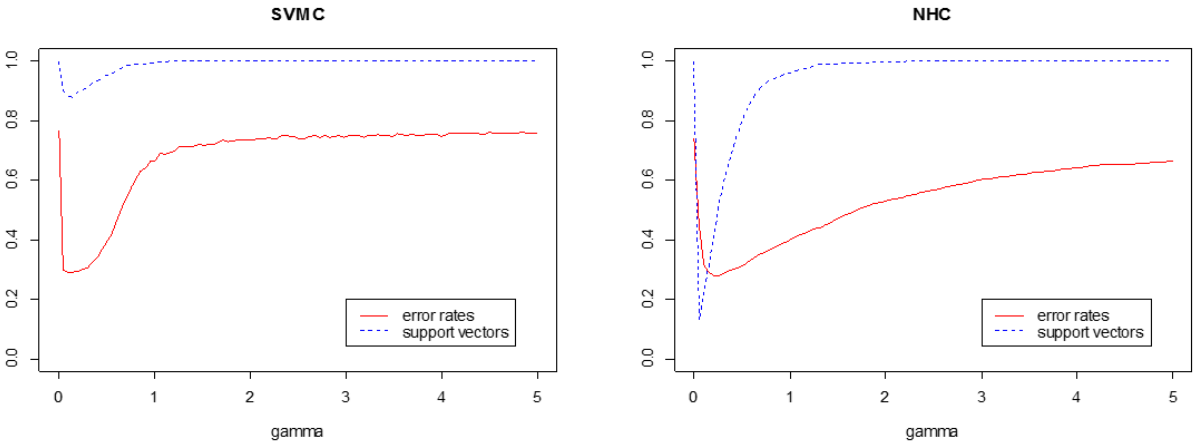

Figure 4.19: Behaviour of fraction of support vectors and error rates for Vehicle dataset with $C = 0.1$.

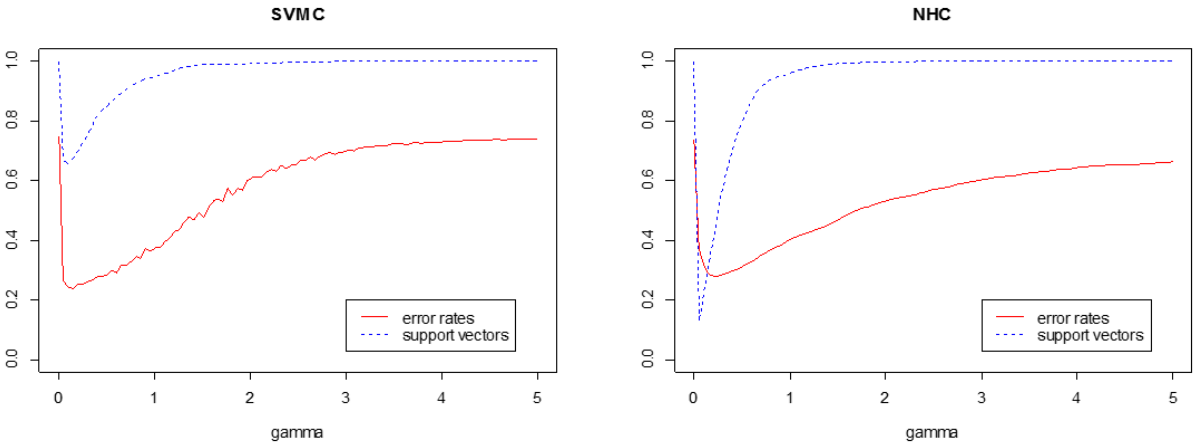

Figure 4.20: Behaviour of fraction of support vectors and error rates for Vehicle dataset with $C = 1$.

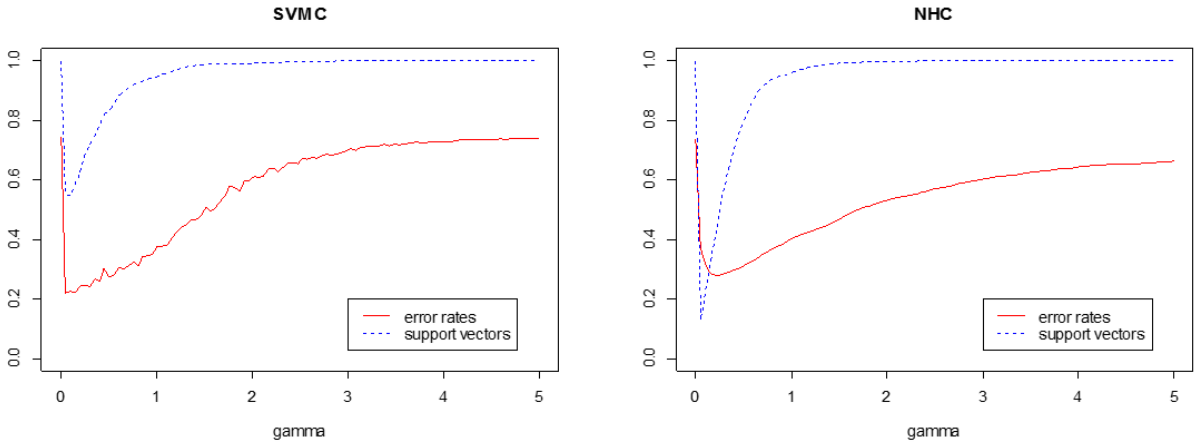

Figure 4.21: Behaviour of fraction of support vectors and error rates for Vehicle dataset with $C = 5$.

4.4.3 Discussion of real-world data results

The error rates for the Iris dataset are very low and do not increase much as γ increases. For $C = 0.1$, NHC uses fewer support vectors for small γ values than when $C = 1$, but the error rate curves do not differ for the different values of C. As C increases, SVMC uses less support vectors and for $C = 0.1$, the error rates increase more quickly than for the larger values for C.

For the Glass dataset, we used $C = 0.3$ instead of $C = 0.1$, because when finding the hypersphere for each class, there were too few objects in some of the classes to use $C = 0.1$. In Chapter 2 we showed that $C > \frac{1}{n}$. For all values of C, the error rates and fraction of support vectors do not differ noticeably for NHC. For $C = 0.3$, the error rates increase smoothly as γ increases for SVMC, but as C increases, the behaviour of the error rates become more erratic. As C increases, the number of support vectors used for SVMC decreases.

When looking at the Vehicle dataset, for NHC and SVM, the error rates increase quickly as γ increases. There are no noticeable differences between the graphs for NHC for the different values for C. It can once again be seen that the number of support vectors used decreases as C increases for SVMC and it seems like the minimum error rate decreases slightly as C increases.

It is very clear that the error rates and fraction of support vectors used behave in the same manner as in the simulated dataset study. For both SVMC and NHC there is clearly a minimum value for the error rates. For SVMC it seems as if the error rates behave more erratic for a large C parameter and the error rate curve is smoother for small values of the C parameter. The C parameter does not seem to have a great influence on the error rates and fraction of support vectors used for NHC. For all three datasets, it can be seen that NHC will always use fewer support vectors at the minimum error rate than SVMC. It seems like the minimum error rate is slightly lower for SVMC than for NHC.

4.5 Conclusion

The error rates and the fraction of support vectors used have very similar behaviour for the study performed on the simulated datasets and the real-world datasets. It is very clear from this study that there exists a γ value that will minimise the error rate. In the next chapter, it becomes important that we can estimate the optimal γ value by minimising the error rate as a function of γ. The behaviour of the error rates and the fraction of support vectors used for NHC does not show a noticeable difference for the different values of the C parameter. We will therefore only use $C = 1$ in the next chapter for NHC (the all enclosing hypersphere) and $C = 1$ for SVMC.

CHAPTER 5
COMPARING TECHNIQUES

5.1 Introduction

In this thesis, we have now discussed four techniques that can used for classification. In this chapter, we want to compare the performance of NHC to that of the other three classification techniques. We will use five real-world datasets to perform the investigation. The application in R (of the techniques we studied in Chapter 3) will be discussed in Section 5.2. In Chapter 4 we studied the behaviour of the error rates as a function of the kernel hyper-parameter. In Section 5.3 we will look at ways to estimate the hyper-parameter. Section 5.4 will give the results for the comparison among the classification techniques for each of the real-world datasets and we will discuss the results in Section 5.5.

5.2 Application of techniques in R

5.2.1 Support Vector Machine Classification in R

The R package needed for support vector machines is `kernlab`. Information about the package can be found in Karatzoglou, Smola and Hornik (2016). We will use the `ksvm()` function to fit the model.

The usage in R is:

```
ksvm(x,data, kernel ="rbfdot",kpar=list(sigma),
       type="kbb-svc",C)
```

with arguments

- `x` is a symbolic description of the model to be fit,
- `data` is the data to be used to fit the model,
- the `kernel` is the kernel function used and in this thesis, we will use the Radial Basis (Gaussian) kernel,
- `kpar=list(sigma)` is the hyper-parameter of the Gaussian kernel which is the γ value,

- `type="kbb-svc"` is the Weston, Watkins native multi-class SVM which can be used for the classification of our datasets that have more than two classes, and
- `C` is the cost parameter as discussed in Chapter 3.

For this chapter, we will use $C = 1$ which is the default value. Let `training.data` be the training dataset, `class` the column in `training.data` (which specifies the class) and `gamma.value` the hyper-parameter value. The following function will be used to fit the model:

```
ksvm(class~.,training.data, kernel ="rbfdot", type="kbb-svc",
     kpar=list(sigma=gamma.value),C=1)
```

To classify a test dataset we can use the `predict()` function and the fitted model. If `svmc.model` is the output of the fitted model and `test.data` is the test dataset, then the following function can be used to predict the test dataset classes:

```
predict(svmc.model,test.data)
```

5.2.2 Random Forest in R

The R package needed for Random Forest is `randomForest`. More information about the package can be found in Liaw and Wiener (2015). We will use the `randomForest()` function to fit the model.

The usage in R is:

```
randomForest(formula, data, ..., subset,
       na.action=na.fail)
```

where

- `formula` is a symbolic description of the model to be fit, and
- `data` is the data to be used to fit the model.

The function uses $m = \sqrt{p}$ as default. If `training.data` is the training dataset and `class` is the vector containing the classes in `training.data`, the following function will be used to fit the model:

```
randomForest(class~.,training.data)
```

To classify a test dataset, we can again use `predict()` and the fitted model. If `rf.model` is the output of the fitted model and `test.data` is the test dataset, then the following function can be used to predict the test dataset classes:

```
predict(rf.model,test.data)
```

5.2.3 Penalised LDA in R

The R package needed for Penalised LDA is `penalizedLDA`. More information about the package can be found in Witten (2015). We will use the `PenalizedLDA.cv()` function which uses cross-validation to find the best parameters to fit the model.

The usage in R is:

```
function (x, y, lambdas = NULL, K = NULL, nfold = 6, folds =
        NULL, type = "standard", chrom = NULL, lambda2 = NULL)
```

where

- `x` is the data matrix, and
- `y` is the vector containing the class labels. The class labels must be numeric.

If `training.data` is the training dataset and `class` is the vector containing the class labels of `training.data`, the following function will be used to find the best parameters for the fitted model:

```
PenalisedLDA.cv(training.data, class)
```

We need the best K (number of projections for penalised LDA) and best lambda values (as in equation (3.22)) to fit the model. If `plda.model` is the output from the function above, we can get the value for the best K as `plda.model$bestK` and the best lambda value as `plda.model$bestlambda`. The function `PenalizedLDA` is now used to predict the test dataset using the best parameters determined in the cross validation function.

The usage in R is:

```
PenalizedLDA(x, y, xte = NULL, type = "standard", lambda,
        K = 2, chrom = NULL, lambda2 = NULL,
          standardized = FALSE, wcsd.x =   NULL,
            ymat = NULL, maxiter = 20,trace = FALSE)
```

In the above function call the `xte` argument is the test data that will be used to predict the classes. The classes of the test data that are called `test.data` can be found by using the function as follows:

```
PenalizedLDA(training.data, as.factor(training.class),
     test.data, lambda=plda.model$bestlambda,
          K=plda.model$bestK)
```

If the output of this function is `plda.predict`, then the predicted classes of the test data are found by using `plda.predict$ypred`.

5.3 Estimation of the hyper-parameter

We need to specify a value for the hyper-parameter of the Gaussian kernel to be able to perform classification under NHC and SVMC. We will discuss two methods that can be used to estimate this hyper-parameter.

5.3.1 Using DEoptim()

It is clear from Chapter 4 that the error rates for SVMC and NHC have turning points at the minimum error rate. We want to determine the hyper-parameter (γ) value where the error rate is a minimum. In Coetzer (2015) different general purpose optimisation functions in R were compared to find the optimal hyper-parameter at the minimum error rate for SVMC. It was found that `DEoptim()` (Ardia, Mullen, Peterson, Ulrich, and Boudt (2016)) performs well and is computationally faster. We will use `DEoptim()` in this chapter to find the optimal γ value for SVMC and NHC.

The `DEoptim()` function is available in the `DEoptim` package. A differential evolution algorithm is used in `DEoptim()` and the function to be minimised does not have to be

continuous or differentiable. Figure 5.1 illustrates how `DEoptim()` works. The red curve is the function (`fn`) that has to be minimised. Let `L` be the lower value and `U` the upper value of the interval which is used to search for the minimum value of the function.

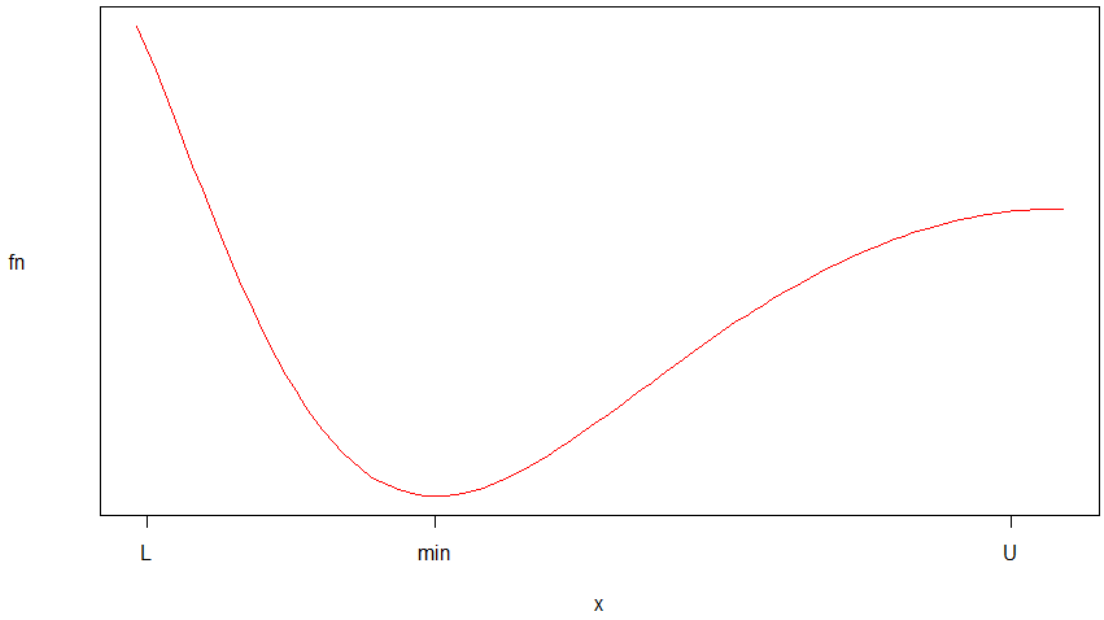

Figure 5.1: Illustration of finding the minimum using DEoptim().

The usage in R is:

```
DEoptim(fn, lower, upper, control = DEoptim.control(), ...,
      fnMap = NULL)
```

The function `fn` will be minimised and this function does not have to be continuous or differentiable. The function searches for the minimum of `fn` between the values given by the parameters `lower` and `upper`. Let `output` be the output from the `DEoptim()` function. The optimal value will be found using `output$optim$bestmem`.

Since we saw in Chapter 4 that the error rates for SVMC and NHC also have a minimum, we will use `DEoptim()` to find the minimum error and the corresponding γ value.

5.3.2 Using sigest()

Another method to estimate the hyper-parameter is by using the function `sigest()` in the `kernlab` package (Karatzoglou et al. (2016)). This function gives three possible values for the hyper-parameter of the Gaussian kernel. The function estimates a range of hyper-parameter values that would return good results for the `ksvm()` function. The function will return the $0.1, 0.5$ and 0.9 quantiles of the estimated hyper-parameter values. These quantiles will be referred to as S1, S2 and S3.

The R usage:

```
sigest(x, frac = 0.5, scaled = TRUE, na.action = na.omit)
```

where

- `x` is the matrix containing the training dataset, and
- `frac` is the fraction of the data to be used for estimation.

When we have the training data and training class as above, we can use the function as follows:

```
sigest(class~., training.data)
```

We will use the optimal value from `DEoptim()` and all three values of `sigest()` in this chapter. Although `sigest()` is meant for SVMC, we will also use these estimated hyper-parameter values to see whether they could be useful for NHC as well.

5.4 Comparison of classifiers

The real-world datasets that will be used in this chapter are the Glass dataset, Vehicle dataset, Abalone dataset, Yeast dataset and Khan dataset. These datasets will be introduced later in this section. To be able to perform classification, we first need to build a classification model using training data. When we have the classification model, we can classify the test data. For SVMC and NHC we will use a validation dataset to find the optimal γ value. This γ value can then be used to classify the test data. We will use 50% of the data as the training dataset, 25% as the validation dataset and the remaining 25% as the test dataset. For the comparison, we will standardise the datasets before doing any analyses.

The following steps will be followed for the experiments with all four techniques:

- Divide the dataset randomly into a training dataset (50%), validation dataset (25%) and test dataset (25%).
- Fit the model on the training dataset.
- The next three steps are for SVMC and NHC only:
    - Estimate the error for model selection with the validation dataset.
    - Estimate the optimal hyper-parameter at the minimum validation error.
    - Fit the model to the training dataset using the optimal hyper-parameter. Also determine the fraction of support vectors used.
- Use the models that were fitted on the training dataset to find the test error using the test dataset.

These steps will be repeated 50 times. In Table 5.1 to Table 5.5 we have the results for the five datasets. The first column contains the name of the techniques and how the hyper-parameter was obtained. For example SVMC DEoptim refers to the SVM classifier with `DEoptim()` being used to obtain γ; SVMC S1 refers to SVM classifier with the S1 value of `sigest()` being used to obtain γ. For the random forest, we chose $m = \sqrt{p}$ as the number of random variables selected. For penalised LDA we chose the parameters using the cross-validation values from `penalizedLDA.cv()`.

The means and standard deviations of the test errors will be calculated for the 50 repetitions of each technique and reported in the second column. The values without the brackets are the mean values and the values in brackets are the standard deviations. The value in bold is the smallest test error when all the techniques are compared. The third column contains the mean and standard deviation of the fraction of support vectors used for SVMC and NHC. The mean and standard deviation of the optimal hyper-parameter for SVMC and NHC is contained in the fourth column. The standard deviations are once again the values in brackets.

All the classification techniques will be tested using the same randomised data to ensure comparability between the techniques. NHC and SVMC will each have four outputs. We will find the hyperparameters using `DEoptim()` as well as using the three `sigest()` values. The classification techniques will be compared on the five different datasets and then the techniques will be compared per dataset. We will display the test error rates, fraction of support vectors used and γ values in boxplots as a graphical representation of the results in the tables.

5.4.1 Glass dataset

The first dataset to be used is the Glass dataset. It is a relatively small dataset and can be found in the `mlbench` R package. The dataset has 9 variables and one pair of variables are highly correlated (correlation is above 0.7). There are 6 types of glass that we will use for the classification. The Glass dataset is summarised in the following table:

Number of classes	Number of objects	Number of variables
6	214	9

Table 5.1: Summary of output for Glass dataset.

Technique	Error rate	Fraction of SVs	Hyper-parameter
SVMC DEoptim	0.3768	0.8943	1.2351
	(0.0805)	(0.045)	(1.6762)
SVMC S1	0.4818	0.9297	0.0255
	(0.1137)	(0.0222)	(0.0062)
SVMC S2	0.3965	0.8775	0.0862
	(0.1152)	(0.0275)	(0.0226)
SVMC S3	0.3723	0.885	0.6243
	(0.0936)	(0.0305)	(0.2931)
NHC DEoptim	0.4028	0.6549	0.6465
	(0.0551)	(0.1359)	(0.6634)
NHC S1	0.5168	0.2621	0.0255
	(0.088)	(0.0263)	(0.0062)
NHC S2	0.4358	0.3634	0.0862
	(0.0763)	(0.0401)	(0.0226)
NHC S3	0.3933	0.6899	0.6243
	(0.0631)	(0.0887)	(0.2931)
Random Forest	**0.2565**	-	-
	(0.0534)	-	-
Penalized LDA	0.4193	-	-
	(0.0511)	-	-

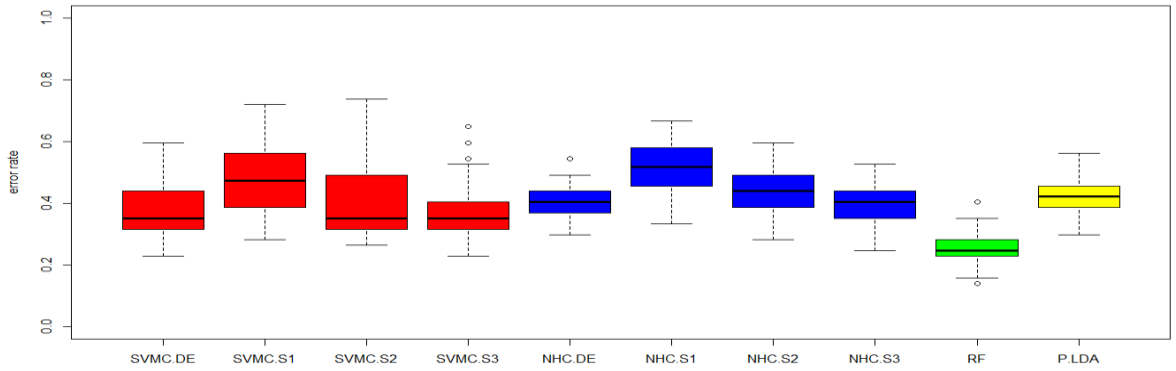

Figure 5.2: Boxplots of the error rate values for Glass data.

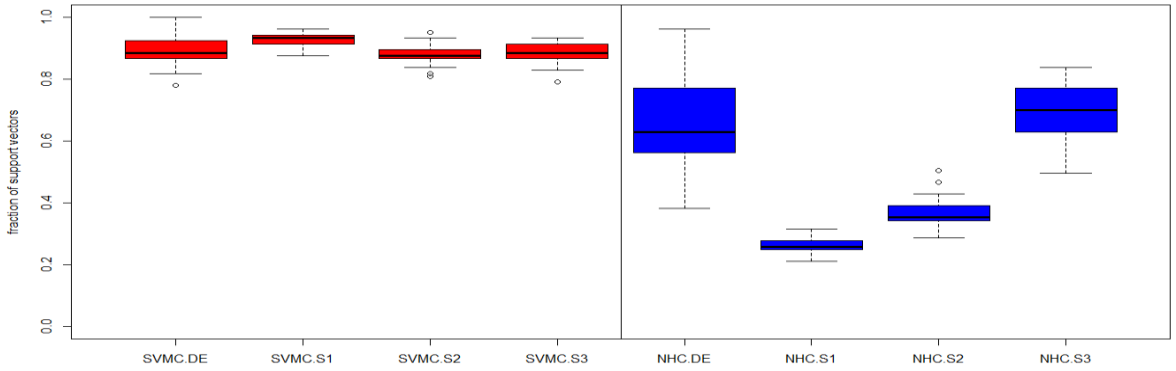

Figure 5.3: Boxplots of the fraction of support vectors for Glass data.

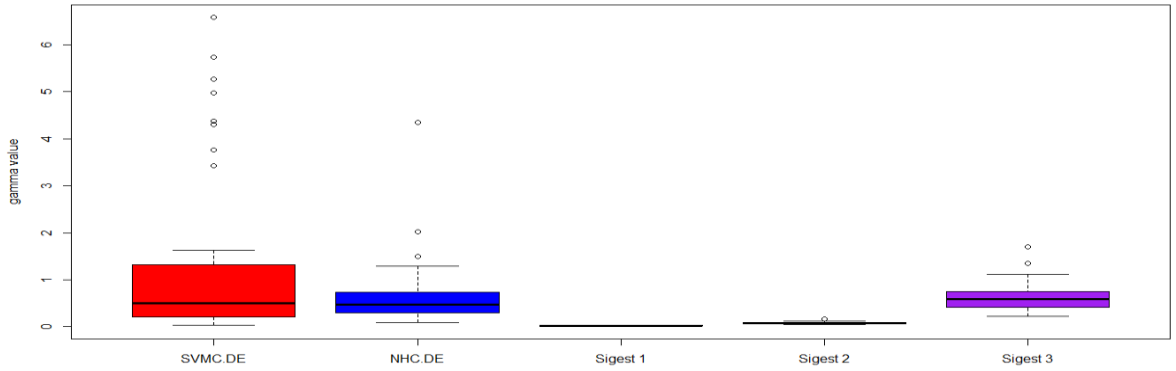

Figure 5.4: Boxplots of the gamma values for Glass data.

5.4.2 Vehicle dataset

The second dataset we used is the Vehicle dataset. It is a larger dataset than the Glass dataset and can also be found in the `mlbench` R package. The dataset has 18 variables and 51 pairs of variables that are highly correlated (correlation is above 0.7). There are 4 classes that we will use for the classification. The Vehicle dataset is summarised in the following table:

Number of classes	Number of objects	Number of variables
4	846	18

Table 5.2: Summary of output for Vehicle dataset.

Technique	Error rate	Fraction of SVs	Hyper-parameter
SVMC DEoptim	0.2664	0.7778	0.1383
	(0.0391)	(0.0505)	(0.1268)
SVMC S1	0.3402	0.8345	0.0141
	(0.1258)	(0.0159)	(0.0011)
SVMC S2	0.2899	0.756	0.0378
	(0.0742)	(0.0126)	(0.0031)
SVMC S3	0.2715	0.7499	0.1217
	(0.047)	(0.014)	(0.0126)
NHC DEoptim	0.3143	0.573	0.2395
	(0.0279)	(0.1569)	(0.0908)
NHC S1	0.6732	0.0831	0.0141
	(0.0341)	(0.0078)	(0.0011)
NHC S2	0.4508	0.1481	0.0378
	(0.0525)	(0.0113)	(0.0031)
NHC S3	0.3204	0.3529	0.1217
	(0.0315)	(0.0344)	(0.0126)
Random Forest	**0.2574**	-	-
	(0.0228)	-	-
Penalized LDA	0.5994	-	-
	(0.0487)	-	-

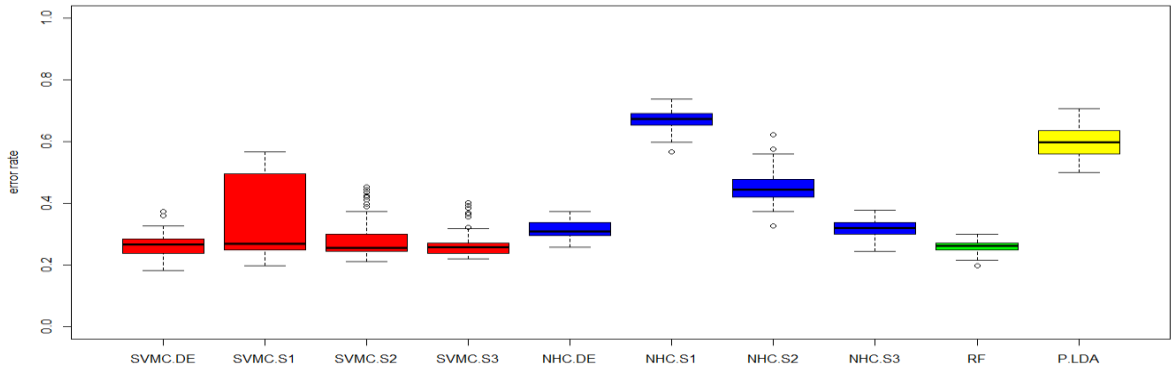

Figure 5.5: Boxplots of the error rate values for Vehicle data.

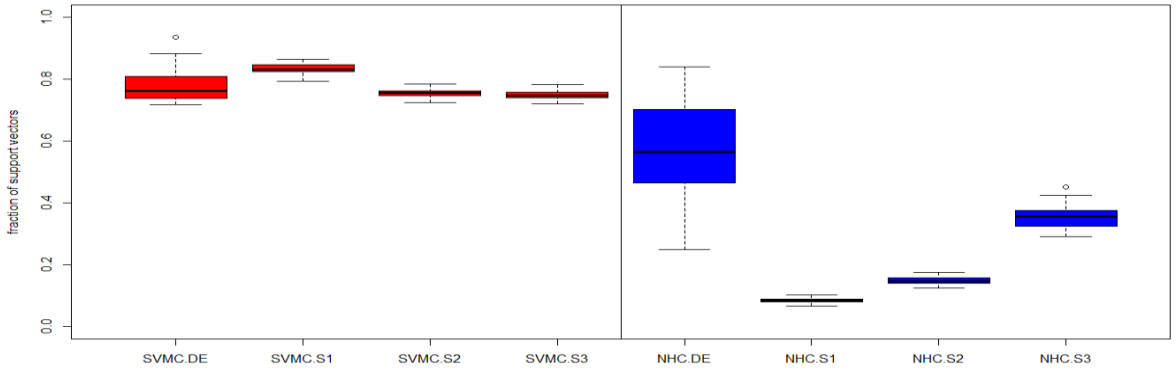

Figure 5.6: Boxplots of the fraction of support vectors for Vehicle data.

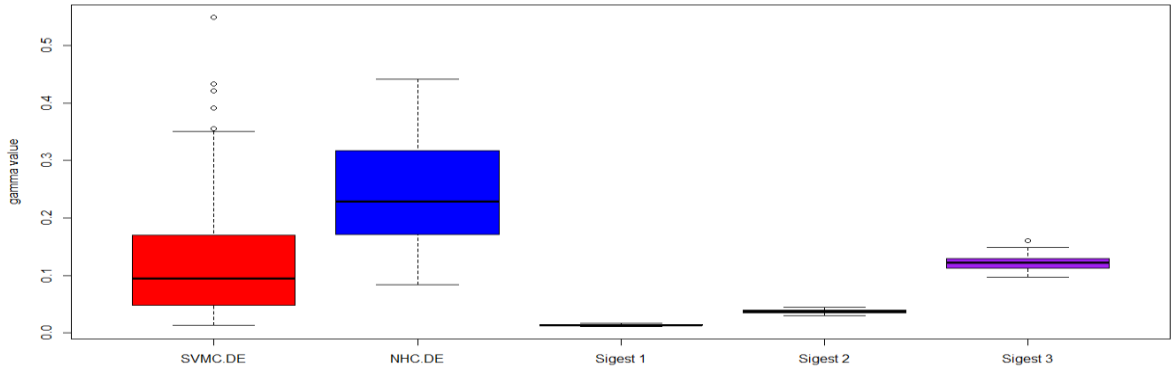

Figure 5.7: Boxplots of the gamma values for Vehicle data.

5.4.3 Abalone dataset

The third dataset we worked with was the Abalone dataset. It is a very large dataset compared to the previous datasets and can be found at the URL http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data. The dataset has 8 variables and 21 pairs of variables are highly correlated (correlation is above 0.7). There are 3 classes that we will use for the classification. The Abalone dataset is summarised in the following table:

Number of classes	Number of objects	Number of variables
3	4177	8

Table 5.3: Summary of output for Abalone dataset.

Technique	Error rate	Fraction of SVs	Hyper-parameter
SVMC DEoptim	0.4518	0.8354	1.0497
	(0.0115)	(0.0114)	(0.7402)
SVMC S1	0.4937	0.8616	0.0253
	(0.0136)	(0.0082)	(0.0014)
SVMC S2	0.4652	0.8374	0.1132
	(0.012)	(0.0098)	(0.0071)
SVMC S3	**0.4502**	0.829	0.7805
	(0.0104)	(0.0073)	(0.0659)
NHC DEoptim	0.485	0.3774	2.3862
	(0.0124)	(0.2171)	(1.6541)
NHC S1	0.588	0.0109	0.0253
	(0.0573)	(0.001)	(0.0014)
NHC S2	0.5007	0.0322	0.1132
	(0.021)	(0.0022)	(0.0071)
NHC S3	0.4965	0.1646	0.7805
	(0.0147)	(0.0132)	(0.0659)
Random Forest	0.4524	-	-
	(0.0161)	-	-
Penalized LDA	0.5994	-	-
	(0.0487)	-	-

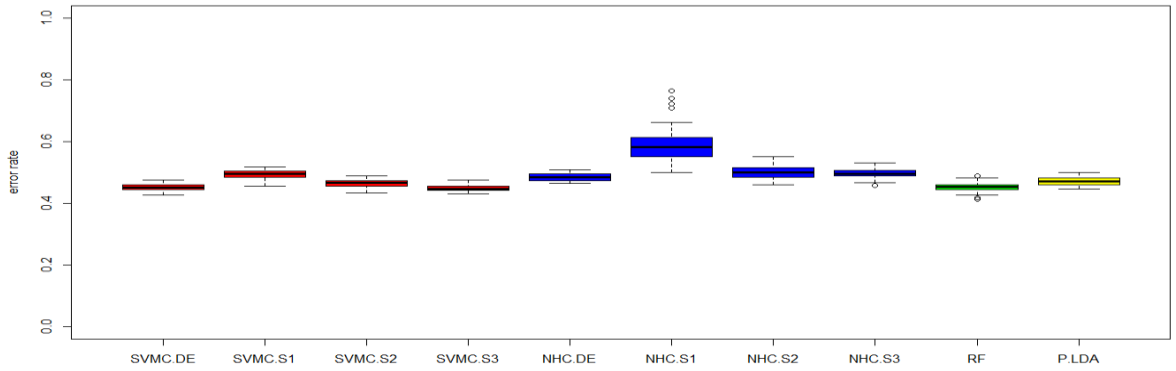

Figure 5.8: Boxplots of the error rate values for Abalone data.

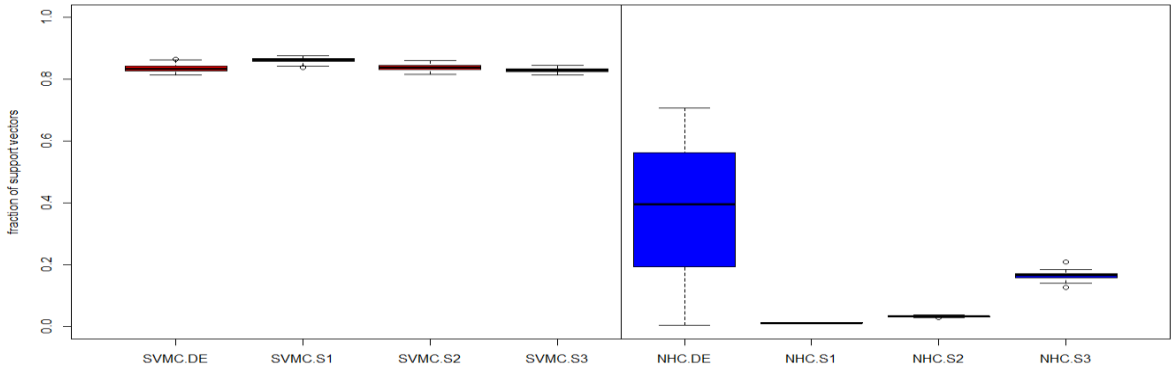

Figure 5.9: Boxplots of the fraction of support vectors for Abalone data.

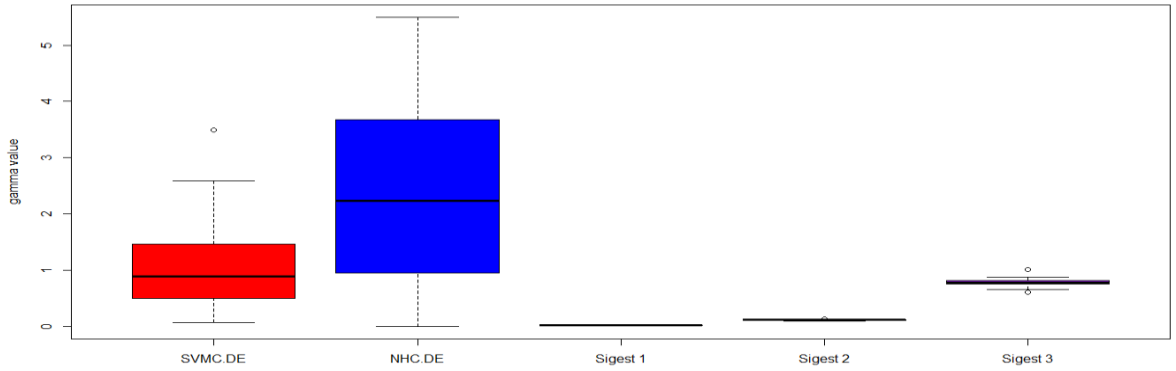

Figure 5.10: Boxplots of the gamma values for Abalone data.

5.4.4 Yeast dataset

The fourth dataset we worked with was the Yeast dataset. It is a larger dataset than the Vehicle and Glass datasets, but smaller than the Abalone dataset. The Yeast dataset can be found in the UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/datasets/Yeast. The dataset has 8 variables and no pairs of variables that are highly correlated. There were 10 classes in the dataset, but for this study we removed one class, because it had only 5 objects in that class. The Yeast dataset is summarised in the following table:

Number of classes	Number of objects	Number of variables
9	1479	8

Table 5.4: Summary of output for Yeast dataset.

Technique	Error rate	Fraction of SVs	Hyper-parameter
SVMC DEoptim	0.2664	0.7778	0.1383
	(0.0391)	(0.0505)	(0.1268)
SVMC S1	0.3402	0.8345	0.0141
	(0.1258)	(0.0159)	(0.0011)
SVMC S2	0.2899	0.756	0.0378
	(0.0742)	(0.0126)	(0.0031)
SVMC S3	0.2715	0.7499	0.1217
	(0.047)	(0.014)	(0.0126)
NHC DEoptim	0.3143	0.573	0.2395
	(0.0279)	(0.1569)	(0.0908)
NHC S1	0.6732	0.0831	0.0141
	(0.0341)	(0.0078)	(0.0011)
NHC S2	0.4508	0.1481	0.0378
	(0.0525)	(0.0113)	(0.0031)
NHC S3	0.3204	0.3529	0.1217
	(0.0315)	(0.0344)	(0.0126)
Random Forest	**0.2574**	-	-
	(0.0228)	-	-
Penalized LDA	0.5994	-	-
	(0.0487)	-	-

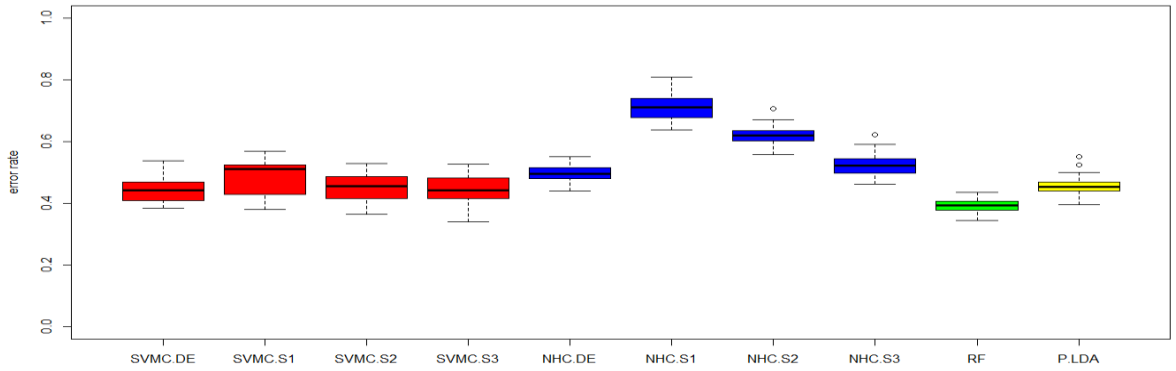

Figure 5.11: Boxplots of the error rate values for Yeast data.

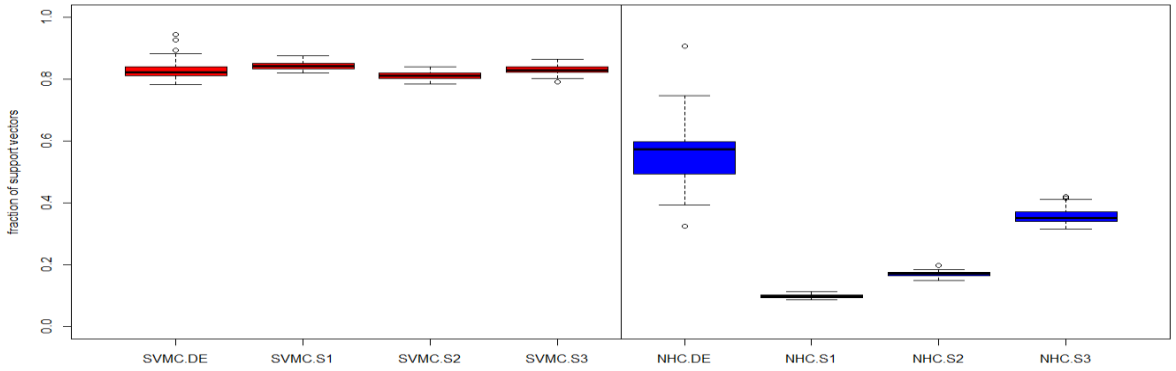

Figure 5.12: Boxplots of the fraction of support vectors for Yeast data.

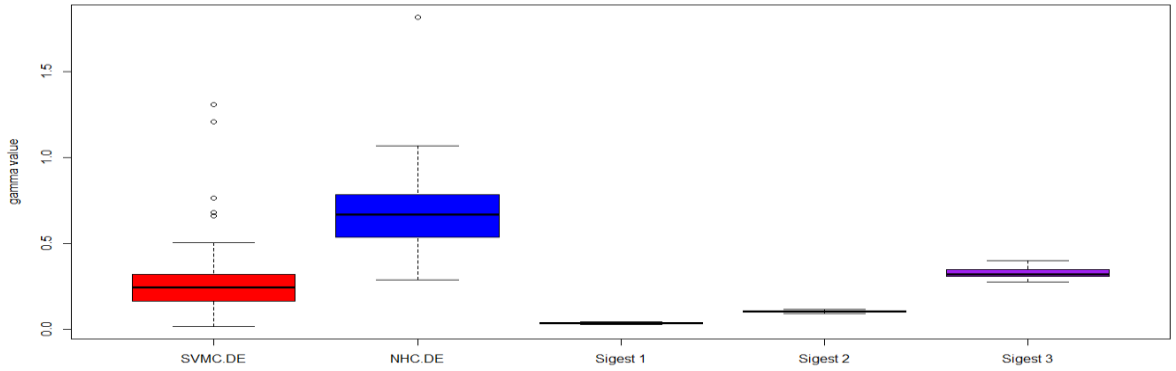

Figure 5.13: Boxplots of the gamma values for Yeast data.

5.4.5 Khan dataset

The fifth dataset we worked with was the Khan dataset. It is a small dataset compared to the previous datasets, but it has a very large number of variables. The Khan dataset can be found in the `ISLR` R package. The dataset has 2308 variables and 2101 pairs of variables are highly correlated (correlation is above 0.7). There are 4 classes that we will use for the classification. The Khan dataset is summarised in the following table:

Number of classes	Number of objects	Number of variables
4	83	2308

Table 5.5: Summary of output for Khan dataset.

Technique	Error rate	Fraction of SVs	Hyper-parameter
SVMC DEoptim	0.0933	0.7794	0.0537
	(0.0255)	(0.0425)	(0.0318)
SVMC S1	0.2683	0.9945	0.0002
	(0.2449)	(0.0105)	(0)
SVMC S2	0.2233	0.992	0.0002
	(0.2055)	(0.0128)	(0)
SVMC S3	0.1933	0.997	0.0003
	(0.1643)	(0.0082)	(0)
NHC DEoptim	0.2288	0.6053	0.0759
	(0.0986)	(0.1497)	(0.1508)
NHC S1	0.2758	0.7	0.0002
	(0.1205)	(0.0479)	(0)
NHC S2	0.2583	0.72	0.0002
	(0.1202)	(0.0446)	(0)
NHC S3	0.2417	0.7665	0.0004
	(0.1088)	(0.0459)	(0)
Random Forest	**0.0745**	-	-
	(0.0188)	-	-
Penalized LDA	0.2142	-	-
	(0.1326)	-	-

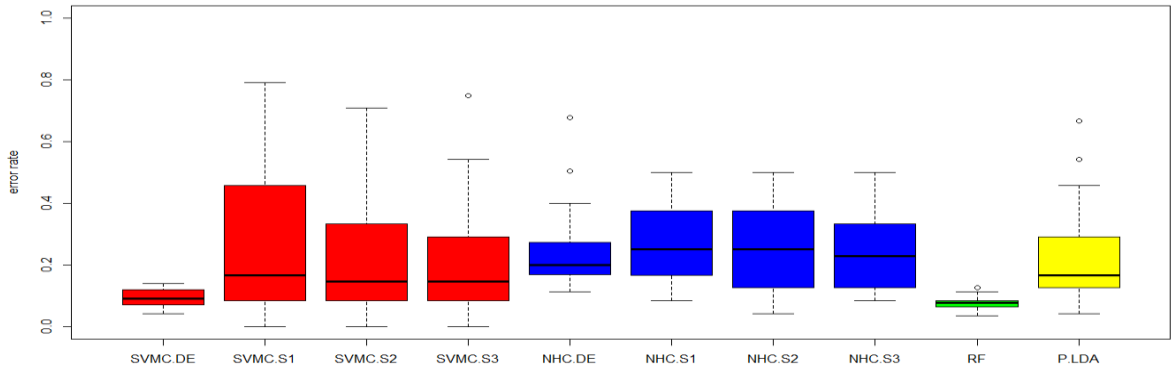

Figure 5.14: Boxplots of the error rates for Khan data.

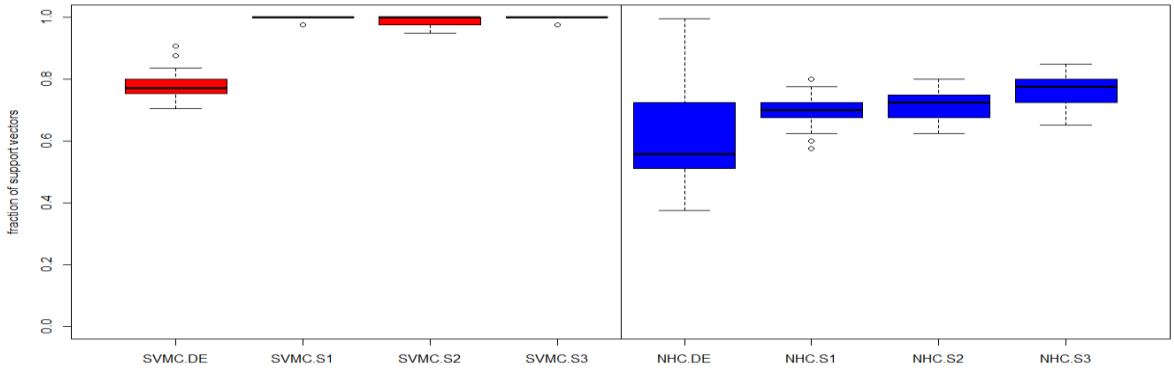

Figure 5.15: Boxplots of the fraction of support vectors for Khan data.

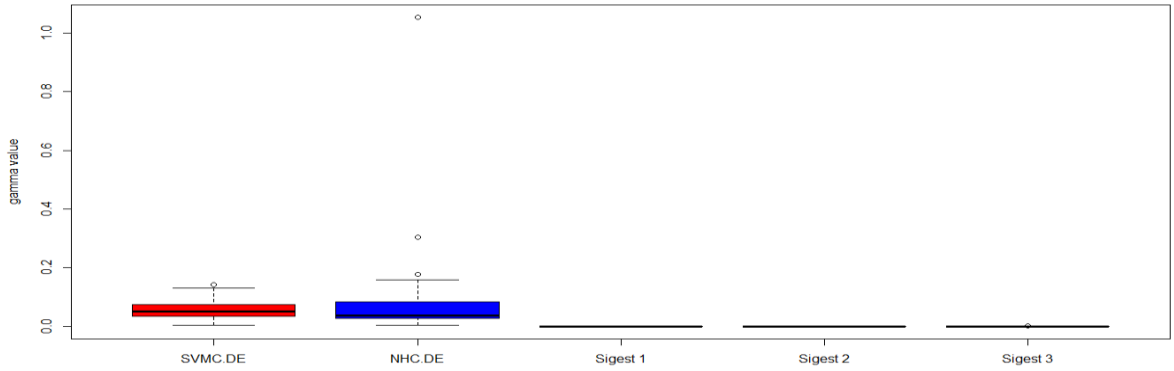

Figure 5.16: Boxplots of the gamma values for Khan data.

5.4.6 Discussion of results

We will first look at the mean error rates. The mean error rates were ranked from small to large for each dataset in Table 5.6. The top three methods are highlighted in bold.

Table 5.6: Ranks of the error rates for data (small to large)

Technique	Glass	Vehicle	Abalone	Yeast	Khan
SVMC DEoptim	**3**	**2**	**2**	**2**	**2**
SVMC S1	9	7	7	7	9
SVMC S2	5	4	4	4	5
SVMC S3	**2**	**3**	**1**	**3**	**3**
NHC DEoptim	6	5	6	5	6
NHC S1	10	10	10	10	10
NHC S2	8	8	9	8	8
NHC S3	4	6	8	6	7
Random Forest	**1**	**1**	**3**	**1**	**1**
Penalised LDA	7	9	5	9	4

For all the datasets except the Abalone dataset, random forest has the smallest error rate. For the Abalone dataset, the SVMC S3 has a slightly smaller error rate than the random forest. For all the other datasets, the error rate for SVMC is only slightly worse than that of random forest. The three techniques with the smallest mean error rate are always Random Forest, SVMC DEoptim and SVMC S3. The worst classifier is NHC S1 for all the datasets. For four of the five datasets, NHC DEoptim performs better that NHC S1, NHC S2 and NHC S3. NHC DEoptim and NHC S3 perform very similarly for all five datasets when the error rates are compared. NHC never performs the best, but it is better than penalised LDA for $n > p$, except for the Abalone dataset where all the techniques performed very similarly. SVMC DEoptim, SVMC S2 and SVMC S3 are good classifiers to use, but SVMC S1 performs badly when compared to the other SVMC techniques.

We will now examine the fraction of support vectors used for SVMC and NHC. For all the datasets, the fraction of support vectors used for NHC is always less than SVMC. This is what we expected from the empirical study we performed in Chapter 4. We will again rank the techniques, but since NHC always uses less support vectors than SVMC, we will rank the techniques separately in Table 5.7.

Table 5.7: Ranks of the fraction of SVs for data (small to large).

Technique	Glass	Vehicle	Abalone	Yeast	Khan
SVMC DEoptim	3	3	2	3	**1**
SVMC S1	4	4	4	4	3
SVMC S2	**1**	2	3	2	2
SVMC S3	2	**1**	**1**	**1**	4
NHC DEoptim	3	4	4	4	**1**
NHC S1	**1**	**1**	**1**	**1**	2
NHC S2	2	2	2	2	3
NHC S3	4	3	3	3	4

SVMC S3 uses the least support vectors for three of the five datasets. SVMC S1 has the most support vectors for four of the five datasets. We know that when we use NHC, the least amount of support vectors will be used for smaller γ values. The smallest fraction of support vectors is mostly at NHC S1, second smallest mostly NHC S2, second largest mostly at NHC S3 and largest fraction of support vectors mostly at NHC DEoptim.

The optimal hyper-parameter values as well as the fraction of support vectors for the NHC varies a lot more than the optimal hyper-parameter values and the fraction of support vectors for SVMC when we look at the `DEoptim()` results. The hyper-parameter values vary more for NHC DEoptim than for SVMC DEoptim for most of the datasets.

To minimise the error rate in SVMC it will be best to use `DEoptim()` and to minimise the fraction of support vectors used for SVMC it will be best to use S3. If we want to minimise the error rate for NHC, the best option is to use `DEoptim()` and S1 will minimise the fraction of support vectors used. The hyper-parameter values for S1, S2 and S3 do not vary a lot.

Looking only at NHC and comparing `DEoptim()` with the `sigest()` results, we can clearly see that it is better to use `DEoptim()` rather than the estimated `sigest()` values. The error rate values are mostly much higher for the `sigest()` values than for `DEoptim()`. The SVMC gets good results when using `sigest()` as well as `DEoptim()` although the variation for `sigest()` is higher. It seems like the `DEoptim()` approach for SVMC gives a better error rate for most of the datasets than the `sigest()` approach.

Penalised LDA performs only slightly worse than SVMC and NHC in all of the datasets except for the Vehicle data where it performs very badly when compared to the rest of the classification techniques. The Khan dataset that has more variables than objects is ideal for the penalised LDA, but RF and SVMC still performed better than the penalised LDA.

CHAPTER 6
CONCLUSION

This thesis investigated mainly the different aspects of multi-class classification. In Chapter 2 we studied the development of a single hypersphere for a single dataset. By fitting a hypersphere around all the data in a Hilbert space, we derived the all enclosing hypersphere. The hypersphere was then adjusted to allow for outliers in the dataset. By doing so, we derived the ν-soft hypersphere which also brought about an extra parameter (C) that needs to be chosen. For $C = 1$ we would obtain the all enclosing hypersphere and for $\frac{1}{n} < C < 1$ we have the ν-soft hypersphere which also resulted in an outlier/novelty detector. Both the all enclosing and ν-soft hyperspheres resulted in a support region in input space. We demonstrated that the solution to the hypersphere is only dependent on the support vectors which are just a small fraction of the dataset. We concluded this chapter by extending the hyperspheres to multi-class classification by fitting a hypersphere around each class in the Hilbert space. From this the multi-class nearest hypersphere classifier (NHC) was derived.

To study the classification performance of NHC, we introduced three popular classification techniques in Chapter 3 for comparison. These techniques were the support vector machine, random forests and penalised LDA. Support vector machines and random forests have been studied extensively by researchers and are well-known for their excellent classification performance. Penalised LDA is a relatively new technique, but is essentially an extension of Fisher's LDA. Penalised LDA uses the LASSO penalty to do variable selection. It was specifically developed to overcome the singularity problem in Fisher's LDA and to select the relevant variables in high-dimensional classification ($n \ll p$).

Chapter 4 is an empirical study of the multi-class NHC. The aim of the study was to investigate the behaviour of the cross-validation error rates and the fraction of support vectors in the training data when doing classification. This was done for different values of the hyper-parameter of the Gaussian kernel (which was used throughout the thesis). Since the support vector machine has similar characteristics to NHC, we also included the SVMC in this investigation for comparison. Very interesting patterns about the error rates under NHC and SVMC resulted from a simulation study as well as a study on real-world data. For most of the results we noted a definite minimum value in the error rate versus the hyper-parameter of the Gaussian kernel. The error rate generally seems to have a minimum for a very small hyper-

parameter value. Interesting patterns have also been discovered if we study the fraction of support vectors for NHC and SVMC. There seems to be a minimum fraction of support vectors if we choose the hyper-parameter appropriately. This minimum is also at a small hyper-parameter value. The hyper-parameter value resulting in the minimum error rate also results in a smaller faction of support vectors. This raised the question of how to obtain the optimal hyper-parameter when the error rate is a minimum.

In Chapter 5 we compared the classification performance of NHC to that of SMVC, random forests and penalised LDA. We used two methods to obtain the optimal hyper-parameter for the Gaussian kernel. One of the methods uses a built-in R function `sigest()` to estimate three values for the hyper-parameter. This method is only recommended when using the Gaussian or Laplacian kernel. This method is not dependent on the classification technique and is used before any model is fitted. The other method uses a general purpose optimisation procedure called Differential Evolution optimization. The function `DEoptim()` in R was used here. Using `DEoptim()` to find the optimal hyper-parameter value is similar to doing a grid search. However, with this procedure we find the optimal value over a continuous interval. This procedure is dependent on the classification technique and finds the optimal hyper-parameter by minimizing the error rate. It can also be used with other kernels besides the Gaussian kernel. In the study we discovered that using `DEoptim()` could still be valuable in searching for the optimal hyper-parameter as it gives smaller error rates than `sigest()` for both SVMC and NHC. As a classifier NHC does not perform as well as SVMC and random forests, but does perform better than penalised LDA. There are however several aspects of NHC that may be useful.

The positive aspects learned from this thesis can be summarised as follows:

- It extends naturally to the multi-class case since you only have to fit a hypersphere around each class and classify cases to the nearest hypersphere.
- It is especially helpful in classification problems where the classes are best separable using a non-linear classifier.
- It is possible to handle problems where $n \ll p$, even though no research has been done yet about how the NHC performs in high-dimensional data settings.
- We can also derive posterior probabilities for NHC analogous to linear discriminant analysis with normal distributions.
- The hyperspheres used in NHC allow for sparsity in the number of observations used (only support vectors are needed). This is a property similar to the support vector machine.

- Hyperspheres can be used to construct an outlier detector. Using this property in NHC allows us to remove the effect of outliers while deriving the NHC classifier.
- NHC is non-parametric and does not make any distribution assumptions about data.

Some of the less positive aspects are:

- NHC makes use of a kernel function. Thus, choosing the appropriate kernel function is an open question.
- NHC requires a few parameters to be estimated. One is the C parameter (which controls the number of outliers) and the other is the kernel hyper-parameter. This adds to the number of computations required by NHC.
- Choosing the appropriate similarity function is also an open question.

Based on the research conducted here the following items are highlighted for further research:

- NHC allows for sparsity in the number of observations used, but not in the variables. Variable selection should be investigated and applied, which will improve the classification performance.
- In this thesis, we used only one similarity function. Other similarity functions should be implemented, which could possibly improve the classification performance of NHC.
- This thesis studied one dataset with $n \ll p$. We saw that NHC performs quite well for this dataset. More extensive research for datasets with $n \ll p$ will give a better idea on the performance of NHC for these types of datasets.
- Other kernel functions (we used the Gaussian kernel in this thesis) might improve the classification performance of NHC.

REFERENCES

1) Ardia, D., Mullen K., Peterson B., Ulrich J., and Boudt, K. (2016). *DEoptim: Global Optimization by Differential Evolution*. R package version 2.2-4, URL https://cran.r-project.org/web/packages/DEoptim/DEoptim.pdf
2) Boser, B.E., Guyon, I.M. and Vapnik, N.V. (1992). *A training algorithm for optimal margin classifiers*. In D. Haussler, editor, Proceedings of the 5th annual ACM workshop on Computational Learning Theory, pp.144-152.
3) Breiman, L. 1996. *Bagging Predictors*. Machine Learning, 26, no 2, 123-140.
4) Breiman, L. 2001. *Random Forests*. Statistics Department of California at Berkley, CA 94720.
5) Breiman, L., Friedman, J., Olshen, R. and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth.
6) Coetzer, F. 2015. *General Purpose Optimisation in R with Applications in Support Vector Machine Classification.* Stellenbosch University.
7) Deng, N., Tian, Y. & Zhang, C. 2013. *Support vector machines.* Minnesota: CRP Press.
8) Fisher, RA. 1936. *The Use of Multiple Measures in Taxonomic Problems.* Annals of Eugenics:179-188
9) Freund, Y and Schapire, R. 1997. *A decision-theoretic generalization of on-line learning and an application to boosting*. Journal of Computer and System Sciences, 55(1):119.
10) Gareth, J., Witten, D., Hastie, T. & Tibshirani, R. 2013. *An Introduction to Statistical Learning with Applications in R*. New York: Springer.
11) Gu, L. and Wu, HZ. 2008. *A kernel-based fuzzy greedy multiple hyperspheres covering algorithm for pattern classification.* Neurocomputing 72: 313-320.
12) Hao PY., Chiang JH., Lin YH. 2009. *A new maximal-margin spherical-structured multi-class support vector machine.* Appl Intell 30(2):98–111
13) Hastie, T., Tibshirani, R. & Friedman, J. 2009. *The Elements of Statistical Learning.* New York: Springer-Verlag
14) Izenman, A.J., 2008. *Modern Multivariate Statistical Techniques*. New York: Springer.
15) Karatzoglou, A., Smola, A., and Hornik, K (2016). *Kernel-Based Machine Learning Lab.* R package version 0.9-25, URL https://cran.r-project.org/web/packages/kernlab/kernlab.pdf
16) Karatzoglou, A., Smola, A., Hornik, K. and Zeileis, A. 2004. Kernlab – an S4 Package for Kernel Methods in R. Paper presented at Research Report Series. August 2004.
17) Liaw, A. and Wiener, M. (2015). *Breiman and Cutler's Random Forests for Classification and Regression.* R package version 4.6-12, URL https://cran.r-project.org/web/packages/randomForest/randomForest.pdf

18) Scholkopf, B., Burges, C. and Vapnik, V. 1995. *Extracting Support data for a given task.* In U. M. Fayyad and R. Uthurusamy, editors, Proceedings, First International Conference on Knowledge Discovery & Data Mining. AAAI Press, Menlo Park, CA.

19) Song, QJ., Xiao, XM., Jiang, HY. and Zhao, XG. 2015. *A new multi-class classification method based on minimum enclosing balls*. Journal of Mechanical Science and Technology 29 (8). 3467-3473.

20) Tang, D. *Building a classification tree in R*. 2013. Dave Tang's Blog. URL https://davetang.org/muse/2013/03/12/building-a-classification-tree-in-r/

21) Tax, D.M.J. & Duin, R.P.W., 1999. *Support Vector Domain Description*. Pattern Recognition Letters 20. 1191-1199.

22) Tax, D.M.J., 2001. "One-class classification." PhD thesis, Technischen Universität Berlin

23) Van der Westhuizen, S. 2014. *Nearest Hypersphere Classification: A Comparison with Other Classification Techniques.* Stellenbosch University.

24) Vapnik, V. *Statistical Learning Theory*. Wiley, New York, 1998.

25) Wang, J., Neskovic, P. and Cooper, LN. 2006. *Bayes classification based on minimum bounding spheres.* Neurocomputing 70: 801-808.

26) Weston, J. and Watkins, C. 1998. *Multi-class Support Vector Machines.* Technical Report. Royal Holloway University of London.

27) Witten, D.M. and Tibshirani, R. 2011. *Penalized classification using Fisher's linear discriminant.* Journal of the Royal Statistical Society, Series B.

28) Witten, D. (2015). *Penalized Classification using Fisher's Linear Discriminant.* R package version 1.1, URL https://cran.r-project.org/web/packages/penalizedLDA/penalizedLDA.pdf

APPENDIX A

FUNCTIONS IN R WRITTEN FOR CHAPTER 2

A.1 Function to plot support regions and decision boundary for hyperspheres

```
Chap2.NHC.plots<-function(x=scale(iris.data)[,1:2],y=iris.class,
             gamma=0.2,Cval=1,col.vec=c("red","green","blue"),
             plot.support.region.only=FALSE,grid.size=100)
{
  class.names=unique(y)
  plot.colours<-as.character(y)
  plot.pch<-as.character(y)
  for (i in 1:length(class.names))
  {
    plot.colours[plot.colours==class.names[i]]<-col.vec[i]
    plot.pch[plot.pch==class.names[i]]<-i
  }
  ### Find minimum x and y values for plot
  xmin<-min(x[,1])-0.5
  xmax<-max(x[,1])+0.5
  ymin<-min(x[,2])-0.5
  ymax<-max(x[,2])+0.5

  xaxis<-c(xmin,xmax)
  yaxis<-c(ymin,ymax)

  ### Create grid for test data
  x1grid<-seq(xmin,xmax,length=grid.size)
  x2grid<-seq(ymin,ymax,length=grid.size)
  test.data<-expand.grid(x1grid,x2grid)

  ### plot training points
  plot(x[,1], x[,2], type="n", xlab="x1", ylab="x2", xlim=xaxis,
             ylim=yaxis)
  points(x,col=plot.colours,pch=as.numeric(plot.pch))
```

```
  ### Plot the support regions

  for(i in 1:num.classes){

    gdata<-x[y==class.names[i],]

    out.test<-
             MultiClass.NHC(gdata,rep(1,nrow(gdata)),test.data,kerne
             l.type=rbfdot,kernel.parameters=gamma,C.val=Cval,
             return.classification.only=FALSE)

    r2<-out.test$radii^2

    xp<-seq(xmin,xmax,length=grid.size)

    np<-length(xp)

    yp<-seq(ymin,ymax,length=grid.size)

    zp<-out.test$points.to.classify.radii-r2

    contour(xp, yp, matrix(zp,np), add=T, drawlabels=F, levels=0,
             lty=1, lwd=2, col=col.vec[i])

  }


  ### plot the decision bounds

  if (!plot.support.region.only)

  {

    zp <-MultiClass.NHC(data=x, class.vector=y,
             points.to.classify=test.data, kernel.type=rbfdot,
                kernel.parameters=gamma, Cval,
                     return.classification.only=FALSE)

    zp.NHC<-zp$class.names.of.points


    for (i in 1:(length(class.names)-1))

      contour(x1grid, x2grid, matrix(zp.NHC==class.names[i],
             grid.size), levels=0.5, add=T, drawlabels=F, lwd=2,
                lty=2, col="black", labex=0)

    line.data<-cbind(zp.NHC,test.data)

    for (i in 1:length(class.names))

    {

      ci<-line.data[line.data[,1]==class.names[i],-1]

      points(ci,col=col.vec[i],pch=".")

    }

  }
}
```

APPENDIX B

FUNCTIONS IN R WRITTEN FOR CHAPTER 3

B.1 Function for producing decision boundary plot for SVMC:

```
library(kernlab)
y<-iris[,5]
x<-scale(iris)[,1:2]


### fit model
iris.svmc<-ksvm(as.factor(y)~.,data=x,
                type="kbb-svc",C=1,kpar=list(sigma=1))


### find minimum and maximum values for x and y axis
xmin<-min(x[,1])
xmax<-max(x[,1])
ymin<-min(x[,2])
ymax<-max(x[,2])


### Create 100 x 100 grid for test dataset
x1<-seq(xmin,xmax,length=100)
x2<-seq(ymin,ymax,length=100)
test.data<-expand.grid(x1,x2)
colnames(test.data)<-c("Sepal.Length","Sepal.Width")


### predict class of test dataset
zp <-predict(iris.svmc,test.data)
### Plot training points
colour<-c(rep("red",50),rep("green",50),rep("blue",50))
plot(x,col=colour, pch=c(rep(1,50),rep(2,50),rep(3,50)))


### Draw decision bounds
contour(x1, x2, matrix(zp=="setosa",100), levels=0.5, add=T,
          drawlabels=F, lwd=2, lty=2, col="black", labex=0)
```

```
contour(x1, x2, matrix(zp=="virginica",100), levels=0.5, add=T,
          drawlabels=F, lwd=2, lty=2, col="black", labex=0)


### Plot test dataset (grid) points
line.data<-cbind(zp,test.data)


c1<-line.data[line.data[,1]=="setosa",-1]
c2<-line.data[line.data[,1]=="versicolor",-1]
c3<-line.data[line.data[,1]=="virginica",-1]


points(c1, col="red", pch=".")
points(c2, col="green", pch=".")
points(c3, col="blue", pch=".")
```

B.2 Function for producing decision boundary plot for Random Forest:

```
library(randomForest)


y<-iris[,5]
x<-scale(iris)[,1:2]


### fit model
iris.rf <- randomForest(x,as.factor(y))


### find minimum and maximum values for x and y axis
xmin<-min(x[,1])
xmax<-max(x[,1])
ymin<-min(x[,2])
ymax<-max(x[,2])


### Create 100 x 100 grid for test dataset
x1<-seq(xmin,xmax,length=100)
x2<-seq(ymin,ymax,length=100)
test.data<-expand.grid(x1,x2)
colnames(test.data)<-c("Sepal.Length","Sepal.Width")


### predict class of test dataset
zp <- predict(iris.rf,newdata=as.matrix(test.data))



### Plot training points
colour<-c(rep("red",50),rep("green",50),rep("blue",50))
plot(x,col=colour,pch=c(rep(1,50),rep(2,50),rep(3,50)))


### Draw decision bounds
contour(x1,x2,matrix(zp=="setosa",100),levels=0.5,add=T,drawlabels=F
             ,lwd=2,lty=2,col="black",labex=0)
```

```
contour(x1,x2,matrix(zp=="virginica",100),levels=0.5,add=T,drawlabel
            s=F,lwd=2,lty=2,col="black",labex=0)


### Plot test dataset (grid) points
line.data<-cbind(zp,test.data)


c1<-line.data[line.data[,1]=="setosa",-1]
c2<-line.data[line.data[,1]=="versicolor",-1]
c3<-line.data[line.data[,1]=="virginica",-1]


points(c1,col="red",pch=".")
points(c2,col="green",pch=".")
points(c3,col="blue",pch=".")
```

APPENDIX C

FUNCTIONS IN R WRITTEN FOR CHAPTER 4

C.1 Function to create k different Training and Validation dataset splits

```
Create.Validation.index<- function(data=iris.data, class=iris.class,
      k.fold=5)
{
  class.names<-unique(class)
  num.groups<-length(class.names)
  data.list<-list()
  length(data.list)<-num.groups
  for (i in 1:num.groups)
    data.list[[i]]<-data[class==class.names[i],]
    validation.index.data<-list()
  length(validation.index.data)<-k
  for (i in 1:num.groups)
  {
    n.per.group<-nrow(data.list[[i]])
    sample.index<-sample(1:n.per.group,n.per.group)
    validation.split<-trunc(n.per.group/k)
    last.group.extra<-n.per.group-k*validation.split
  for (j in 1:k)
  {
    validation.index.data[[j]][[i]]<-
    sample.index[(1+(j-1)*validation.split):(validation.split*j)]
    if (j==k&&last.group.extra>0)
      validation.index.data[[j]][[i]]<-
       c(validation.index.data[[j]][[i]],
        sample.index[(n.per.group-last.group.extra+1):n.per.group])
    }
  }
  return(validation.index.data)
}
```

C.2 Function to return error rates and fraction of support vectors for gamma values between 0 and 5

```
Data.behaviour<-function(data=iris.data,class=iris.class, C.val=1,
    gammas=gamma.values, validation.index.mat=iris.validation.index)
{
  errors.svmc<-rep(0,length(gammas))
  sv.svmc<-rep(0,length(gammas))
  errors.nhc<-rep(0,length(gammas))
  sv.nhc<-rep(0,length(gammas))

  p<-ncol(data)
  class.names<-unique(class)
  num.groups<-length(class.names)

  data.list<-list()
  length(data.list)<-num.groups
  for (i in 1:num.groups)
    data.list[[i]]<-data[class==class.names[i],]

  Test.NHC<-function(gamma, Training.mat,TrainingClass,
     Validation.mat,ValidationClass, C.par)
  {
    output<-MultiClass.NHC(data=Training.mat,
           class.vector=TrainingClass,
                points.to.classify=Validation.mat,
                   kernel.type=rbfdot, kernel.parameters=gamma,
                     C.val=C.par, return.classification.only=FALSE)
    error<-1-sum(ValidationClass==output$class.names.of.points)/
                length(ValidationClass)
    return(list(error=error,num.sv=output$num.support.vec))
  }
  for (j in 1:length(gammas))
  {
    error=0
    sv=0
```

```
    for (k in 1:5)

    {

      validation.mat<-NULL

      validation.class<-NULL

      training.mat<-NULL

      training.class<-NULL

      for (m in 1:num.groups)

      {

        validation.mat<-rbind(validation.mat,
           data.list[[m]][validation.index.mat[[k]][[m]],])

        validation.class<-c(validation.class,
           rep(m,length(validation.index.mat[[k]][[m]])))

        training.mat<-rbind(training.mat,
           data.list[[m]][-validation.index.mat[[k]][[m]],])

        training.class<-c(training.class, rep(m,nrow(data.list[[m]])
           - length(validation.index.mat[[k]][[m]])))

      }

      model.nhc<-Test.NHC(gamma=gammas[j],
           Training.mat=training.mat, TrainingClass=training.class,
                Validation.mat=validation.mat,
                     ValidationClass=validation.class, C.par=C.val)

      error<-error+model.nhc$error

      sv<-sv+sum(model.nhc$num.sv)/(nrow(training.mat))

    }

    errors.nhc[j]<-error/5

    sv.nhc[j]<- sv/5


    model.ksvm<-ksvm(data,class, type="kbb-svc", kernel=rbfdot,
                 kpar=list(sigma=gammas[j]), C=C.val,
                      prob.model=FALSE, cross=5)

    errors.svmc[j]<- cross(model.ksvm)

    sv.svmc[j]<- nSV(model.ksvm)/nrow(data)

  }

  return(cbind(error.svmc=errors.svmc, sv.svmc=sv.svmc,
     error.nhc=errors.nhc, sv.nhc=sv.nhc))

}
```

C.3 Function to plot error rates and fraction of support vectors vs gamma values

```
plot.graphs<-function(output,C,gammas=gamma.values)

{

  par(mfrow=c(1,2))

  plot(gammas,output[,"error.svmc"], ty="l", ylim=c(0,1),
      main=paste("SVMC C=",C,sep=""), ylab="", xlab="gamma")

  lines(gammas,output[,"sv.svmc"],lty=2)

  legend(x=2, y=0.3, legend=c("error rates", "support vectors"),
      lty=c(1,2), cex=0.5)

  plot(gammas, output[,"error.nhc"], ty="l", ylim=c(0,1),
      main=paste("NHC C=",C,sep=""), ylab="", xlab="gamma")

  lines(gammas,output[,"sv.nhc"],lty=2)

  legend(x=2, y=0.3, legend=c("error rates","support vectors"),
      lty=c(1,2), cex=0.5)

}
```

C.4 R script for Simulation Study

```
# Create 3 mean vectors for the 3 classes

mu1<-rep(0,5)

mu2<-rep(1,5)

mu3<-c(0,0,1,1,1)


# Create 2 Sigma matrices.

Sigma.with.corr<-matrix(0.7,nrow=5,ncol=5)

diag(Sigma.with.corr)<-rep(1,5)

Sigma.no.corr<-diag(rep(1,5))


### Create the 4 simulation datasets ###


# The simulation study will be repeated 20 times and we will
# create 20 different datasets for each simulated dataset
# configuration

# Each dataset will have 20 rows. Each row is a different simulated
# dataset.


data1.1<-matrix(mvrnorm(100*20,mu1,Sigma.no.corr), nrow=20,
     byrow=TRUE)

data1.2<-matrix(mvrnorm(100*20,mu2,Sigma.no.corr),nrow=20,
     byrow=TRUE)

data1.3<-matrix(mvrnorm(100*20,mu3,Sigma.no.corr),nrow=20,
     byrow=TRUE)

data1<-cbind(data1.1,data1.2,data1.3)

class1<-c(rep(1,100),rep(2,100),rep(3,100))


data2.1<-matrix(mvrnorm(100*20,mu1,Sigma.with.corr),nrow=20,
     byrow=TRUE)

data2.2<-matrix(mvrnorm(100*20,mu2,Sigma.with.corr),nrow=20,
     byrow=TRUE)

data2.3<-matrix(mvrnorm(100*20,mu3,Sigma.with.corr),nrow=20,
     byrow=TRUE)

data2<-cbind(data2.1,data2.2,data2.3)

class2<-c(rep(1,100),rep(2,100),rep(3,100))
```

```
data3.1<-matrix(mvrnorm(400*20,mu1,Sigma.no.corr),nrow=20,
      byrow=TRUE)

data3.2<-matrix(mvrnorm(400*20,mu2,Sigma.no.corr),nrow=20,
      byrow=TRUE)

data3.3<-matrix(mvrnorm(400*20,mu3,Sigma.no.corr),nrow=20,
      byrow=TRUE)

data3<-cbind(data3.1,data3.2,data3.3)

class3<-c(rep(1,400),rep(2,400),rep(3,400))


data4.1<-matrix(mvrnorm(400*20,mu1,Sigma.with.corr),nrow=20,
      byrow=TRUE)

data4.2<-matrix(mvrnorm(400*20,mu2,Sigma.with.corr),nrow=20,
      byrow=TRUE)

data4.3<-matrix(mvrnorm(400*20,mu3,Sigma.with.corr),nrow=20,
      byrow=TRUE)

data4<-cbind(data4.1,data4.2,data4.3)

class4<-c(rep(1,400),rep(2,400),rep(3,400))


### Create a validation index for small and large dataset ###

validation.index.mat.100<-list()

length(validation.index.mat.100)<-5

validation.index.mat.400<-validation.index.mat.100


for (i in 1:20)

{

  sample100<-
     list(sample(1:100,100),sample(101:200,100),sample(201:300))

  sample400<-
    list(sample(1:400,400),sample(401:800,400),sample(801:1200,400))


  for (j in 1:5)

  {

    validation.index.mat.100[[j]]<-
           rbind(validation.index.mat.100[[j]],
                 c(sample100[[1]][(1+(j-1)*20):(20*j)],
                       sample100[[2]][(1+(j-1)*20):(20*j)],
                             sample100[[3]][(1+(j-1)*20):(20*j)]))
```

```
    validation.index.mat.400[[j]]<-
           rbind(validation.index.mat.400[[j]],
                c(sample400[[1]][(1+(j-1)*80):(80*j)],
                     sample400[[2]][(1+(j-1)*80):(80*j)],
                          sample400[[3]][(1+(j-1)*80):(80*j)]))
  }
}


find.errors<-function(data.to.use=data1, class=class1, C.val=1,
                gammas=gamma.vals,
                     validation.index.mat=validation.index.mat.100,
                          n.sim=20)
{
  errors.svmc<-matrix(0,ncol=n.sim,nrow=length(gammas))
  sv.svmc<-matrix(0,ncol=n.sim,nrow=length(gammas))
  errors.nhc<-matrix(0,ncol=n.sim,nrow=length(gammas))
  sv.nhc<-matrix(0,ncol=n.sim,nrow=length(gammas))


  Test.NHC<-function(gamma, Training.mat,TrainingClass,
     Validation.mat,ValidationClass, C.par)
  {
    output<-MultiClass.NHC(data=Training.mat,
           class.vector=TrainingClass,
                points.to.classify=Validation.mat,
                   kernel.type=rbfdot, kernel.parameters=gamma,
                     C.val=C.par, return.classification.only=FALSE)
    error<-1-sum(ValidationClass==output$class.names.of.points)/
              length(ValidationClass)
    return(list(error=error,num.sv=output$num.support.vec))
  }


  for (i in 1:n.sim)
  {
    use.data<-matrix(data.to.use[i,],ncol=5,byrow=TRUE)


    for (j in 1:length(gammas))
    {
      error=0
      sv=0
```

```
      for (k in 1:5)

      {

        val.index<-validation.index.mat[[k]][i,]

        model.nhc<-Test.NHC(gamma=gammas[j],
           Training.mat=use.data[-val.index,],
             TrainingClass=class[-val.index],
               Validation.mat=use.data[val.index,],
                 ValidationClass=class[val.index], C.par=C.val)

        error<-error+model.nhc$error

        sv<-sv+sum(model.nhc$num.sv)/
                (nrow(use.data)-length(val.index))

      }

      errors.nhc[j,i]<-error/5

      sv.nhc[j,i]<- sv/5


      model.ksvm<-ksvm(use.data,class, type="kbb-svc",
           kernel=rbfdot, kpar=list(sigma=gammas[j]), C=C.val,
                prob.model=FALSE, cross=4)

      errors.svmc[j,i]<- cross(model.ksvm)

      sv.svmc[j,i]<- nSV(model.ksvm)/nrow(use.data)

    }

  }


  return(cbind(error.svmc=apply(errors.svmc,1,mean),
     sv.svmc=apply(sv.svmc,1,mean),
          error.nhc=apply(errors.nhc,1,mean),
               sv.nhc=apply(sv.nhc,1,mean)))

}


gamma.values<-seq(from=0.00000001,to=5,length=100)


out1_0.1<-find.errors(data.to.use=data1, class=class1, C.val=0.1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.100, n.sim=20)

out1_1<-find.errors(data.to.use=data1, class=class1,C.val=1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.100, n.sim=20)

out1_5<-find.errors(data.to.use=data1, class=class1,C.val=5,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.100, n.sim=20)
```

```
out2_0.1<-find.errors(data.to.use=data2, class=class2, C.val=0.1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.100, n.sim=20)

out2_1<-find.errors(data.to.use=data2, class= class2,C.val=1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.100, n.sim=20)

out2_5<-find.errors(data.to.use=data2, class= class2,C.val=5,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.100, n.sim=20)


out3_0.1<-find.errors(data.to.use= data3, class=class3, C.val=0.1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.400, n.sim=20)

out3_1<-find.errors(data.to.use= data3, class= class3,C.val=1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.400, n.sim=20)

out3_5<-find.errors(data.to.use=data3, class=class3,C.val=5,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.400, n.sim=20)


out4_0.1<-find.errors(data.to.use= data4, class=class4, C.val=0.1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.400, n.sim=20)

out4_1<-find.errors(data.to.use= data4, class=class4,C.val=1,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.400, n.sim=20)

out4_5<-find.errors(data.to.use=data4, class=class4,C.val=5,
     gammas=gamma.values,
          validation.index.mat=validation.index.mat.400, n.sim=20)
```

C.5 R script for real-world data

```
### Iris dataset ###

iris.data<-scale(iris[,-5])

iris.class<-iris[,5]


iris.validation.index<-
Create.Validation.index(iris.data,iris.class)


iris.behaviour.C0.1<-Data.behaviour(C.val=0.1)

iris.behaviour.C1<-Data.behaviour(C.val=1)

iris.behaviour.C5<-Data.behaviour(C.val=5)


plot.graphs(iris.behaviour.C0.1,C=0.1)

plot.graphs(iris.behaviour.C1,C=1)

plot.graphs(iris.behaviour.C5,C=5)


### Glass dataset ###

glass.data<-scale(Glass[,1:9])

glass.class<-Glass[,10]


glass.validation.index<-
     Create.Validation.index(glass.data,glass.class)

glass.behaviour.C1<-Data.behaviour(data=glass.data,
     class=glass.class, C.val=1, gammas=gamma.values,
          validation.index.mat=glass.validation.index)

glass.behaviour.C0.3<-Data.behaviour(data=glass.data,
     class=glass.class,C.val=0.3,gammas=gamma.values,
          validation.index.mat=glass.validation.index)

glass.behaviour.C5<-Data.behaviour(data=glass.data,
     class=glass.class,C.val=5,gammas=gamma.values,
          validation.index.mat=glass.validation.index)


plot.graphs(glass.behaviour.C0.3,C=0.3)

plot.graphs(glass.behaviour.C1,C=1)

plot.graphs(glass.behaviour.C5,C=5)
```

```
### Vehicle dataset ###

vehicle.data<-scale(Vehicle[,-19])

vehicle.class<-Vehicle[,19]


vehicle.validation.index<-
     Create.Validation.index(vehicle.data,vehicle.class)

vehicle.behaviour.C1<-Data.behaviour(data=vehicle.data,
     class=vehicle.class,C.val=1,gammas=gamma.values,
          validation.index.mat=vehicle.validation.index)

vehicle.behaviour.C0.1<-Data.behaviour(data=vehicle.data,
     class=vehicle.class,C.val=0.1,gammas=gamma.values,
          validation.index.mat=vehicle.validation.index)

vehicle.behaviour.C5<-Data.behaviour(data=vehicle.data,
     class=vehicle.class,C.val=5,gammas=gamma.values,
          validation.index.mat=vehicle.validation.index)

plot.graphs(vehicle.behaviour.C0.1,C=0.1)

plot.graphs(vehicle.behaviour.C1,C=1)

plot.graphs(vehicle.behaviour.C5,C=5)
```

APPENDIX D

FUNCTIONS IN R WRITTEN FOR CHAPTER 5

D.1 Function to split data into training dataset, validation dataset and test dataset

```
Train.Valid.Test.Split<-function(data, class, train.proportion=0.5,
     validation.proportion=0.25)
{
  class.names<-unique(class)
  num.groups<-length(class.names)

  data.list<-list()
  length(data.list)<-num.groups
  for (i in 1:num.groups)
    data.list[[i]]<-data[class==class.names[i],]

  out.list<-list()
  length(out.list)<-3
  names(out.list)<-c("Training.data","Validation.data","Test.data")

  for (i in 1:num.groups)
  {
    n.per.group<-nrow(data.list[[i]])
    sample.index<-sample(1:n.per.group,n.per.group)
    train.size<-trunc(n.per.group*train.proportion)
    validation.size<-trunc(n.per.group*validation.proportion)

    out.list[[1]][[i]]<-sample.index[1:train.size]
    out.list[[2]][[i]]<-
      sample.index[(train.size+1):(train.size+validation.size)]
    out.list[[3]][[i]]<-
      sample.index[-(1:(train.size+validation.size))]
  }
  return(out.list)
}
```

D.2 Code to perform study in Chapter 5

```
empty.list<-list()

length(empty.list)<-50

glass.index50<-empty.list

vehicle.index50<-empty.list

abalone.index50<-empty.list

khan.index50<-empty.list

yeast.index50<-empty.list


for(i in 1:50)

{

  glass.index50[[i]]<-Train.Valid.Test.Split(glass.data,glass.class)

  vehicle.index50[[i]]<-
      Train.Valid.Test.Split(vehicle.data,vehicle.class)

  abalone.index50[[i]]<-
      Train.Valid.Test.Split(abalone.data,abalone.class)

  khan.index50[[i]]<-Train.Valid.Test.Split(khan.data,khan.class)

  yeast.index50[[i]]<-Train.Valid.Test.Split(yeast.data,yeast.class)

}


datasets.data.list<-
     list(glass.data,vehicle.data,abalone.data,khan.data,yeast.data)

datasets.class.list<-
      list(glass.class,vehicle.class,abalone.class,khan.class,
            yeast.class)

datasets.index.list<-
      list(glass.index50,vehicle.index50,abalone.index50,
            khan.index50,yeast.index50)

errors.list<-list()

length(errors.list)<-5

gamma.list<-errors.list

gamma.matrix<-matrix(0,nrow=50,ncol=8)

colnames(gamma.matrix)<-c("svmc.DE","svmc.S1","svmc.S2", "svmc.S3",
      "nhc.DE", "nhc.S1","nhc.S2", "nhc.S3")

empty.matrix<-matrix(0,nrow=50,ncol=18)
```

```
colnames(empty.matrix)<-c("errors.svmc.DE", "sv.svmc.DE",
     "errors.svmc.S1","sv.svmc.S1", "errors.svmc.S2", "sv.svmc.S2",
          "errors.svmc.S3","sv.svmc.S3", "errors.nhc.DE",
               "sv.nhc.DE", "errors.nhc.S1","sv.nhc.S1",
                    "errors.nhc.S2", "sv.nhc.S2",
                         "errors.nhc.S3","sv.nhc.S3", "errors.rf",
                              "errors.fisher")


for (i in 1:5)
{
  errors.list[[i]]<-empty.matrix
  gamma.list[[i]]<-gamma.matrix
}


Data.behaviour<-function(C.val=1,n.rep=50)
{
  for (i in 1:5)
  {
    for (j in 1:n.rep)
    {
      data<-datasets.data.list[[i]]
      class<-datasets.class.list[[i]]
      data.index<-datasets.index.list[[i]][[j]]
      p<-ncol(data)

      class.names<-unique(class)
      num.groups<-length(class.names)
      for (k in 1:num.groups)
        data.list[[k]]<-data[class==class.names[k],]

      validation.mat<-NULL
      validation.class<-NULL
      training.mat<-NULL
      training.class<-NULL
      test.mat<-NULL
      test.class<-NULL
```

```
for (m in 1:num.groups)

{

  validation.mat<-rbind(validation.mat,
     data.list[[m]][data.index$Validation.data[[m]],])

  validation.class<-c(validation.class,
     rep(m,length(data.index$Validation.data[[m]])))

  training.mat<-rbind(training.mat,
     data.list[[m]][data.index$Training.data[[m]],])

  training.class<-c(training.class,
     rep(m,length(data.index$Training.data[[m]])))

  test.mat<-rbind(test.mat,
     data.list[[m]][data.index$Test.data[[m]],])

  test.class<-c(test.class,
     rep(m,length(data.index$Test.data[[m]])))

}


####### NHC function to find gamma using DEoptim ##########

Test.NHC.error<-function(gamma.value,
     Training.mat=training.mat,
          Training.class=training.class,
               Validation.mat=validation.mat,
                    Validation.class=validation.class,
                         error.only=TRUE)

{

  output<-MultiClass.NHC(data=Training.mat,
     class.vector=Training.class,
          points.to.classify=Validation.mat,
               kernel.type=rbfdot,
                    kernel.parameters=gamma.value, C.val,
                         return.classification.only=FALSE)

  error<-1-
     sum(Validation.class==output$class.names.of.points)/
          length(Validation.class)

  if (error.only)

  {

    return(error=error)

  }else{

    return(c(error=error,
        proportion.sv=sum(output$num.support.vec)/
           (nrow(Training.mat))))

  }

}
```

```
################## DE optim for NHC ########################

DE.nhc.out<-DEoptim(fn=Test.NHC. error,lower=0.000000001,
     upper=7, DEoptim.control(itermax=5))

nhc.optimal.gamma<-DE.nhc.out$optim$bestmem[[1]]

model.nhc<-Test.NHC.error(nhc.optimal.gamma,
     Training.mat=training.mat, Training.class=training.class,
          Validation.mat=test.mat,
            Validation.class=test.class, error.only = FALSE)

errors.list[[i]][j,c("errors.nhc.DE","sv.nhc.DE")]<-model.nhc

gamma.list[[i]][j,"nhc.DE"]<-nhc.optimal.gamma


######## SVMC function to find gamma using DEoptim ##########

Test.SVMC.error<-function(gamma.value,
     Training.mat=training.mat, Training.class=training.class,
          Validation.mat=validation.mat,
               Validation.class=validation.class,
                    error.only=TRUE)

{

  output<-ksvm(as.factor(Training.class)~., data=Training.mat,
     type="kbb-svc", kpar=list(sigma=gamma.value), C=1)

  error<-1-
     sum(predict(output,Validation.mat)==Validation.class)/
          length(Validation.class)


  if (error.only)

  {

    return(error=error)

  }else{

   return(c(error=error,
          proportion.sv=nSV(output)/(nrow(Training.mat))))

  }

}



################ DE optim for SVMC ######################

DE.svmc.out<-DEoptim(fn=Test.SVMC.error, lower=0.000000001,
     upper=7, DEoptim.control(itermax=5))


svmc.optimal.gamma<-DE.svmc.out$optim$bestmem[[1]]
```

```
model.svmc<-Test.SVMC.error(svmc.optimal.gamma,
          Training.mat=training.mat, Training.class=training.class,
               Validation.mat=test.mat,
                    Validation.class=test.class, error.only=FALSE)

     errors.list[[i]][j,c("errors.svmc.DE","sv.svmc.DE")]<-
          model.svmc

     gamma.list[[i]][j,"svmc.DE"]<-svmc.optimal.gamma


     ################# sigest for NHC and SVMC ##################

     sigest.values <- sigest(as.factor(training.class)~.,
                          training.mat)

     for (h in 1:3)

     {

       gamma.list[[i]][j,
          c(paste("nhc.S",h,sep=""),paste("svmc.S",h,sep=""))]<-
                     sigest.values[[h]]

       nhc.model<-Test.NHC.error(sigest.values[[h]],
          Training.mat=training.mat, Training.class=training.class,
               Validation.mat=test.mat,
                  Validation.class=test.class, error.only = FALSE)

        errors.list[[i]][j,c(paste("errors.nhc.S",h,sep=""),
          paste("sv.nhc.S",h,sep=""))]<-nhc.model


       svmc.model<-ksvm(as.factor(training.class)~.,
          data=training.mat, type="kbb-svc",
               kpar=list(sigma=sigest.values[[h]]), C=1)

       svmc.test.error<-1-
          sum(predict(svmc.model,test.mat)==test.class)/
               length(test.class)

       svmc.test.sv<-nSV(svmc.model)/length(training.class)

       errors.list[[i]][j,c(paste("errors.svmc.S",h,sep=""),
          paste("sv.svmc.S",h,sep=""))]<-
               c(svmc.test.error,svmc.test.sv)

     }


     ############### random forests ####################

     rf.model<-randomForest(training.mat,as.factor(training.class))

     rf.test.predict<-predict(rf.model,newdata=test.mat)

     errors.list[[i]][j,"errors.rf"]<-1-
     sum(predict(rf.model,newdata=test.mat)==test.class)/
          length(test.class)
```

```
      ############## Penalized LDA ######################

      plda.model<-PenalizedLDA.cv(training.mat,
            as.factor(training.class))

      out<-PenalizedLDA(training.mat, as.factor(training.class),
                  test.mat,lambda=plda.model$bestlambda,
                        K=plda.model$bestK)


     Mode.predict<-function(x)

      {

        unique.x <- unique(x)

        unique.x[which.max(tabulate(match(x, unique.x)))][[1]]

      }

      plda.mode.predict<-
            apply(matrix(out$ypred, nrow=length(test.class)), 1,
                  Mode.predict)

      errors.list[[i]][j,"errors.fisher"]<-1-
            sum(plda.mode.predict==test.class)/length(test.class)

    }

  }

  return(list(errors.and.sv=errors.list,gammas=gamma.list))

}


i=1

boxplot(errors.list[[i]][, c(1,3,5,7,9,11,13,15,17,18)],
     col=c(rep("red",4),rep("blue",4),"green","yellow"),
            ylim=c(0,1), ylab="error rate", names=c("SVMC.DE",
                  "SVMC.S1", "SVMC.S2", "SVMC.S3", "NHC.DE", "NHC.S1",
                        "NHC.S2", "NHC.S3", "RF", "P.LDA"))


boxplot(errors.list[[i]][,c(2,4,6,8,10,12,14,16)],
     col=c(rep("red",4),rep("blue",4)), ylab="fraction of support
            vectors", names=c("SVMC.DE", "SVMC.S1", "SVMC.S2",
                  "SVMC.S3", "NHC.DE", "NHC.S1", "NHC.S2", "NHC.S3"),
                        ylim=c(0,1))


abline(v=4.5)


boxplot(gamma.list[[i]][,c(1,5,2,3,4)],
     col=c("red","blue",rep("purple",3)), ylab="gamma value",
            names=c("SVMC.DE", "NHC.DE", "Sigest 1", "Sigest 2",
                  "Sigest 3"))
```

```
Final.Output<-function(ER.mat=errors.list[[2]],
                 Gam.mat=gamma.list[[2]])

{

  Final.Mat<-matrix(0,ncol=4,nrow=20)

  Final.Mat<-as.data.frame(Final.Mat)

  Final.Mat[,1]<-c("SVMC DEoptim", "", "SVMC S1", "", "SVMC S2", "",
     "SVMC S3", "", "NHC DEoptim", "", "NHC S1", "", "NHC S2", "",
          "NHC S3", "", "Random Forest", "", "Penalized LDA", "")

  colnames(Final.Mat)<-c("Technique", "Error rate",
     "Fraction of SV's", "Hyperparameter")

  ER.mean<-apply(ER.mat,2,mean)

  ER.sd<-apply(ER.mat,2,sd)

  Gam.mean<-apply(Gam.mat,2,mean)

  Gam.sd<-apply(Gam.mat,2,sd)

  for (i in 1:8)

  {

    Final.Mat[(i*2-1),2:3]<-round(ER.mean[(i*2-1):(i*2)],4)

    Final.Mat[(i*2),2:3]<-paste("(",
           round(ER.sd[(i*2-1):(i*2)],4),")",sep="")

    Final.Mat[(i*2-1),4]<-round(Gam.mean[i],4)

    Final.Mat[(i*2),4]<-paste("(",round(Gam.sd[i],4),")",sep="")

  }


  Final.Mat[c(17,19),2]<-round(ER.mean[17:18],4)

  Final.Mat[c(18,20),2]<-paste("(",round(ER.sd[17:18],4),")",sep="")

  return(Final.Mat)

}

glass.Final.Mat<-Final.Output(errors.list[[1]], gamma.list[[1]])

vehicle.Final.Mat<-Final.Output(errors.list[[2]], gamma.list[[2]])

abalone.Final.Mat<-Final.Output(errors.list[[3]], gamma.list[[3]])

khan.Final.Mat<-Final.Output(errors.list[[4]], gamma.list[[4]])

yeast.Final.Mat<-Final.Output(errors.list[[5]], gamma.list[[5]])
```